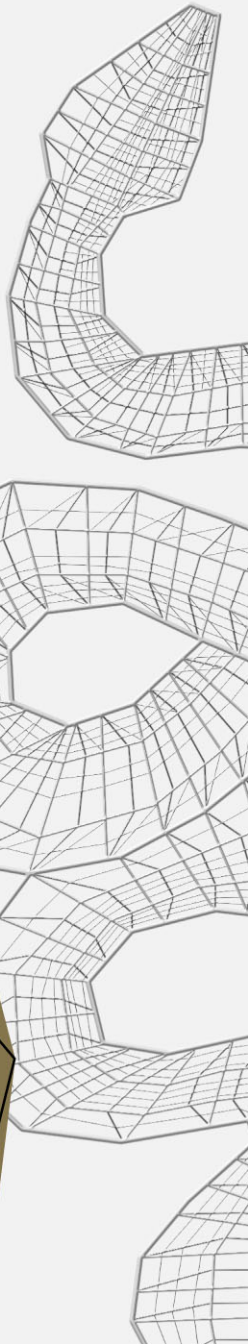
Васильев А.Н.

ПРОГРАММИРОВАНИЕ НА PYTHON

В ПРИМЕРАХ И ЗАДАЧАХ



- Основные синтаксические конструкции языка
- От работы с данными разных типов до создания потоков
- Разбор примеров и задачи для проверки пройденного
- Подходит для студентов и самостоятельного изучения

РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Васильев А.Н.

ПРОГРАММИРОВАНИЕ НА PYTHON

В ПРИМЕРАХ И ЗАДАЧАХ

Москва
2021

УДК 004.438
ББК 32.973.26-018.1
В19

Васильев, Алексей.

В19 Программирование на Python в примерах и задачах / Алексей Васильев. — Москва : Эксмо, 2021. — 616 с. — (Российский компьютерный бестселлер).

ISBN 978-5-04-103199-2

Сегодня существует много разных языков программирования. Некоторые из них популярны, а некоторые — не очень. Обычно популярность языка определяют по количеству программистов, которые используют его в своей работе на постоянной основе, или по запросам работодателей, которые ищут сотрудников-программистов. Долгие годы традиционно популярными являются языки программирования Java, C++, C#, JavaScript и PHP. В последнее время в этой великолепной компании все чаще упоминается язык программирования Python. Даже больше — по некоторым опросам язык Python уже занимает лидирующие позиции. Именно этому языку посвящена книга.

УДК 004.438
ББК 32.973.26-018.1

ISBN 978-5-04-103199-2

ОГЛАВЛЕНИЕ

Вступление
КНИГА О ЯЗЫКЕ ПРОГРАММИРОВАНИЯ PYTHON

Пошли, Скрипач, в открытый космос.

Из к/ф «Кин-дза-дза»

Сегодня существует много разных языков программирования. Некоторые из них популярны, другие не очень. Обычно популярность языка определяют по количеству программистов, которые на постоянной основе используют его в своей работе, или по запросам работодателей, ищущих сотрудников-программистов. Долгие годы традиционно популярными являются языки программирования Java, C++, C#, JavaScript и PHP. В последнее время в этой великолепной компании все чаще упоминается язык программирования Python. Более того, согласно некоторым опросам, язык Python уже занимает лидирующие позиции. Именно ему посвящена эта книга.

Язык Python

Обо мне придумано столько небылиц, что я устаю их опровергать.

Из к/ф «Формула любви»

Тенденции таковы, что даже если язык Python и не является самым популярным на сегодня, то все равно нет сомнений в том, что масштабы его применения постоянно растут. Соответственно, увеличивается спрос на программистов, работающих с языком Python. Такая возрастающая популярность языка во многом объясняется его простотой, красотой и эффективностью. Спектр задач, решаемых с использованием Python, довольно внушителен. Поэтому изучение Python — выбор вполне разумный и многообещающий.

Чем же замечателен Python? Что в нем особенного? Ответы на эти вопросы не такие уж и простые. Тем более что многое зависит от того, с каким языком мы будем его сравнивать. Среди наиболее важных характеристик языка Python можно выделить следующие.

- Язык *интерпретируемый*. При первом запуске программы на выполнение для нее создается промежуточный код. Именно промежуточный код используется при выполнении программы. Если впоследствии в программу вносятся изменения, то при очередном запуске программы создается новый промежуточный код.

НА ЗАМЕТКУ

Языки программирования бывают интерпретируемыми и компилируемыми. Если программа компилируется, то на основе исходного кода создается исполнительный (машинный) код, который и выполняется при запуске программы. Если речь идет об интерпретируемом языке, то программа, написанная на нем, выполняется построчно, без предварительной компиляции. Существует и промежуточный вариант — нечто среднее между компилированием и интерпретированием. В таком случае исходный код программы преобразуется в промежуточный код, который уже затем интерпретируется при выполнении.

Интерпретируемые языки позволяют больше вольности в описании и обработке данных. Программы, написанные на компилируемых языках, характеризуются относительно высокой скоростью выполнения.

- В плане синтаксиса язык Python прост и лаконичен. Он не содержит избыточных конструкций. С другой стороны, язык очень строгий: даже лишний пробел в программном коде может привести к ошибке.
- Язык Python поддерживает парадигму объектно-ориентированного программирования (ООП). Тем не менее он позволяет создавать программы, не использующие классы и объекты.

НА ЗАМЕТКУ

Концепция ООП, реализуемая в языке Python, может стать сюрпризом для читателей, знакомых с такими языками программирования, как Java, C++ и C#. Напротив, те, кто знаком с языком JavaScript, обнаружат для себя некоторые знакомые моменты.

- Язык Python удобен для создания приложений с графическим интерфейсом.

- Еще одним фактором, способствующим популярности языка Python, является большое и дружное сообщество разработчиков, использующих этот язык. Нет недостатка и в свободно распространяемых программных продуктах (включая среды разработки), облегчающих знакомство и использование языка Python.

Выше представлен лишь очень общий и краткий перечень достоинств и особенностей языка. В детали мы погрузимся в основной части книги, когда будем рассматривать конкретные примеры и синтаксические конструкции.

Особенности книги

> Пацак пацака не обманывает. Это некрасиво, родной...
>
> *Из к/ф «Кин-дза-дза»*

Цель этой книги — научить читателя программировать на языке Python. Но учиться можно по-разному. Скажем, можно слушать лекции в университете, можно посещать курсы по программированию, а можно пытаться научиться самостоятельно. Последний вариант — самый трудный, поскольку обычно рядом нет советчика, который мог бы подсказать или объяснить сложный момент. Вот именно для этого «сложного» случая в первую очередь и предназначена книга. Понятно, что совсем исключить «крутые повороты» при «прокладке маршрута» по изучению языка Python не получится. Но мы попытаемся свести к минимуму их количество.

Опыт показывает, что легче всего усвоить различные концепции программирования и подходы, когда они проиллюстрированы примерами. Как раз такая методика использована в этой книге. Принципиальная задача, которая при этом решается, — донести до читателя основную идею, причем не просто на некотором абстрактном уровне, а на уровне ее прикладной реализации с помощью программного кода. Теоретические сведения приводятся в объеме минимальном, но вместе с тем достаточном для качественного усвоения материала.

Структура книги такова, что в первой главе дается краткий обзор основных синтаксических конструкций языка Python. Это позволит читателю практически сразу, еще до завершения чтения книги, приступить к созданию несложных, но вполне функциональных программных

кодов. Этот прием применялся в книгах, посвященных другим языкам программирования, и получил неплохие отзывы читателей. Так что есть основания полагать, что он будет полезен и при изучении языка Python.

Главы после первой посвящены более детальному рассмотрению вопросов, связанных с эффективным программированием в Python. Среди рассмотренных тем: работа с данными разных типов, управляющие инструкции, списки и кортежи, множества и словари, работа с текстом, создание функций, операции с файлами, работа с классами и объектами, наследование и специальные методы, обработка исключительных ситуаций, создание потоков и многое другое. Последняя глава книги содержит полезную информацию, касающуюся создания приложений с графическим интерфейсом (с использованием библиотеки Tkinter). Для удобства усвоения материала каждая глава заканчивается кратким обобщением, в которое вынесены основные положения, рассмотренные и обсуждаемые в соответствующей главе. Также каждая глава содержит список заданий для самостоятельной работы.

НА ЗАМЕТКУ

Материал от главы к главе усложняется постепенно. Некоторые важные моменты достаточно часто повторяются (в разном контексте), особенно в начальных главах. Иногда одни и те же (или похожие) задачи решаются разными методами. Все это сделано намеренно. Цель простая — облегчить процесс усвоения информации и сформировать основы для понимания принципов программирования в Python.

Программное обеспечение

> Показывай свою гравицапу. Если фирменная вещь — возьмем!
>
> *Из к/ф «Кин-дза-дза»*

Для составления программных кодов мало знать язык программирования (в данном случае Python). Понадобится также определенное программное обеспечение. Какое именно? Не помешала бы программа-редактор для набора кода. Хотя собственно программный код мы можем набирать хоть в текстовом редакторе, вроде Notepad. Для этого нам достаточно создать пустой текстовый документ, внести в него команды в соответствии с правилами языка Python и сохранить файл

с расширением .py (стандартное расширение для файлов с программами на языке Python).

> **НА ЗАМЕТКУ**
>
> Помимо расширения .py файлы с Python-программами могут иметь расширение .pyw, если мы имеем дело с программами, в которых используется графический интерфейс (в операционной системе Windows). У файлов, связанных с Python-проектами, могут быть и другие расширения. Например расширение .pyc имеют файлы со скомпилированным промежуточным кодом (файлы с байт-кодом). Оптимизированный байт-код сохраняется в файле с расширением .pyo, а расширение.pyd используется для файлов с бинарным кодом динамических dll-библиотек в операционной системе Windows.

Но даже если мы так поступим, этого все же будет недостаточно. Нам еще как минимум понадобится программа-интерпретатор, которая сможет выполнить команды, написанные на языке Python. Другими словами, нам понадобится специальная программа, которая сможет понять код, который мы написали на языке Python, и исполнит этот код. Как отмечалось выше, такие программы называются *интерпретаторами*. Поэтому обойтись совсем без специального программного обеспечения мы не сможем. А поскольку программное обеспечение все равно придется устанавливать, то разумно воспользоваться всем спектром возможностей, доступных разработчику на языке Python. Тем более что предлагаемые для программирования на Python средства разработки довольно эффективны и часто бесплатны.

Самый разумный подход при создании программ на языке Python состоит в том, чтобы использовать *интегрированную среду разработки* (сокращенно *IDE* от *Integrated Development Environment*). Среда разработки — это специальное приложение, которое позволяет набирать, отлаживать и запускать на выполнение программные коды. Фактически среда разработки объединяет в себе сразу несколько программ. Это очень удобно, поскольку самые разные задачи, начиная с набора кода и до отладки приложения и запуска его на выполнение, реализуются через одну универсальную программу. Использовать среду разработки — разумно и удобно. Поэтому общая рекомендация состоит в том, чтобы использовать ее. Вопрос лишь, какую именно.

Существует довольно много сред разработки для языка Python. Здесь мы кратко остановимся лишь на некоторых, наиболее популярных

(и бесплатных). Но прежде чем перейти к обсуждению сред разработки, мы сделаем несколько замечаний относительно всего процесса установки программного обеспечения, необходимого для программирования на Python.

В первую очередь необходимо установить программу-интерпретатор (и некоторые сопутствующие утилиты). Для этого имеет смысл перейти на страницу поддержки языка `www.python.org`. Эта страница содержит много полезной информации. Там, кроме прочего, в разделе загрузок **Downloads** (адрес `www.python.org/downloads`) можно найти предназначенное для программирования на Python программное обеспечение. Веб-страница с ресурсами, предназначенными для загрузки, представлена на рис. В.1.

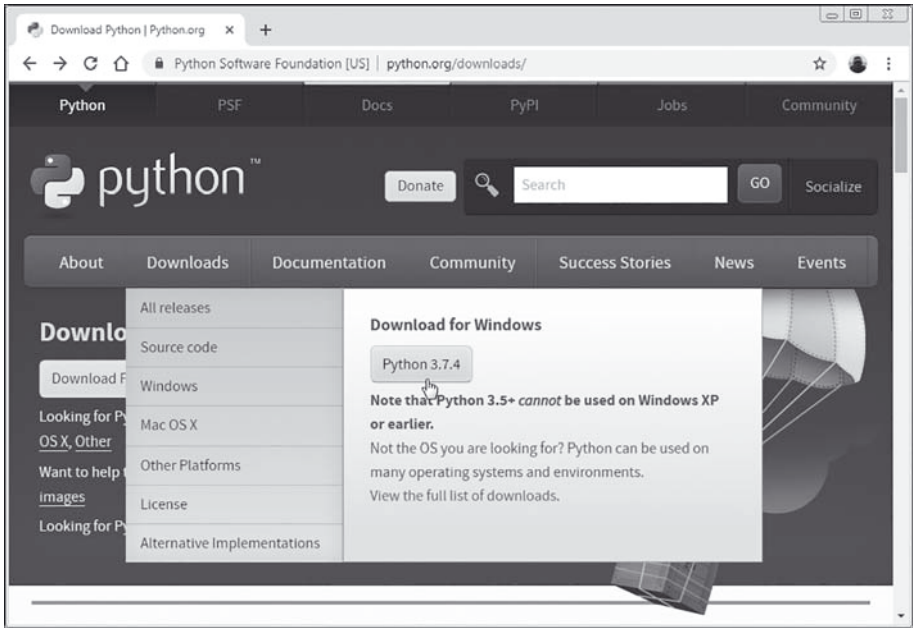


Рис. В.1. Страница *www.python.org/downloads* для загрузки программного обеспечения для программирования на Python

Следует загрузить соответствующие файлы и выполнить установку. Процесс установки простой и интуитивно понятный, поэтому особых комментариев не требует и обычно проходит без проблем. Стоит заметить, что в этом случае автоматически будет установлена и среда разработки, называемая IDLE. Это простая и надежная среда, которая вполне подойдет для эффективной работы с программными кодами на языке Python.

НА ЗАМЕТКУ

Методы работы со средой IDLE, равно как и с другими средами разработки, кратко описываются в первой главе.

Если читателя по каким-либо причинам среда IDLE не устроит, можно воспользоваться другой средой. Благо, выбор достаточно большой.

НА ЗАМЕТКУ

Обычно среды разработки устанавливаются без интерпретатора, поэтому рекомендуется сначала установить интерпретатор (например, загрузив файлы с сайта `www.python.org`), а уже после этого устанавливать среду разработки. В таком случае настройки среды, связанные с интерпретатором, скорее всего, будут выполнены автоматически.

Компания JetBrains предлагает для разработчиков на Python среду разработки, которая называется PyCharm. Информация об этой среде разработки (а также о других многочисленных разработках компании JetBrains) представлена на сайте `www.jetbrains.com`. На рис. В.2 на странице открыт раздел **Tools**, в котором есть ссылка для загрузки установочных файлов среды разработки PyCharm.

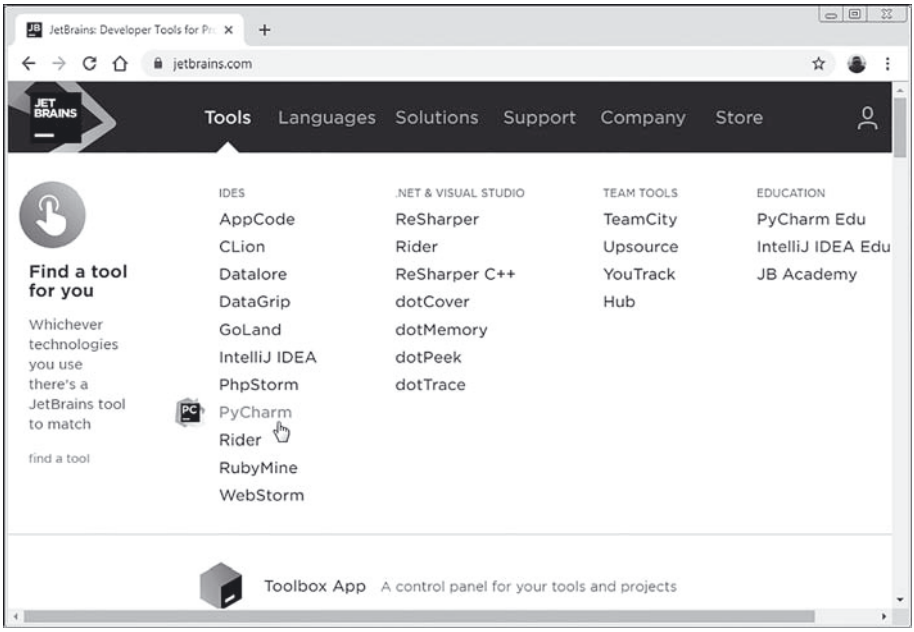


Рис. В.2. Страница *www.jetbrains.com* со ссылкой для загрузки установочных файлов среды разработки PyCharm

Процесс установки среды PyCharm достаточно простой. Это удобная и эффективная среда разработки. Правда, процесс создания приложений (по сравнению с тем, как это происходит при использовании других сред) может показаться немного запутанным, хотя это, конечно субъективное мнение. Вместе с тем среда PyCharm является, на мой взгляд, оптимальным выбором при работе с Python.

Достаточно удобной и функциональной является среда разработки Wing (продукт компании Wingware). На рис. В.3 показано окно браузера, в котором открыта страница `www.wingware.com` поддержки проекта.

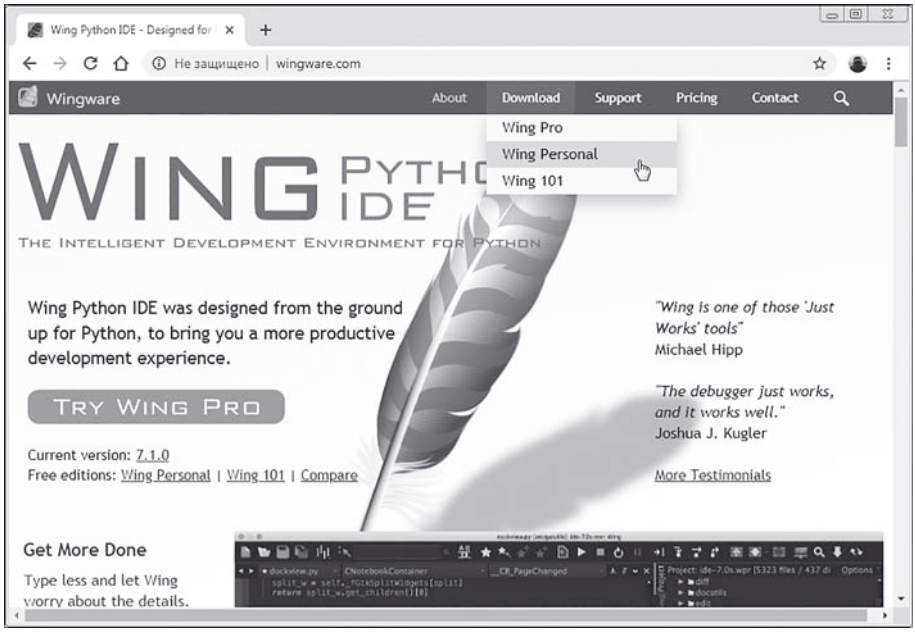


Рис. В.3. Страница *www.wingware.com* со ссылкой для загрузки установочных файлов среды разработки Wing

Среда разработки Wing проста в использовании и содержит все основные утилиты, необходимые для эффективного программирования на Python.

Наконец, стоит упомянуть среду разработки PyScripter. На рис. В.4 показано окно браузера, открытое на странице с адресом `http://sourceforge.net/projects/pyscripter/`.

Это страница ресурса SourceForge, с помощью которого можно загрузить последнюю версию среды разработки PyScripter.

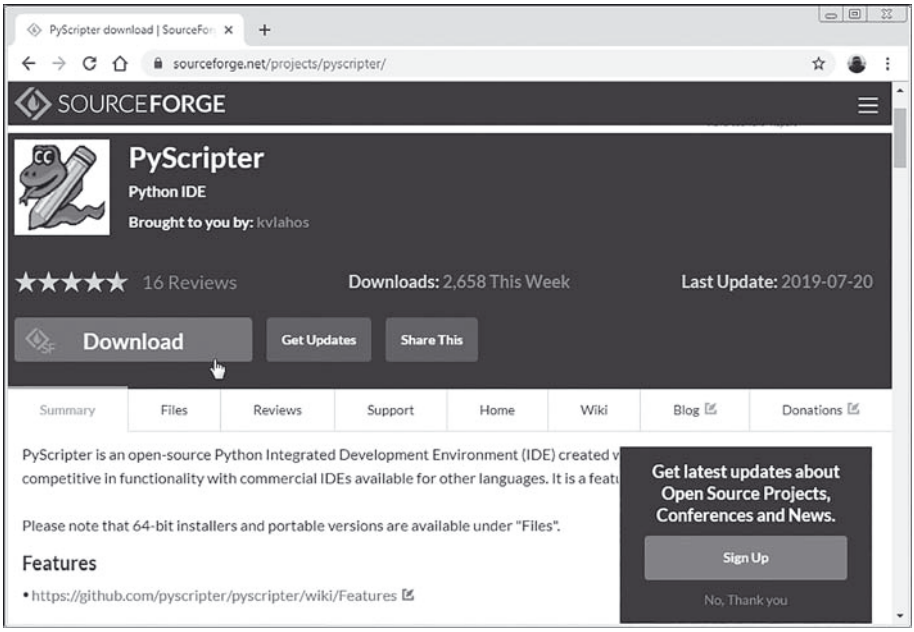


Рис. В.4. Страница *sourceforge.net/projects/pyscripter/* со ссылкой для загрузки установочных файлов среды разработки PyScripter

НА ЗАМЕТКУ

Поскольку страница поддержки среды PyScripter время от времени меняет свой адрес, то перед загрузкой установочных файлов стоит предварительно уточнить актуальный адрес для загрузки.

Характеризуя ситуацию в целом, стоит заметить, что большинство сред разработки предоставляют пользователю практически одинаковый «набор услуг». По крайней мере на начальном этапе, когда читатель только будет знакомиться с языком программирования Python, нет принципиальной разницы в том, какую именно среду разработки использовать. Это скорее вопрос эстетики, а не эффективности.

НА ЗАМЕТКУ

Ситуация со средами разработки довольно изменчива: какие-то среды становятся популярными, другие отходят на второй план. Поэтому следует понимать, что перечень доступных или предпочтительных сред разработки, приведенный выше, достаточно условный. Читатель вполне может использовать и иную среду.

Об авторе

Товарищ, там человек говорит, что он — инопланетянин. Надо что-то делать...

Из к/ф «Кин-дза-дза»

Автор книги — *Васильев Алексей Николаевич*, доктор физико-математических наук, профессор кафедры теоретической физики физического факультета Киевского национального университета имени Тараса Шевченко. Автор книг по программированию и математическому моделированию. Сфера его научных интересов: физика жидкостей и жидких кристаллов, фазовые переходы и критические явления, биофизика, синергетика, математическая экономика, моделирование социально-политических процессов и математическая лингвистика.

Обратная связь

— Слово лечит, разговор мысль отгоняет... Хотите беседовать, сударь?
— О чем?
— О чем прикажете.

Из к/ф «Формула любви»

Высказать свои замечания и предложения относительно этой и других книг автора можно по адресу электронной почты `alex@vasilev.kiev.ua`. Автор заранее благодарен своим читателям за конструктивную критику. Информацию об уже вышедших книгах, а также некоторые полезные материалы, касающиеся этих книг (например, программные коды примеров), можно найти на сайте `www.vasilev.kiev.ua`.

Благодарности

Вельми понеже... Весьма вами благодарен!

Из к/ф «Иван Васильевич меняет профессию»

Книги пишутся для того, чтобы их читали. Лучший стимул — осознание того, что твой труд кому-то нужен. Пользуясь случаем, хочу выразить самую искреннюю благодарность своим читателям: за интерес к книгам, за критические замечания, за желание становиться лучше и умнее.

Глава 1
ЗНАКОМСТВО С PYTHON

— Ну что, не передумали?
— Мне выбирать не приходится.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В этой главе состоится наше знакомство с языком программирования Python. Мы создадим первую программу на этом языке и кратко рассмотрим основные подходы и синтаксические конструкции, характерные для Python. В частности, в этой главе мы узнаем, как создается программа на языке Python, что такое переменные и как они используются, познакомимся со списками и научимся вычислять выражения. Также мы узнаем, что такое условный оператор и как он используется. Мы познакомимся с оператором цикла и научимся создавать функции. Планы у нас большие, поэтому сразу приступим к делу. Начнем с создания очень простой программы.

Первая программа

Нормальные герои всегда идут в обход.

Из к/ф «Айболит-66»

Для написания первой программы нам необходимо знать, во-первых, каковы правила составления программ на языке Python и, во-вторых, как и где набрать программный код и что с ним затем делать. Мы дадим ответы на оба вопроса.

Итак, программа — это набор команд или инструкций. Создание программы, отображающей в окне вывода определенное сообщение, — простой пример, с которого обычно начинается изучение любого языка программирования. В языке программирования Python подобная программа очень проста и состоит всего из одной команды. Соответствующий код представлен в листинге 1.1.

Листинг 1.1. Первая программа

```
print("Приступаем к изучению языка Python.")
```

Команда, из которой состоит программа, представляет собой инструкцию вызова встроенной функции `print()`. Аргументом функции передается текстовое значение (текстовый литерал) `"Приступаем к изучению языка Python."`.

НА ЗАМЕТКУ

Текстовые литералы в Python заключаются в двойные или одинарные кавычки.

Текстовый аргумент, переданный в функцию `print()`, при выполнении соответствующей команды отображается в окне вывода интерпретатора Python. Поэтому при выполнении программы из листинга 1.1 мы должны увидеть следующее сообщение:

Результат выполнения программы (из листинга 1.1)

```
Приступаем к изучению языка Python.
```

Вопрос лишь в том, где именно мы увидим это сообщение. Есть несколько вариантов для ввода и выполнения программного кода. Многое зависит от используемой среды разработки. Но в любом случае сначала эту среду нужно запустить. Если речь идет о среде IDLE, то должно открыться окно, представленное на рис. 1.1.

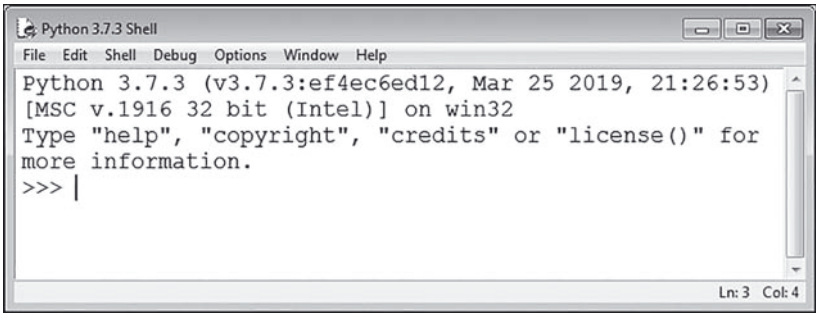


Рис. 1.1. Окно среды разработки IDLE

Окно, которое появляется на экране, является окном оболочки интерпретатора. Его характерная особенность — наличие тройной стрелки >>>, которая является индикатором строки ввода команды. Справа от этого индикатора мигает курсор. В этом месте можно вводить команды. В частности, мы можем ввести туда команду `print("Приступаем к изучению языка Python.")`, как это показано на рис. 1.2.

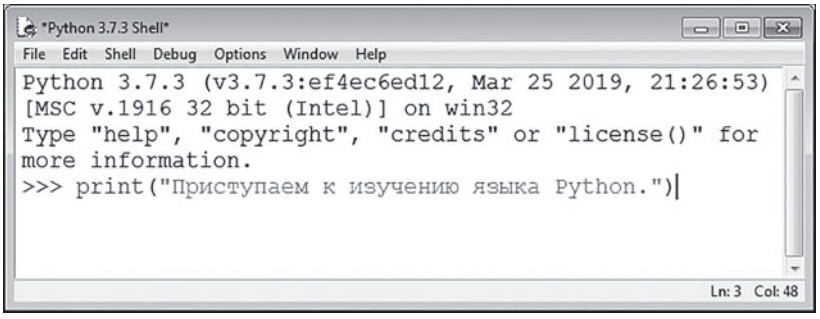


Рис. 1.2. Окно оболочки интерпретатора с введенной командой

Для выполнения команды следует нажать клавишу <Enter>. Каков будет результат, показано на рис. 1.3.

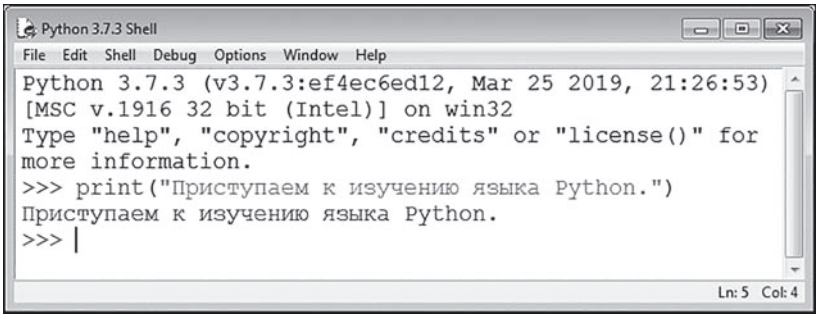


Рис. 1.3. Результат выполнения команды в окне оболочки интерпретатора

Видим, что внизу под введенной командой появилось сообщение, строго в соответствии со значением, переданным аргументом функции `print()`. А еще строкой ниже появляется новый индикатор ввода команды. Там может быть введена новая команда, и после нажатия клавиши <Enter> команда будет выполнена. На рис. 1.4 представлен результат вычисления арифметического выражения $(2 + 3) / 0.25 - 4 \cdot 2.1$ (значение выражения равно *11,6*), которому соответствует команда `(2+3)/0.25-4*2.1`.

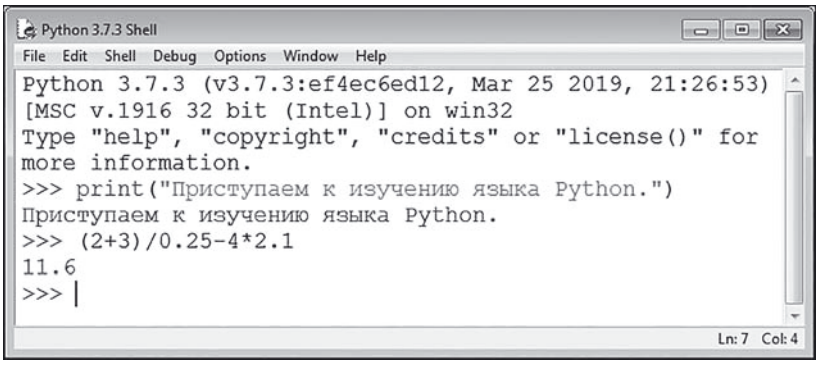


Рис. 1.4. Результат вычисления арифметического выражения

Фактически мы можем использовать окно оболочки интерпретатора языка Python как калькулятор с расширенными возможностями. Но нам не это нужно. Мы пойдем другим путем.

Поскольку в дальнейшем мы планируем создавать программы, состоящие более чем из одной команды, то каждую такую программу станем записывать в отдельный файл, а затем уже программа из файла будет запускаться на выполнение. Рассмотрим весь процесс, начиная от создания файла с программой и до запуска программы на выполнение, на примере программы из листинга 1.1. Поступаем так: в окне оболочки интерпретатора в меню **File** выбираем команду **New File** (или используем комбинацию клавиш <Ctrl>+<N>), как показано на рис. 1.5.

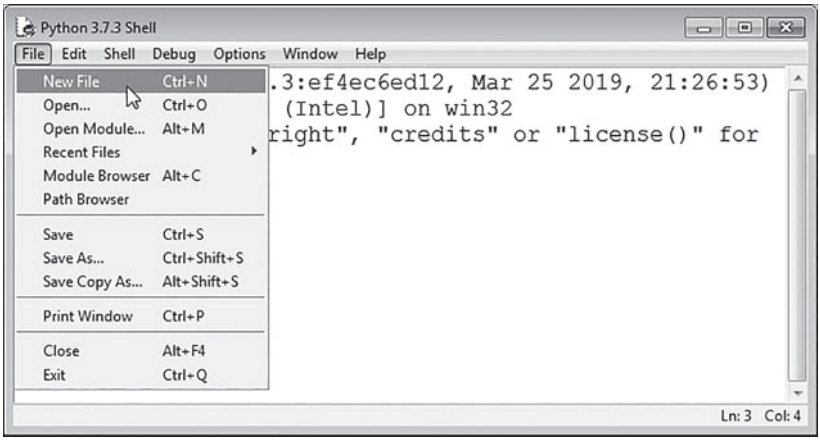


Рис. 1.5. Создание нового файла с кодом
в окне оболочки интерпретатора

Откроется окно редактора кодов. Пустое окно (не содержащее команд) представлено на рис. 1.6.

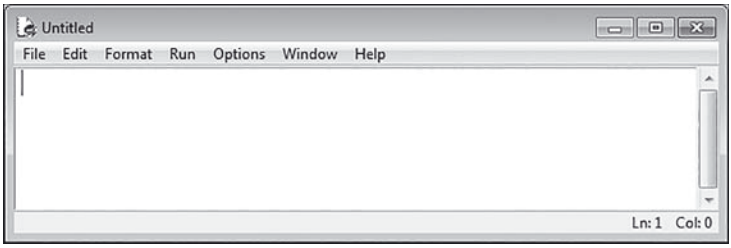


Рис. 1.6. Пустое окно редактора кодов

В окне редактора вводим команды программы. В нашем случае программа состоит всего из одной команды `print("Приступаем к изучению языка Python.")`. Именно ее вводим в окно, как показано на рис. 1.7.

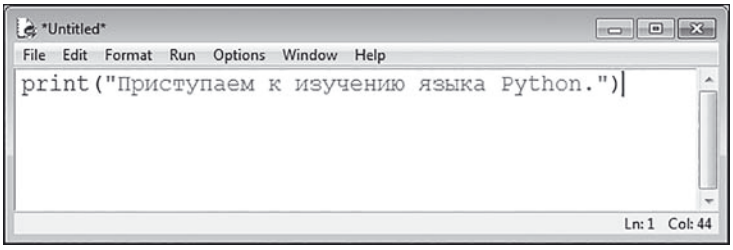


Рис. 1.7. Окно редактора кодов с введенной командой программы

После того как код программы введен, сохраняем файл с программой. Для этого в меню **File** окна редактора кодов выбираем команду **Save** (или используем комбинацию клавиш <Ctrl>+<S>). Процесс проиллюстрирован на рис. 1.8.

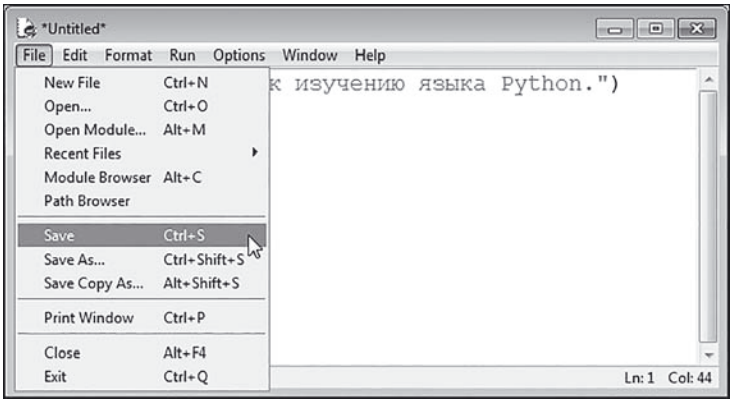


Рис. 1.8. Сохранение файла с программой

Далее следует выбрать место, где будет храниться файл с программой, и название файла (рис. 1.9).

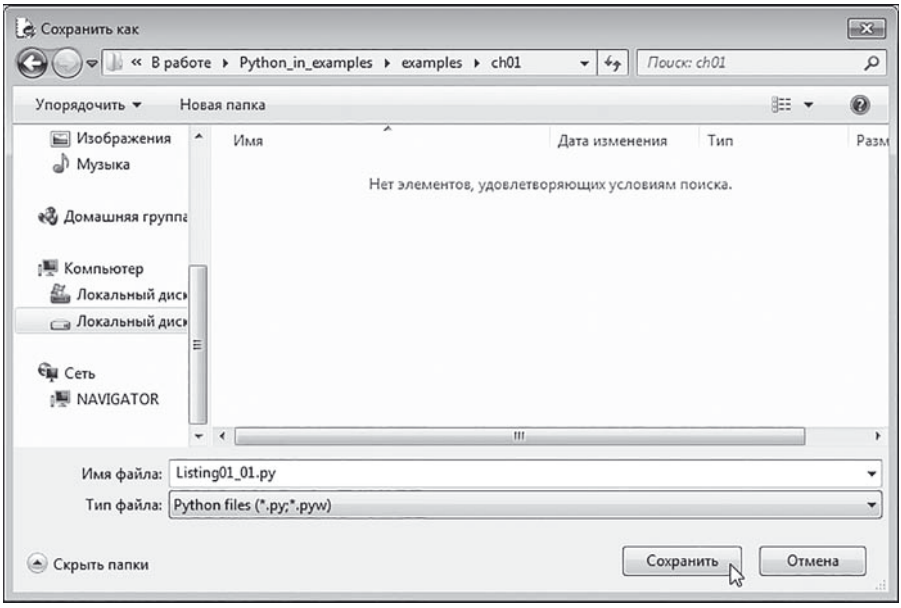


Рис. 1.9. Выбор места хранения и названия для файла с программой

Для операционной системы Windows файлы с программными кодами, написанными на языке Python, сохраняются с расширением `.py` (если речь не идет о приложении с графическим интерфейсом — для приложений с графическим интерфейсом используется расширение `.pyw`).

Для запуска программы на выполнение в окне редактора кодов (с открытым файлом программы) в меню **Run** выбираем команду **Run Module** или нажимаем клавишу <F5>, как показано на рис. 1.10.

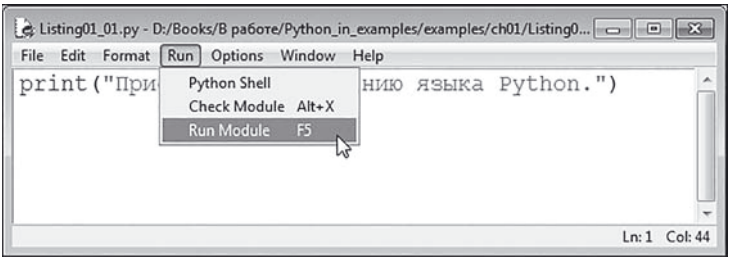


Рис. 1.10. Запуск программы на выполнение

Как следствие, в окне оболочки интерпретатора отобразится результат выполнения программы, как на рис. 1.11.

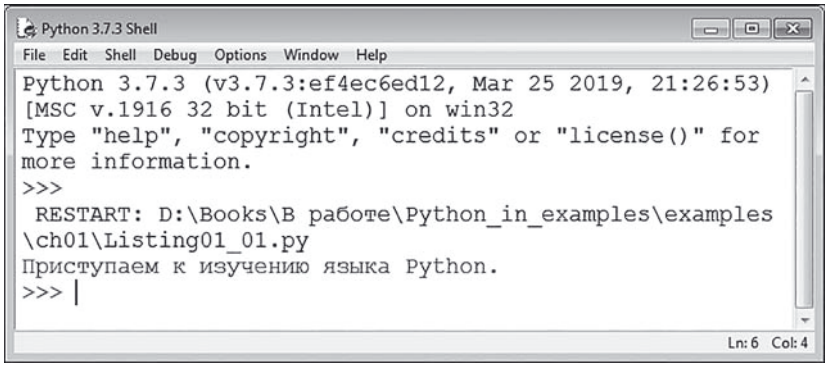


Рис. 1.11. Результат выполнения программы

В общих чертах это те действия, которые необходимо выполнить для создания несложной программы на языке Python в случае, если мы используем среду IDLE.

НА ЗАМЕТКУ

Если мы хотим открыть уже существующий файл с программой, то в меню File окна оболочки интерпретатора можно выбрать команду Open (или воспользоваться комбинацией клавиш <Ctrl>+<O>). Точно такая же команда есть в меню File окна редактора кодов. Вообще, и окно оболочки интерпретатора, и окно редактора кодов имеют достаточно простую систему настройки внешнего вида и скромный, но понятный набор команд. Полагаю, что в случае необходимости читатель без труда сможет разобраться в них.

Использование различных сред разработки

> От пальца не прикуривают, врать не буду.
> А искры из глаз летят.
>
> *Из к/ф «Формула любви»*

Выше мы рассмотрели последовательность действий для создания программы в том случае, если читатель решит использовать среду IDLE. Но мы уже знаем, что это далеко не единственная возможность. Существуют альтернативные варианты. Далее мы кратко рассмотрим основные особенности таких сред разработки, как PyCharm, Wing и PyScripter.

НА ЗАМЕТКУ

Читатели, не испытывающие проблем с «обузданием» сред разработки (которые во многих моментах очень похожи), могут пропустить этот раздел и сразу перейти к следующему разделу, посвященному использованию переменных.

Среда разработки PyCharm

Если воспользоваться средой разработки PyCharm, то при ее запуске появляется окно, как показано на рис. 1.12.

Рис. 1.12. Начальное окно среды разработки PyCharm

Окно достаточно простое и содержит всего несколько команд. Если мы планируем открыть уже существующий проект, то следует воспользоваться командой **Open**. Если же в наши планы входит создание нового проекта, то следует воспользоваться командой **Create New Project**.

НА ЗАМЕТКУ

Если ранее в среде уже создавались проекты, то в левой части окна будет список этих проектов. Открыть один из них можно простым нажатием на имя проекта.

Если вы нажмете на команду **Create New Project**, откроется диалоговое окно **New Project**, представленное на рис. 1.13.

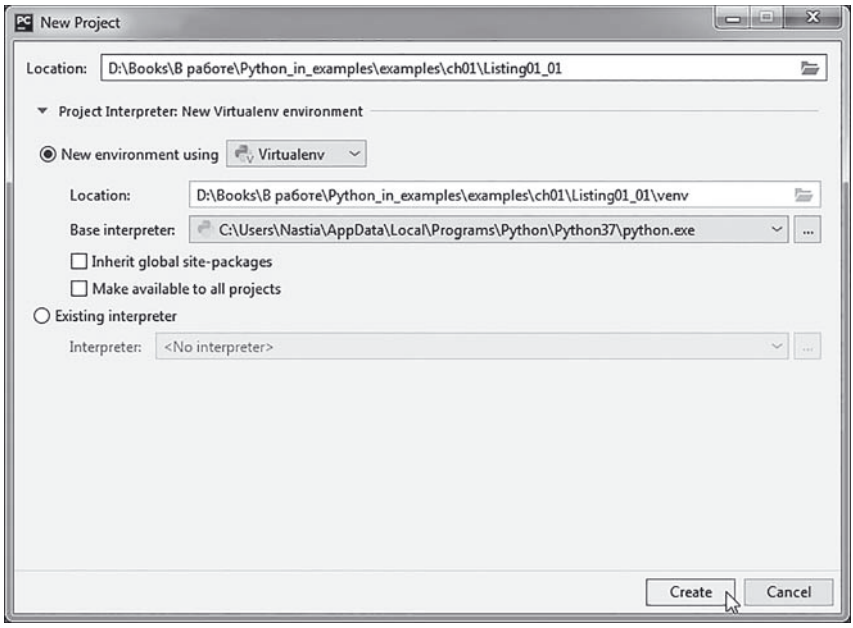


Рис. 1.13. В поле ввода окна **New Project** следует выбрать место для размещения файлов создаваемого проекта

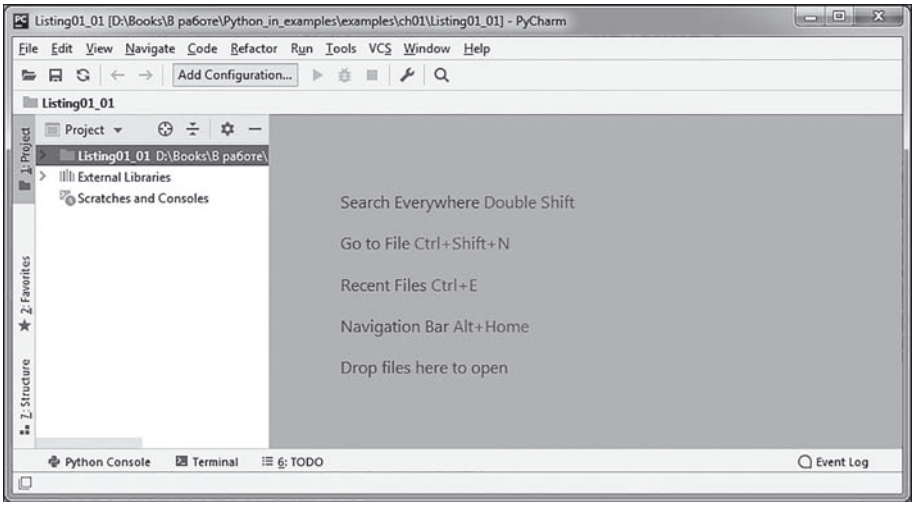


Рис. 1.14. Окно среды PyCharm сразу после создания нового проекта

В поле **Location** следует указать место для записи файлов проекта и название проекта. Пиктограмма с изображением открытой папки (или тремя точками, в зависимости от версии среды) справа от поля облегчает процесс выбора места для проекта: нажатие на пиктограмму приведет к тому, что откроется диалоговое окно для выбора места сохранения проекта. По умолчанию для проекта создается отдельная папка, название которой совпадает с названием проекта.

После того как выбраны место сохранения и название проекта, следует нажать кнопку **Create** в нижней части диалогового окна **New Project** (рис. 1.13). В результате будет создан пустой проект, а окно среды разработки примет вид как на рис. 1.14.

В левой части окна среды разработки находится внутреннее окно проекта. В этом внутреннем окне отображаются раскрывающиеся пункты, соответствующие открытому проекту и вспомогательным библиотекам. Интерес представляет пункт с файлами проекта: нам необходимо добавить в проект новый файл, в который мы затем внесем программный код. Для этого выделяем пункт проекта (его название совпадает с названием проекта) и в контекстном меню выбираем в подменю **New** команду **Python File**, как это показано на рис. 1.15.

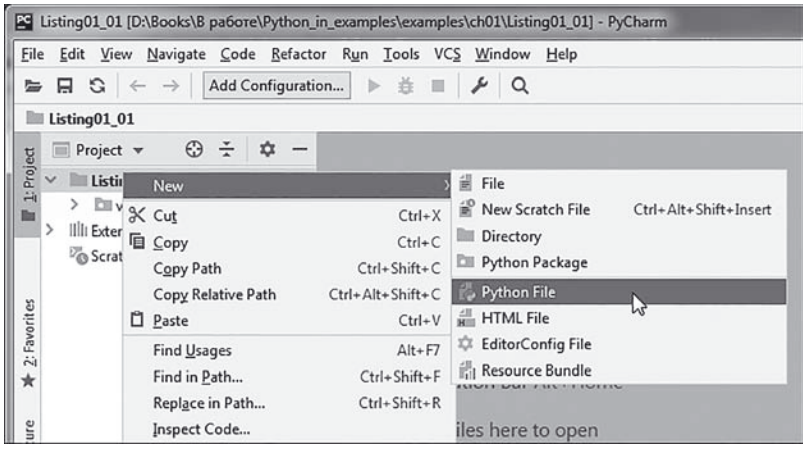


Рис. 1.15. Для добавления файла в проект в контекстном меню проекта **New** следует выбрать команду **Python File**

НА ЗАМЕТКУ

Чтобы добавить файл в проект, можно также воспользоваться командой New из главного меню File.

Для того чтобы в окне приложения отображалась панель инструментов (на рис. 1.14 она расположена под главным меню), необходимо установить флажок настройки Toolbar в подменю Appearance меню View.

Откроется диалоговое окно **New Python File**, в поле ввода которого следует указать название для файла, добавляемого в проект. Процесс ввода названия файла проиллюстрирован на рис. 1.16.

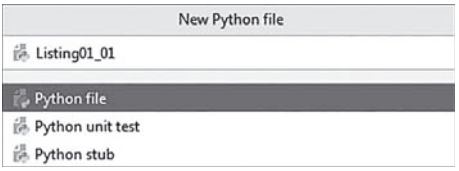


Рис. 1.16. В поле ввода диалогового окна **New Python File** указывается название файла

Если все прошло успешно, то в содержимом пункта проекта появится позиция с названием добавленного в проект файла, как показано на рис. 1.17.

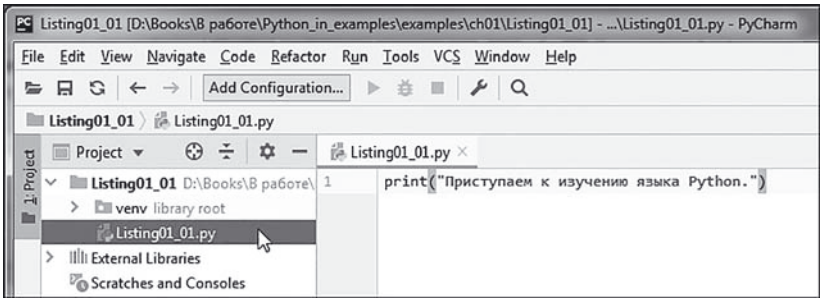


Рис. 1.17. Во внутреннем окне редактора кодов среды PyCharm введен код программы (одна команда)

Если выделить позицию с названием файла, в правой части окна среды разработки отобразится внутреннее окно редактора кодов. Это окно будет содержать код файла. Именно туда нам следует ввести код нашей программы. Для запуска программы на выполнение можно воспользоваться командой **Run** из одноименного главного меню (рис. 1.18).

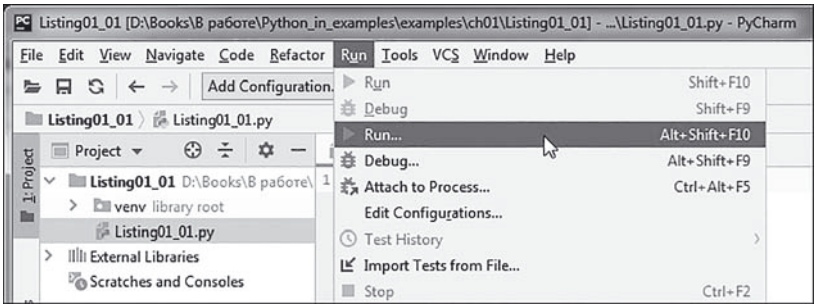


Рис. 1.18. Для запуска программы на выполнение следует выбрать в меню **Run** одноименную команду

Появится внутреннее диалоговое окно **Run**, в котором следует выбрать название запускаемого на выполнение проекта (рис. 1.19).

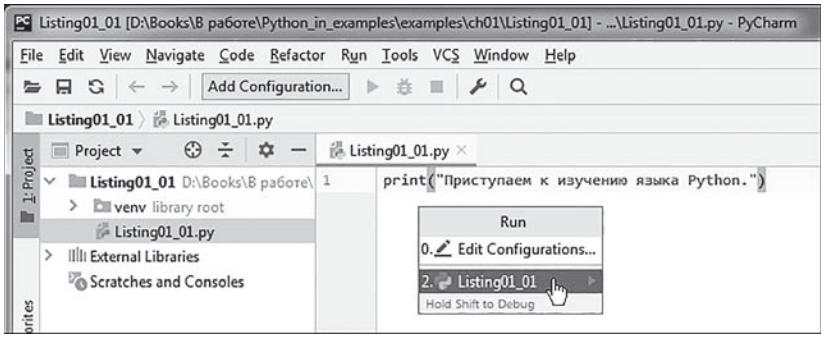


Рис. 1.19. Во внутреннем диалоговом окне **Run** следует выбрать файл для запуска на выполнение

Результат выполнения программы отображается во внутреннем окне вывода, расположенном в нижней части окна среды разработки (рис. 1.20).

Так выглядит процесс создания и запуска на выполнение программы в среде PyCharm. Существуют и другие важные операции, которые предстоит выполнять в процессе работы с программными кодами. Для большинства таких операций есть соответствующие команды в главном меню приложения. Например, чтобы закрыть проект, можно воспользоваться командой **Close Project** из главного меню **File**. С помощью команды **Open** этого меню можно открыть уже существующий проект, а команды **Save as** и **Save All** используются для сохранения изменений. Есть огромное количество иных команд, включая и те, которые связаны с настройкой внешнего вида приложения PyCharm. Однако их описание не входит в наши планы. Полагаю, что читатель сможет разобраться с этими вопросами самостоятельно или с помощью справки по приложению.

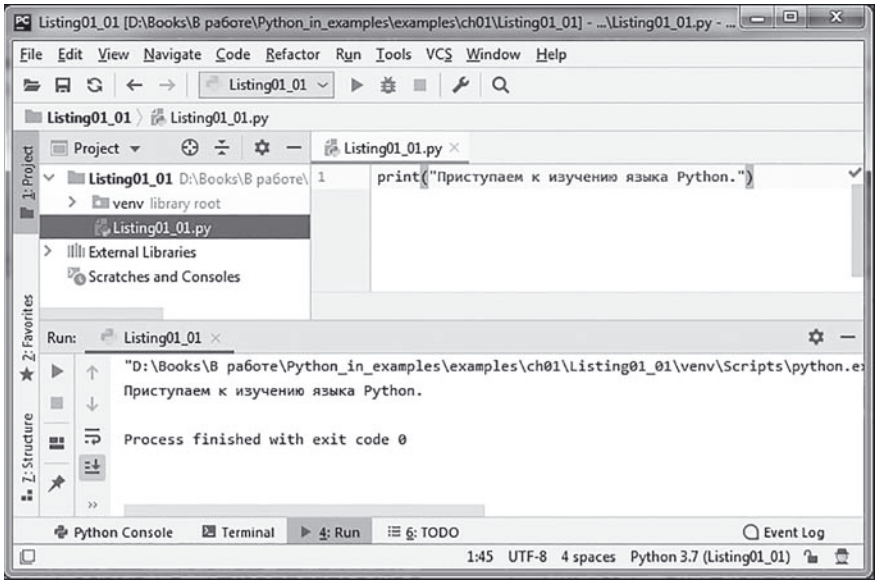


Рис. 1.20. Результат выполнения программы отображается во внутреннем окне вывода (в нижней части) среды разработки PyCharm

Среда разработки Wing

При запуске среды Wing отображается диалоговое окно, показанное на рис. 1.21.

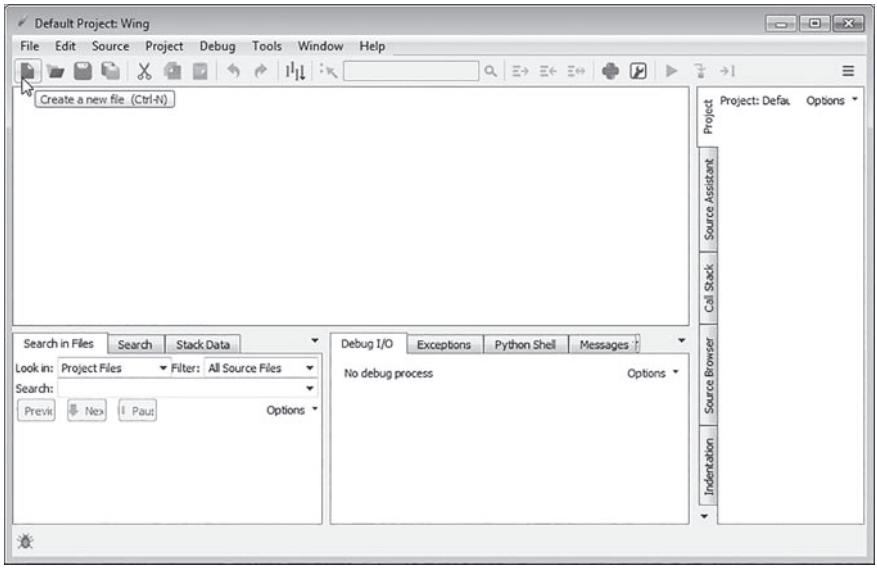


Рис. 1.21. Окно среды разработки Wing

Для создания нового проекта можно нажать специальную кнопку на панели инструментов или воспользоваться командой **New** из меню **File**. В результате во внутреннем окне редактора появится вкладка для ввода программного кода, как показано на рис. 1.22.

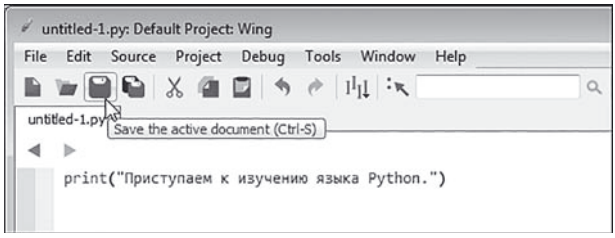


Рис. 1.22. В окне редактора введен код программы

Вводим программный код и сохраняем программу: для этого на панели инструментов есть специальная кнопка (рис. 1.22), но можно также воспользоваться командой **Save** из меню **File**. После этого, в принципе, программа готова к запуску. Один нюанс: если мы собираемся использовать в сообщениях, отображаемых программой, кириллический текст, то могут возникнуть проблемы с кодировкой.

ПОДРОБНОСТИ

Общая проблема связана с тем, что кодировка, используемая в редакторе кодов, может отличаться от кодировки консольного окна или окна вывода. Обычно в таких случаях при отображении сообщения можно увидеть вместо кириллического текста последовательность странных символов. Но в случае со средой Wing ситуация более критичная — программный код может просто не запуститься. Поэтому не исключено, что придется поменять кодировку для файла с программой. Проблема заключается в том, что по умолчанию в редакторе кодов используется кодировка windows cp1251, а в окне вывода используется кодировка utf-8. Поэтому кириллический текст, который выглядит прилично в окне редактора, совершенно по-иному интерпретируется в окне вывода (если до отображения этого текста дело вообще дойдет).

Чтобы поменять кодировку, используемую при сохранении файла с программой, следует в меню **Source** выбрать команду **Current File Properties**, как показано на рис. 1.23.

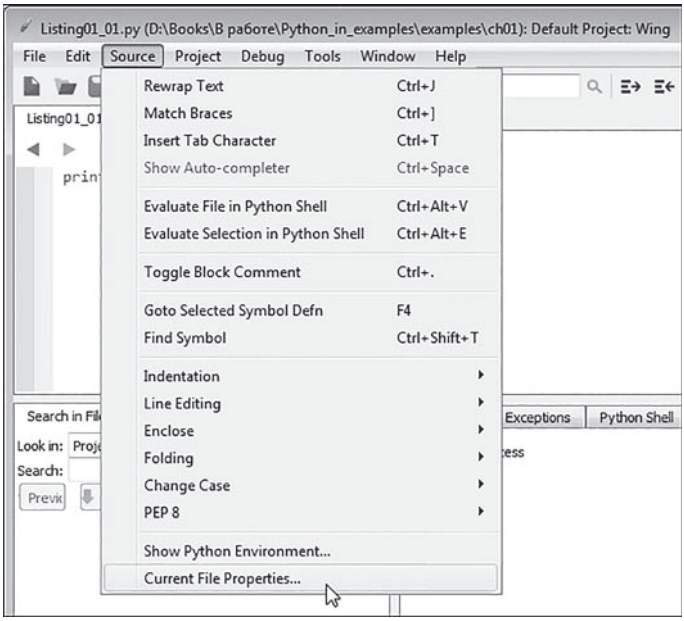


Рис. 1.23. Переход в режим настроек свойств файла программы

Откроется окно **File Properties**, в котором на вкладке **File Attributes** в раскрывающемся списке **Encoding** следует выбрать кодировку для сохранения файла (**Unicode (UTF-8) utf-8**), как показано на рис. 1.24.

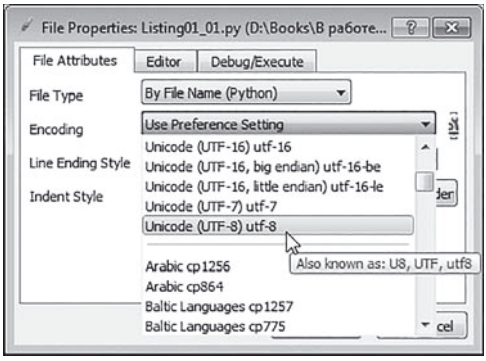


Рис. 1.24. Выбор кодировки для файла программы

Заметим, что выбирать кодировку для файла следует до того, как в файл введен программный код, поскольку в процессе перезагрузки (после изменения кодировки) кириллические символы могут отображаться неправильно.

Проблему с кодировкой можно решить и более радикально. Для этого следует изменить настройки среды так, чтобы, например, в редакторе кодов по умолчанию использовалась кодировка **utf-8**. Поступаем следующим образом: в меню **Edit** выбираем команду **Preferences**, как показано на рис. 1.25.

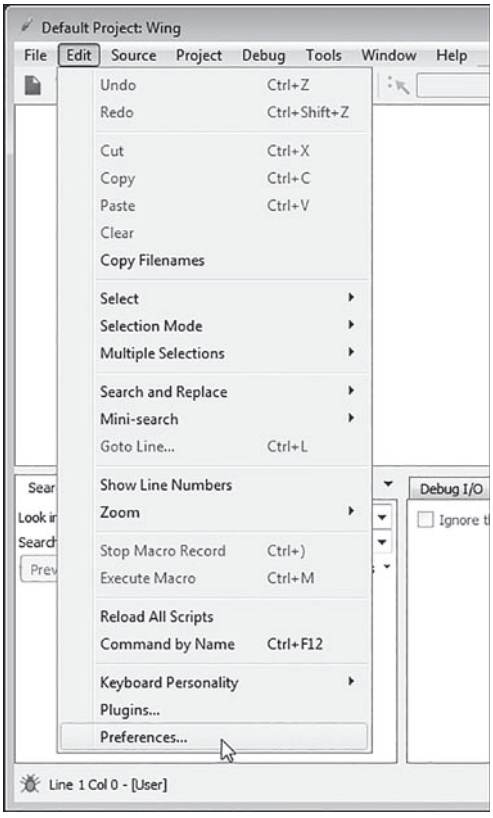


Рис. 1.25. Переход в режим настроек параметров приложения

Откроется диалоговое окно **Preferences**, в котором следует выделить пункт **Files** и в правой части в раскрывающемся списке **Default Encoding** выбрать нужную кодировку (см. рис. 1.26).

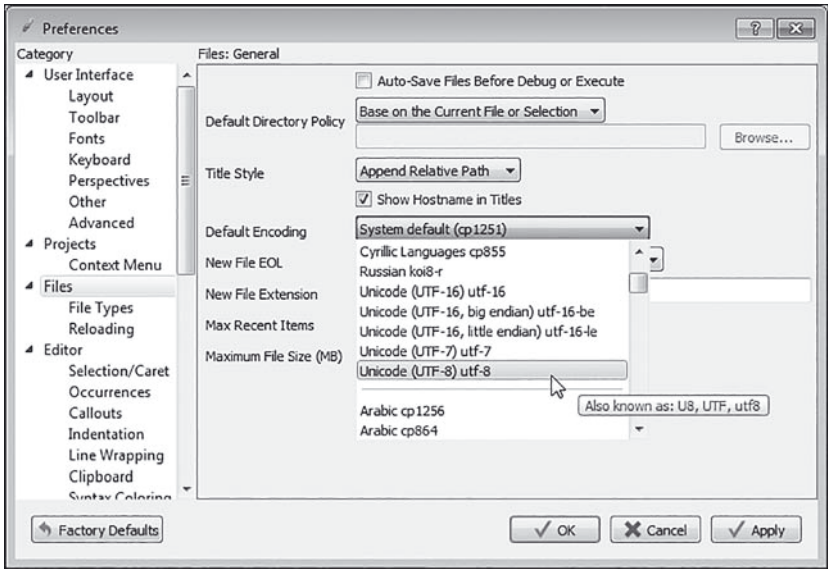


Рис. 1.26. Выбор используемой по умолчанию кодировки для файлов с программным кодом

Эта кодировка по умолчанию будет применяться для всех вновь создаваемых файлов с программным кодом.

НА ЗАМЕТКУ

При необходимости настройка кодировки выполняется и в средах PyCharm и PyScripter.

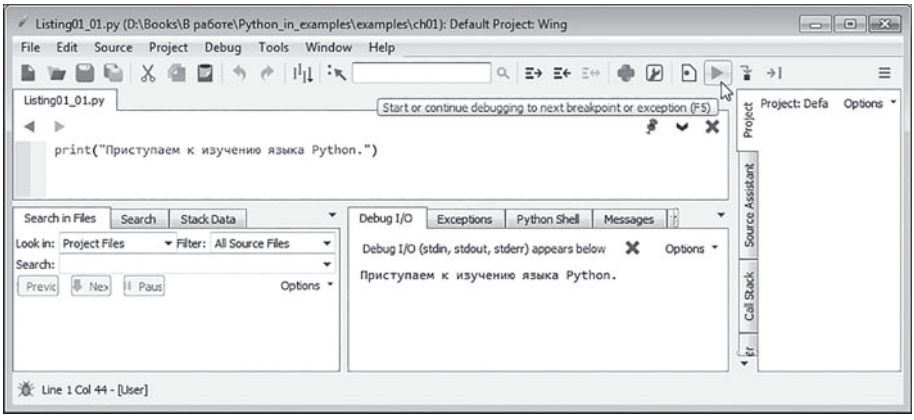


Рис. 1.27. Запуск программы на выполнение и результат выполнения программы в среде Wing

После того как программный код внесен в файл и файл сохранен, можем запускать программу на выполнение. Для этого нажимаем кнопку с зеленой стрелкой на панели инструментов (рис. 1.27) или выбираем команду **Start/Continue** из меню **Debug**.

Результат выполнения программы отображается во внутреннем окне на вкладке **Debug I/O** (находится в нижней части окна приложения Wing).

Среда разработки PyScripter

Окно среды разработки PyScripter показано на рис. 1.28.

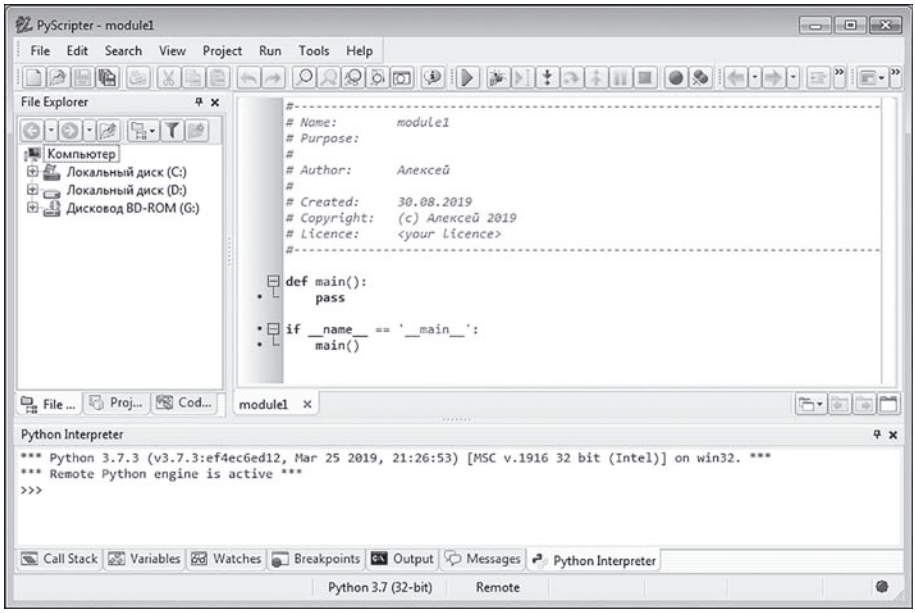


Рис. 1.28. Окно среды разработки PyScripter

Окно приложения PyScripter содержит несколько внутренних окон, в том числе и окно редактора кодов. По умолчанию для нового проекта предлагается шаблонный код. Мы его удаляем и вводим тот код, который нужен нам, и сохраняем проект. Для этого можно нажать соответствующую кнопку на панели инструментов или воспользоваться командой **Save** из меню **File**. На рис. 1.29 показано окно приложения PyScripter с введенным программным кодом перед сохранением.

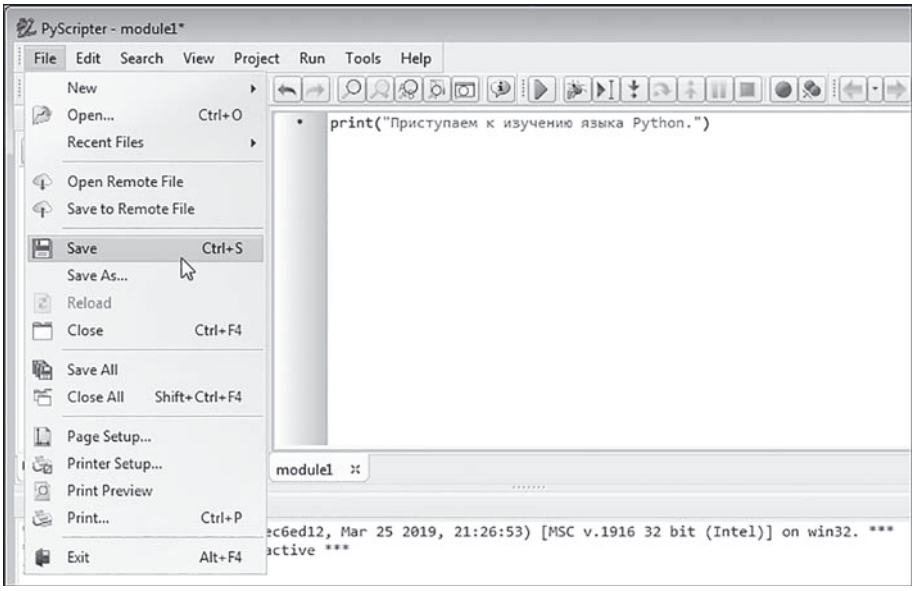


Рис. 1.29. Сохранение проекта в среде PyScripter

После сохранения проекта запускаем программу на выполнение. Для этого используем команду **Run** из одноименного меню или нажимаем кнопку с зеленой стрелкой на панели инструментов, как показано на рис. 1.30.

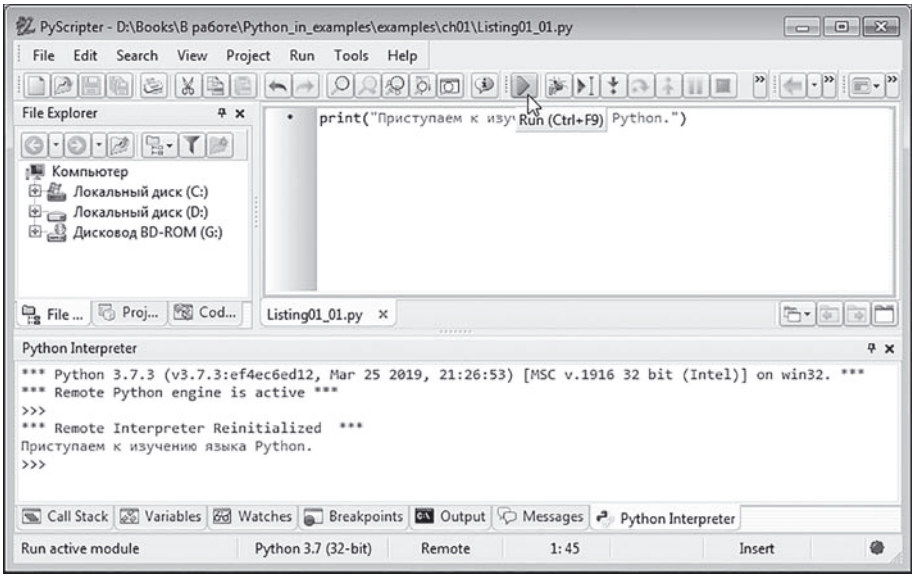


Рис. 1.30. Результат выполнения программы

Результат выполнения программы отображается во внутреннем окне (по умолчанию находится в нижней части окна приложения PyScripter). Разумеется, и в этом случае результат выполнения программы будет такой же, как и во всех предыдущих.

НА ЗАМЕТКУ

Такие операции, как открытие уже существующего документа, закрытие проекта, сохранение проекта и многие другие, во всех средах разработки выполняются однотипно. Каждая среда разработки допускает выполнение всевозможных настроек, затрагивающих как внешний вид окна среды, так и характеристики создаваемых с ее помощью проектов. Полагаю, что выполнение таких операций не составит для читателя большого труда. В крайнем случае, можно порекомендовать обратиться к справочной системе соответствующей среды разработки.

Знакомство с переменными

— Что такое? Я не желаю никаких знакомств.
— Мы предлагаем Вам помощь, а не знакомство.

Из к/ф «Приключения принца Флоризеля»

В программе *переменная* нужна для того, чтобы присвоить ей значение или чтобы прочитать значение переменной. Переменные можно передавать в качестве аргументов функциям, использовать их в выражениях, да и вообще, программа без переменных все равно что песня без слов.

ПОДРОБНОСТИ

Практически любая программа оперирует с данными. Данные в программе сохраняются, кроме прочего, с помощью переменных. Переменная, в общем случае, отождествляется с некоторой областью памяти. В эту область можно записать значение и из такой области можно считать значение. У переменной есть имя. Имя переменной используется для получения доступа к значению, записанному в памяти.

Существуют разные механизмы реализации переменных. Переменная может содержать значение, а может ссылаться на него. Переменную, которая содержит значение, удобно представить в виде коробочки, в которую можно что-то положить, а затем, при

необходимости, посмотреть, что там лежит, или даже положить в коробочку что-то новое. Продолжая эту аналогию, можем отождествить название переменной с именем, написанным на коробочке.
Переменная может ссылаться на значение. В таком случае переменная сама по себе значение не содержит. Ее фактическим значением является адрес другой ячейки, в которой содержится значение или в которую значение записывается. В таком случае переменная играет роль посредника. Но в большинстве случаев иллюзия такая, как если бы переменная сама содержала значение, а не ссылалась на него.

Использовать переменные в программе на языке Python очень просто. Для этого не нужно даже объявлять переменную. Нет необходимости указывать ее тип. Достаточно присвоить значение идентификатору, определяющему имя переменной. Как следствие, в программе появится переменная с соответствующим значением.

ПОДРОБНОСТИ

Во многих языках программирования необходимо объявлять переменные перед использованием. Объявление переменной обычно подразумевает и указание ее типа. Тип переменной влияет на размер области памяти, в которой хранится значение переменной. Для языка Python эти правила не действуют.
В Python переменные ссылаются на значения. Объявлять переменные не нужно. У переменной как таковой типа нет (но есть тип у значения, на которое ссылается переменная).

Небольшой пример программы, в которой используются переменные, представлен в листинге 1.2.

Листинг 1.2. Использование переменных

```
# Переменная с текстовым значением:
txt="Язык программирования Python"
# Отображение значения переменной:
print(txt)
# Переменная с целочисленным значением:
num=123
# Отображение значения переменной:
print("Целое число:", num)
```

Результат выполнения этой программы представлен ниже.

Результат выполнения программы (из листинга 1.2)

```
Язык программирования Python
Целое число: 123
```

В программе, кроме собственно переменных, мы знакомимся с еще несколькими новыми «конструкциями». Так, в нашей программе появились *комментарии*. Комментарий — это текст, который игнорируется при выполнении программы. Комментарий предназначен для тех, кто работает с программным кодом. Обычно комментарии используют для пояснения тех или иных операций, выполняемых в программе. Создать комментарий просто: он начинается с символа #, который и является признаком комментария. Все, что находится справа от символа #, игнорируется при выполнении программного кода.

ПОДРОБНОСТИ

В языке Python пробелы используются для структурирования программного кода. Проще говоря, если пробел перед командой не нужен, то его ставить нельзя. В противном случае последствия могут быть драматическими. В этом отношении Python сильно отличается от прочих языков программирования, в которых наличие пробелов перед командами имеет исключительно эстетический эффект.
Что касается рассматриваемой программы, то каждая команда размещается в новой строке, без отступов слева.

Кроме комментариев, программа содержит четыре команды. Командой `txt="Язык программирования Python"` переменной `txt` присваивается текстовое значение `"Язык программирования Python"`. Стоит заметить, что мы никаким специальным образом не объявляли эту переменную. Ее появление в программе связано с присваиванием значения переменной.

НА ЗАМЕТКУ

Если в программном коде в команде присваивания нового значения уже существующей переменной по ошибке неправильно указать имя этой переменной, то случайно можно объявить новую переменную.

Для отображения в окне вывода значения переменной `txt` используем команду `print(txt)`, в которой имя переменной передается в качестве аргумента функции `print()`. Еще одна переменная, которая

называется num, появляется в программе в результате выполнения команды num=123. Эта переменная получает целочисленное значение. После присваивания значения переменной оно отображается в окне вывода с помощью команды print("Целое число:", num). Особенность команды в том, что функции print() передается два аргумента: текстовый литерал и переменная num. В результате в области вывода в одной строке отображается сначала текстовый литерал, а затем значение переменной num. При этом между текстовым литералом и значением переменной автоматически добавляется пробел.

Как мы увидели из рассмотренного примера, у переменных в Python нет типа. Поэтому теоретически на разных этапах выполнения программы одна и та же переменная может ссылаться не просто на разные значения, но на значения разных типов.

НА ЗАМЕТКУ

Не следует путать тип данных, на которые ссылается переменная, с типом переменной. У данных тип есть. У переменной типа нет. Мы не можем говорить о типе переменной, но можем говорить о типе данных, на которые переменная ссылается.

Целые числа относятся к типу int. Текстовые значения относятся к типу str.

Небольшая иллюстрация к этому утверждению представлена в листинге 1.3. Рассматриваемая там программа является модификацией предыдущего примера. Принципиальное отличие состоит в том, что программа использует одну переменную, которой сначала в качестве значения присваивается текстовый литерал, а затем целое число.

Листинг 1.3. Переменная ссылается на значения разных типов

```
# Текстовое значение переменной:
value="язык Python"
# Отображение значения переменной:
print("Текст:", value)
# Целочисленное значение переменой:
value=321
# Отображение значения переменной:
print("Число:", value)
```

При выполнении программы получаем следующий результат.

Результат выполнения программы (из листинга 1.3)

```
Текст: язык Python
Число: 321
```

В представленном примере одна и та же переменная `value` сначала получает текстовое значение, которое отображается в окне вывода, а затем числовое значение, и оно также отображается в окне вывода.

НА ЗАМЕТКУ

Переменную в программе можно не только «материализовать», присвоив ей значение. Переменную можно удалить. Для этого используют инструкцию `del`, после которой указывается имя удаляемой переменной.

Ввод значения в программу

> Ну и что, что квартет? Добавьте сюда еще людей — будет большой, массовый квартет.
>
> *Из к/ф «Карнавальная ночь»*

Выше мы присваивали значения переменным непосредственно в программном коде (имеется в виду, что значение, присваиваемое переменной, в программном коде было представлено текстовым литералом или числом). Но существует возможность вводить значение через окно оболочки интерпретатора непосредственно в процессе выполнения программы. Для этого используют встроенную функцию `input()`, которая в качестве результата возвращает значение, введенное пользователем с клавиатуры. Особенности ввода значений в процессе выполнения программы иллюстрирует программный код в листинге 1.4.

Листинг 1.4. Ввод значения в программу

```
# Считывание текстового значения:
name=input("Как Вас зовут? ")
# Считывание числового значения:
age=int(input("Сколько Вам лет? "))
```

```
# Отображение считанных значений:
print("Добрый день,", name+"!")
print("Вы родились в",2019-age,"году.")
```

При запуске программы на выполнение сначала появляется первое сообщение с вопросом об имени пользователя. Если мы используем среду IDLE, то запрос будет выглядеть так, как показано на рис. 1.31.

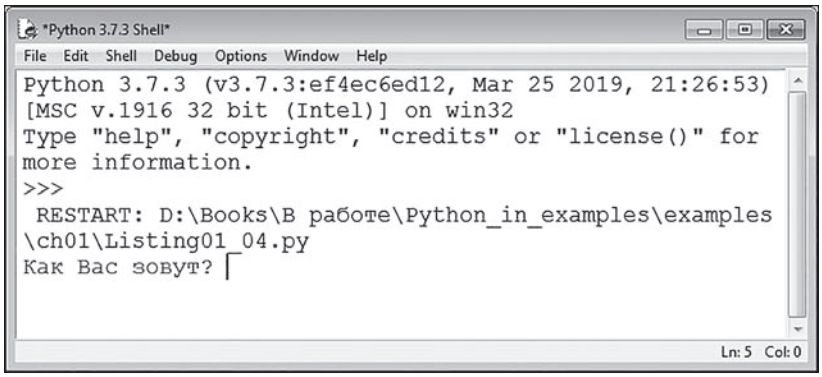


Рис. 1.31. Начало выполнения программы

На этом выполнение программы приостанавливается, пока пользователь не введет имя. На рис. 1.32 показан процесс ввода имени пользователя.

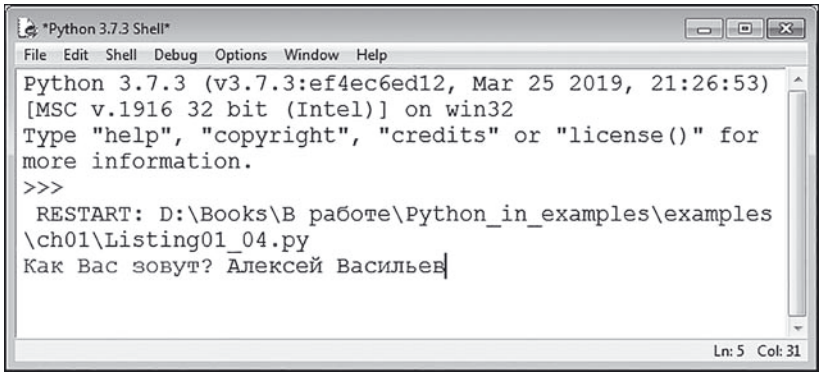


Рис. 1.32. Вводится текст

После того как имя введено, следует нажать клавишу <Enter>. Результат представлен на рис. 1.33.

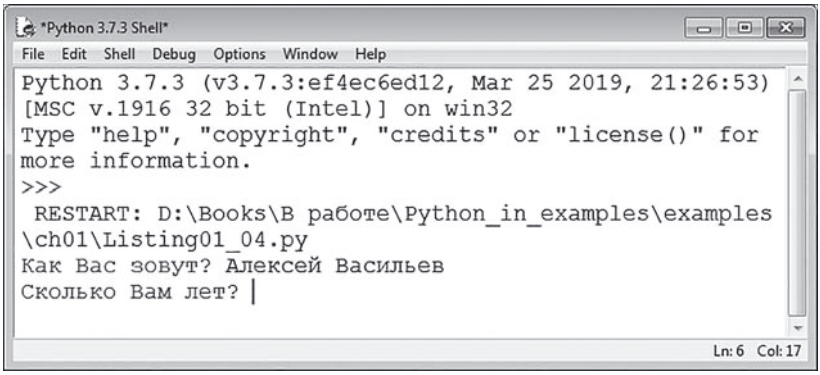


Рис. 1.33. Результат ввода текстового значения

Видим, что появляется еще один вопрос. Теперь необходимо указать возраст пользователя. Предполагается, что это целое число. Процесс ввода целого числа в окне оболочки интерпретатора проиллюстрирован на рис. 1.34.

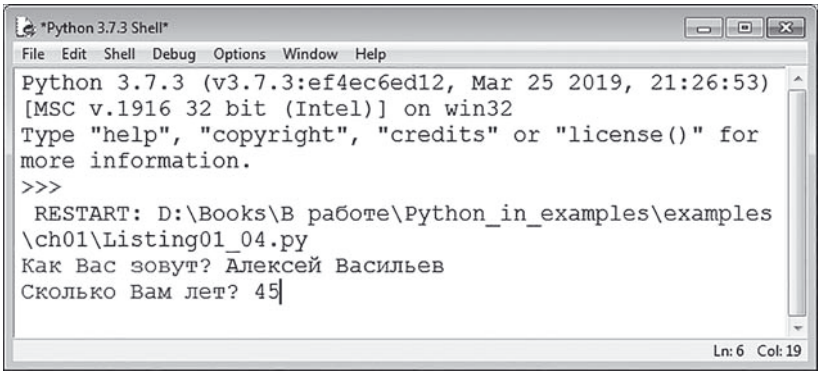


Рис. 1.34. Вводится число

После нажатия клавиши <Enter> в окне вывода появятся два новых сообщения, содержащие введенное имя и вычисленный год рождения пользователя, как на рис. 1.35.

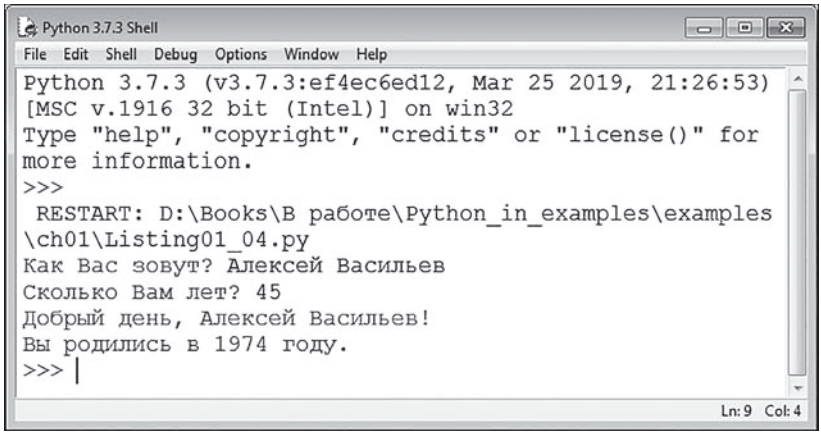


Рис. 1.35. Результат выполнения программы после ввода текста и числа

Если мы используем среду разработки PyCharm, то все происходит аналогичным образом, но только данные вводятся (и отображаются) во внутреннем окне вывода (и ввода). На рис. 1.36 показано окно среды разработки PyCharm с результатом выполнения программы.

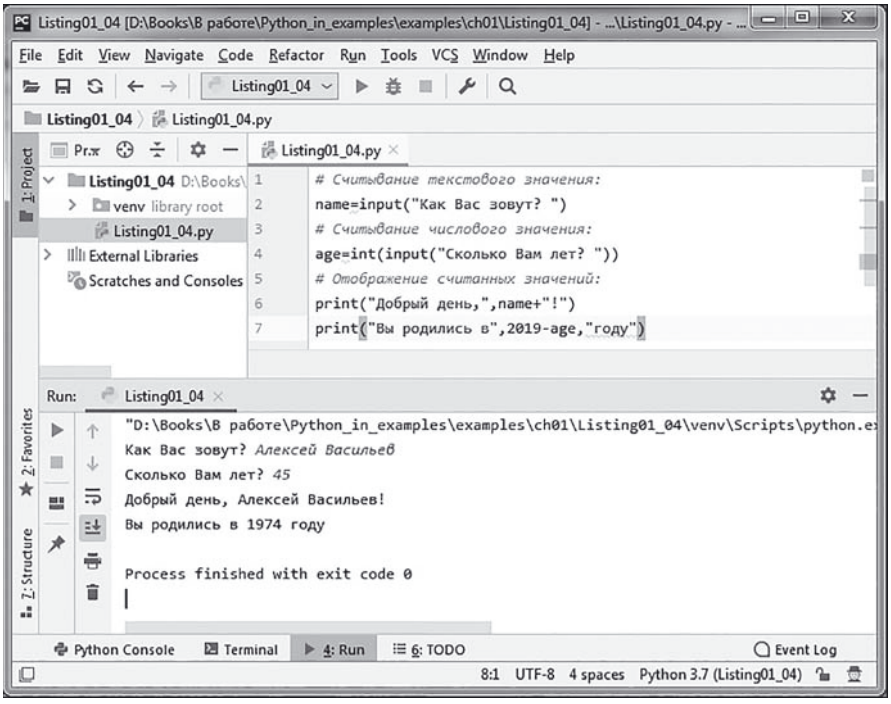


Рис. 1.36. Результат выполнения программы в окне среды разработки PyCharm

Такая же ситуация и со средой разработки Wing. Данные отображаются и вводятся во внутреннем окне **Debug I/O**, как показано на рис. 1.37.

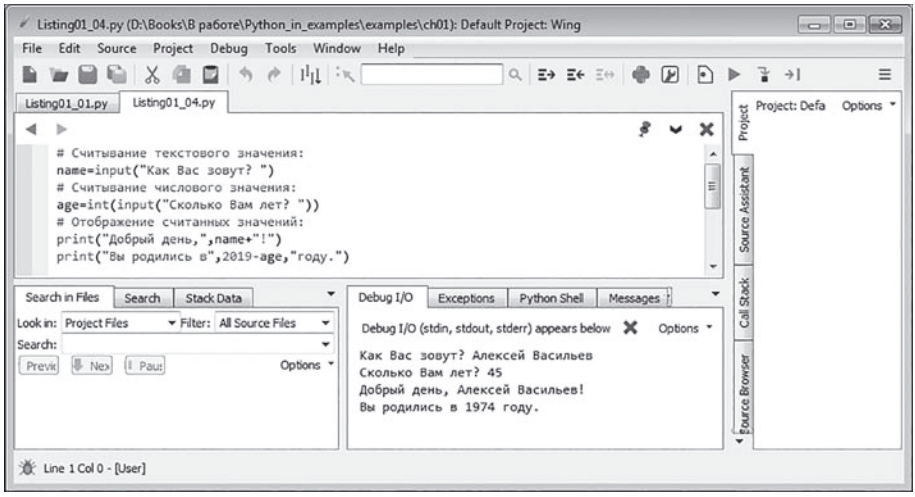


Рис. 1.37. Результат выполнения программы в окне среды разработки Wing

Но вот при работе со средой PyScripter будет сюрприз. Дело в том, что для ввода данных в среде PyScripter по умолчанию отображается специальное диалоговое окно. Так, при запуске программы на выполнение появляется диалоговое окно с первым вопросом и полем ввода, как показано на рис. 1.38.

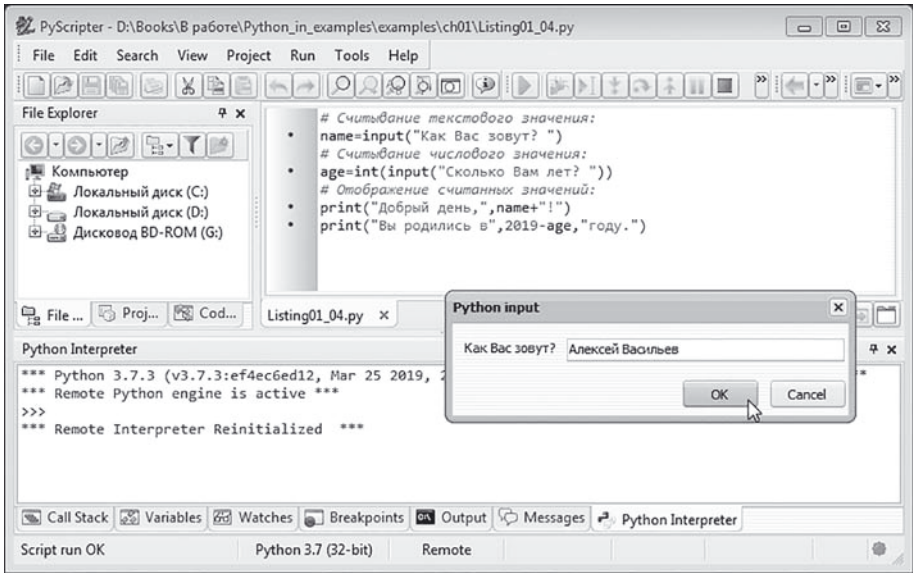


Рис. 1.38. Ввод имени в окно с полем ввода

После того как пользователь вводит имя и нажимает кнопку **OK**, появляется еще одно окно, в котором пользователю предлагается указать возраст (рис. 1.39).

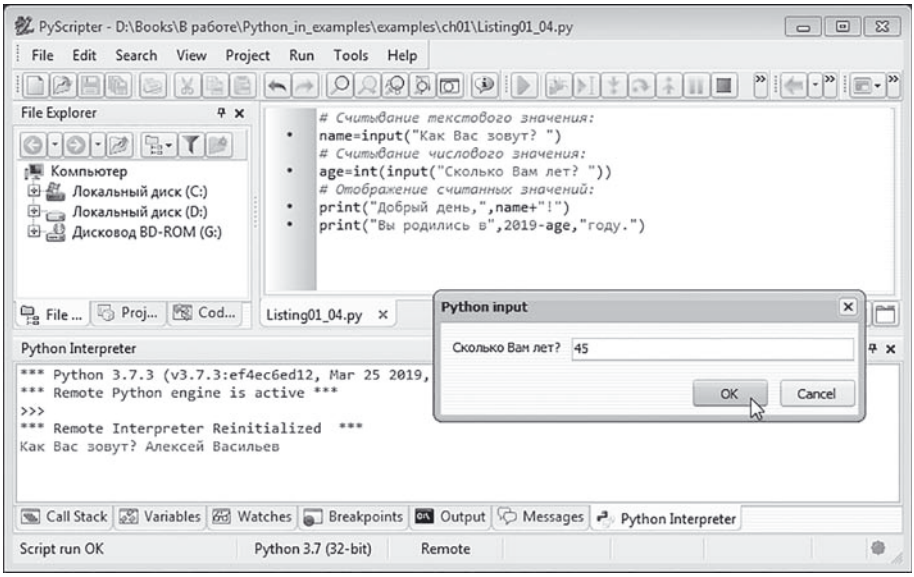


Рис. 1.39. Ввод возраста в поле ввода диалогового окна

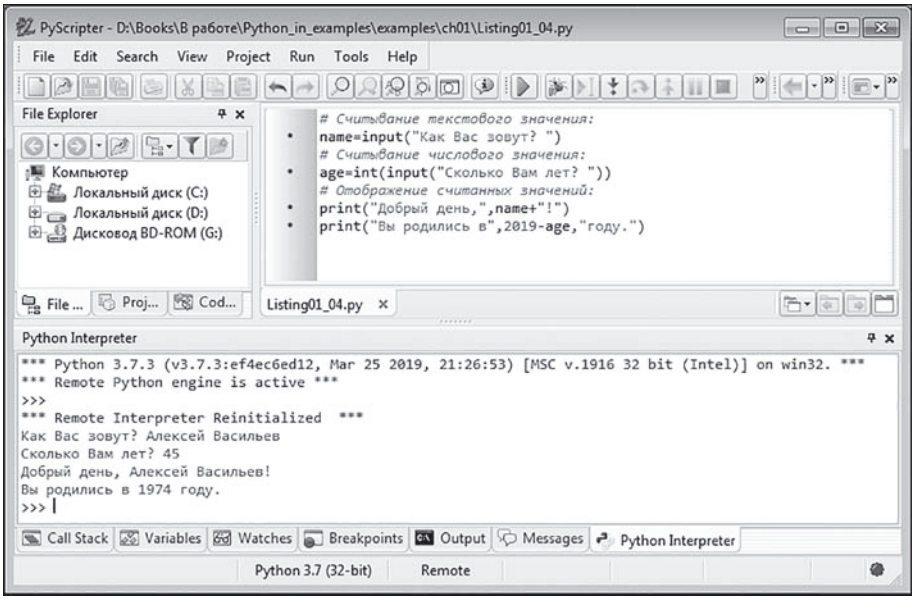


Рис. 1.40. Результат выполнения программы в среде PyScripter

После ввода возраста в области вывода появляются сообщения с указанием имени пользователя и года его рождения (рис. 1.40).

Таким образом, исходные вопросы, на которые отвечал пользователь, и собственно ответы пользователя в области вывода не отображаются.

Как бы там ни было, мы будем исходить из того, что результат выполнения программы (с учетом того, что некоторые значения вводятся пользователем) будет выглядеть следующим образом (данные, которые вводит пользователь, выделены жирным шрифтом).

Результат выполнения программы (из листинга 1.4)

```
Как Вас зовут? Алексей Васильев
Сколько Вам лет? 45
Добрый день, Алексей Васильев!
Вы родились в 1974 году.
```

Теперь проанализируем программный код (см. листинг 1.4), выполнение которого приводит к указанным результатам. При помощи первой команды `name=input("Как Вас зовут? ")` переменной `name` присваивается значение, которое вводит пользователь на запрос, отображаемый при вызове функции `input()`. Текст, отображаемый в окне вывода, передается в качестве аргумента функции `input()`. В результате функция возвращает текст, который был введен пользователем (результат возвращается после нажатия пользователем клавиши <Enter>). Здесь важно подчеркнуть, что возвращается именно текстовое значение. То есть даже если пользователь вводит число, то результатом функции `input()` будет текст, который является текстовым представлением числа. С такой проблемой мы сталкиваемся, когда пытаемся считать возраст пользователя. Для нас важно, чтобы результат был числом, поскольку после считывания значения мы выполняем с ним арифметические вычисления. Для считывания числового значения мы использовали команду `age=int(input("Сколько Вам лет? "))`. В ней инструкция `input("Сколько Вам лет? ")` передана в качестве аргумента функции `int()`, а результат всего выражения присваивается значением переменной `age`. Команда обрабатывается следующим образом. При «вычислении» аргумента `input("Сколько Вам лет? ")` функции `int()` на экране отображается сообщение с просьбой указать возраст. Когда пользователь вводит целое число (например, `45`) и нажимает клавишу

<Enter>, введенное значение в текстовом формате возвращается в качестве результата выражения `input("Сколько Вам лет? ")`. Если пользователь ввел значение `45`, то результатом выражения будет текст `"45"` (текстовое представление числа). И это, напомним, аргумент функции `int()`. Функция `int()` позволяет по текстовому представлению целого числа получить само число. Если аргумент функции `int()` равен `"45"`, то результатом возвращается число `45`. Именно это значение присваивается переменной `age`.

После того как переменные `name` и `age` получили значения, они используются для отображения сообщений в области вывода. Сначала командой `print("Добрый день,", name+"!")` отображается приветствие с именем пользователя. Второй аргумент `name+"!"` функции `print()` представляет собой «сумму» двух текстовых значений. Результатом является текстовая строка, которая получается объединением «суммируемых» текстовых строк (такая процедура называется конкатенацией строк). Командой `print("Вы родились в",2019-age, "году.")` отображается предполагаемый год рождения пользователя. Здесь у функции `print()` три аргумента, которые отображаются в одной строке через пробел. Второй аргумент `2019-age` является разностью двух числовых значений: текущего года (мы исходим из того, что все происходит в 2019 году) и возраста пользователя, что в итоге дает год рождения.

НА ЗАМЕТКУ

При присваивании значения переменной `age` мы выполняли преобразование к числовому формату по одной простой причине: значение переменной `age` используется в арифметическом выражении. Если не выполнить такое преобразование и считать значение в текстовом формате, при вычислении выражения `2019-age` возникнет ошибка.

Функция `eval()`

— Узнаешь, Маргадон?
— Натюрлих, экселенц! Отличная фемина!

Из к/ф «Формула любви»

Язык Python содержит много хороших и полезных функций. Но есть одна, с которой имеет смысл познакомиться сразу. Речь о функции `eval()`. Функция уникальна тем, что возвращает в качестве результата значение выражения, переданного ей в виде текстовой строки. Другими словами, если имеется некоторая текстовая строка, в которую «спрятана» команда, имеющая смысл в языке Python, то, передав эту текстовую строку функции `eval()`, мы получим в результате значение выражения, «спрятанного» в строке. Небольшой пример использования функции `eval()` представлен в программе в листинге 1.5.

Листинг 1.5. Использование функции `eval()`

```
# Текстовое представление для команды:
txt="(2+3)/0.25—4*2.1"
# Отображение выражения и вычисление результата:
print(txt,"=", eval(txt))
# Считывание выражения для вычисления:
res=input("Введите выражение: ")
# Отображение значения выражения:
print("Значение выражения:", eval(res))
```

Результат выполнения программы может быть таким (жирным шрифтом выделено значение, которое вводит пользователь).

Результат выполнения программы (из листинга 1.5)

```
(2+3)/0.25—4*2.1 = 11.6
Введите выражение: 2**3+4*5
Значение выражения: 28
```

В программе командой `txt="(2+3)/0.25—4*2.1"` переменной `txt` значением присваивается текст, содержащий арифметическое выражение.

Тем не менее это все же текст, и именно так обрабатывается значение переменной `txt`. Она используется в команде `print(txt,"=", eval(txt))`, причем в двух аргументах функции `print()`. Первым аргументом передана собственно переменная `txt`. Поскольку переменная текстовая, то отображается ее значение — как оно есть. Затем отображается знак равенства (вторым аргументом функции `print()` является текст `"="`). Третий аргумент — это выражение `eval(txt)`, в котором переменная `txt` передается аргументом функции `eval()`. Что происходит в таком случае? Вычисляется выражение, которое содержится в строке, являющейся значением переменной `txt`. Как следствие, инструкция `eval(txt)` возвращает результат выражения `(2+3)/0.25—4*2.1`, то есть число `11.6`

Программа содержит еще один пример использования функции `eval()`. Происходит все так. Сначала при помощи команды `res=input("Введите выражение: ")` выводится запрос и считывается значение, введенное пользователем. Введенное значение присваивается переменной. Что бы пользователь ни ввел — это будет текст. Но неявно предполагается, что введенный пользователем текст содержит некоторую команду, корректную с точки зрения синтаксиса языка Python. Например, пользователь на запрос программы вводит значение `2**3+4*5`. Это корректная с точки зрения синтаксиса языка Python команда (оператор `**` означает возведение в степень), соответствующая выражению $2^3 + 4 \cdot 5$ (значение выражения равно `28`). Но данное значение будет считано как текст — то есть переменная `res` получит в качестве значения ссылку на текст `"2**3+4*5"`. Зато когда выполняется команда `print("Значение выражения:", eval(res))`, то благодаря инструкции `eval(res)` значение выражения, «спрятанного» в текст из переменной `res`, вычисляется, и вычисленное значение отображается в окне вывода.

Знакомство со списками

> Как обычно, мы занимались государственными делами.
>
> *Из к/ф «Приключения принца Флоризеля»*

Далее мы познакомимся со *списками*. Это один из множественных типов данных в Python. Помимо списков есть еще множества, кортежи и словари. Их мы тоже обсудим, но немного позже. Списки же заслуживают

особого внимания, поскольку в Python они обычно играют ту роль, которую в других языках программирования играют массивы.

> **НА ЗАМЕТКУ**
>
> В Python списки относятся к типу `list`.

Список представляет собой упорядоченный набор значений. Причем значения, формирующие список, могут быть разного типа. Существуют различные способы создания списка. Самый простой, пожалуй, состоит в том, чтобы перечислить в квадратных скобках через запятую значения, входящие в список. Например, командой `nums=[1,5,10]` создается список из трех числовых элементов (`1`, `5` и `10`), а ссылка на него записывается в переменную `nums`. Нередко для создания списков используют функцию `list()`. Если в качестве аргумента функции передать текст, то результатом возвращается список из букв (символов) данного текста. Возможны и другие варианты использования этой функции для создания списков — некоторые мы рассмотрим далее.

Для определения длины списка используют функцию `len()`. Аргументом функции передается ссылка на список, а результатом функция возвращает количество элементов в списке. Доступ к элементу списка можно получить по индексу. Индекс указывается в квадратных скобках после имени списка. Индексация элементов начинается с нуля — то есть первый элемент в списке имеет нулевой индекс. Индекс может быть отрицательным. В таком случае он определяет положение элемента в списке, начиная с конца списка. Индекс `-1` соответствует последнему элементу в списке. У предпоследнего элемента индекс `-2`, и так далее. Также есть очень полезная операция, часто на практике выполняемая со списками, — речь идет о получении *среза*. Например, имеется некоторый `список`, и нам нужно получить последовательность его элементов, начиная с элемента с индексом `i` и заканчивая элементом с индексом `j`. В таком случае мы можем выполнить срез командой вида `список[i: j+1]`. Здесь мы указываем индекс элемента, начиная с которого выполняется срез, а затем через двоеточие — индекс элемента, который следует за последним из элементов, входящих в срез. Результатом выражения вида `список[i: j+1]` является список из элементов, которые в исходном `списке` имеют индексы от `i` до `j` включительно. При этом сам исходный `список` не изменяется.

Инструкция вида `список[i:]` означает создание среза из элементов `списка`, начиная с элемента с индексом `i` и до конца `списка`.

Инструкция в формате `список[: j+1]` возвращает результатом срез из элементов, начиная с первого элемента в `списке` и до элемента с индексом `j` включительно. Наконец, значением выражения `список[:]` является копия `списка` (срез состоит из всех элементов `списка`).

НА ЗАМЕТКУ

Мы рассмотрели наиболее простые способы индексации элементов списка и получения среза. Вообще же соответствующие инструкции могут быть и более замысловатыми. Их мы рассмотрим немного позже в других главах книги.

Также следует отметить, что существует ряд встроенных функций, предназначенных для работы со списками. Например, при работе с числовыми списками полезными могут оказаться функции `max()`, `min()` и `sum()`, предназначенные, соответственно, для вычисления наибольшего и наименьшего числа в списке, а также суммы элементов списка. Функция `sorted()` позволяет отсортировать список (точнее, создает его отсортированную копию), а функция `reversed()` совместно с функцией `list()` дает возможность создать список, в котором элементы следуют в обратном порядке (по сравнению с исходным списком).

Небольшой пример, в котором создаются списки и выполняются наиболее простые операции с ними, представлен в листинге 1.6.

Листинг 1.6. Создание списков и операции с ними

```
# Список из чисел:
nums=[5,10,1,60,25,3]
# Отображение содержимого списка:
print("Список из чисел:", nums)
# Длина списка:
print("Длина списка:", len(nums))
# Первый элемент:
print("Первый элемент:", nums[0])
# Последний элемент:
print("Последний элемент:", nums[-1])
# Наибольшее значение:
print("Наибольшее значение:", max(nums))
```

```
# Наименьшее значение:
print("Наименьшее значение:", min(nums))
# Сумма:
print("Сумма:", sum(nums))
# Список в обратном порядке:
print("Список в обратном порядке:", list(reversed(nums)))
# Сортировка по возрастанию значений:
print("По возрастанию значений:", sorted(nums))
# Сортировка по убыванию значений:
print("По убыванию значений:", sorted(nums, reverse=True))
# Исходный список:
print("Исходный список:", nums)
# Изменение значения элемента списка:
nums[1]="текст"
# Отображение содержимого списка:
print("После внесения изменений:", nums)
# Получение среза:
print("Получение среза:", nums[1: len(nums)-1])
# Замена части элементов списка:
nums[1:-1]=["A","B"]
# Список после замены элементов:
print("После замены элементов:", nums)
# Список чисел от 5 до 10:
nums=list(range(5,11))
print("Список чисел от 5 до 10:", nums)
# Удаление элементов из списка:
nums[2:4]=[]
print("После удаления двух элементов:", nums)
# Удаление последнего элемента:
del nums[len(nums)-1]
print("Удален последний элемент:", nums)
# Нечетные числа:
```

```
nums=[2*k+1 for k in range(5)]
print("Нечетные числа:", nums)
# Список из символов создается на основе текста:
symbs=list("Python")
# Отображение содержимого списка:
print("Список из символов:", symbs)
# Два первых символа:
print("Два первых символа:", symbs[:2])
# Прочие символы:
print("Остальные символы:", symbs[2:])
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 1.6)

```
Список из чисел: [5, 10, 1, 60, 25, 3]
Длина списка: 6
Первый элемент: 5
Последний элемент: 3
Наибольшее значение: 60
Наименьшее значение: 1
Сумма: 104
Список в обратном порядке: [3, 25, 60, 1, 10, 5]
По возрастанию значений: [1, 3, 5, 10, 25, 60]
По убыванию значений: [60, 25, 10, 5, 3, 1]
Исходный список: [5, 10, 1, 60, 25, 3]
После внесения изменений: [5, 'текст', 1, 60, 25, 3]
Получение среза: ['текст', 1, 60, 25]
После замены элементов: [5, 'A', 'B', 3]
Список чисел от 5 до 10: [5, 6, 7, 8, 9, 10]
После удаления двух элементов: [5, 6, 9, 10]
Удален последний элемент: [5, 6, 9]
Нечетные числа: [1, 3, 5, 7, 9]
```

```
Список из символов: ['P', 'y', 't', 'h', 'o', 'n']
Два первых символа: ['P', 'y']
Остальные символы: ['t', 'h', 'o', 'n']
```

Проанализируем код этой программы и результат ее выполнения. В программе командой `nums=[5,10,1,60,25,3]` создается список из чисел. Чтобы проверить содержимое списка, используем инструкцию `print("Список из чисел:", nums)`. По умолчанию при отображении списка его элементы выводятся через запятую, а вся конструкция заключена в квадратные скобки. Длина списка (количество элементов в списке) вычисляется выражением `len(nums)`. Считывание значения первого (начального) элемента в списке выполняется с помощью инструкции `nums[0]`. Выражение `nums[-1]` представляет собой ссылку на последний элемент в списке. Также для списка `nums` вычисляется ряд характеристик: максимальное значение в списке возвращается выражением `max(nums)`, минимальное значение в списке вычисляем с помощью инструкции `min(nums)`, сумма элементов списка вычисляется командой `sum(nums)`.

Командой `list(reversed(nums))` вычисляется список, получающийся из списка `nums` обращением порядка следования элементов. При этом создается новый список, а исходный список не меняется.

ПОДРОБНОСТИ

Выражением `reversed(nums)` является объект итерационного (или итерируемого) типа (его элементы можно перебирать), который соответствует инвертированному списку `nums`. Этот итерируемый объект передается в качестве аргумента функции `list()`, в результате чего мы получаем новый список.

Для создания списка, отсортированного в порядке возрастания значений элементов, используем инструкцию `sorted(nums)`, а для сортировки в порядке убывания значений элементов задействована инструкция `sorted(nums, reverse=True)`. Здесь вторым аргументом функции `sorted()` передается инструкция `reverse=True`, являющаяся индикатором того, что сортировка должна выполняться в обратном порядке (в порядке убывания значений). И в том и в другом случае список `nums` не меняется. Убеждаемся в этом с помощью команды `print("Исходный список:", nums)`.

Команда `nums[1]="текст"` показывает, как можно изменить значение элемента списка: второй по порядку (с индексом `1`) элемент получает значение `"текст"`. Здесь мы сталкиваемся с ситуацией, когда список содержит значения разных типов. При получении среза с помощью команды `nums[1: len(nums)-1]` получаем список из всех элементов списка `nums`, за исключением первого и последнего элементов. Мы учли, что количество элементов в списке `nums` может быть вычислено инструкцией `len(nums)`, а поскольку индексация элементов начинается с нуля, то индекс последнего элемента равен `len(nums)-1` (это тот элемент, который уже не попадает в срез).

НА ЗАМЕТКУ

Для последнего элемента вместо индекса `len(nums)-1` можно было использовать индекс `-1`.

Командой `nums[1:-1]=["A","B"]` срезу `nums[1:-1]` в качестве значения присваивается список `["A","B"]`. Общий эффект такой: в списке `nums` вместо элементов, попадающих в срез `nums[1:-1]` (а это все элементы, за исключением первого и последнего), вставляются элементы списка `["A","B"]`.

Командой `nums=list(range(5,11))` создается список из натуральных чисел в диапазоне от `5` до `10` включительно, и ссылка на этот список записывается в переменную `nums`. Теперь данная переменная ссылается на новый список.

ПОДРОБНОСТИ

Когда переменной в качестве значения присваивается список, то на самом деле переменная получает в качестве значения ссылку на этот список. При присваивании переменной другого значения (например, другого списка) переменная получает значением новую ссылку. На исходный список она больше не ссылается. Если на этот исходный список в программе других ссылок нет, то такой список автоматически удаляется из памяти.

Результатом выражения `range(5,11)` является объект, реализующий последовательность чисел от `5` (первый аргумент) до `10` (значение, на единицу меньшее второго аргумента) включительно. Но этот объект — еще не список. Чтобы получить список, необходимо объект передать аргументом функции `list()`.

После выполнения команды `nums[2:4]=[]` в списке `nums` удаляются элементы с индексами от `2` до `3` включительно. Формально здесь мы присваиваем срезу пустой список, что фактически означает удаление соответствующих элементов из списка. Для удаления элемента из списка можно использовать инструкцию `del`. Например после выполнения команды `del nums[len(nums)-1]` из списка `nums` удаляется последний элемент.

В программе есть пример использования *генератора списка*. Речь идет о команде `nums=[2*k+1 for k in range(5)]`, с помощью которой создается список из пяти нечетных чисел в диапазоне значений от `1` до `9` включительно. Ссылка на список присваивается переменной `nums`. Приведенная выше команда заслуживает особых пояснений.

Итак, в выражении с генератором списка имеются квадратные скобки, внутри которых размещена инструкция такого вида: `выражение for переменная in диапазон`. То есть начинается все с некоторого `выражения`, после которого следует ключевое слово `for`, затем имя `переменной`, ключевое слово `in` и, наконец, конструкция, задающая `диапазон` изменения `переменной`. Список формируется следующим образом: `переменная`, указанная после ключевого слова `for`, последовательно принимает значения из `диапазона`, указанного после ключевого слова `in`. В нашем случае после ключевого слова `for` указана переменная `k`, а после ключевого слова `in` указана инструкция `range(5)`. Поскольку в выражении `range(5)` присутствует только один аргумент `5`, то значением выражения является объект, реализующий последовательность чисел от `0` до `4` (значение, на единицу меньшее аргумента). Таким образом, переменная `k` последовательно принимает значения от `0` до `4`. При каждом из указанных значений переменной `k` создается элемент и вносится в формируемый генератором список. Значение элемента определяется выражением, указанным перед ключевым словом `for`. Там мы указали выражение `2*k+1`. В него входит переменная `k`. Подставляя в выражение `2*k+1` значения для переменной `k` от `0` до `4`, получаем элементы списка.

Командой `symbs=list("Python")` создается список из символов, входящих в текстовое значение `"Python"`, переданное аргументом функции `list()`. Инструкцией `symbs[:2]` вычисляется срез, состоящий из двух начальных символов из списка (элементы с индексами от `0` до `1` включительно). Инструкция `symbs[2:]` определяет срез, состоящий из символов, начиная с символа с индексом `2` и до конца списка.

НА ЗАМЕТКУ

К вопросу использования списков мы еще будем возвращаться. Пока же нам будет достаточно тех сведений о списках, которые представлены выше.

Знакомство с условным оператором

> Значит, такое предложение: сейчас мы нажимаем на контакты и перемещаемся к вам. Но если эта машинка не сработает, тогда уж вы с нами переместитесь, куда мы вас переместим!
>
> *Из к/ф «Кин-дза-дза»*

Условный оператор позволяет выполнять разные блоки команд в зависимости от истинности или ложности некоторого условия. Синтаксис условного оператора `if` следующий:

```
if условие:
    команды
else:
    команды
```

После ключевого слова `if` указывается выражение (условие), истинность которого проверяется. Далее следует двоеточие и блок команд. Каждая команда в блоке выделяется отступом (обычно это четыре пробела). Команды данного блока выполняются в том случае, если условие истинно.

После первого блока команд следует ключевое слово `else`, которое завершается двоеточием. Ключевое слово `else` по горизонтали размещается на том же расстоянии, что и ключевое слово `if`. Под ключевым словом `else` размещается блок команд, выполняемых при ложном условии. Команды выделяются отступом (обычно в четыре пробела).

ПОДРОБНОСТИ

Здесь мы впервые сталкиваемся с механизмом, используемым в Python для структурирования программного кода. В таких языках программирования, как Java, C++, C# и некоторых других, блоки команд выделяются с помощью фигурных скобок. В языке Python

код структурируется с помощью отступов. В принципе, количество пробелов при выделении блока команд может быть любым, но оно должно быть одним и тем же для всей программы. Однако рекомендуется выполнять отступ в четыре пробела. Именно такие отступы мы и будем использовать в программных кодах.

Пример использования условного оператора приведен в листинге 1.7.

Листинг 1.7. Знакомство с условным оператором

```
# Считывание целого числа:
number=int(input("Введите целое число: "))
# Если число — четное:
if number%2==0:
    print("Вы ввели четное число.")
# Если число — нечетное:
else:
    print("Вы ввели нечетное число.")
```

При запуске программы на выполнение появляется сообщение с предложением ввести целое число. Далее все зависит от того, четное или нечетное число вводит пользователь. Ниже показано, каким будет результат выполнения программы, если пользователь вводит четное число (введенное пользователем значение выделено жирным шрифтом).

Результат выполнения программы (из листинга 1.7)

```
Введите целое число: 8
Вы ввели четное число.
```

Если же пользователь вводит нечетное число, то результат следующий.

Результат выполнения программы (из листинга 1.7)

```
Введите целое число: 9
Вы ввели нечетное число.
```

Таким образом, в программе введенное пользователем число проверяется на четность/нечетность, и в зависимости от результатов проверки

в окне вывода появляется соответствующее сообщение. Реализуется все это с помощью условного оператора. Сначала в программе командой `number=int(input("Введите целое число: "))` считывается, преобразуется в целочисленный формат и записывается в переменную `number` значение, введенное пользователем. После этого в игру вступает условный оператор. В условном операторе проверяется условие `number%2==0`. Это выражение, которое может принимать значения `True` (условие истинно) и `False` (условие ложно). Значение выражения рассчитывается следующим образом.

- Результатом выражения `number%2` является остаток от целочисленного деления значения переменной `number` на `2` (здесь мы использовали оператор% вычисления остатка от деления). Для четных чисел остаток от деления на `2` равен `0`, а для нечетных чисел остаток от деления на `2` равен `1`.
- Оператор сравнения значений на предмет равенства `==` позволяет определить, равно ли значение выражения `number%2` нулю или нет. Значение выражения `number%2==0` равно `True`, если значение выражения `number%2` равно `0`. В противном случае (если значение выражения `number%2` не равно `0`) результатом выражения `number%2==0` является логическое значение `False`.

Если при проверке в условном операторе значение условия равно `True` (условие истинно), то выполняется команда `print("Вы ввели четное число.")`. Если же значение условия в условном операторе равно `False` (условие ложно), то выполняется команда `print("Вы ввели нечетное число.")`.

ПОДРОБНОСТИ

Мы рассмотрели достаточно простую форму условного оператора. Но вообще ситуация не такая тривиальная. Например условные операторы могут быть вложенными, причем для таких случаев есть специальная форма условного оператора. Существуют некоторые особенности, связанные с описанием условия в условном операторе. Все это мы рассмотрим немного позже. Здесь мы только познакомились с самыми общими принципами использования условного оператора.

Знакомство с оператором цикла

> Это великолепно просто. По простоте это напоминает античность.
>
> *Из к/ф «Приключения принца Флоризеля»*

Нередко случается так, что необходимо определенное количество раз выполнить некоторый блок команд. В таких случаях полезно использовать *операторы цикла*. Далее мы кратко рассмотрим оператор цикла `while`. Синтаксис у этого оператора простой:

```
while условие:
    команды
```

После ключевого слова `while` указывается условие, двоеточие и затем блок команд, которые выполняются, пока условие истинно. Команды, относящиеся к оператору цикла, выделяются отступом в четыре пробела. Выполняется оператор цикла `while` следующим образом. Сначала проверяется условие, указанное после ключевого слова `while`. Если условие истинно, то выполняются команды в теле оператора цикла (то есть команды из блока, относящегося к оператору цикла). Затем снова проверяется условие. Если оно истинно, снова выполняются команды в теле оператора цикла, опять проверяется условие и так далее. Процесс продолжается до тех пор, пока при проверке условия не окажется, что оно ложно. В таком случае выполнение оператора цикла завершается, и выполняется команда, следующая после оператора цикла.

В рассматриваемом в листинге 1.8 примере с помощью оператора цикла вычисляется сумма натуральных чисел. Верхняя граница суммы вводится пользователем с клавиатуры.

Листинг 1.8. Вычисление суммы

```
# Считывание верхней границы суммы:
n=int(input("Укажите верхнюю границу суммы: "))
# Начальное значение для суммы:
s=0
# Начальное значение индексной переменной:
k=0
```

```
# Оператор цикла для вычисления суммы:
while k<n:
    # Увеличение значения индексной переменной на единицу:
    k=k+1
    # Прибавление слагаемого к сумме:
    s=s+k
# Отображение результата:
print("Сумма чисел от 1 до", n,"равна", s)
```

Результат выполнения программы может быть таким, как показано ниже (жирным шрифтом выделено значение, которое вводит пользователь).

Результат выполнения программы (из листинга 1.8)

```
Укажите верхнюю границу суммы: 100
Сумма чисел от 1 до 100 равна 5050
```

Код программы начинается с инструкции `n=int(input("Укажите верхнюю границу суммы: "))`, которая считывает целочисленное значение для верхней границы суммы (это фактически значение последнего слагаемого в сумме). Командами `s=0` и `k=0` в программе создаются две переменные с начальными нулевыми значениями. В первую предполагается записывать значение суммы натуральных чисел, а с помощью второй переменной (которую мы будем называть индексной) выполняется подсчет количества слагаемых, добавленных в сумму.

В операторе цикла после ключевого слова `while` в качестве условия указано выражение `k<n`. Значение выражения равно `True`, если значение переменной `k` меньше значения переменной `n`. Если условие истинно, то команда `k=k+1` в теле оператора цикла увеличивает на единицу текущее значение переменной `k`, после чего команда `s=s+k` прибавляет это новое значение к сумме.

ПОДРОБНОСТИ

Чтобы понять, как все это «работает», рассмотрим процесс с самого начала. Допустим, что пользователь ввел в программу положительное целочисленное значение для переменной `n`, например значение `100`. Тогда, поскольку начальное значение переменной `k` равно `0`,

при первой проверке условия `k<n` в операторе цикла оно будет истинным. Следовательно, начнут выполняться команды в теле оператора цикла. Там сначала после выполнения команды `k=k+1` переменная `k` получит значение `1`, после чего, вследствие выполнения команды `s=s+k`, к текущему нулевому значению переменной `s` будет прибавлено значение `1` (значение переменой `k`). При проверке условия `k<n` оно окажется истинным (число `1` строго меньше числа `100`), и команды в теле оператора цикла будут выполнены еще раз. Значение переменной `k` станет равным `2`, а значение переменной `s` будет равно сумме чисел `1` (добавка за первый цикл) и `2` (добавка за второй цикл). Далее, условие `k<n` остается истинным (число `2` строго меньше числа `100`), поэтому команды в теле оператора цикла выполняются еще раз. После того как команды выполнены `n-1` раз, значение переменной `k` равно `n-1`, а значение переменной `s` равно сумме чисел от `1` до `n-1` включительно. Проверка условия `k<n` показывает, что оно истинно. Команды в теле цикла выполняются еще раз. Переменная `k` получает значение `n`, значение переменной `s` равно сумме чисел от `1` до `n` включительно. Но очередная проверка условия `k<n` дает значение `False` (значение переменной `k` равно `n`, что не меньше, чем значение переменной `n`). Поэтому команды в теле оператора цикла больше не выполняются.

После завершения оператора цикла сумма вычислена, и ее значение записано в переменную `s`. Поэтому команда `print("Сумма чисел от 1 до", n,"равна", s)` отобразит вычисленное значение в окне вывода.

НА ЗАМЕТКУ

Сумма натуральных чисел от 1 до n равна n(n + 1) / 2.

Рассмотренный выше способ вычисления суммы натуральных чисел в каком-то смысле является «классическим» — так вычисляют сумму в большинстве языков программирования. В языке Python этот алгоритм тоже работает. Но вообще язык Python очень гибкий и красивый. Многие задачи в нем решаются просто и элегантно. Например, чтобы вычислить сумму натуральных чисел от `1` до `n` (при условии, что значение переменной `n` задано), можно было бы воспользоваться командой `sum(list(range(1, n+1)))`. Еще один вариант, характерный именно для языка Python и подразумевающий использование оператора цикла, представлен в листинге 1.9.

Листинг 1.9. Альтернативный способ вычисления суммы

```
# Считывание количества слагаемых:
n=input("Укажите количество слагаемых: ")
# Переменная с текстовым значением:
txt="1"
# Индексная переменная:
k=1
# Оператор цикла для формирования текста с
# выражением для суммы натуральных чисел:
while str(k)!=n:
    # Увеличение значения индексной переменной:
    k=k+1
    # Добавление фрагмента к тексту с выражением
    # для суммы натуральных чисел:
    txt=txt+"+"+str(k)
# Отображение результата:
print(txt,"=", eval(txt))
```

Результат выполнения программы может быть таким, как представлено ниже (жирным шрифтом выделено введенное пользователем значение).

Результат выполнения программы (из листинга 1.9)

```
Укажите количество слагаемых: 10
1+2+3+4+5+6+7+8+9+10 = 55
```

Как и в предыдущем случае, в этой программе мы сначала считываем значение для количества слагаемых в сумме (верхняя граница суммы). Специфика ситуации в том, что считанное значение к целочисленному формату не преобразуется, а сохраняется в текстовом виде. Таким образом, переменная `n`, значение которой определяется командой `n=input("Укажите количество слагаемых: ")`, ссылается на текстовое значение.

ПОДРОБНОСТИ

Предполагается, что пользователь вводит положительное целое число. Оно считывается. Но функция `input()` возвращает результат в текстовом виде. Поэтому значением переменной `n` является текстовое представление для целого положительного числа. Последнее («целость» и «положительность») всецело ложится на пользователя.

Также командой `txt="1"` мы создаем в программе переменную `txt`, начальное значение которой является текстовым представлением для единицы. Переменная `k` создается с начальным значением `1`. Она будет использована для подсчета количества слагаемых в сумме. Общая же стратегия вычисления суммы в данном случае состоит в том, чтобы сформировать текстовую строку, содержащую явное выражение для суммы натуральных чисел. Затем с помощью функции `eval()` эта сумма вычисляется.

Формирование текстовой строки с выражением для суммы выполняется в операторе цикла. В нем проверяется условие `str(k)!=n`. Его основу составляет оператор `!=` «не равно». Значение выражения на основе этого оператора равно `True`, если операнды справа и слева от оператора имеют разные значения. В данном случае выражение `str(k)!=n` возвращает значение `True`, если значение выражения `str(k)` отличается от значения переменной `n`. Значением выражения `str(k)` является текстовое представление для числового значения переменной `k`. Таким образом, условие `str(k)!=n` истинно, если значение переменной `k` отличается от числа, «спрятанного» в текстовом значении переменной `n`.

ПОДРОБНОСТИ

Напомним, что целочисленные значения относятся к типу `int`, а текстовые значения относятся к типу `str`. Соответственно, для приведения к целочисленному типу используют функцию `int()`, а для приведения к текстовому типу используют функцию `str()`. Нам необходимо сравнить числовое значение переменной `k` со значением, которое содержится в текстовой строке, являющейся значением переменной `n`. То есть надо сравнить два числа, но одно из них реализовано как число, а второе — как текст. Понятно, что следует либо преобразовать текст к числовому типу, либо число — к текстовому типу. Мы идем вторым путем и при сравнении значений с помощью команды `str(k)` получаем текстовое представление для числового значения переменной `k`. При этом значение самой переменной не меняется, поэтому в теле оператора цикла мы можем выполнять команду `k=k+1`.

В теле оператора цикла командой `k=k+1` значение индексной переменной увеличивается на единицу, после чего командой `txt=txt+"+"+str(k)` к текущему текстовому значению переменной `txt` дописывается знак плюс и текстовое представление для числового значения переменной `k`. В результате после завершения оператора цикла текстовое значение переменной `txt` представляет собой строку с явным выражением для суммы натуральных чисел. Командой `print(txt,"=", eval(txt))` отображается как само текстовое значение, так и значение «спрятанного» в нем выражения, вычисленного с помощью функции `eval()`.

> **НА ЗАМЕТКУ**
>
> В Python существуют и другие способы реализации циклов. Их мы обсудим в следующих главах книги.

Знакомство с функциями

> Дальше следует непереводимая игра слов с использованием местных идиоматических выражений.
>
> *Из к/ф «Бриллиантовая рука»*

Функция представляет собой именованный блок программного кода, который можно выполнить, вызвав функцию. При вызове функции могут передаваться параметры, которые называются *аргументами* функции. В Python функции описываются исключительно просто. Общий шаблон описания функции представлен ниже.

```
def имя(аргументы):
    команды
```

Начинается описание с ключевого слова `def`, после которого указывают название функции, ее аргументы (в круглых скобках), двоеточие и, наконец, блок команд, которые выполняются при вызове функции. Как и в случае с условным оператором и оператором цикла, блок команд, относящихся к телу функции, выделяется отступом (в четыре пробела). Для вызова функции необходимо в соответствующей команде указать имя функции и, при необходимости, аргументы, которые передаются функции (в круглых скобках после имени функции). Вызов функции означает,

что выполняются команды из тела функции, причем в качестве аргументов используются те значения, которые переданы функции при вызове.

Функция может возвращать значение, а может не возвращать. Если функция не возвращает значение, то речь идет о выполнении определенного набора команд при вызове функции. Если функция возвращает результат, то инструкцию вызова функции можно использовать как операнд в более сложном выражении. При этом инструкция вызова функции отождествляется со значением, возвращаемым функцией. Чтобы в теле функции определить значение, которое функция возвращает, используют инструкцию `return`, после которой указывается возвращаемое функцией значение. Выполнение инструкции `return` в теле функции приводит к завершению выполнения кода функции. Поэтому очень часто (но не всегда) команда с `return`-инструкцией в теле функции является последней. Инструкция `return` может использоваться и в функции, которая не возвращает значение. В таком случае после инструкции `return` никакое значение не указывается, а ее выполнение просто приводит к завершению выполнения кода функции.

Описание функции может размещаться в любом месте программы, но только до того места, где функция впервые вызывается. В листинге 1.10 представлен программный код, в котором объявляются (а затем вызываются) две функции.

Листинг 1.10. Использование функций

```
# Функция для отображения букв из переданного
# аргументом текста:
def show(txt):
    # Преобразование текста в список и его сортировка:
    symbs=sorted(list(txt))
    # Отображение содержимого списка:
    print(symbs)
# Вызов функции:
show("Python")
# Функция для вычисления суммы квадратов натуральных чисел:
def sqsum(n):
    # Создание списка из квадратов натуральных чисел:
    nums=[k*k for k in range(1, n+1)]
```

```
    # Результат функции:
    return sum(nums)
# Переменная с числовым значением:
m=10
# Вызов функции для вычисления суммы квадратов чисел:
print("Сумма квадратов чисел от 1 до", str(m)+":", sqsum(m))
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 1.10)

```
['P', 'h', 'n', 'o', 't', 'y']
Сумма квадратов чисел от 1 до 10: 385
```

Программный код начинается с описания функции, которая называется `show()` (здесь и далее названия функций будут приводиться с пустыми круглыми скобками). В описании функции у нее объявлен один аргумент, который называется `txt`. Неявно предполагается, что это текстовое значение. В теле функции командой `symbs=sorted(list(txt))` на основе текстового (так мы думаем) аргумента создается список, состоящий из символов текстовой строки, причем список сортируется в порядке возрастания значений (для символов сравниваются их коды в кодовой таблице). После создания списка он отображается командой `print(symbs)`.

НА ЗАМЕТКУ

Переменные, которые «появляются» в теле функции, включая аргумент (или аргументы) функции, доступны только в теле функции. Такие переменные называются локальными.

Примером вызова функции `show()` служит команда `show("Python")`, в которой функции передается текстовая строка `"Python"`. В результате на основе текста формируется упорядоченный список и отображается в области вывода.

Функция для вычисления суммы квадратов натуральных чисел называется `sqsum()`, и ее аргумент, обозначающий верхнюю границу суммы, обозначен как `n`.

ПОДРОБНОСТИ

Речь идет о функции для вычисления суммы квадратов натуральных чисел $1^2 + 2^2 + 3^2 + \ldots + n^2 = n(n + 1)(2n + 1) / 6$. Аргументом функции передается значение n. Результатом функция возвращает значение для суммы $1^2 + 2^2 + 3^2 + \ldots + n^2$, причем формула $n(n + 1)(2n + 1) / 6$ при вычислении результата функции не используется.

В теле функции командой `nums=[k*k for k in range(1, n+1)]` создается список, состоящий из квадратов натуральных чисел от `1` до `n` (аргумент функции) включительно. При создании списка использован генератор списков: значение элементов списка вычисляется выражением `k*k`, и при этом переменная `k` последовательно принимает значения от `1` до `n`.

НА ЗАМЕТКУ

Диапазон изменения переменной `k` определяется выражением `range(1, n+1)`. Напомним, что это объект итерируемого типа, через который реализуется последовательность чисел от `1` до `n`.

После того как список из квадратов натуральных чисел создан и ссылка на него записана в переменную `nums`, командой `return sum(nums)` вычисляется сумма квадратов натуральных чисел, и полученное значение возвращается в качестве результата функции.

НА ЗАМЕТКУ

Здесь мы использовали встроенную функцию `sum()`, которая позволяет вычислить сумму значений элементов списка, переданного аргументом функции.

Функция для вычисления суммы квадратов чисел используется в команде `print("Сумма квадратов чисел от 1 до", str(m)+":", sqsum(m))`. Функции `sqsum()` в качестве аргумента передается переменная `m` со значением `10`. Легко проверить, что значение для суммы квадратов вычислено правильно.

Резюме

> Солнце есть. Песок есть. Притяжение есть. Где мы? Мы на Земле.
>
> *Из к/ф «Кин-дза-дза»*

- Программа — это последовательность команд. В процессе выполнения программы команды выполняются одна за другой.
- В языке Python есть несколько полезных встроенных функций, предназначенных для ввода и вывода данных: вывод данных в окно интерпретатора осуществляется с помощью функции `print()`, для ввода данных используют функцию `input()`. Функция `eval()` позволяет вычислить выражение в текстовой строке, которая передается аргументом функции.
- Переменные в Python не имеют типа. Для использования переменной в программе ей просто присваивается значение. На разных этапах выполнения программы переменная может ссылаться на данные разных типов. Для удаления переменной используют инструкцию `del`, после которой указывается имя удаляемой переменной.
- В программе могут использоваться комментарии. Комментарий начинается с символа `#` и игнорируется при выполнении программы.
- Целочисленные значения относятся к типу `int`, текстовые значения относятся к типу `str`. Для преобразования к целочисленному типу используют функцию `int()`, для преобразования к текстовому типу используют функцию `str()`. Текстовые литералы заключаются в двойные или одинарные кавычки.
- Список представляет собой упорядоченный набор элементов, которые, в принципе, могут относиться к разным типам. Создается список перечислением значений элементов в квадратных скобках или, например, с использованием функции `list()`. Могут использоваться и иные подходы (например, с помощью генератора списка). Длина списка определяется с помощью функции `len()`. Для получения доступа к элементу массива после имени массива в квадратных скобках указывается индекс элемента. Индексация элементов начинается с нуля. Отрицательный индекс используется для определения позиции элемента, начиная с конца списка. При работе со списками можно получать срезы, представляющие собой список из элементов исходного массива. Индекс первого входящего в срез элемента

и первого не входящего в срез элемента указываются в квадратных скобках после имени массива и разделяются двоеточием.

- Условный оператор `if` позволяет выполнять разные блоки команд в зависимости от истинности или ложности определенного условия. Оператор цикла `while` используется для многократного выполнения определенного блока команд. Команды выполняются, пока истинно условие, указанное в операторе цикла.
- Функция представляет собой именованный блок программного кода, который можно вызвать по имени. У функции могут быть аргументы. При описании функции после ключевого слова `def` указывается имя функции, ее аргументы (в круглых скобках) и через двоеточие — блок команд, которые выполняются при вызове функции. Инструкция `return` в теле функции завершает выполнение функции. Если после этой инструкции указано значение, оно возвращается результатом функции. Для вызова функции указывается ее значение и, если необходимо, в круглых скобках аргументы, которые передаются функции.

Задания для самостоятельной работы

> Ну, барин, ты задачи ставишь! За десять дён одному не справиться. Тут помощник нужен — хомо сапиенс.
>
> *Из к/ф «Формула любви»*

1. Напишите программу, в которой программа запрашивает у пользователя день и месяц, а затем выводит сообщение о текущем дне и месяце.

2. Напишите программу, в которой пользователь должен ввести текущий год и год своего рождения. Программа вычисляет возраст пользователя и выводит соответствующее сообщение.

3. Напишите программу, в которой расстояние, указанное в милях, переводится в километры. Учесть, что одна миля примерно равна *1,6* километра. Расстояние в милях вводится пользователем с клавиатуры.

4. Напишите программу, в которой создается и отображается список, содержащий степени двойки (числа 2^0, 2^1, 2^2, 2^3 и так далее). Размер списка (количество чисел в списке) вводится пользователем с клавиатуры.

5. Напишите программу, создающую список из чисел, которые при делении на *5* дают в остатке *3* (такие числа вычисляются по формуле $5k + 3$, где $k = 0,1,2...$). Отобразить этот список в прямом и обратном порядке.

6. Напишите программу, в которой проверяется, делится ли введенное пользователем число на *3*. Учесть, что если число делится на *3*, то остаток от деления этого числа на *3* равен нулю.

7. Напишите программу, в которой вычисляется факториал числа. Факториалом $n!$ числа n называется произведение всех чисел от единицы до этого числа: $n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$. Число, для которого вычисляется факториал, вводится пользователем с клавиатуры. В программе должна выполняться проверка того, что пользователь ввел положительное число (используйте условный оператор).

8. Напишите программу, в которой отображаются числа из последовательности Фибоначчи. Первые два числа в последовательности равны единице, а каждое следующее равно сумме двух предыдущих значений (то есть речь о числах *1*, *1*, *2*, *3*, *5*, *8*, *13*, *21*, *34*, *55* и так далее). Количество чисел в последовательности вводится с клавиатуры.

9. Напишите программу, в которой описана функция, возвращающая результатом второе по величине число в списке, переданном функции в качестве аргумента.

10. Напишите программу, в которой описана функция, возвращающая результатом сумму нечетных чисел. Количество чисел передается аргументом функции.

Глава 2
ОСНОВНЫЕ ОПЕРАЦИИ

> История, которую мы хотим рассказать, не опирается на факты: она настолько невероятна, что в нее просто нельзя не поверить.
>
> *Из к/ф «О бедном гусаре замолвите слово»*

В этой главе речь в основном пойдет об управляющих инструкциях языка Python. С некоторыми из них (например, условным оператором и оператором цикла) мы уже сталкивались в предыдущей главе, но в ней знакомство было поверхностным. Здесь мы более детально обсудим условный оператор, операторы цикла, тернарный оператор, познакомимся с принципами использования переменных и узнаем об особенностях операции присваивания. Также будут рассмотрены некоторые аспекты обработки исключительных ситуаций и специфика работы с разными типами данных.

Начнем с операторов цикла. Оператор цикла `while` мы рассматривали в предыдущей главе. В Python это не единственный оператор цикла. Есть еще оператор цикла `for`. Да и оператор `while` имеет разные модификации.

Оператор цикла `while`

> — За сколько сделаешь?
> — За день сделаю.
> — А за два?
>
> *Из к/ф «Формула любви»*

У *оператора цикла* `while` в действительности есть две формы (с одной мы познакомились ранее). Наиболее общий шаблон оператора цикла представлен ниже (жирным шрифтом выделены ключевые элементы шаблона):

```
while условие:
    # команды
else:
    # команды
```

Весь блок начинается ключевым словом `while`, после которого указывают некоторое `условие` (выражение, которое может интерпретироваться как истинное или ложное). После `условия` ставится двоеточие. Далее следует блок команд. После блока команд идет ключевое слово `else` (заканчивается двоеточием). После ключевого слова `else` следует еще один блок команд. Ключевое слово `else` находится на том же уровне (по горизонтали), что и ключевое слово `while`. Блоки команд также выделяются отступами по отношению к ключевым словам `while` и `else` (четыре отступа по умолчанию).

Выполняется условный оператор `while` следующим образом: проверяется `условие`, и если оно истинно, то выполняется блок команд после ключевого слова `while`. После этого снова проверяется условие. Если оно истинно, снова выполняются команды из `while`-блока, и так далее. Процесс продолжается до тех пор, пока `условие` при проверке не окажется ложным. Если условие ложное, то выполняются команды после ключевого слова `else`. На этом выполнение оператора цикла завершается.

Сделаем несколько замечаний. Во-первых, в операторе цикла `while` можно не использовать `else`-блок. Именно с такой версией оператора цикла мы познакомились в предыдущей главе. Во-вторых, есть две важные инструкции, которые могут использоваться в операторах цикла. Выполнение инструкции `break` приводит к завершению оператора цикла, причем без выполнения команд в `else`-блоке. Инструкция `continue` используется в том случае, если необходимо завершить текущий цикл и сразу перейти к проверке условия в `while`-инструкции. Сначала мы рассмотрим небольшой пример, в котором используется упрощенная версия оператора цикла (без `else`-блока). В этом примере, кроме прочего, мы будем иметь дело с двумя арифметическими операторами: с помощью оператора `//` выполняется целочисленное деление, а с помощью оператора `%` вычисляется остаток от деления.

Программа такая: пользователь вводит целое число, а программа печатает его представление (имеются в виду цифры, входящие в состав числа)

в обратном порядке, причем между цифрами в качестве разделителя используется вертикальная черта. Рассмотрим пример в листинге 2.1.

Листинг 2.1. Отображение состава числа

```
# Вводится число:
number=int(input("Введите число: "))
# Пока число больше нуля:
while number>0:
    # Последняя цифра в числе:
    digit=number%10
    # Отображение цифры:
    print("|"+str(digit), end="")
    # Отбрасывается последняя цифра в числе:
    number=number//10
# Отображается последний разделитель:
print("|")
```

При выполнении программы результат может быть следующим (жирным шрифтом выделено значение, которое вводит пользователь).

Результат выполнения программы (из листинга 2.1)

```
Введите число: 31049265
|5—6|2—9|4—0|1—3|
```

Алгоритм простой. Сначала пользователь должен ввести число. Командой `number=int(input("Введите число: "))` это число считывается, преобразуется из текстового формата в целочисленный формат, а ссылка на это число записывается в переменную `number`. После этого выполняется оператор цикла `while`, который выполняется до тех пор, пока число, на которое ссылается переменная `number`, больше нуля (условие `number>0`). За каждый цикл выполняются следующие команды. Сначала командой `digit=number%10` вычисляется последняя цифра в числе, на которое ссылается переменная `number`. Значение выражения `number%10` — это остаток от деления значения переменной `number` на `10`. Несложно сообразить, что это последняя цифра в представлении соответствующего числа. Результат записывается в переменную `digit`. Это значение отображается в окне

вывода. Функции `print()` передаются два аргумента. Первый аргумент определяется выражением `"|"+str(digit)`. Здесь мы имеем сумму двух текстовых значений: литерала `"|"` и выражения `str(digit)`, которое представляет собой результат преобразования к текстовому формату числового значения `digit`. При вычислении суммы двух текстовых значений результатом будет текстовое значение, получающееся в результате объединения суммируемых текстов.

ПОДРОБНОСТИ

При вычислении выражений `number%10` и `str(digit)` значения переменных `number` и `digit` не меняются.

Второй аргумент функции `end=""` является *именованным*. Указывается название аргумента `end` и, через знак равенства, собственно значение этого аргумента `""` (пустая текстовая строка). Назначение аргумента следующее: по умолчанию, если функции `print()` передается один текстовый аргумент, то значение данного аргумента отображается в окне вывода и курсор автоматически переводится в новую строку (поэтому следующее сообщение будет отображаться в новой строке). Формально это означает, что после отображения «полезного текста» (определяется аргументом функции `print()`) автоматически «печатается» еще и инструкция перехода к новой строке. Такой режим можно изменить. Для этого указывается именованный аргумент `end`, который определяет текст, отображаемый вместо инструкции перехода к новой строке. Таким образом, аргумент `end=""` функции `print()` означает, что после отображения значения первого аргумента переход к новой строке не выполняется, а вместо этого отображается текст `""` (то есть ничего не отображается). В итоге в процессе работы оператора цикла при выполнении команды `print("|"+str(digit), end="")` все сообщения отображаются в одной строке.

При выполнении команды `number=number//10` в числе, на которое ссылается переменная `number`, отбрасывается последняя цифра, и ссылка на новое значение записывается в переменную `number`.

ПОДРОБНОСТИ

Значение выражения `number//10` вычисляется как результат целочисленного деления значения переменной `number` на `10`. Целочисленное деление числа на `10` сводится к тому, что в делимом числе отбрасывается последняя цифра. При вычислении выражения `number//10` значение переменной `number` не меняется.

Таким образом, за один цикл последняя цифра в текущем значении переменной `number` отображается в окне вывода, после чего в этом значении отбрасывается последняя цифра и переменная `number` ссылается на новое значение. Поэтому значение переменной `number` постоянно уменьшается (при условии, что исходное значение этой переменной положительно). Как только значение переменной `number` станет нулевым, оператор цикла завершится (при очередной проверке условия `number>0`). К этому моменту в окне вывода будут отображены все цифры из исходного значения переменной `number`. Не хватает только завершающей черточки. Она отображается командой `print("|")`, которая выполняется после завершения оператора цикла.

НА ЗАМЕТКУ

Вообще, эту задачу можно было решить несколько иначе: получить текстовое представление числа, инвертировать его и отобразить символы в окне. Другими словами, можно было обрабатывать введенное пользователем значение не как число, а как текст. Но для нас в данном случае важен был не только результат, но и метод его получения.

Далее рассматривается еще один пример использования оператора цикла `while` (на этот раз с использованием `else`-блока и ключевого слова `break`). Речь идет о программе, при выполнении которой пользователь вводит число, после чего выполняется проверка того, является ли это число простым.

ПОДРОБНОСТИ

Число является простым, если оно не имеет никаких делителей, кроме единицы и самого себя. Например, число `7` является простым, поскольку оно делится только на `1` и `7`. А число `9` простым не является, поскольку кроме `1` и `9` оно еще делится и на `3`.

Рассмотрим программный код, представленный в листинге 2.2.

Листинг 2.2. Простые числа

```
# Вводится число:
number=int(input("Введите число: "))
# Верхняя граница для делителя:
num=number//2
```

```
# Начальное значение делителя:
k=2
# Поиск делителей числа:
while k<=num:
    # Если число делится на k:
    if number%k==0:
        print("Число не является простым")
        # Завершение оператора цикла:
        break
    # Если условие ложно:
    else:
        # Увеличивается значение делителя:
        k=k+1
# Блок выполняется, если не выполнена инструкция break:
else:
    print("Это простое число")
# Сообщение отображается всегда:
print("Проверка завершена")
```

Результат выполнения программы может быть таким (жирным шрифтом выделено введенное пользователем значение).

Результат выполнения программы (из листинга 2.2)

```
Введите число: 97
Это простое число
Проверка завершена
```

Или таким.

Результат выполнения программы (из листинга 2.2)

```
Введите число: 95
Число не является простым
Проверка завершена
```

В программе реализован следующий алгоритм: сначала вводится число (которое проверяется), а затем последовательно вычисляется остаток от деления этого числа на 2, 3, 4 и так далее. Если остаток от деления равен нулю, это означает, что у числа есть соответствующий делитель. В таком случае число не является простым. Если окажется, что у числа делителей нет, значит, оно простое.

Проверяемое число вводится (а ссылка на это число записывается в переменную number) командой number=int(input("Введите число: ")). Очевидно, что самый большой делитель не может превышать половины от числа. Поэтому при помощи команды num=number//2 определяем переменную num, значение которой вычисляется как результат целочисленного деления значения number на 2. Переменная num определяет верхнюю границу для делителей числа number. Командой k=2 определяем переменную k с начальным значением 2. Эту переменную отождествляем с возможным делителем числа number. Поиск делителей выполняется с использованием оператора цикла. В нем проверяется условие k<=num (возможный делитель не может превышать половины от делимого числа). Тело оператора цикла состоит из условного оператора. В условном операторе проверяется условие number%k==0 (остаток от деления number на число num равно нулю — то есть число number делится на число k без остатка). Если так, то число number не является простым. В этом случае командой print("Число не является простым") выводится соответствующее сообщение, после чего инструкцией break завершается выполнение оператора цикла (поскольку дальше продолжать проверку нет смысла). Важное обстоятельство: если оператор цикла завершается в результате выполнения инструкции break, то команда print("Это простое число") в else-блоке оператора цикла не выполняется (наличие else-блока игнорируется).

Если условие в условном операторе ложно (число number не делится без остатка на k), то в else-блоке условного оператора командой k=k+1 значение переменной k увеличивается на единицу. Если в процессе перебора разных значений переменной k ни один делитель числа number так и не будет найден (то есть проверяемое число простое), и при очередной проверке условие k<=num окажется ложным, оператор цикла завершит свою работу выполнением команды print("Это простое число") в else-блоке. Что касается команды print("Проверка завершена"), то она размещена после оператора цикла и выполняется всегда.

НА ЗАМЕТКУ

Стоит заметить, что в рассмотренном выше примере в условном операторе (внутри оператора цикла) `else`-блок можно было не использовать, а разместить команду `k=k+1` в теле оператора цикла (после условного оператора в сокращенной форме). Другими словами, мы могли использовать такой программный код:

```
number=int(input("Введите число: "))
num=number//2
k=2
while k<=num:
    if number%k==0:
        print("Число не является простым")
        break
    k=k+1
else:
    print("Это простое число")
print("Проверка завершена")
```

Выполняться этот код будет так же, как и код из листинга 2.2. Объяснение простое: если на каком-то цикле условие в условном операторе окажется истинным, то будет выполнена инструкция `break`, оператор цикла завершит выполнение, и до выполнения команды `k=k+1` дело не дойдет.

Далее на примере похожей программы мы проиллюстрируем использование инструкции `continue`. В программе, представленной в листинге 2.3, для введенного пользователем числа определяются все его делители.

Листинг 2.3. Поиск делителей числа

```
# Вводится число:
number=int(input("Введите число: "))
# Сообщение о первом делителе числа:
print("Делится на",1)
```

```
# Начальное значение для делителя:
k=1
# Поиск делителей числа:
while k<number//2:
    # Значение делителя увеличивается на единицу:
    k=k+1
    # Если k не является делителем числа:
    if number%k!=0:
        # Завершение текущего цикла:
        continue
    # Сообщение о делителе числа:
    print("Делится на", k)
# Сообщение о последнем делителе числа:
print("Делится на", number)
```

Результат выполнения программы может быть таким (жирным шрифтом выделено значение, которое вводит пользователь).

Результат выполнения программы (из листинга 2.3)

```
Введите число: 28
Делится на 1
Делится на 2
Делится на 4
Делится на 7
Делится на 14
Делится на 28
```

Вот еще один пример выполнения программы.

Результат выполнения программы (из листинга 2.3)

```
Введите число: 29
Делится на 1
Делится на 29
```

Пользователь вводит число, и в процессе выполнения программы отображаются все его делители. Поскольку каждое число делится на 1 и на себя, то в программе есть два «отдельных» сообщения: командой `print("Делится на",1)` в качестве делителя числа указано значение 1, а командой `print("Делится на", number)` делителем числа `number` указано само число `number`.

ПОДРОБНОСТИ

Напомним, что если функции `print()` передать несколько неименованных аргументов (как, например, в случае с командой `print("Делится на",1)` или `print("Делится на", number)`), то значения этих аргументов будут напечатаны в одну строку через пробел.

Но главный интерес представляет оператор цикла. В операторе проверяется условие `k<number//2` (делитель должен быть меньше половины от делимого числа). За каждый цикл командой `k=k+1` значение делителя увеличивается на единицу.

НА ЗАМЕТКУ

Поскольку начальное значение переменной `k` равно `1`, то первый делитель, для которого выполняется проверка, будет равен `2`.

Для проверки того, является ли `k` делителем числа, используем условный оператор в упрощенной форме (без `else`-блока). Истинность условия `number%k!=0` (остаток от деления числа `number` на число `k` не равен нулю) означает, что число `number` не делится на число `k` без остатка. Если так, то инструкцией `continue` выполнение текущего цикла завершается. Команда `print("Делится на", k)`, которая выводит сообщение о делителе числа, расположена после условного оператора. Команда выполняется, только если в условном операторе не выполнялась инструкция `continue`.

Строго говоря, необходимости использовать инструкцию `continue` в данном случае не было. Как иллюстрация к сказанному — программа в листинге 2.4, с помощью которой решается та же задача (по определению делителей числа), но только на этот раз инструкция `continue` не используется.

Листинг 2.4. Делители числа

```
number=int(input("Введите число: "))
k=1
while k<=number//2:
    if number%k==0:
        print("Делится на", k)
    k=k+1
print("Делится на", number)
```

Результат выполнения программы такой же, как в предыдущем случае. Программа простая, и хочется верить, что ее код особых комментариев не требует.

Оператор цикла `for`

> Да, это от души. Замечательно. Достойно восхищения.
>
> *Из к/ф «Формула любви»*

Помимо оператора цикла `while` в Python есть еще и *оператор цикла* `for`. Оператор цикла `for` удобно использовать в ситуациях, когда необходимо перебирать элементы некоторой последовательности или списка (для большей определенности далее будем говорить именно о списке). В наиболее простой форме шаблон вызова этого оператора следующий (жирным шрифтом выделены ключевые элементы шаблона):

```
for переменная in список:
    # команды
```

Начинается команда вызова оператора цикла с ключевого слова `for`, после которого указывается идентификатор (название переменной). Затем следует ключевое слово `in` и ссылка на список (последовательность значений в общем случае). После списка ставится двоеточие, а блок команд, относящийся к оператору цикла, выделяется отступом.

Выполняется оператор цикла `for` так. Переменная, указанная после ключевого слова `for`, последовательно принимает значения из списка,

указанного после ключевого слова `in`. При каждом значении переменной выполняются команды в теле оператора цикла. Как пример использования оператора цикла `for` рассмотрим задачу о переборе списка, состоящего из текстовых значений. Программа представлена в листинге 2.5.

Листинг 2.5. Перебор элементов списка

```
# Список с текстовыми элементами:
colors=["Синий","Желтый","Зеленый"]
# Отображение содержимого списка:
print(colors)
# Перебор элементов списка:
for s in colors:
    print(s,"->", len(s))
```

Результат выполнения программы следующий.

Результат выполнения программы (из листинга 2.5)

```
['Синий', 'Желтый', 'Зеленый']
Синий -> 5
Желтый -> 6
Зеленый -> 7
```

Пример простой и «прямолинейный». В программе создается список с текстовыми элементами (команда `colors=["Синий","Желтый","Зеленый"]`). Содержимое списка проверяем командой `print(colors)`. После этого запускается оператор цикла `for`, в котором переменная `s` принимает значения из списка `colors`. За каждый цикл командой `print(s,"->", len(s))` отображается собственно элемент списка и длина соответствующего текстового значения (вычисляется с помощью функции `len()`).

ПОДРОБНОСТИ

Переменная в операторе цикла `for` принимает значения элементов списка/последовательности. Значение переменной — это копия значения элемента из списка. Иногда это обстоятельство является важным, поскольку переменная не дает прямого доступа

к элементу списка, а лишь позволяет узнать значение элемента. Поэтому, например, через объявленную в операторе цикла переменную не удастся изменить элемент перебираемого списка.

Конечно, способы использования оператора цикла `for` не ограничиваются банальным перебором уже существующего списка. В листинге 2.6 представлена программа, в которой вычисляются числа из последовательности Фибоначчи.

НА ЗАМЕТКУ

В последовательности Фибоначчи первые два числа — единицы, а каждое следующее число вычисляется как сумма двух предыдущих. В итоге получаем последовательность из чисел 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 и так далее.

Рассматриваемая далее программа, кроме иллюстрации собственно использования оператора цикла `for`, содержит еще и пример *множественного присваивания*. Это одна из многих «визитных карточек» языка Python.

Листинг 2.6. Числа Фибоначчи

```
# Количество чисел в последовательности:
n=15
# Первые два числа:
a, b=1,1
# Отображение первых двух чисел:
print(a, b, end=" ")
# За каждый цикл вычисляется одно новое число:
for k in range(n-2):
    # Вычисление нового числа в последовательности:
    a, b=b, a+b
    # Отображение нового числа:
    print(b, end=" ")
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 2.6)

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

В переменную n записываем значение, которое определяет количество чисел в последовательности (в данном случае 15). Первые два числа в последовательности задаем в явном виде: переменные a и b получают значение 1. Для присваивания значений переменным мы использовали команду a, b=1,1. Это то, что называется множественным присваиванием. Переменные a и b, указанные слева от оператора присваивания, получают значения, указанные справа от оператора присваивания. В данном случае это две единицы, поэтому переменные a и b получают значение 1. Командой print(a, b, end=" ") отображаются значения переменных a и b. Автоматически между отображаемыми значениями добавляется пробел. Третий (именованный) аргумент end=" " функции print() означает, что после отображения значений переменных a и b следует добавить пробел (вместо перехода к новой строке). Таким образом, первые два числа в последовательности Фибоначчи отображаются сразу. Для вычисления прочих чисел из последовательности используем оператор цикла for. При выполнении оператора цикла переменная k пробегает значения из виртуальной последовательности, которая формируется инструкцией range(n-2). Вообще, инструкцией range(n-2) формируется итерируемый объект, определяющий последовательность целых чисел в диапазоне от 0 до n-3 включительно. Но в данном случае это не так важно. Важно то, что всего в последовательности n-2 значения, и, соответственно, такое количество циклов будет выполнено. За каждый цикл вычисляется одно новое число в последовательности Фибоначчи. Если учесть, что перед запуском оператора цикла два числа уже было вычислено (имеются в виду два начальных единичных значения), то всего получается n чисел, что нам и нужно.

В операторе цикла выполняются всего две команды. Сначала командой a, b=b, a+b вычисляется новое число в последовательности. Здесь мы снова встречаемся с множественным присваиванием. Команда a, b=b, a+b выполняется так: сначала вычисляются значения выражений, указанных через запятую в правой части от оператора присваивания, а затем эти значения последовательно присваиваются переменным, указанным через запятую в левой части выражения. Более конкретно, сначала вычисляется текущее значение переменной b, вычисляется значение выражения a+b (новое число в последовательности, равное сумме двух предыдущих), а потом переменная a получает первое вычисленное значение (текущее значение переменной b), а переменная b получает второе вычисленное значение (значение выражения a+b). Таким образом, в переменную b записывается новое вычисленное значение числа Фибоначчи, а то значение, которое до этого было записано в переменную b,

«перекочевывает» в переменную a. Поэтому после выполнения каждого цикла получается так, что в переменную b записано последнее (на данный момент) число из последовательности Фибоначчи, а предыдущее число из последовательности записано в переменную a.

После вычисления нового числа в последовательности и записи его в переменную b, командой `print(b, end=" ")` это число отображается в области вывода. Благодаря наличию аргумента `end=" "` в команде вызова функции `print()` после отображения числового значения переменной b переход к новой строке не выполняется, а вместо этого после числа добавляется пробел.

> **НА ЗАМЕТКУ**
>
> Таким образом, значение переменной k в операторе цикла не используется. Нам эта переменная нужна исключительно как счетчик циклов.

Иногда на практике используется форма оператора for с блоком else. Шаблон вызова оператора for в этом случае выглядит следующим образом (ключевые элементы шаблона выделены жирным шрифтом):

```
for переменная in список:
    # команды
else:
    # команды
```

Фактически после уже знакомой нам конструкции for-in со списком для перебора и блоком команд указывается ключевое слово else (заканчивается двоеточием) и еще один блок команд. Команды в else-блоке выполняются только один раз и после того, как перебраны все элементы из списка в for-in инструкции. Проще говоря, сначала выполняется оператор цикла так, как если бы блока else не было вовсе, а только затем выполняются команды в блоке else.

Здесь вполне уместно задаться вопросом: а какой вообще смысл в блоке else, если мы с таким же успехом могли бы просто разместить после оператора цикла обычный блок команд? Ответ такой: если оператор цикла прекращает свое выполнение вследствие вызова инструкции break, то команды в else-блоке не выполняются. Поясним это на примере, который представлен в листинге 2.7.

Листинг 2.7. Поиск букв в тексте

```
# Текст для поиска букв:
mytext=input("Введите текст для проверки: ")
# Буквы для поиска:
symbs=['а','у','я']
print("Ищем такие буквы:", symbs)
# Поиск букв:
for s in symbs:
    # Если буква найдена:
    if s in mytext:
        print("В тексте есть буква '"+s+"'")
        # Завершение оператора цикла:
        break
    # Если буквы нет:
    else:
        print("В тексте нет буквы '"+s+"'")
# Блок else оператора цикла:
else:
    print("Таких букв в тексте нет")
# Последнее сообщение программы:
print("Поиск завершен")
```

Программа предназначена для поиска букв в тексте. Текст, в котором предполагается искать буквы, вводится пользователем с клавиатуры. Соответствующее значение записывается в переменную `mytext` (команда `mytext=input("Введите текст для проверки: ")`). Буквы, которые мы собираемся искать в тексте, реализуем в виде списка, ссылку на который записываем в переменную `symbs` (команда `symbs=['а','у','я']`).

ПОДРОБНОСТИ

Мы указали элементами списка литералы `'а'`, `'у'` и `'я'` в одинарных кавычках. В Python как одинарные, так и двойные кавычки используются в текстовых литералах. Это означает, что литерал,

например 'a', является текстом. То есть в действительности мы реализуем буквы как текст, состоящий из одной буквы. Отметим также, что проверить тип значения можно с помощью встроенной функции type().

Командой print("Ищем такие буквы:", symbs) отображается содержимое списка с буквами, поиск которых выполняется в тексте. После этого запускается оператор цикла, в котором перебираются элементы из списка symbs. Тело оператора цикла состоит из условного оператора. Условие, которое проверяется в условном операторе, реализовано инструкцией s in mytext. Условие истинно, если в тексте mytext содержится элемент со значением, определяемым переменной s. Эта переменная, в свою очередь, последовательно принимает значения элементов из списка symbs.

ПОДРОБНОСТИ

При вычислении выражения вида значение in список выполняется проверка наличия в списке элемента с данным значением. Если такой элемент есть, то значение выражения равно True (истина). Если такого элемента в списке нет, то значение выражения равно False (ложь).

Если в тексте содержится искомая буква, то командой print("В тексте есть буква '"+s+"'") в окне вывода отображается сообщение соответствующего содержания. Затем инструкцией break завершается выполнение оператора цикла.

Если же буквы в тексте нет (при проверке условия оно оказалось ложным), то в else-блоке условного оператора командой print("В тексте нет буквы '"+s+"'") выводится сообщение о том, что данной буквы в тексте нет.

НА ЗАМЕТКУ

Как отмечалось выше, текст можно заключать как в одинарные, так и в двойные кавычки. В командах print("В тексте есть буква '"+s+"'") и print("В тексте нет буквы '"+s+"'") мы формируем текстовые литералы так, чтобы символ (записанный в переменную s) отображался в одинарных кавычках. Для этого мы включаем символ одинарной кавычки ' в текстовый литерал, который выделяется двойными кавычками.

У оператора цикла есть свой `else`-блок. В этом блоке всего одна команда `print("Таких букв в тексте нет")`. Эта команда выполняется только в том случае, если работа оператора цикла завершилась без вызова инструкции `break`. В нашем случае это означает, что `else`-блок будет выполняться, только если текст не содержит искомых букв. Если же в тексте есть хоть одна буква из списка `symbs`, то работа оператора цикла будет завершена инструкцией `break` и, соответственно, команда в `else`-блоке оператора цикла выполняться не будет.

НА ЗАМЕТКУ

Команда `print("Поиск завершен")` размещена после `else`-блока оператора цикла. Поэтому она выполняется в любом случае.

Как может выглядеть результат выполнения программы, показано ниже (жирным шрифтом выделено введенное пользователем значение).

Результат выполнения программы (из листинга 2.7)

```
Введите текст для проверки: любой текст
Ищем такие буквы: ['а', 'у', 'я']
В тексте нет буквы 'а'
В тексте нет буквы 'у'
В тексте нет буквы 'я'
Таких букв в тексте нет
Поиск завершен
```

В данном случае введенный текст не содержит искомых букв. Результат может быть и другим. Ниже показано, какие сообщения появляются при выполнении программы, если пользователь вводит текст (выделен жирным шрифтом) с искомыми буквами.

Результат выполнения программы (из листинга 2.7)

```
Введите текст для проверки: язык Python
Ищем такие буквы: ['а', 'у', 'я']
В тексте нет буквы 'а'
В тексте нет буквы 'у'
В тексте есть буква 'я'
Поиск завершен
```

Здесь из трех искомых букв (`'a'`, `'y'`, `'я'`) в тексте есть только буква `'я'`. Но поиск букв осуществляется строго в том порядке, в котором они размещены в списке `symbs`. Поэтому сначала появляются сообщения о том, что в тексте нет букв `'a'` и `'y'`, и только после этого сообщение о том, что текст содержит букву `'я'`. Поскольку теперь «срабатывает» инструкция `break`, то `else`-блок оператора цикла не выполняется.

> **НА ЗАМЕТКУ**
>
> Поиск букв выполняется до первого совпадения. Например, если бы в тексте не было буквы `'a'`, но были бы буквы `'y'` и `'я'`, то сначала выполнялся бы поиск буквы `'a'` (появляется сообщение, что такой буквы нет). Затем поиск буквы `'y'` (появляется сообщение о наличии в тексте этой буквы). До поиска буквы `'я'` дело вообще не дойдет.

Условный оператор `if`

— Теперь вот такое предложение. А что, если...
— Не стоит.

Из к/ф «Бриллиантовая рука»

В предыдущей главе мы уже познакомились с *условным оператором* `if` и использовали его не один раз. Теперь рассмотрим более сложные варианты использования этого оператора: во-первых, нас будет интересовать механизм реализации сложных условий для проверки в операторе, и, во-вторых, мы рассмотрим варианты применения вложенных условных операторов (включая специальную форму условного оператора с проверкой нескольких условий). Для начала рассмотрим небольшую программу, в которой решается алгебраическое уравнение $Ax = B$ относительно переменной x. Параметры A и B вводятся пользователем. С математической точки зрения это уравнение интереса не представляет, но в плане использования условного оператора дает нам большой простор для деятельности.

Интрига состоит в том, какие значения для параметров A и B введет пользователь. Мы выделим следующие ситуации.

- В принципе, пользователь может ввести нечисловые значения для параметров (и тогда вопрос о решении уравнения теряет смысл).
- Если значение параметра A отлично от нуля, то у уравнения есть решение $x = B / A$.

- Если $A = 0$, то возможны две ситуации. При нулевом значении параметра B уравнение $Ax = B$ превращается в тождество. Поэтому решением уравнения может быть любое число. А вот при ненулевом значении параметра B уравнение решений вообще не имеет.

Все эти ситуации мы постараемся отследить и обработать в программе. Программный код представлен в листинге 2.8.

Листинг 2.8. Решение линейного уравнения

```
# Вводятся параметры уравнения:
a, b=eval(input("Введите (через запятую) два числа: "))
# Проверка типа введенных параметров:
if (type(a)==int or type(a)==float) and (type(b)==int or type(b)==float):
    print("Уравнение "+str(a)+"x="+str(b))
    # Если первый параметр ненулевой:
    if a!=0:
        print("Решение x="+str(b/a))
    # Если первый параметр нулевой:
    else:
        # Если второй параметр нулевой:
        if b!=0:
            print("Решений нет!")
        # Если оба параметра нулевые:
        else:
            print("Решение — любое число!")
# Если параметры нечисловые:
else:
    print("Введены некорректные значения!")
    # Завершение выполнения программы:
    raise SystemExit(0)
print("Поиск решения завершен.")
```

Кроме собственно условного оператора, в программе есть еще несколько интересных моментов. Первый из них связан с вводом значений для параметров уравнения. Мы для этой цели использовали команду, `a,`

`b=eval(input("Введите (через запятую) два числа: "))` с множественным присваиванием. В левой части выражения через запятую указаны переменные `a` и `b`. В правой части указано выражение на основе функции `eval()`. Аргументом функции передается выражение `input("Введите (через запятую) два числа: ")`. Предполагается, что пользователь вводит два числа (целых или действительных). Числа разделяются запятой. Вся эта конструкция считывается как текст, который возвращается результатом функции `input()`. Выходит, что функция `eval()` получает текст в качестве аргумента. Выражение, содержащееся в тексте, вычисляется. Если это два числа, разделенные запятой, то результатом вычисления является последовательность из двух числовых (именно числовых!) значений. Эта последовательность присваивается последовательности из переменных `a` и `b`. В результате переменные получают значения из присваиваемой последовательности. Если пользователь введет нечисловое значение (одно или оба), то в процессе вычислений могут возникнуть проблемы. Поэтому после считывания значений для параметров уравнения мы проверяем их тип. Мы исходим из того, что пользователь может ввести как целое число (тип `int`), так и действительное число в формате с плавающей точкой (тип `float`).

ПОДРОБНОСТИ

В формате числа с плавающей точкой в качестве разделителя целой и дробной части используется точка (например, `23.5` или `12.0`). Также можно использовать экспоненциальную нотацию. В этом случае указывается мантисса и показатель экспоненты. Мантисса и показатель экспоненты разделяются символом `E` или `e`. Значение определяется так: мантисса умножается на 10 в соответствующей степени. Например, выражение `1.5e-2` означает число $1{,}5 \cdot 10^{-2}$, а выражение `0.25E3` соответствует числу $0{,}25 \cdot 10^{3} = 250$.

Узнать тип значения, на которое ссылается переменная, можно с помощью функции `type()`. Аргументом функции передается соответствующая переменная.

НА ЗАМЕТКУ

Напомним, что переменные в Python не имеют определенного типа. Но переменные не содержат значение, а ссылаются на него. Это примерно так, как если бы в переменную был записан адрес памяти, по которому хранится значение. И вот у этого значения тип есть. В процессе выполнения программы на разных этапах переменная

может ссылаться не просто на разные значения, а на значения разных типов. В любой момент мы можем проверить тип значения, на которое в данный момент ссылается переменная. Для этого аргументом функции `type()` передается переменная. Результатом является объект, определяющий тип значения, на которое ссылается переменная. Ключевые слова `int` и `float`, которыми мы определяли целочисленный тип и тип чисел с плавающей точкой, на самом деле являются ссылками на объекты (что такое объекты и как они реализуются в Python — рассказывается немного позже).

Также стоит заметить, что кроме целых чисел и чисел с плавающей точкой в Python есть встроенная поддержка для работы с комплексными числами (тип `complex`).

В условном операторе в качестве проверяемого условия указано выражение `(type(a)==int or type(a)==float) and (type(b)==int or type(b)==float)`. В этом выражении мы использовали логические операторы `or` (*логическое или*) и `and` (*логическое и*).

ПОДРОБНОСТИ

Если `X` и `Y` — некоторые логические выражения (возможные значения — `True` или `False`), то значение выражения `X or Y` равно `True`, если хотя бы один из операндов `X` или `Y` равен `True`. Значение выражения `X or Y` равно `False`, если оба операнда `X` и `Y` равны `False`. Проще говоря, «сложное» условие `X or Y` на основе оператора `or` истинно, если истинно хотя бы одно из условий `X` или `Y`.

Значением выражения вида `X and Y` на основе оператора `and` является логическое значение `True`, только если значения операндов `X` и `Y` равны `True`. Если хотя бы один из операндов `X` или `Y` равен `False`, то значение выражения `X and Y` равно `False` («сложное» выражение истинно, если истинны оба условия `X` и `Y`).

Кроме операторов `or` и `and`, которые являются бинарными (они используются с двумя операндами), нередко используется логический оператор `not`. Это унарный (используется с одним операндом) оператор логического отрицания. Если операнд `X` имеет значение `True`, то значением `not X` является значение `False`. Если значение операнда `X` равно `False`, то значение выражения `not X` равно `True`.

Значение выражения `type(a)==int` равно `True`, если тип значения, на который ссылается переменная `a`, равен `int` (то есть значение — целое число). Значение выражения `type(a)==float` равно `True`, если переменная `a` ссылается на значение типа `float` (число в формате с плавающей точкой). Выражение `type(a)==int or type(a)==float`

истинно, если переменная a ссылается на значение типа int или float. Аналогично и для переменной b. Наконец, выражение (type(a)==int or type(a)==float) and (type(b)==int or type(b)==float) истинно, если переменная a ссылается на значение типа int или float, и при этом переменная b тоже ссылается на значение типа int или float.

Если пользователь ввел два числовых значения, то командой print ("Уравнение "+str(a)+"x="+str(b)) отображается решаемое уравнение.

ПОДРОБНОСТИ

При формировании текстового литерала (который передается аргументом функции print()) использована функция str(), с помощью которой числовые значения преобразуются к текстовому формату.

После этого в игру вступает еще один условный оператор. В нем проверяется условие a!=0, истинное в том случае, если значение переменной a не равно 0. Если так, то командой print("Решение x="+str(b/a)) отображается решение уравнения.

ПОДРОБНОСТИ

В Python есть различные операторы для сравнения значений — операторы сравнения. По своей сути они аналогичны соответствующим математическим операторам, и, в принципе, их назначение понятно просто из названия. Вот эти операторы: < (меньше), > (больше), <= (меньше или равно), >= (больше или равно), == (равно),!= (не равно). Результатом сравнения является значение True или False, в зависимости от того, выполняется или нет соответствующее соотношение между сравниваемыми значениями.

Кроме перечисленных выше операторов сравнения есть еще два: is (оператор проверки идентичности) и is not (оператор проверки неидентичности). Они используются в том случае, когда нужно проверить, ссылаются ли две переменные не просто на одинаковые значения, а на одно и то же значение. Обычно такую операцию выполняют, когда сравнивают объекты.

Если условие a!=0 ложное (означает, что значение переменной a равно нулю), то запускается еще один условный оператор, в котором проверяется условие b!=0. Оно истинно, если значение переменной b не равно нулю. Это случай, когда у уравнения решений нет (выполняется команда print("Решений нет!")). Может статься, что условие b!=0

ложно. Получается, что обе переменные a и b имеют нулевые значения, и решение — любое число (выполняется команда `print("Решение — любое число!")`).

Все описанное происходит при условии, что пользователь ввел два числовых значения. Если это не так, то выполняются команды в `else`-блоке самого первого, внешнего условного оператора. В нем есть команда `print("Введены некорректные значения!")`, после чего выполняется инструкция `raise SystemExit(0)`. Это команда, которой завершается выполнение программы. Речь идет о том, что выполнение данной команды приводит к завершению выполнения всего кода, в отличие от случаев, когда мы использовали инструкцию `break` для завершения выполнения оператора цикла.

ПОДРОБНОСТИ

Инструкция `raise` используется для генерирования исключений — специальных сообщений, которые могут обрабатываться в программе. Исключения бывают разными, и у них есть тип (или класс). В известном смысле тип/класс исключения похож на тип данных. При генерировании исключения после инструкции `raise` указывается тип исключения, а в круглых скобках — параметры, которые используются при создании объекта исключения. При генерировании исключения класса `SystemExit` выполнение программы прекращается. Нулевое значение, указанное в круглых скобках после названия класса исключения, означает, что выполнение программы завершается в штатном режиме (без ошибок).

В Python существуют и иные способы «досрочно» завершить выполнение программы. Так, можно вызвать функцию `exit()` (с нулевым аргументом) из модуля `site` (обычно, но не всегда, этот модуль автоматически загружается средой исполнения). Более «надежной» является одноименная функция из модуля `sys`. Для подключения модуля используется инструкция `import sys`, а сама функция вызывается командой `sys.exit(0)`. Но в любом случае все сводится к генерированию исключения класса `SystemExit`.

Обработка исключений кратко обсуждается далее в этой главе. Более подробно исключения будут рассмотрены после того, как мы познакомимся с принципами реализации классов, объектов и механизмом наследования.

Таким образом, последняя команда `print("Поиск решения завершен.")` в программе выполняется, только если пользователь вводит два числовых значения.

Каким может быть результат выполнения программы, показано ниже (здесь и далее жирным шрифтом выделено то, что вводит пользователь).

Результат выполнения программы (из листинга 2.8)

```
Введите (через запятую) два числа: 2.5,3.6
Уравнение 2.5x=3.6
Решение x=1.44
Поиск решения завершен.
```

В данном случае пользователь ввел два действительных числа (не равных нулю). Ниже представлен результат выполнения программы в случае, если пользователь вводит два отличных от нуля целочисленных значения.

Результат выполнения программы (из листинга 2.8)

```
Введите (через запятую) два числа: 2,3
Уравнение 2x=3
Решение x=1.5
Поиск решения завершен.
```

А теперь случай, когда первый коэффициент нулевой, а второй — нет.

Результат выполнения программы (из листинга 2.8)

```
Введите (через запятую) два числа: 0,3.5
Уравнение 0x=3.5
Решений нет!
Поиск решения завершен.
```

Ситуация, когда пользователь вводит два нулевых значения.

Результат выполнения программы (из листинга 2.8)

```
Введите (через запятую) два числа: 0,0
Уравнение 0x=0
Решение — любое число!
Поиск решения завершен.
```

Еще один характерный случай: пользователь в качестве значения для первого параметра ввел текстовое значение.

Результат выполнения программы (из листинга 2.8)

```
Введите (через запятую) два числа: "2",3
Введены некорректные значения!
```

Как видим, в этом случае программа выводит сообщение о том, что пользователь ввел некорректные значения для параметров уравнения.

Рассмотренная выше конструкция из вложенных условных операторов является вполне законной, но для подобных случаев в Python есть специальная версия условного оператора. Шаблон этой версии условного оператора имеет следующий вид (жирным шрифтом выделены ключевые элементы шаблона):

```
if условие:
    # команды
elif условие:
    # команды
elif условие:
    # команды
…
elif условие:
    # команды
else:
    # команды
```

В общем-то запомнить эту конструкцию несложно. Сначала идет инструкция `if` с условие и двоеточием, затем блок команд. После этого следует несколько `elif`-блоков: ключевое слово `elif`, условие, двоеточие и блок команд. Завершается все это `else`-блоком: ключевое слово `else`, двоеточие и блок команд. Выполняется оператор так.

- Сначала проверяется условие в `if`-блоке. Если оно истинно, выполняются команды этого блока, и на этом выполнение условного оператора заканчивается.

- Если условие в `if`-блоке ложное, то проверяется условие в первом `elif`-блоке. Если условие истинное, то выполняются команды этого блока и завершается выполнение условного оператора.
- Если условие в первом `elif`-блоке ложное, то проверяется условие во втором `elif`-блоке. И так далее.
- Если условия в `if`-блоке и во всех `elif`-блоках ложные, то выполняются команды из блока `else`.

НА ЗАМЕТКУ

Блок `else` не является обязательным. Если `else`-блок отсутствует, то условный оператор выполняется так же, с поправкой на то, что если все условия (в `if`-блоке и `elif`-блоках) окажутся ложными, то никакие команды выполняться не будут.

Как иллюстрацию к использованию описанной конструкции рассмотрим предыдущий пример, но в новой реализации. Программный код представлен в листинге 2.9.

Листинг 2.9. Еще один способ решить уравнение

```
# Вводятся параметры уравнения:
a, b=eval(input("Введите (через запятую) два числа: "))
# Если параметры некорректные:
if (type(a)!=int and type(a)!=float) or (type(b)!=int and type(b)!=float):
    print("Введены некорректные значения!")
    # Завершение выполнения программы:
    raise SystemExit(0)
# Если первый параметр ненулевой:
elif a!=0:
    txt="Решение x="+str(b/a)
# Если второй параметр ненулевой (при нулевом первом):
elif b!=0:
    txt="Решений нет!"
# Если оба параметра нулевые:
else:
    txt="Решение — любое число!"
```

```
# Вид уравнения:
print("Уравнение "+str(a)+"x="+str(b))
# Результат поиска корня:
print(txt)
print("Поиск решения завершен.")
```

Результат выполнения программы такой же, как и в предыдущем случае. Но алгоритм получения этого результата немного изменился. Параметры уравнения вводятся, как и ранее, — два числовых значения, разделенные запятой. После считывания значений и записи их в переменные `a` и `b` в условном операторе проверяется условие `(type(a)!=int and type(a)!=float) or (type(b)!=int and type(b)!=float)`. Условие истинно, если хотя бы одно из значений, на которые ссылаются переменные `a` и `b`, не является числовым (то есть если пользователь ввел некорректные значения). В этом случае выводится сообщение о некорректных значениях параметров и завершается выполнение программы.

ПОДРОБНОСТИ

Условие `type(a)!=int` истинно, если переменная `a` ссылается на значение типа, отличного от `int`. Условие `type(a)!=float` истинно, если переменная `a` ссылается на значение типа, отличного от `float`. Условие `type(a)!=int and type(a)!=float` будет истинным, если значение, на которое ссылается переменная `a`, не относится ни к типу `int`, ни к типу `float`. Условие `type(b)!=int and type(b)!=float` истинно, если переменная `b` ссылается на нечисловое значение. Поэтому все «большое» условие является истинным, если переменная `a` или переменная `b` ссылаются не на числа.

Если условие в `if`-блоке оказалось ложным, то проверяется условие `a!=0` в первом `elif`-блоке. При истинном условии командой `txt="Решение x="+str(b/a)` переменной `txt` присваивается текстовое значение (содержащее информацию о корне уравнения). Если условие `a!=0` ложное, то проверяется условие `b!=0` в следующем `elif`-блоке. Если так, то значение переменной `txt` определяется командой `txt="Решений нет!"`. При ложном условии `b!=0` выполняется команда `txt="Решение — любое число!"` в `else`-блоке.

НА ЗАМЕТКУ

Таким образом, до выполнения `else`-блока дело доходит, только если ложными окажутся все три условия (в `if`-блоке и `elif`-блоках). Методом исключения получается, что команды (точнее, она там одна) из `else`-блока будут выполняться, только если пользователь ввел числовые значения, причем оба они нулевые.

После выполнения условного оператора командой `print("Уравнение "+str(a)+"x="+str(b))` отображается выражение для решаемого уравнения. Результат поиска решения отображается с помощью команды `print(txt)`, после чего команда `print("Поиск решения завершен.")` выводит сообщение о завершении работы программы.

НА ЗАМЕТКУ

Таким образом, сначала, в зависимости от значений параметров уравнения, формируется текст и записывается в переменную `txt`, а после завершения выполнения условного оператора этот текст отображается в окне вывода.

Еще один небольшой пример использования условного оператора с последовательной проверкой нескольких условий представлен в листинге 2.10.

Листинг 2.10. Идентификация числа

```
# Текстовая переменная:
res="Это число "
# Вводится текст:
txt=input("Введите название числа: ")
# Преобразование в нижний регистр:
txt=txt.lower()
# Идентификация числа:
if txt=="один" or txt=="единица":
    res+="1"
elif txt=="два" or txt=="двойка":
    res+="2"
elif txt=="три" or txt=="тройка":
```

```
    res+="3"
else:
    res+="не идентифицировано"
# Результат идентификации:
print(res)
```

Программа очень простая: пользователь вводит название для целого числа, а программа определяет это число. Правда, определять она умеет только числа `1`, `2` и `3`. Но в данном случае важен сам принцип. В общих чертах выполняется программа так: считывается текст, введенный пользователем (название числа), по этому названию определяется число, после чего отображается сообщение со значением числа. Для реализации этого алгоритма в программе командой `res="Это число "` создается текстовая переменная `res` (с начальным значением, которое затем будет «уточнено»). Мы планируем использовать данную переменную при отображении сообщения с результатом идентификации числа. Затем выполняется команда `txt=input("Введите название числа: ")`. В результате в переменную `txt` записывается текст, который вводит пользователь. Мы исходим из того, что пользователь введет название числа. При этом он может использовать при наборе названия как большие (прописные), так и маленькие (строчные) буквы. Чтобы выполнить сравнение текстовых значений без учета регистра символов (то есть строчный и прописной символ интерпретируется как один и тот же), используем небольшую хитрость: командой `txt=txt.lower()` все буквы из тестового значения переменной `txt` преобразуем в нижний регистр (то есть все буквы становятся маленькими).

ПОДРОБНОСТИ

Инструкцией `txt.lower()` из текстового объекта `txt` вызывается метод `lower()`. Методы очень напоминают функции. Но если функция вызывается «сама по себе», то для вызова метода нужен объект. В известном смысле объект играет роль аргумента (одного из аргументов) метода. И пока что к объекту, из которого вызывается метод, мы так и будем относиться — как к аргументу.

Мы уже знаем, что в Python переменные не содержат значение, а ссылаются на него. Каждый раз, когда речь идет о значении переменной, имеется в виду значение, на которое переменная ссылается. Что касается текстовой переменной `txt`, то в действительности она ссылается не просто на текст, а на объект, который содержит

текст. Можно сказать и иначе: текст реализуется с помощью объекта. Именно из этого объекта вызывается метод `lower()`.

При вызове методов используется «точечный» синтаксис: указывается имя объекта и, через точку, имя вызываемого метода (с круглыми скобками, в которых могут указываться аргументы). Это общее правило.

Что касается собственно метода `lower()`, то его «работа» состоит в том, что он «берет» текст из объекта, из которого вызвали метод, и возвращает в качестве результата такой же текст, в котором все буквы маленькие. Проще говоря, результатом выражения `txt.lower()` является текст как в переменной `txt`, но только маленькими буквами. При этом текст в переменной `txt` не меняется. Чтобы изменить текст в переменной `txt`, мы используем команду `txt=txt.lower()`. Команда выполняется так: считывается текст из переменой `txt`, буквы текста преобразуются в нижний регистр, и полученный результат записывается в переменную `txt`.

Отметим также, что для перевода букв в верхний регистр используется метод `upper()`. Более детально методы и объекты обсудим позже.

Основную работу выполняет условный оператор. Первым проверяется условие `txt=="один" or txt=="единица"` в `if`-блоке. Условие истинно, если пользователь ввел текст `"один"` или `"единица"`, причем регистр букв значения не имеет. Если так, то число идентифицируется как `1`, и командой `res+="1"` к текущему значению текстовой переменной дописывается текст `"1"`.

ПОДРОБНОСТИ

В Python есть сокращенные операции присваивания. Например, если нужно использовать команду вида `x=x+y`, то ее можно записать более компактно как `x+=y`. Это правило относится не только к оператору сложения. Например, если обозначить символом ☺ какой-то бинарный арифметический или побитовый (описываются немного позже оператор), то команду `x=x ☺ y` можно представить как `x ☺= y`. Поэтому команда `res+="1"` является аналогом команды `res=res+"1"`.

НА ЗАМЕТКУ

Таким образом, для обозначения единицы можно использовать два слова (без учета регистра букв): `"один"` или `"единица"`. Аналогично и для двух других чисел — каждое из них определяется двумя словами.

Если первое условие оказалось ложным, проверяется условие `txt=="два" or txt=="двойка"` в первом `elif`-блоке. При истинности условия выполняется команда `res+="2"` (к текущему значению переменной `res` дописывается текст `"2"`). Во втором `elif`-блоке проверяется условие `txt=="три" or txt=="тройка"`. Понятно, что для этого необходимо, чтобы в предыдущем `elif`-блоке условие оказалось ложным. Во втором `elif`-блоке при истинном условии выполняется команда `res+="3"`. Наконец, если все условия оказались ложными, в `else`-блоке выполняется команда `res+="не идентифицировано"`.

По завершении условного оператора командой `print(res)` отображается значение переменной `res`.

Ниже представлен результат выполнения программы, когда число идентифицируется как `1` (здесь и далее жирным шрифтом выделено значение, которое вводит пользователь).

Результат выполнения программы (из листинга 2.10)

```
Введите название числа: Единица
Это число 1
```

Так может выглядеть результат выполнения программы, если число идентифицируется как `2`:

Результат выполнения программы (из листинга 2.10)

```
Введите название числа: Два
Это число 2
```

Здесь число идентифицируется как `3`:

Результат выполнения программы (из листинга 2.10)

```
Введите название числа: ТРОЙКА
Это число 3
```

Наконец, ниже представлен случай, когда программа «не узнает» число.

Результат выполнения программы (из листинга 2.10)

```
Введите название числа: Пять
Это число не идентифицировано
```

Еще раз стоит подчеркнуть, что регистр букв при вводе пользователем названия числа значения не имеет.

Тернарный оператор

— Один крокодил. Два крокодила. Три крокодила...
— Бредит, бедняга. Похоже на тропическую лихорадку.

Из м/ф «Приключения капитана Врунгеля»

Оператор — это некоторый символ (или символы), обозначающий определенную операцию. У оператора есть *операнд* или операнды — те значения, с которыми выполняется операция. Обычно операнды используются с одним операндом или двумя операндами. Такие операторы называются, соответственно, *унарными* и *бинарными*. Но есть в Python один оператор, у которого три операнда. Это *тернарный* оператор. В некотором смысле тернарный оператор является «мини-реализацией» условного оператора. Синтаксис тернарного оператора следующий (жирным шрифтом выделены ключевые элементы шаблона тернарного оператора):

```
значение if условие else значение
```

Используются два ключевых слова: `if` и `else` (как и в условном операторе). Но по сравнению с условным оператором в данном случае есть важная особенность: тернарный оператор возвращает результат. Другими словами, вся описанная выше конструкция имеет значение, и это значение, например, может быть присвоено переменной. Значение на основе тернарного оператора вычисляется так. Сначала проверяется `условие`, указанное после ключевого слова `if`. Если `условие` истинно, то результатом всего выражения является `значение`, указанное перед ключевым словом `if`. Если `условие` окажется ложным, то результатом всего выражения будет `значение`, указанное после ключевого слова `else`.

> **НА ЗАМЕТКУ**
>
> Таким образом, операндами тернарного оператора являются условие, указанное после ключевого слова `if`, а также значения, указанные перед ключевым словом `if` и после ключевого слова `else`.

Как пример использования тернарного оператора рассмотрим программу, в которой введенное пользователем число проверяется на предмет четности/нечетности.

> **НА ЗАМЕТКУ**
>
> В предыдущей главе мы рассматривали аналогичную программу, но в ней использовался условный оператор. Здесь мы вместо условного оператора воспользуемся тернарным оператором.

Рассмотрим программный код в листинге 2.11.

Листинг 2.11. Проверка числа на четность/нечетность

```
# Вводится число:
num=int(input("Введите целое число: "))
# Использование тернарного оператора:
res="четное" if num%2==0 else "нечетное"
# Сообщение:
print("Это "+res+" число")
```

Результат выполнения программы может быть таким (жирным шрифтом выделено введенное пользователем значение).

Результат выполнения программы (из листинга 2.11)

```
Введите целое число: 123
Это нечетное число
```

Или таким:

Результат выполнения программы (из листинга 2.11)

```
Введите целое число: 124
Это четное число
```

Программа сначала считывает введенное пользователем целое число. Число записывается в переменную `num`.

НА ЗАМЕТКУ

Число считывается в текстовом формате. Для преобразования текстового представления целого числа собственно в число используется функция `int()`.

Затем переменной `res` присваивается текстовое значение. Мы использовали команду `res="четное" if num%2==0 else "нечетное"`, основу которой составляет тернарный оператор. Рассмотрим выражение `"четное" if num%2==0 else "нечетное"` более детально. Его значение вычисляется следующим образом. Сначала проверяется условие `num%2==0`. Это условие истинно, если остаток от деления значения переменной `num` на `2` равен нулю (то есть введенное пользователем число делится на два). В таком случае значением выражения будет текст `"четное"`. Если условие `num%2==0` ложное (значение переменной `num` на `2` не делится — то есть число нечетное), то значением выражения будет текст `"нечетное"`. Таким образом, если значение переменной `num` четное, то переменная `res` получает значение `"четное"`. Если значение переменной `num` нечетное, то переменная `res` получает значение `"нечетное"`. Командой `print("Это "+res+" число")` отображается сообщение о результате тестирования числа.

В принципе, тернарный оператор представляет собой достаточно гибкую и эффективную конструкцию. Скажем, в зависимости от истинности/ложности условия он может возвращать не просто разные значения, а значения разных типов (просто констатация факта — целесообразность подобного способа программирования обсуждать не будем). Существуют и другие «трюки», которые делают код простым и красивым. Небольшая иллюстрация к сказанному представлена в листинге 2.12.

Листинг 2.12. Значения разных типов и тернарный оператор

```
# Вводится выражение:
val=eval(input("Введите выражение: "))
# Используется тернарный оператор:
a, b=(val[0], val[-1]) if type(val)==str else (val, type(val))
# Значения переменных:
print(a)
print(b)
```

В этой программе всего несколько строк кода, но интерес представляет одна (с тернарным оператором). Все происходит следующим образом. Пользователь вводит какое-либо выражение (текст, число, список), это выражение считывается и вычисляется (для чего использована функция `eval()`), а результат записывается в переменную `val`. Затем выполняется команда `a, b=(val[0], val[-1]) if type(val)==str else (val, type(val))`. Команда не самая тривиальная, и мы ее проанализируем «по частям». Этой командой значение присваивается сразу двум переменным (`a` и `b`). В правой части размещен тернарный оператор. В тернарном операторе проверяется условие `type(val)==str`. Оно истинно, если пользователь ввел текстовое выражение. В этом случае тернарный оператор возвращает в качестве значения конструкцию `(val[0], val[-1])`. Фактически это последовательность из двух элементов (первая и последняя буквы в текстовом выражении, на которое ссылается переменная `val`). Получается, что в правой части — последовательность `(val[0], val[-1])`, а в левой части — переменные `a` и `b`. В итоге переменная `a` в качестве значения получает `val[0]`, а переменной `b` присваивается значение `val[-1]`. Если условие `type(val)==str` ложное, то тернарный оператор возвращает в качестве значения последовательность `(val, type(val))`. Поэтому переменной `b` присваивается значением переменная `val`, а переменной `b` присваивается выражение `type(val)`.

ПОДРОБНОСТИ

Несколько пояснений. Во-первых, внешние круглые скобки в выражениях `(val[0], val[-1])` и `(val, type(val))` мы использовали для группировки выражения, чтобы оно при интерпретации обрабатывалось как операнд тернарного оператора. Мы такую конструкцию называли последовательностью, хотя корректнее было бы назвать это кортежем (тип `tuple`). Кортеж в Python — это практически то же, что и список (то есть упорядоченный набор элементов). Но в отличие от списка кортеж после создания изменить нельзя. Напомним, что список в Python можно создать, если в квадратных скобках перечислить через запятую элементы списка. А если вместо квадратных скобок использовать круглые, то получим кортеж. Более того, если скобки совсем не использовать, а просто перечислить элементы через запятую (как мы это делали ранее при множественном присваивании), то тоже получим кортеж. Вывод простой: то, что мы называли и называем последовательностью элементов, на самом деле является кортежем. Но для понимания сути происходящего в нашем

случае это не так важно. Кортеж вместе с другими множественными типами данных мы рассмотрим в одной из следующих глав книги.

Напомним, что доступ к элементам списка, а также текста можно получить с помощью индекса. Индексация начинается с нуля, поэтому первый элемент в списке (первая буква в тексте) имеет нулевой индекс (как в выражении `val[0]`). Но индекс может быть и отрицательным. В таком случае элементы отсчитываются с конца списка/текста. Индекс `-1` соответствует последнему элементу, индекс `-2` соответствует предпоследнему элементу и так далее. Выражение `val[-1]` означает последний символ в тексте, на который ссылается переменная `val`.

Значением выражения вида `type(val)` является ссылка на объект, содержащий информацию о типе объекта, на который ссылается переменная `val`. Если попытаться «распечатать» такой объект, передав выражение `type(val)` аргументом функции `print()`, то появится сообщение вида `<class 'тип'>`, где `тип` является идентификатором, определяющим тип данных (например, `int` для целых чисел, `float` для действительных чисел, `list` для списков, `tuble` для кортежей, `str` для текста и так далее).

Результат выполнения программы в случае, если пользователь вводит число (здесь и далее жирным шрифтом выделено введенное пользователем значение), ниже.

Результат выполнения программы (из листинга 2.12)

```
Введите выражение: 123
123
<class 'int'>
```

Далее представлен результат выполнения программы для случая, когда пользователь вводит список.

Результат выполнения программы (из листинга 2.12)

```
Введите выражение: [1,2,3]
[1, 2, 3]
<class 'list'>
```

Если пользователь вводит текст, то значениями переменных будут первая и последняя буквы в тексте.

Результат выполнения программы (из листинга 2.12)

```
Введите выражение: "Python"
P
n
```

НА ЗАМЕТКУ

Есть важный момент, связанный с вводом текста. Если мы считываем введенное пользователем значение просто с помощью функции `input()`, то что бы ни ввел пользователь, это считывается как текст. Образно выражаясь, то, что ввел пользователь, заключается в кавычки, и это становится текстовым результатом, возвращаемым функцией `input()`. Если мы, например, введем слово `Python`, то функция `input()` вернет в качестве результата текст `"Python"`. А вот если мы команду вызова функции `input()` передаем аргументом функции `eval()`, то выполняется попытка вычислить значение выражения, «спрятанного» в тексте. В нашем примере это текст `"Python"`, а вычисляться будет выражение `Python`, которое не имеет смысла, и поэтому возникнет ошибка. Чтобы в результате вычислений получить текст, необходимо вводить не `Python`, а `"Python"` (то есть при вводе текста его следует заключить в кавычки). Тогда в функцию `eval()` будет передано текстовое значение `'"Python"'` (текст `"Python"` в кавычках). Соответственно, вычисляться будет выражение `"Python"`. Этот текст. При «вычислении» текста получаем тот же текст. Еще раз подчеркнем, что все это — следствие использования функции `eval()`.

Существуют и другие способы эффективного (и эффектного) применения тернарного оператора. Тем не менее следует понимать, что тернарный оператор все же не может рассматриваться как полноценная замена условному оператору.

Обработка исключительных ситуаций

Рапортуйте мне, откуда и куда идет «Беда».

Из м/ф «Приключения капитана Врунгеля»

Чтобы понять суть следующего механизма, начнем с простой и очень конкретной ситуации. Допустим, мы пишем программу, в которой пользователь должен ввести целое число. Мы считываем это число в текстовом

формате и хотим преобразовать в числовой формат. Для этого используем функцию `int()`. Если пользователь ввел целое число, то преобразование будет выполнено. Но если пользователь введет нечто, отличное от целого числа, то возникнет ошибка. То есть в данном случае мы потенциально можем столкнуться с проблемой, причем возникновение или не возникновение этой проблемы не связано с тем, как мы напишем код. Мы всецело в руках пользователя. Только от его действий зависит, будет ли корректно выполняться код.

> **НА ЗАМЕТКУ**
>
> Конечно, мы можем воспользоваться функцией `eval()` для вычисления значения выражения, которое вводит пользователь, а затем проверить тип соответствующего значения с помощью функции `type()`. Однако все равно нет гарантии, что пользователь корректно введет выражение. Так что описанная выше проблема является принципиальной.

Если при выполнении программы произошла ошибка, то говорят, что возникла *исключительная ситуация*. Хорошая новость состоит в том, что в языке Python есть механизм, который позволяет перехватывать и обрабатывать исключительные ситуации. Другими словами, можно это выразить так: имеется возможность так организовать программный код, что даже в случае возникновения ошибки программа продолжит работу в штатном режиме.

> **НА ЗАМЕТКУ**
>
> Здесь мы рассмотрим наиболее простые приемы, связанные с использованием системы обработки исключительных ситуаций. Более детальный разбор темы мы проведем после того, как познакомимся с классами и объектами.

Обработка исключений в языке Python базируется на использовании конструкции `try-except`. Идея очень простая и красивая: программный код, при выполнении которого может возникнуть ошибка, помещается в специальный блок. Этот блок помечается ключевым словом `try`. Код, размещенный в `try`-блоке, будем называть *контролируемым* кодом. После `try`-блока размещается блок, помеченный ключевым словом `except`. Это блок *обработки исключения*. Общий шаблон для конструкции `try-except`, таким образом, выглядит достаточно просто (жирным шрифтом выделены ключевые элементы шаблона).

```
try:
    # команды
except:
    # команды
```

Если подобная конструкция использована в программном коде, то происходит все следующим образом. Выполняется контролируемый код в `try`-блоке. Если при выполнении этого кода ошибки не возникают, то программный код в `except`-блоке игнорируется. Если же при выполнении `try`-блока возникла ошибка, то дальнейшее выполнение команд в этом блоке прекращается и начинают выполняться команды в `except`-блоке. После того как команды в `except`-блоке выполнены, работа программы продолжается в обычном режиме — то есть выполняются команды после конструкции `try-except`.

НА ЗАМЕТКУ

В некотором смысле конструкция `try-except` напоминает условный оператор, в котором роль `else`-блока играет блок `except`, а вместо проверки условия используется факт наличия/отсутствия ошибки. Хотя, конечно, аналогия довольно грубая.

Фактически `except`-блок содержит код, который выполняется, только если при выполнении `try`-блока возникла ошибка. Стоит также заметить, что в случае возникновения ошибки в `try`-блоке его выполнение прекращается на той команде, которая стала причиной ошибки. Команды, которые размещены в `try`-блоке после нее, не выполняются. В листинге 2.13 представлена программа, в которой пользователю предлагается ввести целое число, и затем это число отображается в окне вывода. Для обработки возможных ошибок, связанных с некорректным вводом значения, мы использовали конструкцию `try-except`.

Листинг 2.13. Знакомство с обработкой исключений

```
print("Обработка исключений")
# Контролируемый код:
try:
    num=int(input("Введите целое число: "))
    print("Вы ввели число "+str(num))
# Обработка исключения:
```

```
except:
    print("Нужно было ввести целое число!")
print("Спасибо за сотрудничество!")
```

Результат выполнения программы может быть таким (жирным шрифтом выделено значение, введенное пользователем).

Результат выполнения программы (из листинга 2.13)

```
Обработка исключений
Введите целое число: 123
Вы ввели число 123
Спасибо за сотрудничество!
```

Здесь было введено целочисленное значение, и при выполнении программы ошибка не возникла. Ниже показана ситуация, когда вместо целого числа вводится действительное число с плавающей точкой.

Результат выполнения программы (из листинга 2.13)

```
Обработка исключений
Введите целое число: 123.0
Нужно было ввести целое число!
Спасибо за сотрудничество!
```

Видим, что результат программы несколько иной, но важно другое: программа выполняется в «штатном» режиме. Теперь проанализируем программный код.

Первой выполняется команда `print("Обработка исключений")`, от которой сюрпризов ожидать не приходится. Далее очередь доходит до `try`-блока с двумя командами. Командой `num=int(input("Введите целое число: "))` считывается введенное пользователем значение и выполняется попытка преобразовать его в целое число. Именно на этом этапе может возникнуть ошибка. Допустим, все прошло нормально, и число считано и записано в переменную `num`. Тогда выполняется следующая команда `print("Вы ввели число "+str(num))` в `try`-блоке. Команды (она там одна) в блоке `except` не выполняются, а сразу выполняется команда `print("Спасибо за сотрудничество!")` после конструкции `try-except`.

Теперь другая ситуация: при выполнении команды `num=int(input("Введите целое число: "))` в `try`-блоке возникает ошибка. В таком случае выполнение `try`-блока прекращается (и команда `print("Вы ввели число "+str(num))` в `try`-блоке не выполняется), а начинают выполняться команды в `except`-блоке (будет выполнена команда `print("Нужно было ввести целое число!")`). После завершения работы конструкции `try-catch` выполняется команда `print("Спасибо за сотрудничество!")`.

Выше мы рассмотрели наиболее простой и лаконичный способ использования конструкции `try-except`. В действительности тема перехвата и обработки исключений (а также их искусственного генерирования, как это ни странно может прозвучать) очень многогранна, а соответствующие механизмы гибки и продуктивны. Мы их обсудим, но позже (поскольку для понимания сути механизмов необходимо иметь хотя бы общее представление о классах, объектах и наследовании). Но один прием рассмотрим прямо сейчас. Связан он с тем, что ошибки разных типов можно обрабатывать по-разному.

Дело в том, что ошибки бывают разными: у каждой ошибки/исключения есть тип (или класс). В конструкции `try-except` можно использовать несколько блоков `except`, каждый из которых будет обрабатывать ошибки определенного типа. Шаблон `try-except` конструкции в этом случае такой:

```
try:
    # команды
except Тип_ошибки:
    # команды
except Тип_ошибки:
    # команды
…
except Тип_ошибки:
    # команды
```

Существует довольно много типов ошибок (классов исключений). Например, ошибка, связанная с преобразованием введенного пользователем значения к целочисленному формату, относится к классу `ValueError`. Если в программе предпринимается попытка обратиться к несуществующей переменной (переменной, которой предварительно

не присвоено значение), то генерируется исключение класса `NameError`. Ошибка `TypeError` возникает при несовместимости типов, попытке выполнить некорректную операцию (операцию, которая недопустима для данных определенного типа). При попытке деления на ноль генерируется исключение класса `ZeroDivisionError`. Если неправильно указать индекс элемента списка, будет сгенерирована ошибка класса `IndexError`. Ошибка класса `SyntaxError` возникает при попытке вычислить некорректное (в плане синтаксиса) выражение. Как вся эта информация может использоваться на практике, иллюстрирует программа, представленная в листинге 2.14.

Листинг 2.14. Обработка ошибок разных типов

```
print("Операции со списком чисел…")
# Контролируемый код:
try:
    nums=eval(input("Введите числовой список: "))
    print("Получено значение: "+str(nums))
    a=int(nums[0])
    b=int(nums[3])
    print(str(a)+"/"+str(b)+"="+str(a/b))
# Обработка исключений:
except ValueError:
    print("ValueError: ошибка при преобразовании!")
except ZeroDivisionError:
    print("ZeroDivisionError: попытка деления на ноль!")
except TypeError:
    print("TypeError: недопустимая операция!")
except IndexError:
    print("IndexError: неверный индекс элемента!")
except SyntaxError:
    print("SyntaxError: невозможно вычислить выражение!")
except NameError:
    print("NameError: неверный идентификатор!")
print("Завершение программы.")
```

Программа, в общем-то, простая, но содержит много «скрытых опасностей». Начинается все с выполнения команды `print("Операции со списком чисел...")`, после чего начинают выполняться команды из `try`-блока. Командой `nums=eval(input("Введите числовой список: "))` считывается и вычисляется значение выражения, которое вводит пользователь (мы ожидаем, что пользователь введет выражение для списка с целочисленными элементами). Уже на этом этапе могут возникнуть проблемы, если пользователь введет выражение, которое интерпретатор не сможет вычислить. Далее командой `print("Получено значение: "+str(nums))` отображается «конструкция», которая вычислена на основе введенного пользователем выражения (эта команда нужна для того, чтобы легче было понять, что же «прочитала» программа). После этого выполняются две однотипные команды `a=int(nums[0])` и `b=int(nums[3])`. Какие ошибки могут возникнуть здесь? Во-первых, мы индексируем переменную `nums`, что не всегда возможно делать. Во-вторых, если список не пустой, то начальный элемент `nums[0]` существует, но элемента `nums[3]` в списке может и не быть (имеется в виду, что элемент с индексом `3` будет, только если в списке не меньше четырех элементов). В-третьих, даже если элементы с указанными индексами есть, не факт, что они могут быть преобразованы в целочисленный тип `int`.

При выполнении команды `print(str(a)+"/"+str(b)+"= "+str(a/b))` может возникнуть проблема, если значение переменой `b` равно нулю (а на ноль делить нельзя). Такой вот опасный программный код.

В `except`-блоках обрабатываются ошибки шести разных типов (`ValueError`, `ZeroDivisionError`, `TypeError`, `IndexError`, `SyntaxError` и `NameError`). Обработка очень простая: при возникновении ошибки появляется сообщение с классом ошибки (чтобы легче было идентифицировать «сработавший» `except`-блок) и краткое описание причины ошибки.

В конце выполнения программы командой `print("Завершение программы.")` выводится последнее сообщение.

Ниже представлены разные варианты выполнения программы в зависимости от того, какое выражение вводит пользователь (здесь и далее оно выделено жирным шрифтом). Сначала рассмотрим случай, когда пользователь вводит целочисленный список с достаточным (не меньше четырех) количеством элементов в списке, и четвертый по счету (с индексом `3`) элемент отличен от нуля.

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: [2,4,6,5,1]
Получено значение: [2, 4, 6, 5, 1]
2/5=0.4
Завершение программы.
```

В таком случае при выполнении программы исключения не генерируются, и ни один `except`-блок не выполняется.

Следующая ситуация — пользователь вводит текстовое значение (в кавычках — это важно!).

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: "Python"
Получено значение: Python
ValueError: ошибка при преобразовании!
Завершение программы.
```

Это значение считывается как текст. Текст, как и список, можно индексировать. Но при попытке преобразовать первую букву текста в целое число (команда `a=int(nums[0])`) возникает ошибка класса `ValueError`. Причем ошибка возникает именно на этапе преобразования в число — выражение `nums[0]` вычисляется нормально.

Теперь пользователь вводит текст, но уже без двойных кавычек.

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: Python
NameError: неверный идентификатор!
Завершение программы.
```

По сравнению с предыдущей ситуацией есть изменения. Чтобы понять причину, следует вспомнить, как работает функция `eval()`. Эта

функция вычисляет значение выражения, содержащегося в тексте, который является аргументом функции. Если пользователь вводит выражение `"Python"`, то в функцию `eval()` передается выражение `'"Python"'`, и вычисляться будет выражение `"Python"`. Это текст, именно он и возвращается функцией `eval()`. А вот если пользователь вводит `Python`, то в функцию `eval()` передается выражение `"Python"`, а вычисляться будет выражение `Python`. Интерпретатор воспринимает это выражение как название переменной. Но такой переменной нет, поэтому на этапе вычисления введенного пользователем значения (команда `nums=eval(input("Введите числовой список: "))`) возникает ошибка класса `NameError`.

Ниже представлена ситуация, сходная с предыдущей, но теперь пользователь вводит не одно слово, а два (фраза `Hello World!`).

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: Hello World!
SyntaxError: невозможно вычислить выражение!
Завершение программы.
```

Интерпретатор пытается вычислить значение выражения `Hello World!`, но на этот раз он данную конструкцию, как имя переменной не воспринимает, а просто не может понять, что это за выражение. В результате возникает синтаксическая ошибка класса `SyntaxError`.

Теперь пользователь вводит список, в котором всего три элемента (и поэтому элемента с индексом `3` нет).

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: [1,2,3]
Получено значение: [1, 2, 3]
IndexError: неверный индекс элемента!
Завершение программы.
```

Ошибка класса `IndexError` возникает на этапе вычисления выражения `nums[3]`.

А вот что будет, если в списке элемент с индексом 3 равен нулю.

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: [1,2,3,0,5]
Получено значение: [1, 2, 3, 0, 5]
ZeroDivisionError: попытка деления на ноль!
Завершение программы.
```

Ошибка класса `ZeroDivisionError` возникает при вычислении выражения `a/b`, поскольку предпринимается попытка деления на ноль.

Еще одна ситуация — пользователь вводит целое число.

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: 12345
Получено значение: 12345
TypeError: недопустимая операция!
Завершение программы.
```

При попытке проиндексировать значение переменной `nums` (команда `nums[0]`) возникает ошибка класса `TypeError`, поскольку к целым числам операция индексирования неприменима.

Понятно, что механизм «персональной» обработки ошибок — это хорошо. Но одна проблема очевидна: довольно сложно предусмотреть, какого типа ошибки в принципе могут возникнуть. Поэтому обычно «подстраховываются»: создают `except`-блоки для обработки ошибок определенного типа, а последним добавляют `except`-блок без указания типа ошибки. Этот блок обрабатывает все ошибки, которые не обрабатываются «персональными» блоками. В листинге 2.15 представлен программный код из предыдущего примера, но с небольшими изменениями: в нем оставлены `except`-блоки для обработки ошибок классов `ZeroDivisionError` и `IndexError`, а прочие ошибки обрабатываются последним `except`-блоком.

Листинг 2.15. Еще один способ обработки ошибок

```
print("Операции со списком чисел…")
try:
    nums=eval(input("Введите числовой список: "))
    print("Получено значение: "+str(nums))
    a=int(nums[0])
    b=int(nums[3])
    print(str(a)+"/"+str(b)+"="+str(a/b))
except ZeroDivisionError:
    print("ZeroDivisionError: попытка деления на ноль!")
except IndexError:
    print("IndexError: неверный индекс элемента!")
except:
    print("Что-то пошло не так!")
print("Завершение программы.")
```

Ниже результат выполнения программы, если ошибки в процессе вычислений не возникают.

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: [2,4,6,5,1]
Получено значение: [2, 4, 6, 5, 1]
2/5=0.4
Завершение программы.
```

Пользователь вводит текст в кавычках.

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: "Python"
Получено значение: Python
Что-то пошло не так!
Завершение программы.
```

Пользователь вводит текст (из одного слова) без кавычек.

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: Python
Что-то пошло не так!
Завершение программы.
```

Пользователь вводит фразу из нескольких слов.

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: Hello World!
Что-то пошло не так!
Завершение программы.
```

Пользователь вводит список из трех элементов.

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: [1,2,3]
Получено значение: [1, 2, 3]
IndexError: неверный индекс элемента!
Завершение программы.
```

Вводится список с нулевым четвертым элементом.

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: [1,2,3,0,5]
Получено значение: [1, 2, 3, 0, 5]
ZeroDivisionError: попытка деления на ноль!
Завершение программы.
```

Пользователь вводит целое число.

Результат выполнения программы (из листинга 2.14)

```
Операции со списком чисел…
Введите числовой список: 12345
Получено значение: 12345
Что-то пошло не так!
Завершение программы.
```

Удобство такого подхода (когда используется `except`-блок без указания типа ошибки) состоит в том, что нет необходимости идентифицировать типы генерируемых ошибок.

НА ЗАМЕТКУ

Этот блок, который «ловит» все, что в принципе можно «поймать», должен быть последним в списке `except`-блоков. Объяснение такое: при генерировании исключения начинают проверяться `except`-блоки на предмет совпадения типа ошибки, указанной в блоке, и типа ошибки, которая возникла. Если совпадение есть, то выполняются команды соответствующего `except`-блока. Блоки проверяются в том порядке, в котором они указаны в коде. Если бы, гипотетически, блок `except` без идентификатора типа ошибки размещался не последним, то он бы обрабатывал все ошибки, которые не были обработаны до него. Поэтому не было бы смысла размещать после такого `except`-блока другие блоки.

Резюме

Не надо громких слов, они потрясают воздух, но не собеседника.

Из к/ф «Формула любви»

- Оператор цикла `while` позволяет многократно выполнять определенный набор команд. Команды выполняются, пока истинно условие, указанное в операторе цикла.
- Оператор цикла `for` удобно использовать для перебора содержимого списка или иной упорядоченной последовательности элементов.

Для создания виртуальных последовательностей (объект, по своим свойствам напоминающий список) используют функцию `range()`.

- При работе с операторами цикла для досрочного завершения одного цикла можно использовать инструкцию `continue`. Для досрочного завершения работы оператора цикла используют инструкцию `break`.
- В операторах цикла можно использовать `else`-блок. Команды этого блока выполняются в конце работы оператора цикла, и только один раз. Если работа оператора цикла завершается с помощью инструкции `break`, команды в `else`-блоке не выполняются.
- В условном операторе `if` можно использовать сложные условия (в которых задействованы такие логические операторы, как `and`, `or` и `not`). Условные операторы могут быть вложенными. Также существует специальная форма условного оператора с `elif`-блоками, благодаря чему в условном операторе можно последовательно проверять несколько условий.
- У тернарного оператора три операнда: условие и два значения. Тернарный оператор возвращает результат, который зависит от истинности или ложности условия. По своим возможностям тернарный оператор дополняет (но не заменяет) условный оператор.
- Для обработки исключительных ситуаций используется конструкция `try-except`. Контролируемый код, который может вызвать ошибку, помещается в блок `try`. Блок `except` содержит команды, которые выполняются при возникновении ошибки. Если при выполнении команд в `try`-блоке ошибки не возникли, то блок `except` игнорируется.
- Можно использовать несколько блоков `except` для обработки ошибок разных типов. В таком случае тип ошибки (класс исключения) указывается после ключевого слова `except`. Если последний `except`-блок не содержит идентификатора с названием класса исключения, то такой блок обрабатывает все ошибки, которые не обрабатываются предыдущими `except`-блоками.

Задания для самостоятельной работы

Как советовать, так все чатлане, а как работать, так...

Из к/ф «Кин-дза-дза»

1. Напишите программу, в которой пользователь вводит целое число, а программа определяет, сколько в этом числе цифр `0`, `1`, `2` и так далее, до `9`.

2. Напишите программу, в которой пользователь вводит целое число, а программа каждую цифру в этом числе меняет на «дополнение до 9»: цифра `0` меняется на `9`, цифра `1` меняется на `8`, цифра `2` меняется на `7`, и так далее — цифра `8` меняется на `1`, а цифра `9` меняется на `0`.

3. Напишите программу, которая на основе списка из натуральных чисел формирует целое число. Число формируется «объединением» элементов списка: например, если исходный список `[12,3,456,78]`, то программа должна получить число `12345678`.

4. Напишите программу, в которой сравниваются (на предмет равенства) два числовых списка. Два списка равны, если они одинакового размера и у них совпадают соответствующие элементы.

5. Напишите следующую программу. Пользователь вводит список целочисленных значений, а также верхнюю границу для вычисления суммы. Программа вычисляет сумму натуральных чисел, но за исключением тех, которые входят в список. Например, если пользователь ввел список `[2,5,6]` и `10` в качестве верхней границы суммы, то программа должна вычислить сумму чисел от `1` до `10`, но без учета чисел `2`, `5` и `6`.

6. Напишите программу, в которой пользователь вводит три числа, а программа определяет, может ли существовать треугольник со сторонами, длина которых равняется введенным значениям. Условие существования треугольника такое: сумма двух любых (из трех введенных) чисел должна быть больше третьего числа.

7. Напишите программу, в которой пользователь вводит три целых числа, а программа проверяет, являются ли эти числа тремя последовательными элементами арифметической последовательности. В арифметической последовательности каждый новый член получается прибавлением к предыдущему определенного фиксированного числа.

8. Напишите программу, в которой пользователь вводит целое число от `1` до `7` включительно, а программа выводит название дня недели, соответствующее этому числу (`"Понедельник"` для `1`, `"Вторник"` для `2`, и так далее).

9. Напишите программу, в которой пользователь вводит два действительных числа, а программа проверяет, какие из чисел больше. Используйте тернарный оператор и обработку исключительных ситуаций.

10. Напишите программу для решения уравнения $Ax = B - A - 1$. Параметры A и B вводятся пользователем. Уравнение имеет решение $x = (B - 1) / A - 1$ если $A \neq 0$. При $A = 0$ и $B = 1$ решением является любое число, а при $A = 0$ и $B \neq 1$ решений у уравнения нет. Предложите разные варианты программы, в том числе и с использованием обработки исключительных ситуаций.

Глава 3
СПИСКИ И КОРТЕЖИ

А чем они друг от друга отличаются?

Из к/ф «Кин-дза-дза»

В этой главе мы столкнемся с такими типами данных, как списки и кортежи: узнаем, чем список отличается от кортежа, какие операции с ними можно выполнять и какие функции при этом могут использоваться. Начнем со знакомства с *кортежами*, которые очень кратко упоминались в предыдущей главе.

Знакомство с кортежами

А нормального входа в этот универсам нет?

Из к/ф «Кин-дза-дза»

Кортеж (тип `tuple`), как и список (тип `list`), представляет собой упорядоченную последовательность элементов (не обязательно одного типа). С точки зрения «природы» этих объектов кортежи и списки практически эквивалентны. Различия между ними проявляются на «техническом» уровне. Сводится все к тому, какие операции можно выполнять со списками, а какие — с кортежами. Кортежи относятся к *неизменяемым* типам данных. В известном смысле про кортеж можно думать как про список, элементы которого изменить нельзя. Это, кстати, совсем не означает, что с кортежами нельзя выполнять операции. Но обо всем по порядку.

НА ЗАМЕТКУ

Кортежи напоминают списки, но арсенал доступных операций для них скромнее. Зачем же тогда кортежи нужны? Ответ состоит в том, что кортежи проще реализуются. С другой стороны, для большинства прикладных задач вполне достаточно операций, доступных при работе с кортежами.

При создании элементы списка указываются в квадратных скобках, а элементы кортежа — в круглых. Самый простой способ создать кортеж состоит в том, чтобы перечислить в круглых скобках через запятую элементы. Например, командой `A=(1,2,3)` создается кортеж (ссылка на кортеж записывается в переменную `A`), который состоит из трех целочисленных элементов (`1`, `2` и `3`). Кортеж (как и список) может содержать элементы разных типов. Так, команда `B=(100, "Python",[1,2,3],(10,20))` создает кортеж (и ссылка на него записывается в переменную `B`), который содержит три элемента: целое число `100`, текст `"Python"`, список `[1,2,3]` и кортеж `(10,20)`.

Для создания пустого кортежа (кортеж, который не содержит элементов) можно использовать пустые круглые скобки. Например, командой `A=()` в переменную `A` записывается ссылка на пустой кортеж.

ПОДРОБНОСТИ

С учетом того, что кортежи относятся к неизменяемым типам данных, может показаться, что создавать пустой кортеж нет смысла. Однако это не так. Как мы увидим далее, несмотря на свою «неизменность», с кортежами можно выполнять очень «колоритные» операции, в том числе с использованием пустых кортежей (например, в качестве начального значения переменной).

Также есть особенности у создания кортежа из одного элемента. Например, список из одного элемента мы можем создать командой `A=[10]` (список `A`, в котором один целочисленный элемент `10`) или `B=["Python"]` (список `B`, в котором один текстовый элемент `"Python"`). Но если мы захотим создать кортеж из одного элемента, то квадратные скобки нужно заменить на круглые, а после единственного элемента следует поставить запятую. Например, мы можем использовать команду `A=(10,)` или `B=("Python",)`. Причина, по которой после единственного элемента нужно указывать запятую, состоит в том, что в Python выражение в круглых скобках интерпретируется так же, как и соответствующее выражение баз скобок. Скажем, в результате выполнения команды `A=(10)` переменная `A` будет ссылаться на целочисленное значение `10`, а переменная `B` после выполнения команды `B=("Python")` будет ссылаться на текст `"Python"`.

При выводе с помощью функции `print()` содержимого кортежа его элементы отображаются в круглых скобках и разделяются запятыми. Если кортеж состоит из одного элемента, то после этого элемента отображается запятая.

Круглые скобки при создании кортежа можно не использовать вовсе. Так, если некоторой переменной в качестве значения присвоить последовательность разделенных запятой элементов, то в результате переменная будет ссылаться на кортеж, созданный на основе данной последовательности.

НА ЗАМЕТКУ

Например, если воспользоваться командой `B="Python"`, то переменная `B` в качестве значения получит ссылку на текст `"Python"`. А если воспользоваться командой `B="Python",` (то есть после текста `"Python"` поставить запятую), то будет создан кортеж из одного текстового элемента `"Python"` и ссылка на этот кортеж записывается в переменную `B`.

Для создания кортежей может использоваться функция `tuple()`. Если вызвать функцию без аргументов, то получим пустой кортеж. Если передать аргументом функции `tuple()` текст, то в результате получим кортеж, элементами которого являются буквы из текста (каждый элемент кортежа реализуется как текст). Чтобы создать кортеж на основе списка или другого, уже существующего кортежа, этот список или кортеж указывают в качестве аргумента функции `tuple()`.

К кортежам, как и к спискам, можно применять индексирование, включая процедуру получения среза. Главное отличие кортежей от списков стоит в том, что с помощью индексирования кортежа или получения среза мы можем получить (прочитать) значение или значения, но не можем присвоить новые значения элементам кортежа.

НА ЗАМЕТКУ

Напомним основные правила индексирования и получения срезов (в данном случае в отношении списков и кортежей). Итак, индекс указывается в квадратных скобках. Индексация начинается с нуля, поэтому первый элемент в списке/кортеже имеет индекс `0`. Количество элементов можно определить с помощью функции `len()`. Индекс последнего элемента на единицу меньше количества элементов в списке/кортеже. Можно указывать отрицательный индекс. В таком случае элементы отсчитываются с конца списка/кортежа: индекс `-1` соответствует последнему элементу, индекс `-2` соответствует предпоследнему элементу и так далее.

Если не указать первый и/или последний индекс в инструкции получения среза, то в качестве пропущенного индекса будет использован

индекс соответственно первого или последнего элемента. Например, выражение вида `[: j+1]` означает срез от начального элемента до элемента с индексом `j`, выражение `[i:]` означает срез от элемента с индексом `i` до последнего элемента, а выражение `[:]` означает срез от начального до последнего элемента.

При получении среза, если нас интересуют элементы с индексами от `i` до `j` включительно, в квадратных скобках через двоеточие указываются значения `i` (индекс первого из считываемых элементов) и `j+1` (индекс первого из не считываемых элементов), то есть инструкция, используемая для индексирования, выглядит как `[i: j+1]`. Если срез выполняется для списка, то получаем список, а если срез выполняется для кортежа, то получаем кортеж.

Небольшой пример, в котором создаются кортежи и затем с ними выполняются очень простые операции (такие как считывание значения элемента кортежа или получение среза), представлен в листинге 3.1.

Листинг 3.1. Знакомство с кортежами

```
# Разные способы создания кортежей:
Alpha=(5,10,15,"двадцать")
Bravo=100,["один","два","три"],200
Charlie=tuple([1,2,3,(4,5,6,7,8,9)])
Delta=tuple("ABCDEF")
Echo=tuple(2**k for k in range(11))
# Считывание значений элементов и получение среза:
print("Alpha:", Alpha)
print("Элементов:", len(Alpha))
print("Первый:", Alpha[0])
print("Последний:", Alpha[-1])
print("Bravo:", Bravo)
print("Элементов:", len(Bravo))
print("Bravo[1]:", Bravo[1])
print("Bravo[1][2]:", Bravo[1][2])
print("Charlie:", Charlie)
print("Элементов:", len(Charlie))
print("Charlie[3]:", Charlie[3])
```

```
print("Charlie[3][1:4]:", Charlie[3][1:4])
print("Delta:", Delta)
print("Элементов:", len(Delta))
print("Delta[-3:]:", Delta[-3:])
print("Echo:", Echo)
Foxtrot=tuple(Echo[k] for k in range(len(Echo)) if k%2==0)
print("Foxtrot:", Foxtrot)
Golf=Echo[2:5]
print("Golf:", Golf)
```

При выполнении программы получаем такой результат.

Результат выполнения программы (из листинга 3.1)

```
Alpha: (5, 10, 15, 'двадцать')
Элементов: 4
Первый: 5
Последний: двадцать
Bravo: (100, ['один', 'два', 'три'], 200)
Элементов: 3
Bravo[1]: ['один', 'два', 'три']
Bravo[1][2]: три
Charlie: (1, 2, 3, (4, 5, 6, 7, 8, 9))
Элементов: 4
Charlie[3]: (4, 5, 6, 7, 8, 9)
Charlie[3][1:4]: (5, 6, 7)
Delta: ('A', 'B', 'C', 'D', 'E', 'F')
Элементов: 6
Delta[-3:]: ('D', 'E', 'F')
Echo: (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024)
Foxtrot: (1, 4, 16, 64, 256, 1024)
Golf: (4, 8, 16)
```

Программа достаточно простая. Например, команда `Alpha=(5, 10, 15, "двадцать")` создает кортеж из четырех элементов (три числа

и текст). Для определения количества элементов используем функцию `len()`. Значение первого элемента считывается инструкцией `Alpha[0]`. Значение последнего элемента считывается инструкцией `Alpha[-1]`.

> **НА ЗАМЕТКУ**
>
> Еще раз обращаем внимание, что с помощью выражений вида `Alpha[0]` или `Alpha[-1]` мы можем прочитать значения элементов. Присвоить новые значения элементам не получится.

Еще один кортеж создается командой `Bravo=100,["один", "два", "три"], 200`. В данном случае мы круглые скобки не используем. Сам кортеж состоит из трех элементов, одним из которых является список `["один","два","три"]`. Поэтому значением выражения `Bravo[1]` является ссылка на этот список. Тогда несложно догадаться, что значением выражения `Bravo[1][2]` является ссылка на элемент с индексом `2` в списке, ссылка на который возвращается выражением `Bravo[1]`. Это элемент с текстовым значением `"три"`.

При выполнении команды `Charlie=tuple([1, 2, 3, (4, 5, 6, 7, 8, 9)])` кортеж создается на основе списка `[1,2,3,(4,5,6,7,8,9)]` (последним элементом которого является кортеж). В итоге получаем кортеж из четырех элементов: числа `1`, `2` и `3`, а также кортеж `(4,5,6,7,8,9)` (элемент кортежа `Charlie` с индексом `3`). Значением выражения `Charlie[3]` — ссылка на кортеж `(4,5,6,7,8,9)`. При вычислении выражения `Charlie[3][1:4]` получаем срез из кортежа `(4,5,6,7,8,9)`, в который включаются элементы с индексами от `1` до `3` включительно. Поскольку срез создается на основе кортежа, то результатом также является кортеж. Более конкретно, это кортеж `(5,6,7)`.

Примером создания кортежа на основе текстового значения является команда `Delta=tuple("ABCDEF")`. Кортеж в этом случае получается разбивкой исходного текста на символы. Для получения среза использована инструкция `Delta[-3:]`. Первый индекс `-3` означает третий с конца элемент кортежа `Delta`. Это элемент, начиная с которого выполняется срез. Поскольку второй индекс (индекс после двоеточия) не указан, то срез выполняется включительно до последнего элемента кортежа `Delta`. В итоге `Delta[-3:]` является ссылкой на кортеж, который состоит из трех последних элементов кортежа `Delta`.

В команде `Echo=tuple(2**k for k in range(11))` мы передали аргументом функции `tuple()` выражение `2**k for k in range(11)`,

которое генерирует числовую последовательность (число 2 в степени от 0 до 10 включительно). На основе этой последовательности формируется кортеж. Мы его используем для создания еще одного кортежа. Речь о команде `Foxtrot=tuple(Echo[k] for k in range(len(Echo)) if k%2==0)`. Она похожа на команду, которой создавался кортеж `Echo`, но аргумент, переданный функции `tuple()`, теперь выглядит немного сложнее. Выражением `Echo[k] for k in range(len(Echo)) if k%2==0` формируется последовательность из элементов кортежа `Echo`. Индекс `k` пробегает значения из последовательности целых чисел от 0 до `len(Echo)-1` (индекс последнего элемента в кортеже `Echo`) включительно, но не все, а только те из них, для которых истинно условие `k%2==0` (значение индекса делится нацело на 2), указанное после ключевого слова `if`. Проще говоря, в формируемую последовательность входят только те элементы кортежа `Echo`, у которых индекс четный (то есть 0, 2, 4 и так далее). В итоге в кортеж `Foxtrot` включаются, через один, элементы из кортежа `Echo`, начиная с первого элемента.

Наконец, командой `Golf=Echo[2:5]` выполняется срез кортежа `Echo`. Срез содержит элементы с индексами от 2 до 4 включительно и сам является кортежем.

Основные операции со списками и кортежами

А здесь из луца воду делают.

Из к/ф «Кин-дза-дза»

Теперь мы переходим к рассмотрению операций, которые могут выполняться со списками и кортежами. Хотя кортежи реализуются иначе, чем списки, но с точки зрения их назначения разница не такая уж принципиальная: в обоих случаях имеем дело с упорядоченной последовательностью элементов. Поэтому вопрос сводится к тому, как та или иная операция выполняется при использовании списков и как она выполняется при использовании кортежей. Со списками операции выполнять проще, для работы с ними имеется множество специальных функций и методов. Арсенал поддержки операций с кортежами скромнее. Поэтому нередко при использовании кортежей приходится прибегать к изобретательности, а то и изощренности.

НА ЗАМЕТКУ

Как неоднократно отмечалось, кортежи относятся к неизменяемым типам данных и кортеж после создания изменить нельзя. Скажем, можно узнать значение элемента кортежа, но нельзя элементу кортежа присвоить новое значение. А что же можно? На самом деле можно очень много. Главное понимать, как реализуются кортежи и как обрабатываются операции, связанные с использованием кортежей.

Базовый принцип, который обычно используется при выполнении операций с кортежами, состоит в следующем. Допустим, имеется некоторый кортеж. Доступ к кортежу мы получаем с помощью переменной, которая ссылается на кортеж. Кортеж мы изменить не можем. Но мы можем записать в переменную ссылку на другой кортеж. Получается, что обращаясь все к той же переменной, мы получаем уже другой кортеж. Формально все выглядит так, как если бы исходный кортеж изменился. Но на самом деле это уже другой кортеж.

Основных операций, которые выполняются с упорядоченными последовательностями элементов (то есть со списками и кортежами), не так уж и много. Можно выделить такие:

- считывание значения элемента (или группы элементов);
- изменение значения элемента;
- вставка элемента (или группы элементов) в последовательность;
- удаление элемента (или группы элементов) из последовательности;
- изменение порядка элементов в последовательности.

Прочие манипуляции со списками и кортежами в той или иной степени можно рассматривать как частные случаи для указанных выше операций.

НА ЗАМЕТКУ

Напомним, что узнать значение элемента можно, проиндексировав список/кортеж или выполнив срез (если нужно получить значения для группы элементов). Допустим, имеется некоторый `список`. Если мы хотим узнать значение элемента в этом списке с индексом `i`, то используем инструкцию вида `список[i]`. Если мы выполним срез инструкцией вида `список[i: i+1]`, то получим список, который состоит из одного элемента (это элемент с индексом `i`).

По отношению к спискам и кортежам может применяться оператор сложения +. Если сложить с помощью оператора + два списка, получим список, который является объединением исходных (суммируемых) списков. Это же относится к кортежам, с поправкой на то, что суммируются два кортежа, и результатом также является кортеж. Одной только этой операции, в комбинации с выполнением срезов, достаточно для решения многих задач. Для начала рассмотрим небольшой пример, в котором выполняются простые операции со списками. Некоторые операции читателю уже должны быть знакомы. Код программы представлен в листинге 3.2.

Листинг 3.2. Операции со списками

```
# Создание списков:
A=[10,20,30]
print("A:", A)
B=["Python",[1,2]]
print("B:", B)
# Вычисление суммы списков:
C=A+B
print("C:", C)
# Добавление элемента в конец списка:
C+=[100]
print("C:", C)
# Удаление элемента списка:
C[1:2]=[]
print("C:", C)
# Добавление элемента в начало списка:
C=[200]+C
print("C:", C)
# Замена нескольких элементов в списке:
C[:3]=["A","B"]
print("C:", C)
# Вставка элементов в список:
C[2:2]=[8,9]
print("C:", C)
```

```
# Присваивание значения элементу списка:
C[2:3]=[7]
print("C:", C)
```

При выполнении программы получаем следующий результат.

Результат выполнения программы (из листинга 3.2)

```
A: [10, 20, 30]
B: ['Python', [1, 2]]
C: [10, 20, 30, 'Python', [1, 2]]
C: [10, 20, 30, 'Python', [1, 2], 100]
C: [10, 30, 'Python', [1, 2], 100]
C: [200, 10, 30, 'Python', [1, 2], 100]
C: ['A', 'B', 'Python', [1, 2], 100]
C: ['A', 'B', 8, 9, 'Python', [1, 2], 100]
C: ['A', 'B', 7, 9, 'Python', [1, 2], 100]
```

Командами `A=[10,20,30]` и `B=["Python",[1,2]]` мы создаем два списка. Они использованы в инструкции `C=A+B`, которой вычисляется новый список. Он получается объединением содержимого списков `A` и `B`, а ссылка на этот созданный список записывается в переменную `C`. Списки `A` и `B` при этом не меняются. Они просто служат основой для вычисления нового списка.

Команда `C+=[100]` добавляет в конец списка `C` целочисленный элемент `100`. Аналогичной командой `C=[200]+C` новый целочисленный элемент `200` добавляется в начало списка `C`.

ПОДРОБНОСТИ

Команда `C+=100` является эквивалентом команды `C=C+[100]`. При вычислении выражения `C+[100]` создается новый список, который получается объединением содержимого списка `C` и списка `[100]` (список из одного элемента). Ссылка на созданный список присваивается в качестве значения переменной `C`. В итоге получается, что переменная `C` ссылается на список, который отличается от исходного (на который до этого ссылалась переменная `C`) наличием в конце элемента `100`.

Нечто похожее происходит при выполнении команды `C=[200]+C`, только теперь объединяются списки `[200]` и `C`, поэтому элемент `200` оказывается в начале вновь созданного списка.

Команда `C[1:2]=[]` удаляет из списка `C` элемент с индексом `1` (второй по порядку элемент в списке). А вот командой `C[:3]=["A","B"]` первые три элемента в списке `C` (элементы с индексами `0`, `1` и `2`) заменяются на два элемента из списка `["A","B"]`.

ПОДРОБНОСТИ

Выражение `C[1:2]` означает срез, состоящий всего из одного элемента (элемент списка `C` с индексом `1`). Этому срезу присваивается в качестве значения пустой список (обозначен пустыми квадратными скобками `[]`). Данная операция обрабатывается так, что элементы, входящие в срез, из исходного списка удаляются.

Инструкция `C[:3]` соответствует срезу списка `C`, в который входят первые три элемента с индексами `0`, `1` и `2` (пропущенный первый индекс в инструкции получения среза означает, что срез выполняется, начиная с начального элемента). Этому срезу в качестве значения присваивается список `["A","B"]` из двух элементов (`"A"` и `"B"`). В итоге элементы, определяемые срезом, из списка удаляются, а вместо них в список вставляются элементы из списка, который присваивается срезу.

Новой для нас является команда `C[2:2]=[8,9]`. Она напоминает предыдущие, но обращает на себя внимание инструкция получения среза `C[2:2]`. Особенного в ней то, что оба индекса совпадают. Формально это пустой срез. То есть если бы мы захотели отобразить содержимое такого среза, то список был бы пустым. Но если такого вида инструкции присваивается значение (список), то обрабатывается команда так, что в ту позицию, которая определяется совпадающими индексами в квадратных скобках, вставляются элементы из присваиваемого списка. В нашем случае инструкция `C[2:2]=[8,9]` обрабатывается следующим образом: в список `C` в позицию элемента с индексом `2` вставляются элементы `8` и `9`, а все элементы списка, начиная с элемента с индексом `2`, смещаются вправо на две позиции.

ПОДРОБНОСТИ

По правилам языка Python при выполнении команды вида `A[i: i]=B` в список `A` в позицию с индексом `i` вставляются элементы из списка `B`, а все элементы списка `A`, начиная с элемента с индексом `i`, сдвигаются вправо.

Контрастирует с предыдущей командой инструкция `C[2:3]=[7]`. Срез `C[2:3]` состоит из одного элемента (с индексом `2`) списка `C`. Вместо этого

элемента в список вставляется целочисленный элемент 7. Но по сути получается, что элемент списка C с индексом 2 получает значение 7.

НА ЗАМЕТКУ

Конечно, вместо команды `C[2:3]=[7]` мы могли бы использовать команду `C[2]=7`. Эффект в данном случае был бы такой же.

Теперь мы посмотрим, как аналогичные операции могли бы быть выполнены в случае, если вместо списков используются кортежи. Соответствующий программный код представлен в листинге 3.3.

Листинг 3.3. Операции с кортежами

```
# Создание кортежей:
A=(10,20,30)
print("A:", A)
B=("Python",(1,2))
print("B:", B)
# Вычисление суммы кортежей:
C=A+B
print("C:", C)
# "Добавление" элемента в конец кортежа:
C+=(100,)
print("C:", C)
# "Удаление" элемента кортежа:
C=C[:1]+C[2:]
print("C:", C)
# "Добавление" элемента в начало кортежа:
C=(200,)+C
print("C:", C)
# "Замена" нескольких элементов в кортеже:
C=("A","B")+C[3:]
print("C:", C)
# "Вставка" элементов в кортеж:
C=C[:2]+(8,9)+C[2:]
```

```
print("C:", C)
# "Присваивание" значения элементу кортежа:
C=C[:2]+(7,)+C[3:]
print("C:", C)
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 3.3)

```
A: (10, 20, 30)
B: ('Python', (1, 2))
C: (10, 20, 30, 'Python', (1, 2))
C: (10, 20, 30, 'Python', (1, 2), 100)
C: (10, 30, 'Python', (1, 2), 100)
C: (200, 10, 30, 'Python', (1, 2), 100)
C: ('A', 'B', 'Python', (1, 2), 100)
C: ('A', 'B', 8, 9, 'Python', (1, 2), 100)
C: ('A', 'B', 7, 9, 'Python', (1, 2), 100)
```

Результат выполнения этой программы (с поправкой на то, что мы использовали кортежи) совпадает с результатом выполнения предыдущей программы.

Исходным для вычислений являются кортежи, которые создаются командами `A=(10,20,30)` и `B=("Python",(1,2))`. При выполнении команды `C=A+B` на основе кортежей `A` и `B` (путем объединения) создается новый кортеж, и ссылка на него записывается в переменную `C`. С помощью команды `C+=(100,)` создается эффект того, что в конец кортежа добавляется новый элемент.

ПОДРОБНОСТИ

Строго говоря, что при выполнении команды `C+=(100,)` в рассматриваемом примере, что при выполнении команды `C+=[100]` в предыдущем примере, ничего никуда не добавляется. В действительности при выполнении команды `C+=(100,)` создается новый кортеж, и ссылка на этот кортеж записывается в переменную `C`. Точно то же случается, когда мы в предыдущем примере используем команду

`C+=[100]`: создается новый список и ссылка на него записывается в переменную `C`. То есть здесь принципиальной разницы между списками и кортежами как бы и нет. Но это кажущаяся одинаковость. Связана она исключительно с тем, как мы организовали код. А организовать его мы могли по-разному. Для кортежей действительно нет альтернативного способа «добавить» элемент — только создать новый кортеж. Но для списка такая возможность имеется. Даже несколько возможностей. Поясним на маленьком примере.

Допустим, командой `X=[1,2,3]` создается список. Мы хотим добавить в конец этого списка элемент `100`. Для этого подойдет команда `X[len(X): len(X)]=[100]`. В этой команде выражение `X[len(X): len(X)]` соответствует пустому срезу. Более того, элемента с индексом `len(X)` в списке `X` вообще нет, поскольку индекс последнего элемента на единицу меньше размера списка. Но поскольку срезу присваивается значение, то команда обрабатывается так: в позицию с индексом `len(X)` (то есть в конец списка) вставляется элемент `100`. С другой стороны, мы могли воспользоваться командой `X=X+[100]`. Внешний эффект будет одинаков: какую бы команду ни использовали, переменная `X` в итоге будет ссылаться на список `[1,2,3,100]`. Так есть ли между командами принципиальная разница? Разница есть. При выполнении команды `X=X+[100]` создается новый список, а при выполнении команды `X[len(X): len(X)]=[100]` изменения вносятся в существующий список. В большинстве случаев это не важно. Но «в большинстве» не означает «всегда».

Пускай, как и раньше, список `X` создается командой `X=[1,2,3]`. А теперь выполним команду `Y=X`. То есть появляется еще одна переменная (мы ее назвали `Y`), и этой переменной в качестве значения присваивается `X`. Если мы проверим значение переменных `X` и `Y`, то обе они будут ссылаться на список `[1,2,3]`. Затем выполняется команда `X[len(X): len(X)]=[100]`. Если после этого мы снова проверим значение переменных `X` и `Y`, то для многих будет сюрприз: не только переменная `X` ссылается на список `[1,2,3,100]`, но и переменная `Y` ссылается на такой же (точнее, этот же) список. Почему? Все очень просто. При выполнении команды `Y=X` переменной `Y` присваивается такое же значение, как у переменной `X`. Но значением переменной `X` является не список. Значением переменной `X` является ссылка на список (проще говоря, адрес списка в области памяти). Именно эта ссылка копируется в переменную `Y`. В итоге обе переменные ссылаются на один и тот же список. Получается, что переменных две, а список один. Командой `X[len(X): len(X)]=[100]` в этот список вносятся изменения. Но поскольку физически это все тот же список, то переменные `X` и `Y` продолжают на него ссылаться.

> Но если бы мы вместо команды `X[len(X): len(X)]=[100]` использовали команду `X=X+[100]`, то переменная `X` ссылалась бы на список `[1,2,3,100]`, а переменная `Y` ссылалась бы на список `[1,2,3]`. Все потому, что при выполнении команды `X=X+[100]` создается новый список, и ссылка на него записывается в переменную `X`. Получается, что переменная `X` ссылается уже на новый список, а переменная `Y` продолжает ссылаться на исходный список, который не изменился.

Создать эффект «удаления» элемента из кортежа позволяет команда `C=C[:1]+C[2:]`. Мы хотим «исключить» из кортежа элемент с индексом `1`. Наша стратегия состоит в том, чтобы выполнить два среза и объединить их в новый кортеж (при этом элемент с индексом `1` «выпадает» из финального кортежа). Первый кортеж вычисляется как срез `C[:1]` — начальные элементы до индекса `0` включительно (то есть всего один элемент). Второй кортеж вычисляется как срез `C[2:]` — все элементы, начиная с элемента с индексом `2`, и до конца кортежа.

Командой `C=(200,)+C` на основе кортежа `C` формируется новый кортеж, а ссылка записывается в переменную `C`. Новый кортеж по сравнению с исходным содержит в самом начале целочисленный элемент `200`.

При выполнении команды `C=("A","B")+C[3:]` новый кортеж формируется так: кортеж `("A","B")` объединяется со срезом `C[3:]`, в который входят все элементы кортежа `C`, начиная с элемента с индексом `3`. Общий эффект такой, как если бы мы удалили из исходного кортежа первых три элемента, и вместо них вставили два (`"A"` и `"B"`).

Срезами `C[:2]` и `C[2:]` кортеж `C` разбивается (условно, конечно) на два кортежа: в первый включаются элементы с индексами до `1` включительно, а во второй включаются элементы, начиная с элемента с индексом `2`. Поэтому в результате выполнения команды `C=C[:2]+(8,9)+C[2:]` получаем кортеж, в котором между элементами с индексами `1` и `2` (в исходном кортеже) вставляются элементы `8` и `9`.

Похожая ситуация имеет место при выполнении команды `C=C[:2]+(7,)+C[3:]`, но на этот раз из исходного кортежа выпадает элемент с индексом `2`, а вместо него вставляется целочисленный элемент `7`.

Создание выборки на основе списков и кортежей

Нечестная игра! Ты специально мои ходы плохо думал!

Из к/ф «Кин-дза-дза»

Достаточно общей и часто встречающейся является такая задача: имеется список или кортеж, и на основе этого списка или кортежа необходимо создать некоторую новую последовательность элементов. Данная последовательность может быть реализована как список или кортеж — это не важно. Принципиальный момент состоит в том, что есть один набор элементов, а нам нужно на его основе создать другой. Понятно, что вариаций здесь достаточно много, но обычно все сводится к тому, что следует выбрать не все элементы, а только некоторые, или нужно выбрать все элементы, но в другом порядке (например, в обратном). Именно такого класса операции мы и обсудим.

Прежде всего рассмотрим возможности, которые у нас есть при использовании срезов. Дело в том, что при выполнении среза можно указывать третий индекс. В таком случае команда выполнения среза выглядит как `список[i: j: k]` или `кортеж[i: j: k]`. Срез выполняется следующим образом. Формируется список/кортеж (в зависимости от того, для списка или кортежа выполняется срез), и в этот список/кортеж добавляются элементы из исходного списка/кортежа, начиная с элемента, позиция которого определяется первым индексом `i`. Индекс элементов, включаемых в срез, изменяется дискретно. Шаг дискретности определяется третьим индексом `k`. Второй индекс `j` определяет позицию, ограничивающую срез (элемент с соответствующим индексом в срез не включается).

ПОДРОБНОСТИ

Третий индекс в инструкции получения среза может быть как положительным, так и отрицательным. Например, если мы используем инструкцию вида `список[2:12:3]`, то в срез будут включены элементы с индексами `2`, `5`, `8` и `11`, и именно в том порядке, как перечислены индексы. Если третий индекс отрицательный, то элементы включаются в срез в обратном порядке. Например, в результате выполнения среза с помощью инструкции вида `список[12:2:-3]` в срез будут включены элементы с индексами `12`, `9`, `6` и `3`. Если исходить из того, что элементы в списке с увеличением индекса размещаются слева направо, то при выполнении среза с использованием

трех индексов правило следующее. Если третий индекс (определяющий шаг дискретности при формировании среза) положительный, то элемент, определяемый первым индексом, должен находиться левее элемента, определяемого вторым индексом. Если третий индекс отрицательный, то элемент, определяемый первым индексом, должен находиться правее элемента, определяемого вторым индексом. Нарушение этого правила приводит к тому, что вычисляемый срез будет пустым.

Так, при вычислении среза инструкцией вида `список[0:-1:2]` элементы в срез включаются «через один» (с шагом дискретности `2`): в срез попадают элементы с индексами `0`, `2`, `4` и так далее, пока включаемый в срез элемент находится левее элемента с индексом `-1` (а это последний элемент в списке).

Инструкцией `список[-1:0:-2]` срез формируется из элементов «через один», но в обратном порядке, начиная с последнего элемента (с индексом `-1`) в списке, причем первый элемент (с индексом `0`) в срез уже не попадает. Поскольку в инструкции `список[-1:0:-2]` третий индекс отрицательный, первые два индекса указываются так, чтобы первый элемент был правее второго. Если бы мы воспользовались инструкцией `список[0:-1:-2]`, то срез оказался бы пустым.

Если первый и/или второй индекс не указан, то автоматически подразумевается «правильный» граничный элемент. «Правильность» элемента определяется по знаку третьего индекса. Если третий индекс положительный, то отсутствующий первый индекс соответствует начальному элементу списка, а отсутствующий второй индекс соответствует конечному элементу списка. Если третий индекс отрицательный, то все наоборот: отсутствующий первый индекс соответствует конечному элементу, а отсутствующий второй индекс соответствует начальному элементу. Например, инструкция `список[0::2]` означает получение среза с шагом `2`, начиная с первого элемента и до конца списка (последний элемент может быть включен в срез). Инструкцией `список[-1::-2]` срез формируется выбором элементов с шагом `2` в обратном порядке, начиная с последнего элемента и заканчивая первым (он может быть включен в срез). Эквивалентом инструкции `список[0::2]` является команда вида `список[::2]`, а инструкцию `список[-1::-2]` можно заменить на `список[::-2]`.

Еще одна возможность по созданию как списков и кортежей, так и выборок на их основе связана с использованием генератора последовательностей. Общий шаблон генератора выглядит следующим образом (здесь и далее жирным шрифтом выделены ключевые элементы шаблона).

```
выражение for переменная in диапазон if условие
```

С подобной конструкцией мы уже сталкивались (в примере из листинга 3.1). Сначала указывается выражение, определяющее значение элемента. Это выражение зависит от некоторой переменной. После ключевого слова `for` указывается название этой переменной, ключевое слово `in` и итерируемая последовательность (список, кортеж, виртуальная последовательность, создаваемая, например, с помощью функции `range()`). Переменная последовательно принимает значения из этой последовательности. Для каждого из значений переменной вычисляется элемент, который включается в последовательность, формируемую генератором. Также после ключевого слова `if` можно указать условие, используемое в качестве фильтра: переменная принимает только те значения из указанного диапазона, которые удовлетворяют указанному после ключевого слова `if` условию. Стоит также заметить, что в качестве выражения, определяющего значения для формируемой выборки, можно использовать тернарный оператор. Шаблон всей конструкции при этом может быть таким:

```
значение if условие else значение for переменная in диапазон
```

В данном случае значение элемента вычисляется выражением `значение if условие else значение` (это выражение на основе тернарного оператора). Еще более сложный вариант, когда используется и тернарный оператор, и условие для фильтрации, представлен ниже.

```
значение if условие else значение for переменная in диапазон if условие
```

Если так, то значение элементов вычисляется выражением `значение if условие else значение`, и при этом `переменная` принимает значения из `диапазона`, но только если выполняется `условие` после второго ключевого слова `if`.

Пример использования срезов и генераторов последовательностей для создания списков, кортежей и выборок на их основе, представлен в листинге 3.4.

Листинг 3.4. Создание выборки

```
# Кортеж чисел:
A=tuple(k for k in range(1,21) if k%3!=0)
print(A)
# Список чисел:
```

```
B=[2**(k//2) if k%2==0 else 3**(k//2) for k in range(15)]
print(B)
# Список чисел:
C=[0 if k==0 or k==1 else k**2 for k in range(13) if not k in [2,5,7]]
print(C)
# Кортеж в обратном порядке:
Alpha=A[::-1]
print(Alpha)
# Элементы выбираются "через один", начиная с первого:
Bravo=B[::2]
print(Bravo)
# Элементы выбираются "через один", начиная со второго:
Charlie=B[1::2]
print(Charlie)
```

Как выглядит результат выполнения программы, показано ниже.

Результат выполнения программы (из листинга 3.4)

```
(1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20)
[1, 1, 2, 3, 4, 9, 8, 27, 16, 81, 32, 243, 64, 729, 128]
[0, 0, 9, 16, 36, 64, 81, 100, 121, 144]
(20, 19, 17, 16, 14, 13, 11, 10, 8, 7, 5, 4, 2, 1)
[1, 2, 4, 8, 16, 32, 64, 128]
[1, 3, 9, 27, 81, 243, 729]
```

Код достаточно простой, но мы его все равно проанализируем. Так, команда `A=tuple(k for k in range(1,21) if k%3!=0)` создает кортеж из чисел из диапазона от `1` до `20` включительно — но не всех, а только тех, которые не делятся нацело на `3` (условие `k%3!=0` истинно, если остаток от деления значения переменной `k` на `3` отличен от `0`). А команда `B=[2**(k//2) if k%2==0 else 3**(k//2) for k in range(15)]` формирует список из `15` чисел. Это чередующиеся степени двойки и тройки. Более детально список формируется в соответствии со следующим алгоритмом. Переменная `k` пробегает значения от `0` до `14` включительно. Для каждого значения `k` элемент списка определяется как

`2**(k//2)`, если истинно условие `k%2==0`. Если же условие `k%2==0` ложное, то значение элемента списка вычисляется как `3**(k//2)`. Условие `k%2==0` истинно, если остаток от деления значения переменой `k` на `2` равен нулю. Поэтому для четных значений `k` условие истинное, а для нечетных значений `k` условие ложное. Получается, что элементы поочередно вычисляются как `2**(k//2)` или `3**(k//2)`. Значением выражения `k//2` является результат деления нацело (без остатка) значения переменной `k` на `2`. При изменении значения переменой `k` от `0` до `14` выражением `k//2` принимает значения от `0` до `7` включительно, причем каждое — по два раза. Эти значения определяют показатель степени, в которую возводятся числа `2` и `3`.

Теперь проанализируем команду `C=[0 if k==0 or k==1 else k**2 for k in range(13) if not k in [2,5,7]]`. Список формируется так: переменная `k` пробегает значения из диапазона от `0` до `12` включительно, но при этом исключаются значения `2`, `5` и `7`.

ПОДРОБНОСТИ

Истинность условия `k in [2,5,7]` формально означает, что переменная `k` принадлежит списку `[2,5,7]`. Фактически речь идет о том, что значение переменой `k` рано или `2`, или `5`, или `7`. Соответственно, истинность условия `not k in [2,5,7]` означает, что переменная `k` не принадлежит списку `[2,5,7]`. Если так, то значение этой переменной не равно ни `2`, ни `5`, ни `7`.

Значение элемента списка при данном значении переменной `k` вычисляется как значение выражения `0 if k==0 or k==1 else k**2`. А значение этого выражения равно `0`, если значение переменной `k` равно `0` или `1`. Для прочих значений переменной `k` значение элемента списка вычисляется как `k**2`.

Команда `Alpha=A[::-1]` формирует на основе кортежа `A` новый кортеж (и записывает ссылку на него в переменную `Alpha`), в котором элементы размещены в обратном порядке. Третий элемент `-1` в инструкции получения среза означает, что индексы уменьшаются с единичной дискретностью. А поскольку первые два элемента не указаны, то автоматически это означает, что срез берется от последнего до первого элемента.

При выполнении команды `Bravo=B[::2]` на основе списка `B` создается новый список. Элементы в него включаются «через один» (шаг дискретности по индексу равен `2`), начиная с первого (с нулевым индексом)

и до конца списка. Нечто похожее происходит и при выполнении команды `Charlie=B[1::2]`, но только на этот раз элементы выбираются начиная не с первого, а со второго (его индекс равен `1`).

К спискам и кортежам может применяться оператор умножения `*`. Если список или кортеж умножается на целое число, то результатом такой операции является соответственно список или кортеж, который получается повторением умножаемого списка или кортежа, количество повторений определяется значением числа, на которое производится умножение. Небольшой пример использования этой операции представлен в листинге 3.5.

Листинг 3.5. Умножение списков и кортежей на число

```
Alpha=5*[0]
print(Alpha)
Bravo=(1,)*3
print(Bravo)
Charlie=[1,2]*3
print(Charlie)
Delta=[[1,2]]*3
print(Delta)
Echo=4*(1,[2,3])
print(Echo)
Foxtrot=([1]*2)*3
print(Foxtrot)
Golf=([1]*2,)*3
print(Golf)
```

Результат выполнения программы такой, как показано ниже.

Результат выполнения программы (из листинга 3.5)

```
[0, 0, 0, 0, 0]
(1, 1, 1)
[1, 2, 1, 2, 1, 2]
[[1, 2], [1, 2], [1, 2]]
```

```
(1, [2, 3], 1, [2, 3], 1, [2, 3], 1, [2, 3])
[1, 1, 1, 1, 1, 1]
([1, 1], [1, 1], [1, 1])
```

В этой программе создается несколько списков и кортежей. Каждый раз используется умножение списка или кортежа на число. Например, команда `Alpha=5*[0]` создает список из пяти нулей.

ПОДРОБНОСТИ

Значение выражения `5*[0]` вычисляется так: содержимое умножаемого списка `[0]` (то есть число `0`) повторяется `5` раз. Получается список из пяти нулей.

Примерно такой же результат выполнения команды `Bravo=(1,)*3`. Но только в этом случае создается не список, а кортеж. Он содержит три единицы.

ПОДРОБНОСТИ

Если воспользоваться выражением `(1)*3`, то его значением является число `3` — результат произведения чисел `1` и `3`. Дело в том, что выражение `(1)` интерпретируется как число `1` (в круглых скобках). Для того чтобы создать кортеж из одного элемента, после этого элемента следует поставить запятую — в нашем случае речь об инструкции `(1,)`. Результатом выражения `(1,)*3` является кортеж, который получается троекратным повторением содержимого кортежа `(1,)`.

В результате выполнения команды `Charlie=[1,2]*3` создается список, который получается троекратным повторением содержимого списка `[1,2]` (числа 1 и 2). Но когда выполняется команда `Delta=[[1,2]]*3`, то список получается другой: троекратно повторяется содержимое списка `[[1,2]]`. Этот список содержит всего один элемент, которым является список `[1,2]`. Поэтому в итоге получаем список, состоящий из трех элементов-списков `[1,2]`.

При выполнении команды `Echo=4*(1,[2,3])` получаем кортеж. Его содержимое — это повторенное 4 раза содержимое кортежа `(1,[2,3])`. Данный кортеж содержит два элемента: число 1 и список `[2,3]`. Наконец, имеет смысл сравнить результат выполнения команд `Foxtrot=([1]*2)*3` и `Golf=([1]*2,)*3`. Они похожи, но результаты различны. В обоих

случаях значение выражения `[1]*2` — это список `[1,1]`, который состоит из двух единиц. Выражение `([1,1])` представляет собой список, заключенный в круглые скобки. То есть это список. Поэтому если его умножить на `3` (выражение выглядело бы как `([1,1])*3`), то это все равно, что умножить на `3` список `[1,1]` (команда `[1,1]*3`). В результате создается список, который получается троекратным повторение двух единиц. А вот значение выражения `([1]*2,)` — это кортеж, который состоит из одного элемента `[1,1]` (результат вычисления выражения `[1]*2`). Поэтому результатом выражения `([1,1])*3` является кортеж, который состоит из трех элементов-списков `[1,1]`.

ПОДРОБНОСТИ

Вложенные списки (списки, элементами которых являются списки), созданные с помощью оператора умножения `*`, являются достаточно «коварными». Например, если мы создадим список с помощью команды `A=[[0]*3]*2`, то получим список, который состоит из двух элементов-списков `[0,0,0]`. То есть переменная `A` ссылается (именно ссылается!) на список `[[0,0,0],[0,0,0]]`. Этот список можно отождествлять с таблицей, в которой две строки, и в каждой строке по три элемента. Здесь все ожидаемо. Сюрпризы начинаются дальше. Допустим, мы командой `A[0][1]=1` меняем значение второго элемента в первой строке. По идее должен был бы получиться список `[[0,1,0],[0,0,0]]`. На самом деле получаем список `[[0,1,0],[0,1,0]]`. В чем причина такого «странного» результата? Чтобы происходящее не казалось произволом, следует учесть важное обстоятельство: если переменной в качестве значения присваивается список, то в действительности в переменную записывается ссылка на список. Например, значением переменной `A`, как отмечалось выше, является не список, а ссылка (адрес в памяти) списка. Но когда мы обращаемся к переменной `A`, то автоматически по ссылке получаем доступ к списку. Это общее правило: доступ к списку получаем через ссылку. Теперь посмотрим, что происходит, когда командой `A=[[0]*3]*2` создается список. Результатом выражения `[0]*3` является список `[0,0,0]`. Фактически этот список умножается на `2`, в результате чего получается список из двух элементов `[0,0,0]`. То есть элемент `[0,0,0]` повторяется дважды. Это «повторение» реализуется через копирование элемента `[0,0,0]`. Но доступ к списку `[0,0,0]` получаем через ссылку на него. Поэтому копируется не список, а ссылка на него. В итоге получается, что список `A` состоит из двух элементов, и каждый из элементов является ссылкой на список `[0,0,0]`. Причем это один и тот же список, а не два одинаковых. Причина,

еще раз, в том, что при обычном копировании списка копируется не список, а ссылка на него. Эта специфика языка Python имеет прямое отношение к созданию копий списков. Она обсуждается далее.

Стоит также заметить, что при создании списка, например, командой `B=[0]*3` такой проблемы нет: результатом является список `[0,0,0]`, и изменение значения любого из элементов на прочие элементы не влияет. Все потому, что элементы списка в данном случае числа, а не списки.

НА ЗАМЕТКУ

Список может быть создан и в результате выполнения операции множественного присваивания. Если в команде множественного присваивания количество переменных слева от оператора присваивания меньше количества значений, указанных справа от операции присваивания и при этом одна из переменных помечена звездочкой `*`, то такой переменной в качестве значения присваивается список, в который «упакованы» все «лишние» значения из правой части. Другими словами, часть переменных в правой части автоматически включается в список так, чтобы команда присваивания имела смысл и количество переменных совпадало с количеством значений. Например при выполнении команды `A,*B,C=1,2,3,4,5,6` переменная `A` получает значение `1`, переменная `C` получает значение `6`, а переменной `B` в качестве значения присваивается список `[2,3,4,5]`.

Вложенные списки и кортежи

> Братцы, кончайте философствовать, он сейчас возникнет уже.
>
> *Из к/ф «Кин-дза-дза»*

Списки в языке Python играют важную роль. Это структуры достаточно гибкие. С их помощью можно реализовать много интересных и даже экзотических идей. Здесь мы рассмотрим ситуацию, когда список состоит из списков (то есть элементами списка являются списки). Такие списки будем называть *вложенными*. Небольшой пример, в котором создаются и используются вложенные списки, представлен в листинге 3.6.

Листинг 3.6. Создание вложенных списков

```
# Импорт функций для генерирования случайных чисел:
from random import *
# Функция для отображения вложенного списка:
def show(A):
    for a in A:
        for s in a:
            print(s, end=" ")
        print()
# Функция для создания вложенного списка
# из случайных чисел:
def rands(m, n):
    res=[[randint(0,9) for i in range(n)] for j in range(m)]
    return res
# Функция для создания вложенного списка из букв:
def symbs(m, n):
    val='A'
    res=[['' for i in range(n)] for j in range(m)]
    for i in range(m):
        for j in range(n):
           res[i][j]=val
           val=chr(ord(val)+1)
    return res
# Создание вложенного списка:
A=[[(j+1)*10+i+1 for i in range(5)] for j in range(3)]
print("Список A:")
# Отображение вложенного списка:
show(A)
# Инициализация генератора случайных чисел:
seed(2019)
# Список случайных чисел:
B=rands(3,4)
```

```
print("Список B:")
# Отображение вложенного списка:
show(B)
# Список с буквами:
C=symbs(3,5)
print("Список C:")
# Отображение вложенного списка:
show(C)
# Список определяет количество строк во вложенном списке:
size=[3,5,4,6]
# Создание вложенного списка:
D=[['*' for k in range(s)] for s in size]
print("Список D:")
# Отображение вложенного списка:
show(D)
```

Результат выполнения программы ниже.

Результат выполнения программы (из листинга 3.6)

```
Список A:
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
Список B:
2 3 7 2
3 3 5 9
4 6 9 3
Список C:
A B C D E
F G H I J
K L M N O
Список D:
* * *
```

```
* * * * *
* * * *
* * * * * *
```

Код в принципе простой, но в нем есть несколько моментов, достойных внимания. Во-первых, мы в программе собираемся генерировать случайные числа, поэтому нам понадобится несколько функций из модуля `random`. Чтобы использовать данные функции, необходимо подключить модуль, в котором они находятся. Для этого в программном коде использована инструкция `from random import *`. После ключевого слова `from` указывается название модуля (в данном случае `random`), из которого выполняется импорт. Затем следует ключевое слово `import` и название импортируемой функции. Если указать звездочку `*`, как в нашем случае, то это означает, что импортируются все функции из модуля.

ПОДРОБНОСТИ

Некоторые функции доступны в программном коде автоматически, по умолчанию. Другие функции, чтобы стать доступными, должны импортироваться. В таком случае используется `import`-инструкция. Существует несколько форматов подключения модулей. Например, после ключевого слова `import` можно просто указать название импортируемого модуля (например, `import random`). В таком случае перед именем функции из модуля следует указывать название модуля. Скажем, если мы хотим вызвать функцию `seed()` с аргументом `2019`, то команда выглядела бы как `random.seed(2019)`. Если воспользоваться командой импорта вида `from модуль import функция`, то `функцию` можно использовать в программе, и название `модуля` можно не указывать. Если вместо названия функции указать звездочку `*`, то доступными станут все функции из `модуля`.

Нас на самом деле интересует две функции из модуля `random`: функция `randint()`, используемая для генерирования случайных целых чисел, и функция `seed()`, которую мы используем для инициализации генератора случайных чисел.

ПОДРОБНОСТИ

В действительности компьютер случайные числа генерировать не может. Числа, которые генерирует компьютер, являются псевдослучайными — впечатление такое, что они случайные, но на самом деле числа образуют некоторую последовательность. Для

генерирования этой последовательности нужна «затравка» — некоторое число, которое будет использовано в вычислениях. Данный процесс называется инициализацией генератора случайных чисел. Как правило, конкретное значение не принципиально. Важно то, что если вы указываете одно и то же «затравочное» число, то последовательность «случайных» чисел получается одна и та же.

Какое число мы указали в качестве аргумента функции `seed()` — не важно. Важно, что если аргумент функции `seed()` один и тот же при разных запусках программы, то генерироваться будет одна и та же последовательность псевдослучайных чисел.

В программе описано несколько вспомогательных функций. Функция `show()` предназначена для отображения содержимого вложенных списков. Предполагается, что аргумент функции (обозначен как `A`) является вложенным списком: это список, элементами которого являются списки. В теле функции с помощью двух вложенных операторов цикла перебираются элементы списка. В частности, переменная `a` во внешнем операторе цикла принимает значения элементов из списка `A` (аргумент функции). Мы исходим из того, что элементами списка `A` являются списки, с помощью которых реализуются строки матрицы (если отождествлять вложенный список с матрицей). Поэтому переменную `a` интерпретируем как список, и элементы этого списка перебираются во внутреннем операторе цикла (переменная `s` принимает значения из списка `a`). Во внутреннем операторе цикла командой `print(s, end=" ")` отображается значение элемента, а после завершения внутреннего оператора цикла командой `print()` выполняется переход к новой строке.

Функция `rands()` предназначена для создания вложенного списка из случайных чисел. У функции два аргумента (`m` и `n`), которые определяют количество строк и столбцов в матрице. Элементы матрицы, как отмечалось, являются случайными числами. Вложенный список создается командой `res=[[randint(0,9) for i in range(n)] for j in range(m)]`, в котором значение элемента вычисляется инструкцией `randint(0,9)`. Значение этого выражения — случайное целое число в диапазоне от `0` до `9` включительно.

ПОДРОБНОСТИ

Функция `randint()` в качестве результата возвращает целое случайное число, равномерно распределенное на интервале, границы которого определяются аргументами функции.

Команда `return res` возвращает ссылку на созданный список в качестве результата функции.

Функция `symbs()` предназначена для создания вложенного списка из букв. Как и в предыдущем случае, аргументы функции определяют количество строк и столбцов в матрице. В теле функции командой `res=[['' for i in range(n)] for j in range(m)]` создается вложенный список соответствующего размера, а элементы списка — пустые текстовые значения. Затем запускаются вложенные операторы цикла, в которых переменная `i` определяет индекс строки, а переменная `j` определяет индекс элемента в троке. Командой `res[i][j]=val` элементу массива присваивается новое значение — это значение переменной `val`. Начальное значение переменной — символ `'A'`. И за каждый цикл командой `val=chr(ord(val)+1)` в переменную `val` записывается следующий символ в кодовой таблице.

ПОДРОБНОСТИ

С помощью функции `ord()` определяется код символа в кодовой таблице. Символ (переменная, значением которой является символ) передается аргументом функции `ord()`.

Функция `chr()` позволяет по коду символа определить символ. Другими словами, если в качестве аргумента функции `chr()` указать код символа, то результатом функция возвращает символ.

Выражение `chr(ord(val)+1)` вычисляется так. Определяется код символа, который является текущим значением переменной `val`. К этому коду прибавляется `1`. В итоге получается код следующего символа в кодовой таблице. Полученное значение (код нового символа) передается аргументом функции `chr()`, в результате чего получаем новый символ (следующий символ в кодовой таблице). Это новое значение записывается в переменную `val`.

Здесь также стоит отметить, что буквы в кодовой таблице размещены подряд, в соответствии с тем, как они размещены в алфавите.

В созданный список последовательно (построчно) записываются символы из кодовой таблицы, а по окончании этого процесса ссылка на список возвращается результатом функции (команда `return res`).

В программе мы создаем четыре вложенных списка. Список `A` создается командой `A=[[(j+1)*10+i+1 for i in range(5)] for j in range(3)]` без привлечения специальных функций. В данном случае значения элементов списка вычисляются явно (выражение `(j+1)*10+i+1`) с использованием значений индексов `i` и `j`.

Командой `B=rands(3,4)` создается список из случайных чисел, а командой `C=symbs(3,5)` создается список, заполненный буквами. Еще один список содержит в строках разное количество элементов. Мы используем числовой список `size=[3,5,4,6]` для определения количества элементов в каждой из строк вложенного списка. Список создается командой `D=[['*' for k in range(s)] for s in size]`. Он содержит в качестве значений элементов символ `'*'` (звездочка). В команде создания списка переменная `s` принимает значения элементов из списка `size` (числа 3, 5, 4 и 6). При каждом значении переменной `s` переменная `k` принимает значения от `0` до `s-1` включительно. Для каждого значения `k` элемент вычисляется выражением `'*'`.

Для отображения содержимого вложенных списков использована функция `show()`.

Копирование списков и кортежей

> Они нам гравицапу дают, а мы организуем взаимовыгодную торговлю.
>
> *Из к/ф «Кин-дза-дза»*

Далее мы рассмотрим вопросы, связанные с присваиванием списков и созданием копий списков. Ситуация здесь не самая тривиальная, в чем у нас уже была возможность убедиться. Чтобы понять суть проблемы, рассмотрим небольшой пример, представленный в листинге 3.7.

Листинг 3.7. Присваивание списков

```
# Исходный список:
A=[1,3,5]
# Присваивание списков:
B=A
# Изменение значений элементов:
B[1]="Python"
A[2]=(10,20)
# Проверка результата:
print(A)
print(B)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 3.7)

```
[1, 'Python', (10, 20)]
[1, 'Python', (10, 20)]
```

Мы начинаем вычисления со списком A, у которого вначале три числовых элемента (1, 3 и 5). Команда B=A выполняет копирование значений (переменная B получает значение переменной A). Далее с помощью команд B[1]="Python" (второму элементу в списке B в качестве значения присваивается текст "Python") и A[2]=(10,20) (третьему элементу в списке A значением присваивается кортеж (10,20)) мы меняем значения элементов в списках B и A соответственно. Однако при проверке значений переменных A и B (команды print(A) и print(B)) оказывается, что они (переменные) ссылаются на одинаковые списки. Объяснение простое: это не просто одинаковые списки, это один и тот же список. Как уже упоминалось ранее, списки (и кортежи) относятся к ссылочным типам данных — переменная, в качестве значения которой присваивается список, не содержит этот список как значение, а ссылается на него. Фактически значением переменной является адрес списка в памяти. Поэтому при выполнении команды B=A копия списка A не создается. В переменную B копируется значение переменной A — то есть адрес списка, на который ссылается переменная A. В результате и переменная A, и переменная B будут ссылаться на один и тот же список. Поэтому, изменяя список через переменную A или B, мы изменяем один и тот же список.

Как же тогда создать копию списка? Есть несколько способов, разных по сложности и надежности. Например, можно создать копию списка с помощью среза. Если речь идет о некотором списке A, то выражение A[:] определяет копию этого списка. Также можно воспользоваться методом copy(), который вызывается из объекта списка: для списка A создать копию можно с помощью выражения A.copy().

НА ЗАМЕТКУ

Метод, как и функция, представляет собой именованный блок команд, которые выполняются при вызове метода. Но если функция вызывается «сама по себе» (то есть просто указываются имя функции и аргументы, которые ей передаются), то метод вызывается

из объекта. В данном случае таким объектом является список. При вызове метода используется «точечный» синтаксис: указывается имя объекта, из которого вызывается метод, и, через точку, собственно имя вызываемого метода (с аргументами, если необходимо). Фактически объект, из которого вызывается метод, играет роль «особого» аргумента, подобно аргументу функции.

Правда, в обоих случаях (и использования среза, и вызова метода `copy()`) создается то, что называется *поверхностной копией*. В чем особенность такой поверхностной копии, иллюстрирует пример в листинге 3.8.

Листинг 3.8. Создание поверхностной копии

```
# Исходный список:
A=[1,3,[10,20],"Python",[40,50]]
# Создание поверхностной копии списка:
B=A[:]
C=A.copy()
print("Исходные значения:")
print("A:", A)
print("B:", B)
print("C:", C)
# Внесение изменений в исходный список:
A[0]=[100,200]
A[2][1]=300
A[3]="Java"
A[4]=90
C[4][1]="C++"
print("После внесения изменений:")
print("A:", A)
print("B:", B)
print("C:", C)
```

Ниже представлен результат выполнения программы.

Результат выполнения программы (из листинга 3.8)

```
Исходные значения:
A: [1, 3, [10, 20], 'Python', [40, 50]]
B: [1, 3, [10, 20], 'Python', [40, 50]]
C: [1, 3, [10, 20], 'Python', [40, 50]]
После внесения изменений:
A: [[100, 200], 3, [10, 300], 'Java', 90]
B: [1, 3, [10, 300], 'Python', [40, 'C++']]
C: [1, 3, [10, 300], 'Python', [40, 'C++']]
```

Отправной точкой является список, который создается командой `A=[1,3,[10,20],"Python",[40,50]]`. Особенность списка в том, что среди его элементов, кроме прочего, есть списки и текст. Мы создаем две поверхностные копии этого списка командами `B=A[:]` (использован срез) и `C=A.copy()` (использован метод `copy()`). Проверка показывает, что все три списка одинаковы. Но далее мы вносим изменения в исходный список `A` (и один раз в список `C`). Нас интересует, отразятся ли эти изменения на списках `B` и `C`. В частности, команда `A[0]=[100,200]` присваивает первому элементу в списке `A` новое значение (список `[100,200]`). Данное изменение на списках `B` и `C` не сказывается. Это ожидаемо, поскольку при выполнении команд `B=A[:]` и `C=A.copy()` создаются копии списка `A`. Командой `A[2][1]=300` изменяется значение второго элемента в списке, который является третьим элементом списка `A`. Именно это изменение сказывается и на списке `B`, и на списке `C`. В чем причина? Причина в том, что третий элемент списка `A` — это список. Технически значением элемента `A[2]` является адрес списка `[10,20]`. Когда создаются копии, элементы `B[2]` и `C[2]` получают такое же значение, что и у элемента `A[2]`. Поэтому во всех трех списках третий элемент ссылается на один и тот же список. Если командой `A[2][1]=300` меняется один из элементов этого списка, то изменение «ощущают» все три списка `A`, `B` и `C`. Собственно, поэтому созданные нами копии являются «поверхностными» — при создании копии не принимается в расчет, что некоторые элементы могут быть списками (в более общем случае — относятся к ссылочным типам). Как следствие, копируются адреса таких объектов, а не создаются копии.

Ситуацию подтверждают и прочие команды, которыми вносятся изменения в списки. Например, командой `A[3]="Java"` меняется значение

четвертого элемента списка A, а командой A[4]=90 меняется значение пятого элемента в списке. На списках B и C эти команды не сказываются, поскольку соответствующие элементы в данных списках создавались копированием. А вот команда C[4][1]="C++" скажется на списке B. Причина в том, что при создании копии элемента A[4] копировался адрес списка [40,50], на который изначально ссылался элемент A[4]. В результате все три элемента A[4], B[4] и C[4] ссылаются на один и тот же список. После выполнения команды A[4]=90 элемент A[4] получает новое значение, а элементы B[4] и C[4] по-прежнему ссылаются на список [40,50]. Когда выполняется команда C[4][1]="C++", в этом списке меняется значение одного элемента, и проверка списков B и C подтверждает это.

В отличие от поверхностной копии, при создании *полной копии* копируются значения элементов, которые относятся к ссылочным типам. Создать полную копию можно с помощью функции deepcopy() из модуля copy. В листинге 3.9 представлен пример создания полной копии. Это вариация на тему предыдущего примера, только на этот раз использована функция deepcopy().

Листинг 3.9. Создание полной копии

```
# Импорт функции для создания полной копии:
from copy import deepcopy
# Исходный список:
A=[1,3,[10,20],"Python",[40,50]]
# Создание полной копии списка:
B=deepcopy(A)
print("Исходные значения:")
print("A:", A)
print("B:", B)
# Внесение изменений в исходный список:
A[0]=[100,200]
A[2][1]=300
A[3]="Java"
A[4]=90
print("После внесения изменений:")
```

```
print("A:", A)
print("B:", B)
```

Как выглядит результат выполнения программы, показано ниже.

Результат выполнения программы (из листинга 3.9)

```
Исходные значения:
A: [1, 3, [10, 20], 'Python', [40, 50]]
B: [1, 3, [10, 20], 'Python', [40, 50]]
После внесения изменений:
A: [[100, 200], 3, [10, 300], 'Java', 90]
B: [1, 3, [10, 20], 'Python', [40, 50]]
```

Основное отличие от предыдущего случая в том, что даже для внутренних списков создаются копии. Все это благодаря использованию функции `deepcopy()`.

НА ЗАМЕТКУ

В модуле `copy` также есть функция `copy()`, позволяющая создавать поверхностные копии списков.

Функции и методы для работы со списками

> Друг, у вас какая система? Разрешите взглянуть?
>
> *Из к/ф «Кин-дза-дза»*

Для работы со списками предназначены различные встроенные функции и методы. Многие из них позволяют выполнять операции, которые в принципе могут быть выполнены с помощью основных арифметических операторов. Тем не менее использование специальных функций и методов обычно значительно упрощает ситуацию. Далее мы познакомимся с некоторыми из них.

НА ЗАМЕТКУ

Напомним, что кортежи относятся к неизменяемым типам данных — после создания кортежа его изменить нельзя. Это обстоятельство накладывает серьезное ограничение на спектр операций, выполняемых ими. Поэтому далее мы в основном будем говорить о списках. Хотя кортежи также будут упоминаться — но исключительно в контексте операций, которые не изменяют исходный кортеж.

Например, есть такие методы, как `insert()`, `append()` и `extend()`. Методы вызываются из объекта списка и не возвращают результат. Метод `insert()` используется для вставки в список нового элемента. Позиция (индекс) вставки нового элемента и сам элемент (его значение) передаются в качестве аргументов методу. Вставка элемента выполняется в тот список, из которого вызывается метод. Если элемент добавляется в конец списка, то можно воспользоваться методом `append()`. Метод вызывается из списка, в который выполняется вставка. Добавляемый элемент указывается аргументом метода.

Метод `extend()` позволяет добавить в конец списка, из которого вызывается метод, элементы списка, указанного аргументом метода `extend()`.

НА ЗАМЕТКУ

Если из списка вызывается метод `append()` и в качестве аргумента метода указан список, то этот список будет добавлен как элемент в исходный список (из которого вызывался метод). Если вместо метода `append()` в такой ситуации использовать метод `extend()`, то в исходный список добавляются элементы того списка, который указан аргументом метода. Например, если список `A` создается командой `A=[1,2]`, то после выполнения команды `A.append([3,4])` список `A` будет таким: `[1,2,[3,4]]`. А если бы мы вместо команды `A.append([3,4])` использовали инструкцию `A.extend([3,4])`, то список `A` был бы таким: `[1,2,3,4]`.

Метод `pop()` позволяет удалить элемент из списка, из которого вызывается метод. Метод возвращает результат — это значение удаляемого элемента. Индекс удаляемого элемента указывается в качестве аргумента метода. Если индекс удаляемого элемента не указан, то удаляется последний элемент в списке. Для удаления элемента из списка также можно использовать метод `remove()`. Аргументом методу передается

значение, определяющее удаляемый из списка элемент. Если из некоторого списка вызывается метод `remove()`, то в этом списке будет удален первый слева элемент, значение которого совпадает со значением, переданным аргументом методу. Метод результат не возвращает.

НА ЗАМЕТКУ

Таким образом, метод `pop()` позволяет удалить из списка элемент с определенным индексом, а метод `remove()` позволяет удалить из списка элемент с определенным значением.

Небольшой пример, в котором использованы перечисленные выше методы, представлен в листинге 3.10.

Листинг 3.10. Вставка и удаление элементов

```
# Размер списка:
n=10
# Начальное значение для списка:
A=[1,1]
# Заполнение списка:
for k in range(n-2):
    A.append(A[-1]+A[-2])
# Проверка содержимого списка:
print("A:", A)
# Изменение порядка элементов в списке:
for k in range(len(A)-1):
    A.append(A.pop(-k-2))
# Проверка содержимого списка:
print("A:", A)
# Удаление наибольшего элемента в списке:
A.remove(max(A))
# Проверка содержимого списка:
print("A:", A)
# Удаление наименьшего элемента в списке:
A.remove(min(A))
```

```
# Проверка содержимого списка:
print("A:", A)
# Добавление элемента в начало списка:
A.insert(0, A[0]+A[1])
# Проверка содержимого списка:
print("A:", A)
# Пустой список:
B=[]
# Часть элементов одного списка переносится в другой:
for k in range(len(A)//2):
    B.insert(0, A.pop(-1))
# Проверка содержимого списков:
print("A:", A)
print("B:", B)
# Добавление элемента в конец списка:
A.append(B)
# Проверка содержимого списка:
print("A:", A)
# Удаление последнего элемента в списке:
A.pop(-1)
# Проверка содержимого списка:
print("A:", A)
# Добавление элементов в список:
A.extend(B)
# Проверка содержимого списка:
print("A:", A)
```

Ниже показано, как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 3.10)

```
A: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
A: [55, 34, 21, 13, 8, 5, 3, 2, 1, 1]
A: [34, 21, 13, 8, 5, 3, 2, 1, 1]
```

```
A: [34, 21, 13, 8, 5, 3, 2, 1]
A: [55, 34, 21, 13, 8, 5, 3, 2, 1]
A: [55, 34, 21, 13, 8]
B: [5, 3, 2, 1]
A: [55, 34, 21, 13, 8, [5, 3, 2, 1]]
A: [55, 34, 21, 13, 8]
A: [55, 34, 21, 13, 8, 5, 3, 2, 1]
```

В программе представлено несколько простых команд. Для иллюстрации работы методов мы сначала создаем список с числами Фибоначчи.

НА ЗАМЕТКУ

Напомним, что в последовательности Фибоначчи два первых числа равны единице, а каждое следующее число равно сумме двух предыдущих.

Сначала командой `A=[1,1]` создается список из двух единиц. После этого мы начинаем заполнять список. Для этого использован оператор цикла, в котором несколько раз выполняется команда `A.append(A[-1]+A[-2])`. Что это за команда? Ссылка `A[-1]` соответствует последнему элементу в списке `A`, а ссылка `A[-2]` означает предпоследний элемент в списке `A`. Методом `append()` в конец списка добавляется новый элемент. Получается, что командой `A.append(A[-1]+A[-2])` в конец списка `A` добавляется новый элемент, значение которого равно сумме значений двух последних (на данный момент) элементов списка. Это именно то, что нам нужно.

Еще один оператор цикла используется для изменения порядка элементов в списке. Переменная `k` принимает значения от `0` до `len(A)-2` включительно (верхняя граница на `2` меньше количеств элементов в списке). Командой `A.append(A.pop(-k-2))` из списка удаляется элемент и добавляется в конец списка. После завершения оператора цикла все элементы будут размещены в обратном порядке.

ПОДРОБНОСТИ

Командой `A.pop(-k-2)` из списка `A` удаляется элемент с индексом `-k-2`. Индекс отрицательный, поэтому позиция элемента отсчитывается с конца списка. Речь об элементе, порядковый номер

которого k+2 с конца списка. Поскольку k принимает значения, начиная с нуля, то справа удаляется 2-й, 3-й (и так далее) элементы, вплоть до первого слева элемента. Значение выражения A.pop(-k-2) — это значение удаленного элемента. Поскольку это значение указано в качестве аргумента метода append(), то удаленный элемент добавляется в конец списка.

НА ЗАМЕТКУ

Для того чтобы изменить (на обратный) порядок элементов в списке можно использовать методы reverse() или функцию reversed(). Метод reverse() вызывается (без аргументов) из списка, который следует инвертировать (изменить порядок элементов). Метод не возвращает результат, а в списке, из которого вызывался метод, изменяется порядок элементов.

При вызове функции reversed() в качестве аргумента ей передается список, который не меняется. Функция возвращает результатом итерируемый объект, который соответствует инвертированному списку. Чтобы получить сам список, этот итерируемый объект можно передать аргументом функции list(). А если передать итерируемый объект в качестве аргумента функции tuple(), то получим кортеж.

Командой A.remove(max(A)) из списка A удаляется элемент с наибольшим значением, а командой A.remove(min(A)) удаляется элемент (первый из них) с наименьшим значением. Здесь мы воспользовались функциями max() и min(), предназначенными для определения значения наибольшего и, соответственно, наименьшего элемента в списке, переданном аргументом функции.

Командой A.insert(0, A[0]+A[1]) в начало списка A (позиция с индексом 0 — первый аргумент метода insert()) добавляется новый элемент, значение которого равно сумме значений первого и второго (на данный момент) элементов списка A.

На следующем этапе командой B=[] создается простой список, и в него «перекочевывают» некоторые элементы из списка A. В частности, мы перебираем половину списка A (значение выражения len(A)//2 — результат целочисленного деления на 2 количества элементов списка A). За каждый цикл выполняется команда B.insert(0, A.pop(-1)), которой из списка A удаляется последний элемент, и этот элемент добавляется в начало списка B. Если затем воспользоваться командой

`A.append(B)`, то список `B` будет добавлен как элемент в конец списка `A` (список `B` при этом не меняется). Удалить этот элемент можно командой `A.pop(-1)`. А вот если воспользоваться командой `A.extend(B)`, то содержимое списка `B` поэлементно добавляется в список `A`.

Есть несколько методов, которые используются для поиска элементов в списке и выполнения иных преобразований. Так, метод `index()` позволяет определить индекс элемента в списке. Значение элемента передается аргументом методу. Результатом метод возвращает индекс первого вхождения элемента в список. Если такого элемента нет, то возникает ошибка. Можно передать методу `index()` два аргумента. Тогда второй аргумент определяет индекс позиции, начиная с которой выполняется поиск.

Метод `count()` позволяет определить количество элементов с определенным значением в списке, из которого вызывается метод. Для сортировки элементов массива можно использовать метод `sort()` или функцию `sorted()`. Если из списка вызвать метод `sort()` (без аргументов или с аргументом `reverse=False`), то список будет отсортирован в порядке возрастания. Чтобы список сортировался в порядке убывания, метод вызывают с аргументом `reverse=True`. Метод не возвращает результат.

Функции `sorted()` при вызове в качестве аргумента передается список (который в результате вызова функции не меняется). Результатом функция возвращает новый отсортированный в порядке возрастания список. Если функции передать еще один аргумент `reversed=True`, то список-результат будет отсортирован в порядке убывания.

ПОДРОБНОСТИ

При вызове функций и методов аргументы обычно указываются в круглых скобках через запятую. Аргументы при этом следуют в той последовательности, как они описывались в функции. В этом случае говорят о позиционной передаче аргументов. Но есть еще один способ передачи аргументов — по ключу. В таком случае указывается название аргумента и, через знак равенства, его значение. Данный механизм обычно используют, когда часть необязательных аргументов функции или методу не передаются. Передача аргументом инструкций вида `reverse=True` или `reverse=False` как раз является примером передачи аргумента по ключу.

Программа, в которой используются описанные выше методы и функции, представлена в листинге 3.11.

Листинг 3.11. Поиск и подсчет количества элементов

```
# Подключение функций для генерирования случайных чисел:
from random import *
# Инициализация генератора случайных чисел:
seed(2019)
# Создание списка из случайных чисел:
A=[randint(10,20) for k in range(15)]
# Содержимое списка:
print("A:", A)
# Подсчет элементов с разными значениями:
for a in range(min(A), max(A)+1):
    print(a,"-", A.count(a))
# Наибольшее, наименьшее и среднее значения:
print("Наименьший:")
print("A[", A.index(min(A)),"]=", min(A), sep="")
print("Наибольший:")
print("A[", A.index(max(A)),"]=", max(A), sep="")
print("Среднее:", sum(A)/len(A))
# Сортировка списка:
B=sorted(A)
A.sort(reverse=True)
# Проверка содержимого списков:
print("A:", A)
print("B:", B)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 3.11)

```
A: [12, 13, 17, 12, 13, 20, 13, 20, 15, 20, 19, 20, 14, 16, 19]
12—2
13—3
14—1
```

```
15—1
16—1
17—1
18—0
19—2
20—4
Наименьший:
A[0]=12
Наибольший:
A[5]=20
Среднее: 16.2
A: [20, 20, 20, 20, 19, 19, 17, 16, 15, 14, 13, 13, 13, 12, 12]
B: [12, 12, 13, 13, 13, 14, 15, 16, 17, 19, 19, 20, 20, 20, 20]
```

В программе создается список `A` со случайными числами. Затем выполняется подсчет элементов с разными значениями. Для этого запускается оператор цикла, в котором переменная `a` принимает значения от `min(A)` (наименьшее значение в списке) до `max(A)` (наибольшее значение в списке). Количество элементов в списке `A` с данным значением `a` вычисляется инструкцией `A.count(a)`.

Далее вычисляется наибольшее и наименьшее значения элементов в списке (и их позиция), а также вычисляется среднее значение элементов в списке (сумма значений всех элементов, деленная на количество элементов в списке).

Индекс элемента с наименьшим значением вычисляется инструкцией `A.index(min(A))`, а для вычисления индекса элемента с наибольшим значением использована инструкция `A.index(max(A))`. Для вычисления суммы значений элементов в списке мы использовали инструкцию `sum(A)`, а количество элементов в списке вычисляется выражением `len(A)`.

ПОДРОБНОСТИ

Одним из аргументов функции `print()` указана инструкция `sep=""`. Это аргумент, переданный по ключу. Данный аргумент определяет разделитель, который автоматически вставляется между значениями аргументов, переданных функции `print()` через запятую. По умолчанию таким разделителем является пробел. Другими

словами, если вызвать функцию `print()` с несколькими аргументами (предназначенными для отображения), разделенными запятой, то при отображении значений этих аргументов между ними автоматически добавляется пробел. Если вместо пробела мы хотим использовать в подобной ситуации другой разделитель, то соответствующее значение для разделителя указывается с помощью ключа `sep` (в формате `sep=значение`), что и было сделано выше.

На следующем этапе выполняется сортировка списка. Командой `B=sorted(A)` на основе списка `A` сортировкой в порядке возрастания создается новый список, и ссылка на него записывается в переменную `B`. При этом список `A` не меняется. А вот при выполнении команды `A.sort(reverse=True)` сортируется список `A`, причем из-за наличия аргумента `reverse=True` сортировка выполняется в порядке убывания значений элементов.

Резюме

> Только быстро. А то одна секунда здесь — это полгода там!
>
> *Из к/ф «Кин-дза-дза»*

- Кортеж, подобно списку, представляет собой упорядоченный набор элементов. В отличие от списков, кортежи относятся к неизменяемым типам данных: после создания кортежа его нельзя изменить. Это накладывает определенные ограничения на операции, которые могут выполняться с кортежами.
- Чтобы создать кортеж, его элементы перечисляют через запятую. Обычно вся конструкция заключается в круглые скобки. Также для создания кортежа используют функцию `tuple()`. В качестве аргумента функции можно передавать, например, текст или список, на основе которых создается кортеж. Чтобы создать кортеж из одного элемента, после этого единственного элемента ставится запятая.
- Кортежи можно индексировать, и для кортежей выполняется срез. Правила такие же, как и в случае со списками. Отличие от списков в том, что значение элементов можно прочитать, но нельзя присвоить новое значение элементам.
- Хотя кортежи изменять нельзя, на основе уже существующего кортежа можно создать новый кортеж и ссылку на этот новый кортеж

записать в переменную, которая ссылалась на исходный кортеж. В результате создается иллюзия, что кортеж изменился.

- Списки создаются с помощью функции `list()`. Еще можно перечислить элементы списка в квадратных скобках. Полезными могут быть также генераторы последовательностей.
- Доступ к элементу списка или кортежа можно получить, указав индекс элемента в списке/кортеже. Индексация начинается с нуля. Индекс может быть и отрицательным. Отрицательные индексы означают, что индексация выполняется с конца списка/кортежа.
- К спискам и кортежам можно применять оператор сложения +. Если суммируются два списка, то результатом является новый список, который получается объединением элементов из первого и второго списка. Если суммируются два кортежа, то результатом является кортеж, который состоит из элементов обоих кортежей.
- Результатом выражения вида `список[i: j+1]` или `кортеж[i: j+1]` является список/кортеж, который состоит из элементов исходного списка/кортежа с индексами от `i` до `j` включительно. В инструкции получения среза можно указать, через двоеточие, третий параметр, который будет определять шаг дискретности по индексу. Отрицательный третий параметр означает, что элементы в срез включаются с конца списка/кортежа в начало. Если не указать параметр, определяющий начальный/конечный индекс элементов для включения в срез, то в соответствии с контекстом команды используется первый/последний элемент списка/кортежа.
- Если срезу присвоить в качестве значения список, то элементы, соответствующие срезу, будут удалены, а на их место вставлены элементы из присваиваемого списка. Если срезу вида `список[i: i]` с совпадающими индексами `i` в качестве значения присваивается список, то элементы этого присваиваемого списка будут вставлены в исходный список в позицию с индексом `i`. При этом элементы исходного списка не удаляются, а сдвигаются вправо (в направлении увеличения индекса).
- Если список/кортеж умножить на число, то результатом является список/кортеж, получающийся повторением элементов исходного (умножаемого на число) списка/кортеж. Количество повторений определяется числом, на которое умножается список/кортеж.
- При присваивании списков копия не создается — получаем две переменные, которые ссылаются на один и тот же список. Для создания поверхностной копии списка выполняют срез в формате `список[:]`

или используют метод `copy()`. Для создания полной копии используется функция `deepcopy()` из модуля `copy`.

- Для работы со списками есть встроенные функции и методы. Метод `append()` позволяет добавить элементы одного списка в конец другого списка. Метод `extend()` используется для добавления списка элементом в другой список. Метод `insert()` позволяет вставить элемент в указанную позицию в списке. Методом `pop()` элемент с указанным индексом удаляется из списка. Удалить из списка элемент с определенным значением можно с помощью метода `remove()` (также для удаления элементов используют инструкцию `del`). Определить индекс элемента с определенным значением можно с помощью метода `index()`. Для подсчета количества элементов с указанным значением используют метод `count()`. Общее количество элементов в списке/кортеже определяют с помощью функции `len()`. Можно определить элемент с наименьшим значением (функция `min()`), наибольшим значением (функция `max()`), сумму значений элементов (функция `sum()`). Для сортировки элементов в списке используют метод `sort()` или функцию `sorted()`, а для изменения порядка элементов в списке применяют метод `reverse()` или функцию `reversed()`.

Задания для самостоятельной работы

— Своими мозгами надо играть.
— Как он может своими мозгами играть, когда он эти куклы первый раз видит?

Из к/ф «Кин-дза-дза»

1. Напишите программу, в которой на основе текста, введенного пользователем, создается кортеж. Затем на основе этого кортежа создается новый кортеж. В новый кортеж включаются равноотстоящие элементы, начиная с первого (с нулевым индексом). Например, в новый кортеж включаются элементы, отстоящие друг от друга на `3` позиции (элементы с индексами `0`, `3`, `6`, `9` и так далее). Расстояние между элементами (приращение по индексу) вводится пользователем.

2. Напишите программу, в которой пользователь вводит целое число, а программа формирует кортеж, который состоит из цифр, входящих в это число. Предложите способы создания кортежа, при котором

цифры, формирующие число, включаются в кортеж в прямом и обратном порядке.

3. Напишите программу с функцией, которая создает вложенный список. Размеры списка указываются аргументами функции. Список заполняется случайными буквами.

4. Напишите программу, в которой есть функция для заполнения вложенного списка. Список заполняется натуральными числами «змейкой»: сначала заполняется первая строка, затем последний столбец (сверху вниз), последняя строка (справа налево), первый столбец (снизу вверх), вторая строка (слева направо), и так далее.

5. Напишите программу, в которой создается вложенный список из случайных чисел. В матрице, которая реализуется данным вложенным списком, удаляется строка и столбец. Номер строки и номер столбца, которые нужно удалить, вводятся пользователем.

6. Напишите программу, в которой выполняется сортировка списка (в порядке возрастания) методом пузырька. Метод такой: последовательно сравниваются значения соседних элементов, и если значение элемента слева больше значения элемента справа, элементы меняются местами. За один полный перебор элементов в списке элемент с самым большим значением оказывается последим в списке. За второй перебор предпоследним оказывается элемент со вторым по величине значением и так далее.

7. Напишите программу с функцией, которая для списка, переданного аргументом, возвращает список из двух элементов: значение наибольшего элемента в списке и индекс этого элемента в списке (если таких элементов несколько, то индекс первого из таких элементов).

8. Напишите программу, в которой создается числовой список. Список заполняется случайными числами. Затем элементы с четными индексами сортируются в порядке возрастания, а элементы с нечетными индексами сортируются в порядке убывания.

9. Напишите программу, в которой создается числовой список. Список заполняется случайными числами. Затем между каждой парой элементов этого списка вставляется новый элемент, равный сумме значений соседних элементов.

10. Напишите программу, в которой создается два списка одинакового размера. На основе этих списков поочередной вставкой элементов из первого и второго списка формируется новый список.

Глава 4
МНОЖЕСТВА И СЛОВАРИ

> Вообще-то я не специалист по этим гравицапам.
>
> *Из к/ф «Кин-дза-дза»*

В этой главе мы познакомимся еще с двумя типами данных, которые можно назвать множественными. Причина в том, что речь идет о наборе значений, которые сохраняются в одной «конструкции». А именно, мы познакомимся с множествами и словарями. В некотором смысле эти типы данных напоминают списки и кортежи, но не совсем — есть принципиальные отличия.

Знакомство с множествами

> Если у меня много КЦ есть, я имею право носить малиновые штаны.
>
> *Из к/ф «Кин-дза-дза»*

Множество — это неупорядоченный набор уникальных элементов. Не имеет значения, в каком порядке элементы добавлялись в множество, — принципиально только, входит ли элемент в множество. Этим множество отличается от списка, в котором важно не только значение элемента, но и его позиция в списке.

> **НА ЗАМЕТКУ**
>
> Таким образом, непосредственно из определения множества следует, что в множестве не может быть двух элементов с одинаковыми значениями. Причина в том, что нет физической возможности различить два элемента с одинаковыми значениями, поскольку такое понятие, как «позиция элемента в множестве», отсутствует. Если в некоторое множество попытаться добавить элемент со значением,

которое уже представлено в множестве, то такая ситуация интерпретируется как попытка добавить уже существующий элемент. В итоге ничего не происходит — множество остается неизменным.

Хотя на первый взгляд может показаться, что множество по функциональным возможностям значительно уступает списку, на самом деле есть класс прикладных задач, решение которых значительно упрощается в случае использования множеств.

Для создания множества его элементы следует перечислить (через запятую) в фигурных скобках. Например, командой `A={1,2,3}` создается множество `A` из трех элементов `1`, `2` и `3`. При этом если среди перечисленных элементов встречается несколько с одинаковыми значениями, то такие значения интерпретируются как один элемент. Скажем, если мы создаем множество командой `A={1,2,3,2,1}`, то, как и в предыдущем случае, получим множество `A` из трех элементов `1`, `2` и `3`. Также не имеет значения порядок, в котором указаны элементы при создании множества.

НА ЗАМЕТКУ

Строго говоря, порядок перечисления элементов при создании множества может иметь некоторое значение, если речь идет об автоматическом преобразовании значений. Например, логическое значение `True` при создании множества отождествляется с числовым значением `1`, а логическое значение `False` отождествляется с числовым значением `0`. Поэтому если мы создадим множество командой `A={True,3,1,2,0,  False}`, а затем проверим содержимое множества `A`, то получим элементы `0`, `True`, `2` и `3`. Объяснение такое: значения `1` и `True`, а также `0` и `False` интерпретируются как один элемент. Значение `True` указано раньше, чем значение `1`, поэтому остается `True`. Значение `0` указано раньше, чем значение `False`, поэтому остается значение `0`.

Для проверки того, принадлежит ли элемент некоторому множеству, используют оператор `in`. Соответствующая команда имеет вид `элемент in множество`. Значение этого выражения равно `True`, если `элемент` принадлежит `множеству`, и `False` в противном случае. При этом, в отличие от списка, индексация множества не допускается — мы не можем узнать значение элемента множества, указав после имени множества индекс. Объяснение простое и связано с тем, что, как отмечалось выше, нет смысла говорить о позиции (индексе) элемента в множестве. Вместе

с тем количество элементов в множестве можно узнать с помощью функции `len()`.

НА ЗАМЕТКУ

Для проверки того, что элемент не принадлежит множеству, используется оператор `not in`. Значением выражения вида `элемент not in множество` является значение `True`, если `элемента` в `множестве` нет. В противном случае (если `элемент` в `множестве` есть) выражение возвращает значение `False`.

Также множество создают с помощью функции `set()`. В качестве аргумента можно передать список, кортеж или текст (а также множество). В результате вызова функции `set()` на основе переданного аргумента создается множество.

ПОДРОБНОСТИ

Если вызвать функцию `set()` без аргументов, будет создано пустое множество. А если использовать пустые фигурные скобки, то будет создан пустой словарь. Причина в том, что фигурные скобки используются не только для создания множеств, но и для создания словарей, о которых речь пойдет далее.

Программа, в которой создаются множества, представлена в листинге 4.1.

Листинг 4.1. Создание множеств

```
# Создание множеств:
A={10,50,20,10,50}
B=set([6,2,6,0,4,0])
C=set("Hello World")
# Проверка содержимого множеств:
print("A:", A)
print("Элементов:", len(A))
print("B:", B)
print("Элементов:", len(B))
print("C:", C)
print("Элементов:", len(C))
```

Результат выполнения программы будет следующим.

Результат выполнения программы (из листинга 4.1)

```
A: {10, 50, 20}
Элементов: 3
B: {0, 2, 4, 6}
Элементов: 4
C: {'W', ' ', 'e', 'd', 'o', 'H', 'l', 'r'}
Элементов: 8
```

Программа очень простая, она содержит всего несколько команд, которыми создаются множества. Для отображения содержимого множества мы передаем ссылку на него в качестве аргумента функции `print()`. Чтобы узнать размер множества (количество элементов), используем функцию `len()`.

Может показаться, что область применения множеств ограничена, в том числе и по причине существования списков, но это не так. Есть задачи, которые удается решить просто и где-то даже эффектно именно за счет использования множеств. В качестве иллюстрации рассмотрим задачу о генерировании случайных чисел. Ранее мы с чем-то подобным сталкивались. Но в тех случаях речь шла о генерировании определенного количества случайных чисел. Здесь мы создадим программу, при выполнении которой генерируется определенное количество *разных* случайных чисел. При этом мы будем использовать множество. Код представлен в листинге 4.2.

Листинг 4.2. Генерирование разных случайных чисел

```
# Подключение функций для генерирования случайных чисел:
from random import *
# Инициализация генератора случайных чисел:
seed(2019)
# Количество разных случайных чисел:
count=10
# Создание пустого множества:
nums=set()
```

```
# Генерирование случайных чисел:
while len(nums)<count:
    nums.add(randint(1, count+5))
# Отображение результата:
print(nums)
```

Результат выполнения программы (с учетом того, что используется генератор случайных чисел) может быть таким.

Результат выполнения программы (из листинга 4.2)

```
{3, 4, 5, 6, 8, 10, 11, 13, 14, 15}
```

Для генерирования случайных чисел подключаем пакет `random` (инструкция `from random import *`). Командой `seed(2019)` выполняется инициализация генератора случайных чисел.

НА ЗАМЕТКУ

Напомним, что конкретное значение аргумента функции `seed()` значения не имеет. Число, указанное аргументом, определяет последовательность, которая будет генерироваться. Инициализируя генератор чисел одним и тем же числом при каждом запуске программы, мы каждый раз получаем одну и ту же последовательность случайных (на самом деле псевдослучайных) чисел.

Количество разных случайных чисел, которые нужно сгенерировать, записываем в переменную `count`. А командой `nums=set()` создается пустое множество (в котором нет элементов). После этого запускается оператор цикла `while`, в котором проверяется условие `len(nums)<count`. Условие истинно, если количество элементов в множестве `nums` меньше значения переменной `count`. В теле оператора цикла командой `nums.add(randint(1, count+5))` генерируется новое случайное число, которое добавляется в множество `nums`. Для добавления элемента в множество из него вызывается метод `add()`. В качестве аргумента методу передается добавляемый элемент. Он вычисляется инструкцией `randint(1, count+5)`. Мы вызываем функцию `randint()` с аргументами `1` и `count+5`. Результатом соответствующей инструкции является случайное число в диапазоне значений от `1` до `count+5` включительно (при значении `10` для переменной `count`

получается, что генерируется число в диапазоне от 1 до 15). Здесь следует учесть, что если в множество добавляется число, которое в нем уже есть, то в действительности новый элемент не добавляется. Именно это обстоятельство позволяет утверждать, что множество будет содержать разные числа.

После того как множество сформировано, командой `print(nums)` отображается его содержимое.

Множество — структура достаточно специфическая и имеет свои особенности, которые во многом объясняются функциональными возможностями и назначением множества. Так, элементами множества могут быть только значения неизменяемого типа. Это означает, что список не может быть элементом множества (но зато элементом множества может быть кортеж). Также элементом множества не может быть множество.

НА ЗАМЕТКУ

Поскольку элементы множества относятся к неизменяемым типам, то нет необходимости создавать полную копию множества — достаточно поверхностной, которую можно создать с помощью метода `copy()` (метод вызывается из множества, для которого создается копия).

Также можно создавать неизменяемые множества — это множества, которые после создания изменить нельзя. Для создания неизменяемого множества вместо функции `set()` используется функция `frozenset()`.

Операции с множествами

— Надо снова проверять...
— Ну проверяй.

Из к/ф «Кин-дза-дза»

Мы уже знаем, что добавить элемент в множество можно с помощью метода `add()`. Обратная операция — удаление элемента из множества — выполняется с помощью методов `remove()` и `discard()`. Методы вызываются из множества, а аргументом передается удаляемый элемент.

ПОДРОБНОСТИ

Различие между методами `remove()` и `discard()` проявляется при попытке удалить из множества элемент, которого в множестве нет. Если используется метод `remove()`, то в таком случае возникает ошибка. Если же используется метод `discard()`, то не происходит ничего.
Стоит отметить, что в Python возникающие в ходе выполнения ошибки можно перехватывать и обрабатывать. Поэтому тот факт, что метод при определенных обстоятельствах генерирует ошибку, вовсе не означает, что данный метод плох. Скорее наоборот — это позволяет создавать в программе блоки, выполняемые в случае возникновения определенных обстоятельств.

С помощью метода `clean()` можно выполнить полную очистку множества. Но эти все операции имеют отношение к добавлению элементов в множество или удалению элементов из множества. Кроме них есть теоретико-множественные операции, в которых операндами выступают множества и результатом является новое множество.

- Например, есть такая операция, как *объединение множеств*. Если объединяются множества `A` и `B`, то результатом является множество, которое состоит из элементов, входящих хотя бы в одно из множеств `A` или `B`. Для вычисления объединения множеств можно использовать оператор `|` или метод `union()` (инструкции вида `A|B` и `A.union(B)`).
- При вычислении *пересечения множеств* `A` и `B` результатом является множество, которое состоит из элементов, входящих одновременно как в множество `A`, так и в множество `B`. Для вычисления пересечения множеств используется оператор `&` или метод `intersection()` (инструкции вида `A&B` и `A.intersection(B)`).
- *Разностью множеств* `A` и `B` называется множество, содержащее только те элементы множества `A`, которых нет в множестве `B`. Разность множеств вычисляется с помощью оператора – или методом `difference()` (инструкции вида `A-B` и `A.difference(B)`).
- *Симметрической разностью множеств* `A` и `B` является множество, содержащее элементы множества `A` или множества `B`, которые при этом не входят в пересечение этих множеств (то есть в результирующее множество входят только те элементы, которые входят только в одно из множеств: или в `A`, или в `B`). Симметрическая разность множеств вычисляется с помощью оператора `^` или

методом `symmetric_difference()` (инструкции вида `A^B` и `A.symmetric_difference(B)`).

НА ЗАМЕТКУ

Операции объединения, пересечения и симметрической разности множеств не зависят от порядка следования операндов: например, результат выражений `A.intersection(B)` и `B.intersection(A)` совпадают. Результат вычисления разности множеств в общем случае зависит от того, какой операнд первый, а какой — второй. Скажем, результатом выражения `A.difference(B)` является множество, состоящее из элементов множества `A`, которые не входят в множество `B`. А результатом выражения `B.difference(A)` является множество, которое состоит из тех элементов множества `B`, которые не входят в множество `A`.

ПОДРОБНОСТИ

При вычислении описанных выше операций вычисляется новое множество. Скажем, в результате вычисления выражения `A.union(B)` создается множество, которое содержит все элементы из множеств `A` и `B`. Множества `A` и `B` при этом не изменяются. Но объединение множеств можно вычислить и с помощью метода `update()`. В результате выполнения команды вида `A.update(B)` в множество `A` добавляются элементы из множества `B`.

Метод `intersection_update()` используется для вычисления пересечения множеств: в результате выполнения операции `A.intersection_update(B)` в множестве остаются только те элементы, которые входят в множество `B`.

Метод `difference_update()` позволяет вычислить симметрическую разность: после выполнения команды вида `A.difference_update(B)` в множестве `A` остаются только те элементы, которых нет в множестве `B`.

Метод `symmetric_difference_update()` используется при вычислении симметрической разности множеств: после выполнения команды `A.symmetric_difference_update(B)` множество `A` будет состоять из тех элементов, которые содержались ранее в этом множестве, но которых нет в множестве `B`, а также элементы из множества `B`, которых не было в множестве `A`.

Небольшой пример, в котором выполняются операции с множествами, представлен в листинге 4.3.

Листинг 4.3. Операции с множествами

```
# Создание множеств:
A={2*k+1 for k in range(5)}
B={2*k for k in range(5)}
C={2*k+1 for k in range(3,8)}
# Содержимое множеств:
print("Созданы множества:")
print("A =", A)
print("B =", B)
print("C =", C)
# Объединение множеств:
print("Объединение множеств:")
print("A | B =", A.union(B))
print("B | A =", B.union(A))
print("A | C =", A|C)
# Пересечение множеств:
print("Пересечение множеств:")
print("A & B =", A.intersection(B))
print("A & C =", A&C)
# Разность множеств:
print("Разность множеств:")
print("A — C =", A-C)
print("C — A =", C.difference(A))
# Симметрическая разность множеств:
print("Симметрическая разность множеств:")
print("A ^ C =", A^C)
print("C ^ A =", C.symmetric_difference(A))
# Исходные множества:
print("Исходные множества:")
print("A =", A)
print("B =", B)
print("C =", C)
```

Ниже показан результат выполнения программы.

Результат выполнения программы (из листинга 4.3)

```
Созданы множества:
A = {1, 3, 5, 7, 9}
B = {0, 2, 4, 6, 8}
C = {7, 9, 11, 13, 15}
Объединение множеств:
A | B = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
B | A = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
A | C = {1, 3, 5, 7, 9, 11, 13, 15}
Пересечение множеств:
A & B = set()
A & C = {9, 7}
Разность множеств:
A — C = {1, 3, 5}
C — A = {11, 13, 15}
Симметрическая разность множеств:
A ^ C = {1, 3, 5, 11, 13, 15}
C ^ A = {1, 3, 5, 11, 13, 15}
Исходные множества:
A = {1, 3, 5, 7, 9}
B = {0, 2, 4, 6, 8}
C = {7, 9, 11, 13, 15}
```

В данном случае для создания множеств мы используем генераторы последовательностей. Множество `A` состоит из пяти нечетных натуральных чисел (от `1` до `9` включительно). Множество `B` состоит из такого же количества четных чисел (от `0` до `8` включительно). Множество `C` состоит из пяти нечетных чисел (от `7` до `15` включительно). Для выполнения операции объединения множеств используем метод `union()` (команды `A.union(B)` и `B.union(A)`), а также оператор `|` (команда `A|C`). При вычислении объединения множеств `A` и `B` получаем множество из чисел от `0` до `9` включительно. Это те числа, которые входят хотя бы в одно из множеств `A` или `B`. При вычислении объединения множеств `A` и `C`

получаем множество, состоящее из нечетных чисел от 1 до 15. Это числа, которые входят хотя бы в одно из множеств A и C.

Пересечение множеств вычисляем с помощью метода intersection() (команда A.intersection(B)) или оператора & (команда A&C). Множества A и B не пересекаются (у них нет общих элементов). Поэтому результатом пересечения этих множеств является пустое множество (множество, в котором нет элементов). Результатом пересечения множеств A и C является множество, состоящее из чисел 7 и 9. Это числа, которые входят как в множество A, так и множество C.

Разность множеств A и C вычисляется выражением A-C. Результатом является множество из элементов множества A, которые не входят в множество C. Это числа 1, 3 и 5. Разностью множеств C и A (вычисляется инструкцией C.difference(A)) представляет собой множество из элементов, которые входят в множество C, но не входят в множество A. Это числа 11, 13 и 15.

Симметрическая разность множеств A и C (вычисляется как A^C или C.symmetric_difference(A)) является множеством из элементов, которые входят только в одно из множеств A или C (то есть в нем нет элементов, которые одновременно входят в множество A и в множество C). В итоге получаем множество из чисел 1, 3, 5, 11, 13 и 15.

Во всех случаях, когда выполняется какая-либо операция с множествами, исходные множества не меняются, а создается новое множество. Но иногда бывает необходимость изменять исходное множество. Пример, аналогичный предыдущему, но теперь с изменением множеств при выполнении операций, представлен в листинге 4.4.

Листинг 4.4. Изменение множеств

```
# Создание множеств:
A={0,5,10,15,20}
B={10,15,20,25,30}
# Содержимое множеств:
print("Созданы множества:")
print("A =", A)
print("B =", B)
# Пересечение множеств:
```

```
print("Пересечение множеств A и B:")
A.intersection_update(B)
print("A =", A)
# Объединение множеств:
print("Объединение с множеством {1,20,100}:")
A.update({1,20,100})
print("A =", A)
# Разность множеств:
print("Разность множеств B и A:")
B.difference_update(A)
print("B =", B)
# Симметрическая разность множеств:
print("Симметрическая разность множеств B и {30,35}:")
B.symmetric_difference_update({30,35})
print("B =", B)
```

Как выглядит результат выполнения программы, показано ниже.

Результат выполнения программы (из листинга 4.4)

```
Созданы множества:
A = {0, 5, 10, 15, 20}
B = {10, 15, 20, 25, 30}
Пересечение множеств A и B:
A = {10, 20, 15}
Объединение с множеством {1,20,100}:
A = {1, 20, 100, 10, 15}
Разность множеств B и A:
B = {25, 30}
Симметрическая разность множеств B и {30,35}:
B = {35, 25}
```

В данном случае мы используем методы `update()`, `intersection_update()`, `difference_update()` и `symmetric_difference_update()`. Методы позволяют вычислять соответственно объединение,

пересечение, разность и симметрическую разность множеств, и при их вызове изменяется множество, из которого вызывается метод. Стоит также отметить, что подобные операции можно выполнять и с помощью операторов |=, &=, -= и ^= (упрощенные формы операций присваивания на основе операторов |, &, – и ^). В листинге 4.5 представлена программа, аналогичная предыдущей, но с использованием операторов вместо методов (для сокращения объема кода комментарии удалены).

Листинг 4.5. Использование операторов для изменения множеств

```
A={0,5,10,15,20}
B={10,15,20,25,30}
print("Созданы множества:")
print("A =", A)
print("B =", B)
print("Пересечение множеств A и B:")
A&=B
print("A =", A)
print("Объединение с множеством {1,20,100}:")
A|={1,20,100}
print("A =", A)
print("Разность множеств B и A:")
B-=A
print("B =", B)
print("Симметрическая разность множеств B и {30,35}:")
B^={30,35}
print("B =", B)
```

Результат выполнения программы такой же, как и в предыдущем случае.

НА ЗАМЕТКУ

Для определения того, является ли одно множество подмножеством другого множества, можно использовать метод `issubset()`. Значением выражения `A.issubset(B)` является `True`, если множество `A` является подмножеством множества `B`. В противном случае результатом будет `False`.

Также полезным может быть метод `issuperset()`. Значением выражения `A.issuperset(B)` является `True`, если множество `B` является подмножеством множества `A`. В противном случае результатом является `False`.

Метод `isdisjoint()` позволяет проверить, является ли пересечение множеств пустым: значением выражения `A.isdisjoint(B)` является `True`, если у множеств `A` и `B` нет общих элементов. Если у множеств `A` и `B` есть общие элементы, то значение выражения `A.isdisjoint(B)` равно `False`.

Для множеств применимы операции сравнения. В частности, для сравнения множеств на предмет равенства используется оператор ==. При этом два множества считаются равными, если они состоят из одинаковых элементов. Другими словами, если `A` и `B` являются множествами, то значением выражения `A==B` является `True`, когда множества `A` и `B` состоят из одинаковых элементов. Если множества `A` и `B` отличаются хотя бы одним элементом, результатом выражения `A==B` является значение `False`. Соответственно, с помощью оператора!= множества сравниваются на предмет неравенства (два множества являются неравными, если они не состоят из одинаковых элементов). Значением выражения `A!=B` является значение `True`, если множества `A` и `B` отличаются хотя бы одним элементом. Если множества `A` и `B` состоят из одинаковых элементов, то результатом выражения `A!=B` будет значение `False`. Также к множествам можно применять операторы <, <=, > и >=.

Значением выражения `A<B` является `True`, если множество `A` является подмножеством множества `B` и при этом не равно множеству `B`. В противном случае значение выражения `A<B` равно `False`.

Значение выражения `A<=B` равно `True`, если множество `A` является подмножеством множества `B`, в том числе множество `A` может быть равно множеству `B`. В противном случае результатом выражения `A<=B` является значение `False`.

Значением выражения `A>B` является `True`, если множество `B` является подмножеством множества `A` и при этом не равно множеству `A`. В противном случае значение выражения `A>B` равно `False`.

Значение выражения `A>=B` равно `True`, если множество `B` является подмножеством множества `A`, в том числе множество `B` может быть равно множеству `A`. В противном случае результатом выражения `A>=B` является значение `False`.

НА ЗАМЕТКУ

Если множество A является подмножеством множества B, то каждый элемент множества A является элементом множества B. Из этого определения следует, что любое множества является своим же подмножеством (например, множество A является подмножеством множества A).

В листинге 4.6 представлена программа, в которой множества сравниваются с использованием означенных выше операторов.

Листинг 4.6. Сравнение множеств

```
# Исходные множества:
A={1,2}
B={1,2,3}
C={3,2,1}
# Содержимое множеств:
print("A =", A)
print("B =", B)
print("C =", C)
# Сравнение множеств:
print("A==B:", A==B)
print("A!=B:", A!=B)
print("B==C:", B==C)
print("B!=C:", B!=C)
print("A<B:", A<B)
print("A>B:", A>B)
print("B<C:", B<C)
print("B<=C:", B<=C)
print("B>=C:", B>=C)
print("B>=A:", B>=A)
```

При выполнении программы получаем следующий результат.

Результат выполнения программы (из листинга 4.5)

```
A = {1, 2}
```

```
B = {1, 2, 3}
C = {1, 2, 3}
A==B: False
A!=B: True
B==C: True
B!=C: False
A<B: True
A>B: False
B<C: False
B<=C: True
B>=C: True
B>=A: True
```

Программа достаточно простая, и, хочется верить, нет необходимости объяснять результат ее выполнения.

> **НА ЗАМЕТКУ**
>
> Если `A` и `B` — некоторые множества, то из того, что значение выражения `A<=B` равно `False`, не следует, что значение выражения `A>=B` будет равно `True`. Например, если `A={1,2,3}` и `B={2,3,4}`, то и значение выражения `A<=B`, и `A>=B` оба равны `False`. Причина в том, что ни множество `A` не является подмножеством множества `B`, ни множество `B` не является подмножеством множества `A`.

Примеры использования множеств

Я тебя полюбил — я тебя научу.

Из к/ф «Кин-дза-дза»

Далее мы рассмотрим несколько программ, в которых для решения прикладных задач используются множества. Сначала программными методами решим такую задачу: пользователя просят ввести целое число, а затем для введенного числа определяются цифры, из которых состоит число. Код программы представлен в листинге 4.7.

Листинг 4.7. Состав числа

```
# Считывание числа:
number=int(input("Введите число: "))
# Если число отрицательное:
if number<0:
    number*= -1
# Создается пустое множество:
digits=set()
# Если был введен ноль:
if number==0:
    digits.add(0)
# Если число не равно нулю:
else:
    # Перебор цифр в представлении числа:
    while number!=0:
        # Последняя цифра в представлении числа:
        digits.add(number%10)
        # В представлении числа отбрасывается
        # последняя цифра:
        number//=10
print("Число состоит из таких цифр:")
# Отображение результата:
for n in digits:
    print(n, end=" ")
# Переход к новой строке:
print()
```

Как может выглядеть результат выполнения программы, показано ниже (жирным шрифтом введено значение, которое вводит пользователь).

Результат выполнения программы (из листинга 4.7)

```
Введите число: 2501175
```

```
Число состоит из таких цифр:
0 1 2 5 7
```

Программа простая. Сначала команда `number=int(input("Введите число: "))` считывает целое число и записывает его в переменную `number`.

> **НА ЗАМЕТКУ**
>
> Функция `input()` в качестве результата возвращает текст, даже если пользователь вводит число. Поэтому нам необходимо текстовое представление числа преобразовать в собственно число. Для этого используется функция `int()`.

Теоретически это число может быть отрицательным. Нас интересуют цифры в представлении числа, а не его знак. Поэтому с помощью условного оператора проверяется условие `number<0` (число отрицательное) и если так, то командой `number*= -1` текущее значение переменной `number` умножается на `-1` (число становится положительным).

Затем командой `digits=set()` создается пустое множество. В это множество мы планируем заносить цифры, из которых состоит число. В первую очередь мы проверяем, является ли введенное пользователем число нулем (проверяется условие `number==0`). Если так, то командой `digits.add(0)` в множество `digits` добавляется цифра `0`. На этом, собственно, вычисления заканчиваются, и мы получаем множество из одного элемента. Если же введенное пользователем число не равно нулю, то выполняется другой блок команд. В частности, запускается оператор цикла `while`, который выполняется, пока истинно условие `number!=0` (значение переменной `number` отлично от нуля). Командой `digits.add(number%10)` в множество добавляется последняя цифра в представлении числа, записанного в переменную `number`. После этого командой `number//=10` в представлении числа, являющегося значением переменной `number`, отбрасывается последняя цифра.

> **НА ЗАМЕТКУ**
>
> Выражением `number%10` вычисляется остаток от деления значения переменной `number` на `10`. Это последняя цифра в представлении числа. Командой `number//=10` текущее значение переменой `number` делится нацело на `10` и полученное значение присваивается переменной `number`. При вычислении результата целочисленного

деления числа на 10 фактически в представлении числа отбрасывается последняя цифра.

После завершения оператора цикла while, когда сформировано множество digits, с помощью оператора цикла for отображается содержимое множества digits.

ПОДРОБНОСТИ

Задачу об определении состава числа (разных цифр, с помощью которых записано число) можно было решить значительно проще. Например, такая программа могла бы состоять всего из двух команд: digits=set(input("Введите число: ")) и print(digits). Инструкция input("Введите число: ") отображает сообщение с просьбой ввести число, а результатом возвращает текстовую строку с представлением этого числа. Строка передается аргументом функции set(). В результате на основе строки создается множество, причем элементами множества являются символы из текстовой строки. Если пользователь ввел положительное целое число, то символами являются цифры (но не как числа, а именно как символы). Таким образом, множество digits будет состоять из символьных значений, являющихся цифрами, из которых сформировано введенное пользователем число. Команда print(digits) нужна для того, чтобы увидеть содержимое этого множества.

В следующей программе пользователь вводит два текстовых значения и для них определяются буквы, которые присутствуют в каждом из текстовых значений. Рассмотрим программу в листинге 4.8.

Листинг 4.8. Общие буквы в тексте

```
# Считывание текстовых значений:
text=input("Первый текст: ")
# Первое множество:
A=set(text)
text=input("Второй текст: ")
# Второе множество:
B=set(text)
# Общие буквы:
C=A&B
```

```
# Результат:
print("Буквы из первого текста: ", A)
print("Буквы из второго текста: ", B)
print("Общие буквы: ", C)
```

Результат выполнения программы следующий (жирным шрифтом выделены значения, введенные пользователем).

Результат выполнения программы (из листинга 4.8)

```
Первый текст: программа
Второй текст: абракадабра
Буквы из первого текста:  {'г', 'о', 'р', 'п', 'а', 'м'}
Буквы из второго текста:  {'а', 'р', 'к', 'б', 'д'}
Общие буквы:  {'а', 'р'}
```

В программе считывается два текстовых значения и на их основе создаются множества `A` и `B`. Эти множества содержат буквы (символы, если точнее) из тех текстовых значений, на основе которых формировались множества. Для определения букв, которые имеются в обоих текстах, вычисляется пересечение множеств (команда `C=A&B`).

В следующей программе в пределах первой полусотни определяются все натуральные числа, которые делятся на `3` или на `11`, но при этом не делятся на `5` и `7`. При этом мы будем использовать операции с множествами. Программа представлена в листинге 4.9.

Листинг 4.9. Числа с заданными свойствами

```
# Верхняя граница для чисел:
n=50
# Множество натуральных чисел:
E=set(range(1, n+1))
# Множество чисел, которые делятся на 3:
A={s for s in E if s%3==0}
# Множество чисел, которые делятся на 11:
B={s for s in E if s%11==0}
# Множество чисел, которые делятся на 5:
```

```
C={s for s in E if s%5==0}
# Множество чисел, которые делятся на 7:
D={s for s in E if s%7==0}
# Множество чисел, которые делятся на 3 или 11,
# но не делятся на 5 и 7:
N=(A|B)-(C|D)
# Отображение результата:
print(N)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 4.9)

```
{33, 3, 36, 6, 39, 9, 11, 12, 44, 48, 18, 22, 24, 27}
```

Эта задача может решаться по-разному. Мы поступаем следующим образом. В переменную `n` записывается значение `50` для верхней границы множества натуральных чисел. Командой `E=set(range(1, n+1))` создается множество натуральных чисел в диапазоне от `1` до значения переменой `n` включительно.

Команда `A={s for s in E if s%3==0}` создает множество чисел, которые делятся на `3`. Множество чисел, которые делятся на `11`, создается командой `B={s for s in E if s%11==0}`. Также мы создаем множество чисел, которые делятся на `5` (команда `C={s for s in E if s%5==0}`), и множество чисел, делящихся на `7` (команда `D={s for s in E if s%7==0}`). Во всех этих случаях в качестве базового используется множество `E` из натуральных чисел. Элементы этого множества перебираются и включаются в новое множество, если значение элемента удовлетворяет определенному критерию.

Командой `N=(A|B)-(C|D)` вычисляется множество, состоящее из чисел, которые делятся на `3` и/или `11`, но не делятся на `5` и `7`. Здесь следует учесть, что выражением `A|B` вычисляется множество, состоящее из чисел, которые делятся хотя бы на одно из чисел `3` и `11`. Значением выражения `C|D` является множество, состоящее из чисел, которые делятся на `5` и/или `7`. Разность этих множеств (выражение `(A|B)-(C|D)`) представляет собой множество, которое состоит из чисел, делящихся на `3` и/или `11`, но нее делящихся на `5` и/или `7`.

Знакомство со словарями

> Господи прости, от страха все слова повыскакивали.
>
> *Из к/ф «Формула любви»*

Важным типом данных являются *словари*. Словарь можно представить в виде некоторого списка, но при этом роль индексов могут играть не только числовые значения. Значения, которые играют роль индексов, называются *ключами* (ключом, например, может быть текст, число или кортеж — но всегда это значение неизменяемого типа). Другими словами, имеется набор *значений*, и каждое значение из этого набора идентифицируется с помощью ключа.

Создать словарь можно с использованием фигурных скобок: в блоке из этих скобок указываются через запятую выражения вида `ключ: значение`, которые определяют ключи и соответствующие им значения в словаре.

Также для создания словаря используется функция `dict()`. Если ключами словаря являются текстовые значения, то в качестве аргументов функции `dict()` могут передаваться конструкции вида `ключ=значение`. Такие выражения разделяются запятыми. В этом случае создается словарь, в котором ключи и значения определяются аргументами, переданными функции.

ПОДРОБНОСТИ

Ключи в таком случае (хотя они являются текстовыми) указываются без кавычек. Причина в том, что на самом деле в данном случае ключи играют роль именованных аргументов (именованные аргументы функций описываются несколько позже, в одной из следующих глав). В этом случае на ключи накладываются ограничения как на названия аргументов (например, это не может быть два слова). Также стоит отметить, что описанный способ передачи аргументов функции `dict()` не является единственным.

НА ЗАМЕТКУ

Вызов функции `dict()` без аргументов приводит к созданию пустого словаря. Пустой блок из фигурных скобок `{}` также соответствует пустому словарю (не множеству).

> Как и в случае со списками, кортежами и множествами, количество элементов в словаре можно определить с помощью функции `len()`.

Если словарь создан, то доступ к значению из этого словаря можно получить по ключу. Доступ выполняется в формате `словарь[ключ]` — то есть после имени словаря в квадратных скобках указывается ключ для доступа к соответствующему значению. Небольшая программа, в которой создаются словари, представлена в листинге 4.10.

Листинг 4.10. Знакомство со словарями

```
# Первый словарь:
nums={100:"сотня",1:"единица",10:"десятка"}
# Содержимое словаря:
print(nums)
print("1:  ", nums[1])
print("10: ", nums[10])
print("100:", nums[100])
# Операции со словарем:
nums[3]="тройка"
nums[10]="десять"
nums.pop(100)
print(nums)
# Второй словарь:
order=dict(Первый=1, Третий=3, Последний=10)
# Содержимое словаря:
print(order)
print("Первый:   ", order["Первый"])
print("Третий:   ", order["Третий"])
print("Последний:", order["Последний"])
# Операции со словарем:
order["Последний"]=12
del order["Третий"]
order["Пятый"]=5
print(order)
```

Ниже показано, как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 4.10)

```
{100: 'сотня', 1: 'единица', 10: 'десятка'}
1:   единица
10:  десятка
100: сотня
{1: 'единица', 10: 'десять', 3: 'тройка'}
{'Первый': 1, 'Третий': 3, 'Последний': 10}
Первый:    1
Третий:    3
Последний: 10
{'Первый': 1, 'Последний': 12, 'Пятый': 5}
```

В программе создается два словаря, и с этими словарями выполняются несложные операции. Первый словарь создается командой `nums={100:"сотня",1:"единица",10:"десятка"}`. В этом словаре три элемента. Их ключи — это целые числа `1`, `10` и `100`, а значения элементов соответственно равны `"единица"`, `"десятка"` и `"сотня"`. Для получения значения элемента словаря после имени словаря в квадратных скобках указывается ключ соответствующего элемента: например, `nums[1]`, `nums[10]` и `nums[100]`. В данном случае считывается значение элемента. В таком же формате элементу можно присвоить новое значение — примером может служить команда `nums[10]="десять"`. Если указать ключ, которого в словаре нет, то в словарь будет добавлен новый элемент с таким ключом. Например, при выполнении команды `nums[3]="тройка"` в словаре `nums` появляется элемент с ключом `3` и значением `"тройка"`. Для удаления элемента с определенным ключом можно использовать метод `pop()`. Так, в результате выполнения команды `nums.pop(100)` из словаря удаляется элемент с ключом `100`.

Еще один словарь создается командой `order=dict(Первый=1, Третий=3, Последний=10)`. Ключами в данном случае являются текстовые значения `"Первый"`, `"Третий"`, `"Последний"`, а элементы имеют значения `1`, `3` и `10` соответственно.

НА ЗАМЕТКУ

Еще раз обращаем внимание, что текстовые значения для ключей при создании словаря указываются без кавычек. Однако когда выполняется обращение к элементу, то кавычки для текстовых ключей указываются.

Для считывания значений элементов словаря используем команды `order["Первый"]`, `order["Третий"]` и `order["Последний"]`. Командой `order["Последний"]=12` элементу с ключом `"Последний"` присваивается значение `12`. Командой `del order["Третий"]` из словаря удаляется элемент с ключом `"Третий"`. А вот при выполнении команды `order["Пятый"]=5` значение присваивается элементу с ключом `"Пятый"`, которого в словаре нет. В результате выполнения команды такой элемент в словаре появляется.

Описанные выше способы создания словарей не являются исчерпывающими. Так, функции `dict()` в качестве аргумента можно передать уже существующий словарь — в этом случае создается его поверхностная копия. Словарь можно создавать и на основе списка. Элементами такого списка, который передается аргументом функции `dict()`, должны быть списки или кортежи, состоящие из двух элементов. Эти элементы определяют ключи и значения для элементов словаря. Еще один пример создания словарей представлен в листинге 4.11.

Листинг 4.11. Другие способы создания словарей

```
# Создание словаря:
age=dict([["Кот Матроскин",5],["Пес Шарик",7],["Дядя Федор",12]])
# Перебор ключей:
for s in age.keys():
    print(s+":", age[s])
# Перебор значений:
for v in age.values():
    print(v, end=" ")
print()
# Создание словаря:
color=dict([[(255,0,0),"Красный"],[(0,255,0),"Зеленый"],[(0,0,255),"Синий"]])
# Обращение к элементам словаря:
```

```
color[(255,255,0)]="Желтый"
print("(255,0,0):", color[(255,0,0)])
print("(255,255,0):", color[(255,255,0)])
print("(0,255,0):", color.get((0,255,0)))
print("(0,0,255):", color.get((0,0,255),"Белый"))
print("(255,255,255):", color.get((255,255,255),"Белый"))
```

Результат выполнения программы такой, как показано ниже.

Результат выполнения программы (из листинга 4.11)

```
Кот Матроскин: 5
Пес Шарик: 7
Дядя Федор: 12
5 7 12
(255,0,0): Красный
(255,255,0): Желтый
(0,255,0): Зеленый
(0,0,255): Синий
(255,255,255): Белый
```

Первый словарь создается командой `age=dict([["Кот Матроскин",5], ["Пес Шарик",7], ["Дядя Федор",12]])`. В этой команде мы передаем аргументом функции `dict()` список, элементами которого являются списки `["Кот Матроскин",5]`, `["Пес Шарик",7]` и `["Дядя Федор",12]`. Первый элемент в каждом из этих списков определяет ключ элемента словаря, а второй элемент в списке определяет значение элемента словаря.

НА ЗАМЕТКУ

В данном случае словарь напоминает мини-базу данных, в которой ключ является именем персонажа, а значение элемента словаря определяет возраст персонажа.

Для каждого словаря с помощью метода `keys()` можно получить итерируемый объект, позволяющий определить значения ключей из словаря.

Методом `values()` возвращается итерируемый объект, позволяющий получить значения элементов из словаря.

ПОДРОБНОСТИ

Итерируемый объект не является последовательностью в прямом смысле этого понятия. Но это объект, который, как и последовательность, можно перебирать. Поэтому во многих ситуациях итерируемый объект используется так, как если бы это была последовательность вроде списка.

Что касается метода `keys()`, то результатом он возвращает итерируемый объект класса `dict_keys`. Метод `values()` результатом возвращает итерируемый объект класса `dict_values`.

Классы и объекты обсуждаются в одной из следующих глав.

В операторе цикла `for` переменная `s` принимает значения ключей из словаря `age` (для получения набора ключей использована инструкция `age.keys()`). Для каждого заданного значения `s` выполняется команда `print(s+":", age[s])`, которой отображается значение ключа и значение соответствующего элемента словаря.

В следующем операторе цикла `for` перебираются значения элементов словаря. Переменная `v` принимает значения из последовательности ключей (вычисляется инструкцией `age.values()`). Команда `print(v, end=" ")` отображает значения элементов из словаря, при этом в качестве разделителя используются пробелы.

Еще один словарь создается командой `color=dict([[(255,0,0), "Красный"], [(0, 255, 0), "Зеленый"], [(0, 0, 255), "Синий"]])`. В качестве аргумента функции `dict()` передан список, элементами которого являются списки `[(255, 0, 0), "Красный"]`, `[(0, 255, 0), "Зеленый"]` и `[(0, 0,255),"Синий"]`. Примечательно здесь то, что ключами являются кортежи (`(255,0,0)`, `(0,255,0)` и `(0,0,255)`). Значения элементов словаря — текстовые значения с названиями цвета (соответственно `"Красный"`, `"Зеленый"` и `"Синий"`).

НА ЗАМЕТКУ

Мы воспользовались тем, что в формате RGB (сокращение от Red Green Blue) цвет задается с помощью трех целых чисел, каждое в диапазоне от `0` до `255` включительно.

При обращении к элементам словаря можно, как мы уже делали ранее, указывать ключ в квадратных скобках после названия словаря (например, как в командах `color[(255,0,0)]` и `color[(255,255,0)]`). Такой же подход использован в команде `color[(255,255,0)]="Желтый"`, которой в словарь добавляется новый элемент (поскольку указан ключ, который на момент выполнения команды в словаре не представлен). Однако для получения значения элемента по ключу можно использовать метод `get()`, при вызове которого ключ передается в качестве аргумента (примером служит команда `color.get((0,255,0))`). Методу можно передать второй аргумент, который будет возвращаться в качестве результата, если ключа, указанного первым аргументом, в словаре нет. Например, результатом выражения `color.get((0,0,255), "Белый")` является значение `"Синий"` элемента с ключом `(0,0,255)`. Но элемента с ключом `(255,255,255)` в словаре нет, поэтому результатом выражения `color.get((255,255,255),"Белый")` является значение `"Белый"`.

ПОДРОБНОСТИ

Если при обращении к элементу словаря в квадратных скобках указать несуществующий ключ, это приведет к ошибке класса `KeyError`. Если доступ к элементу осуществляется с помощью метода `get()` и аргументом методу передается один аргумент, определяющий значение ключа, то результатом возвращается значение соответствующего элемента. Если указан несуществующий ключ, то результатом возвращается значение `None`, и ошибка при этом не возникает.

Как и в случае со списками, кортежами и множествами, для создания словарей можно использовать генераторы последовательностей. Небольшой пример того, как это можно было бы сделать, представлен в листинге 4.12.

Листинг 4.12. Генераторы словарей

```
# Список со значениями ключей:
days=["Пн","Вт","Ср","Чт","Пт","Сб","Вс"]
# Создание словарей:
week={days[s]: s for s in range(len(days))}
myweek={d: days.index(d) for d in days}
# Проверка результата:
```

```
print(week)
print(myweek)
# Создание еще одного словаря:
sqrs={k: k**2 for k in range(1,11) if k%2!=0}
# Проверка результата:
print(sqrs)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 4.12)

```
{'Пн': 0, 'Вт': 1, 'Ср': 2, 'Чт': 3, 'Пт': 4, 'Сб': 5, 'Вс': 6}
{'Пн': 0, 'Вт': 1, 'Ср': 2, 'Чт': 3, 'Пт': 4, 'Сб': 5, 'Вс': 6}
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

В данном случае создается три словаря, причем два из них одинаковые. Более конкретно, мы создаем список `days` с текстовыми значениями для ключей. Словарь `week` создается с использованием генератора последовательности: элемент словаря формируется инструкцией `days[s]: s`, где переменная `s` пробегает значения от `0` до величины `len(days)-1` (количество элементов в списке `days` минус `1`). Выражение `days[s]: s` означает, что при заданном значении `s` ключ элемента определяется элементом списка `days[s]`, а значение элемента словаря при этом равно `s`. Аналогичный словарь `myweek` создается несколько иначе. Элемент словаря формируется выражением `d: days.index(d)`, а переменная `d` принимает значения из списка `days`. Значение переменной задает ключ элемента словаря, а в качестве значения элемента словаря указано выражение `days.index(d)`. Это индекс элемента со значением `d` в списке `days`. В итоге мы создаем два одинаковых словаря `week` и `myweek`, в чем и убеждаемся при проверке.

Еще один словарь состоит из числовых значений: ключами элементов являются нечетные натуральные числа, а значением элементов является квадрат соответствующего числа. Словарь создается командой `sqrs={k: k**2 for k in range(1,11) if k%2!=0}`. Элемент словаря определяется выражением `k: k**2`, а переменная `k` пробегает значения из диапазона от `1` до `10` включительно, причем в словарь соответствующий элемент добавляется, только если истинно условие `k%2!=0` (остаток от деления значения `k` на `2` не равен нулю — то есть число нечетное).

Операции со словарями

Вам поручена эта операция, так что действуйте!

Из к/ф «Бриллиантовая рука»

Мы уже выполняли основные операции со словарями (такие как добавление элемента в словарь или удаление элемента из словаря). Теперь рассмотрим другие операции, применимые к словарям.

В первую очередь остановимся на способах создания копии словаря. Причем эту задачу мы будем интерпретировать в широком смысле, когда на основе уже существующего словаря создается новый словарь (не обязательно совпадающий с исходным). Для этого существуют разные подходы, и некоторые из них реализованы в программе, представленной в листинге 4.13.

Листинг 4.13. Создание словаря на основе словаря

```
# Исходный словарь:
A={"Начальный":1,"Средний":2,"Последний":3}
# Создание копии словаря:
B=dict(A)
C=A.copy()
# Создание словаря на основе словаря:
D={k: v*10 for k, v in A.items()}
# Отображение результата:
print("Созданы множества:")
print("A =", A)
print("B =", B)
print("C =", C)
print("D =", D)
# Изменение исходного словаря:
for k in A:
    A[k]*=100
# Отображение результата:
print("После изменения оригинала:")
```

```
print("A =", A)
print("B =", B)
print("C =", C)
print("D =", D)
```

Результат выполнения программы следующий.

Результат выполнения программы (из листинга 4.13)

```
Созданы множества:
A = {'Начальный': 1, 'Средний': 2, 'Последний': 3}
B = {'Начальный': 1, 'Средний': 2, 'Последний': 3}
C = {'Начальный': 1, 'Средний': 2, 'Последний': 3}
D = {'Начальный': 10, 'Средний': 20, 'Последний': 30}
После изменения оригинала:
A = {'Начальный': 100, 'Средний': 200, 'Последний': 300}
B = {'Начальный': 1, 'Средний': 2, 'Последний': 3}
C = {'Начальный': 1, 'Средний': 2, 'Последний': 3}
D = {'Начальный': 10, 'Средний': 20, 'Последний': 30}
```

Это простая программа, в которой создается словарь `A` с тремя элементами (команда `A={"Начальный":1,"Средний":2,"Последний":3}`). Затем командами `B=dict(A)` и `C=A.copy()` на основе словаря `A` создаются словари-копии. Еще один словарь `D` создается на основе словаря `A`, но на этот раз речь не идет о копии. Словарь создается командой `D={k: v*10 for k, v in A.items()}`. Здесь мы используем генератор последовательности, причем в качестве перебираемого множества указано выражение `A.items()`. Метод `items()` возвращает итерируемый объект класса `dict_items`. В качестве «элементов» этого итерируемого объекта возвращаются кортежи, состоящие из двух элементов: ключ элемента словаря и значение элемента словаря. В операторе цикла указаны через запятую две переменные, `k` и `v`. Поэтому переменная `k` в качестве значения получает ключ элемента словаря, а переменная `v` принимает значение соответствующего элемента словаря.

ПОДРОБНОСТИ

Методы `items()`, `keys()` и `values()` результатом возвращают итерируемые объекты. Чтобы получить на основе этих итерируемых

объектов списки, результат вызова данных методов следует передать аргументом функции `list()`. В результате получим список, который обрабатывается по правилам работы со списками. Аналогично можно воспользоваться функциями `tuple()` и `set()` для создания на основе итерируемого объекта соответственно кортежа и множества. Более того, если результат вызова метода `items()` передать в качестве аргумента функции `dict()`, то в результате получим копию словаря, из которого вызывался метод `items()`.

На следующем этапе перебираются элементы словаря `A`. Причем мы использовали оператор цикла, в котором переменная `k` перебирает значения из словаря `A`. Важно то, что в этом случае переменная `k` принимает значения ключей из словаря `A`. Поэтому при выполнении команды `A[k]*=100` значения элементов из словаря `A` умножаются на `100`. Причем проверка показывает, что изменение содержимого словаря `A` не влияет на содержимое словарей `B`, `C` и `D`.

Однако копия, которая создается с помощью метода `copy()` или при передаче словаря аргументом функции `dict()`, является *поверхностной копией* словаря. Особенности такой копии иллюстрирует программа, представленная в листинге 4.14.

Листинг 4.14. Полная и поверхностная копия словаря

```
# Импорт функции для создания полной копии:
from copy import deepcopy
# Исходный словарь:
A={"один":1,"два":"двойка","три":[3,4]}
# Поверхностная копия словаря:
B=dict(A)
C=A.copy()
# Полная копия словаря:
D=deepcopy(A)
# Отображение результата:
print("A =", A)
print("B =", B)
print("C =", C)
print("D =", D)
```

```
# Изменение значений элементов исходного словаря:
A["два"]=2
A["три"][1]=5
# Проверка результата:
print("A =", A)
print("B =", B)
print("C =", C)
print("D =", D)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 4.14)

```
A = {'один': 1, 'два': 'двойка', 'три': [3, 4]}
B = {'один': 1, 'два': 'двойка', 'три': [3, 4]}
C = {'один': 1, 'два': 'двойка', 'три': [3, 4]}
D = {'один': 1, 'два': 'двойка', 'три': [3, 4]}
A = {'один': 1, 'два': 2, 'три': [3, 5]}
B = {'один': 1, 'два': 'двойка', 'три': [3, 5]}
C = {'один': 1, 'два': 'двойка', 'три': [3, 5]}
D = {'один': 1, 'два': 'двойка', 'три': [3, 4]}
```

В программе командой `A={"один":1,"два":"двойка", "три":[3,4]}` создается словарь, на основе которого создаются словари-копии `B` (команда `B=dict(A)`), `C` (команда `C=A.copy()`) и `D` (команда `D=deepcopy(A)`). Причем в последнем случае создается полная копия словаря.

НА ЗАМЕТКУ

Для использования функции создания полной копии `deepcopy()` она инструкцией `from copy import deepcopy` импортируется из модуля `copy`.

После создания копий командами `A["два"]=2` и `A["три"][1]=5` меняются значения элементов словаря, на основе которого создавались словари-копии.

ПОДРОБНОСТИ

Исходным значением элемента с ключом `"два"` является `"двойка"`. Новым является значение `2`. Значением элемента с ключом `"три"` является ссылка на список `[3,4]`. Соответственно, инструкция `A["три"][1]` является обращением ко второму элементу в списке `[3,4]`. Таким образом, после выполнения команды `A["три"][1]=5` число `4` в списке `[3,4]` заменяется на `5`.

После внесения изменений в словарь `A` мы проверяем содержимое всех созданных словарей. Видим, что во всех словарях-копиях элемент с ключом `"два"` не изменил значение. При этом в словарях `B` и `C` в списке, являющемся значением элемента с ключом `"три"`, второй элемент `4` изменился на `5`. Словарь `D`, который создавался как полная копия словаря `A`, не изменился совсем. В чем причина такого результата?

Для ответа на вопрос следует учесть, что список `[3,4]` на самом деле не является значением элемента с ключом `"три"` в словаре `A`. Значением этого элемента является ссылка на список (фактически адрес списка). При создании поверхностных копий копируется ссылка на список, а не сам список. Поэтому словари `A`, `B` и `C` через элемент с ключом `"три"` ссылаются на один и тот же список. Следовательно, когда через словарь `A` вносятся изменения в список, то эти изменения влияют и на словари `B` и `C`. Здесь важно, что ссылка, записанная в элемент с ключом `"три"`, при выполнении команды `A["три"][1]=5` не меняется. Меняется лишь список, на который выполнена ссылка. Что касается элемента словаря `A` с ключом `"два"`, то при выполнении команды `A["два"]=2` меняется значение этого элемента. Для прочих словарей этот элемент остается со старым значением.

Кроме собственно создания словарей, интерес могут представлять и другие операции со словарями. Например, может возникнуть необходимость добавить в словарь не отдельный элемент, а сразу целый словарь. В таком случае используют метод `update()`. Метод вызывается из словаря, в который добавляется другой словарь, а добавляемый словарь передается в качестве аргумента методу. При этом в результате вызова метода добавляемый словарь не меняется, изменения вносятся в тот словарь, из которого вызывается метод.

ПОДРОБНОСТИ

Аргументом методу `update()` можно также передавать не словарь, а список, который содержит списки или кортежи с парами

значений ключ/значение. В случае если ключи текстовые, можно передавать в качестве аргументов функции `update()` конструкции вида `ключ=значение`. То есть фактически аргументы функции `update()` можно передавать так, как мы передаем аргументы функции `dict()`. Кроме того, у нас остается возможность передать в качестве аргумента функции `update()` инструкцию, которой создается словарь (он и будет добавлен в словарь, из которого вызывается метод). Если нам нужно проверить, содержится ли элемент с данным ключом в словаре, то используется оператор `in`. Проверка того, что элемент не содержится в словаре, выполняется с помощью конструкции `not in`. При этом нужно помнить, что поиск выполняется среди ключей словаря.

Также словари можно сравнивать не предмет равенства (оператор ==) или неравенства (оператор!=). Словари считаются равными, если они содержат элементы с одинаковыми ключами и значениями. Полную очистку словаря можно выполнить с помощью метода `clear()`. Небольшой пример, в котором выполняются простые операции со словарями, представлен в листинге 4.15.

Листинг 4.15. Некоторые операции со словарями

```
# Создание словарей:
A=dict(zip([1,2,3],['K','L','M']))
B=dict.fromkeys([10,20,30],'Z')
# Отображение результата:
print("A =", A)
print("B =", B)
# Сравнение словарей:
print("Сравнение словарей:")
print("A==B:", A==B)
print("A!=B:", A!=B)
# Добавление одного словаря к другому:
A.update(B)
# Проверка результата:
print("Объединение словарей:")
print("A =", A)
# Проверка содержимого словаря:
```

```
print("Проверка содержимого словаря:")
print((20,'Z') in A.items())
print(20 in A)
print('Z' in A)
print(5 not in A)
# Очистка словаря:
A.clear()
# Проверка результата:
print("Словарь после очистки:")
print("A =", A)
```

Ниже показан результат выполнения программы.

Результат выполнения программы (из листинга 4.15)

```
A = {1: 'K', 2: 'L', 3: 'M'}
B = {10: 'Z', 20: 'Z', 30: 'Z'}
Сравнение словарей:
A==B: False
A!=B: True
Объединение словарей:
A = {1: 'K', 2: 'L', 3: 'M', 10: 'Z', 20: 'Z', 30: 'Z'}
Проверка содержимого словаря:
True
True
False
True
Словарь после очистки:
A = {}
```

Кроме прочего, здесь мы встречаемся еще с несколькими способами создания словарей. Так, командой `A=dict(zip([1,2,3],['K','L','M']))` словарь фактически создается на основе двух списков: один содержит значения ключей, другой содержит значения элементов. Эти списки предварительно передаются в качестве аргументов функции

zip(), в результате чего возвращается специальный объект. Если этот объект передать аргументом функции list(), то в результате получим список, который состоит из кортежей, в каждом из них два элемента: один из первого списка, второй из второго.

ПОДРОБНОСТИ

Аргументами функции zip() передается несколько списков. Результатом является объект, который соответствует некоторому новому списку (чтобы увидеть это список, объект нужно передать в качестве аргумента функции list()). Прообраз списка формируется следующим образом. Он состоит из кортежей, в каждом кортеже количество элементов такое, как количество списков, переданных в качестве аргументов функции zip(). При формировании кортежей один элемент берется из первого списка, второй элемент берется из второго списка, и так далее. Например, если аргументом функции zip() передаются списки [1,2,3] и ['A','B','C'], то, передав результат вызова функции zip() в функцию list(), получим список [(1,'A'),(2,'B'),(3,'C')].

Еще один словарь создается командой B=dict.fromkeys([10,20,30],'Z'). Здесь мы из класса dict вызываем метод fromkeys(). Первым аргументом методу передается список со значениями ключей словаря. Второй аргумент определяет значение элементов словаря — в созданном словаре все элементы будут иметь одно и то же значение.

Для сравнения словарей мы используем инструкции A==B и A!=B.

В результате выполнения команды A.update(B) содержимое словаря B добавляется в словарь A (при этом словарь B остается неизменным).

Также программа содержит примеры проверки того, входит ли тот или иной элемент в словарь. Например, значением выражения является (20,'Z') in A.items() является True. Дело в том, что результатом выражения A.items() является объект класса dict_items, который соответствует списку с элементами-кортежами. Элементы кортежа содержат значение ключа и значение элемента.

НА ЗАМЕТКУ

Увидеть «содержимое» объекта, который возвращается методом items(), можно, передав результат вызова метода items() в функцию list() (речь о команде вида list(A.items())).

Среди этих кортежей есть и кортеж `(20,'Z')`. Но при вычислении значения выражения `20 in A` в словаре `A` выполняется поиск элемента с ключом `20`. Такой элемент в словаре есть, поэтому значением выражения является `True`. Элемента с ключом `'Z'` в словаре `A` нет, поэтому значение выражения `'Z' in A` равно `False`. Нет в нем и элемента с ключом `5`. При вычислении значения выражения `5 not in A` получаем значение `True`.

Наконец, командой `A.clear()` выполняется очистка словаря `A`. После этого он не содержит ни одного элемента.

Резюме

Это твое заднее слово?

Из к/ф «Кин-дза-дза»

- Множество представляет собой неупорядоченный набор уникальных элементов. В отличие от списков, при работе с множествами такое понятие, как позиция элемента в множестве, теряет смысл. Можно только сказать, содержится ли в множестве элемент с указанным значением или нет.
- Для создания множеств используют функцию `set()`, в качестве аргументов которой передаются элементы множества. Можно также перечислить элементы множества в фигурных скобках.
- Элементы в множество можно добавлять, а также можно удалять элементы из множества. Кроме этого, есть специальные методы для вычисления объединения множеств, пересечения множеств, разности множеств и симметрической разности множеств.
- Словарь представляет собой набор элементов, доступ к которым выполняется по ключу. В качестве ключей могут использоваться значения неизменяемых типов.
- Для создания словаря используют функцию `dict()`. Аргументом функции передается список с внутренними кортежами по два элемента в каждом. Эти элементы определяют ключи и значения элементов словаря. Также можно создать словарь, указав попарно в фигурных скобках значения для ключей и элементов словаря. При этом ключ и значение элемента разделяются двоеточием. Существуют и другие

способы создания словарей. В частности, при создании множеств и словарей могут использоваться генераторы последовательностей.

- Для обращения к элементу словаря после имени словаря в квадратных скобках указывается ключ словаря.
- В словарь можно добавлять элементы, из словаря можно удалять элементы, а также можно выполнять объединение словарей (когда содержимое одного словаря добавляется в другой словарь).

Задания для самостоятельной работы

— Капу, капу жми!
— Нажал уже.

Из к/ф «Кин-дза-дза»

1. Напишите программу, в которой генерируется `15` случайных целых чисел: `5` чисел попадают в диапазон значений от `1` до `10` и `10` чисел попадают в диапазон от `10` до `30`.

2. Напишите программу, в которой пользователь вводит два целых числа, а программой определяются цифры, которые есть в представлении обоих чисел.

3. Напишите программу, в которой пользователь вводит текстовое значение и для этого текстового значения определяются гласные буквы, представленные во введенном тексте.

4. Напишите программу для вычисления множества чисел (в пределах первой полусотни), которые делятся или на `3`, или на `4`, но при этом не делятся одновременно на `3` и `4`.

5. Напишите программу для создания множества, элементами которого являются кортежи (по два элемента в каждом) с нечетными числами: `(1,3)`, `(3,5)`, `(5,7)` и так далее.

6. Напишите программу, которая выполняется следующим образом. Пользователь вводит целое неотрицательное число. На основе этого числа создается список из натуральных чисел от `1` до этого числа. Затем на основе этого списка создается словарь. Элементы списка служат ключами для элементов словаря. Значения элементов словаря определяются на основе значений элементов списка, но взятых в обратном

порядке. Например, если пользователь вводит число 3, то создается список [1,2,3] и на его основе создается словарь из трех элементов. У элемента с ключом 1 значение 3, у элемента с ключом 2 значение 2, а у элемента с ключом 3 значение 1.

7. Напишите программу, которая выполняется следующим образом. Пользователь вводит текст. На основе этого текста создается словарь. Ключами словаря служат символы из текста, а значениями элементов словаря являются количества вхождений соответствующих символов в текст. Например, если пользователь вводит текст "ABBCAB", то словарь будет состоять из трех элементов с ключами "A", "B" и "C", а значения элементов соответственно равны 2 (в тексте 2 буквы "A"), 3 (в тексте 3 буквы "B") и 1 (в тексте 1 буква "C").

8. Напишите программу, в которой используется словарь. Ключами в словаре являются фамилии писателей, а значение соответствующего элемента — название произведения, написанного автором. В программе перебираются значения всех элементов словаря, и для каждого значения (название произведения) пользователю предлагается указать фамилию автора. После перебора содержимого словаря и получения всех ответов программа отображает количество правильных ответов пользователя.

9. Напишите программу, в которой пользователю предлагается ввести текстовое значение. На основе текста формируется словарь. Ключами элементов словаря являются символы из текста. Значение соответствующего элемента — это исходный текст, в котором «вычеркнут» тот символ, который является ключом. Если при формировании очередного элемента словаря окажется, что такой ключ уже есть, то соответствующий символ пропускается. Например, если пользователь ввел текст "ABCABD", то в словаре будут представлены элементы с ключами "A", "B", "C" и "D" со значениями соответственно "BCABD", "ACABD", "ABABD" и "ABCAB".

10. Напишите программу, в которой на основе двух словарей создается новый словарь. В этот новый словарь включатся те элементы, которые представлены в каждом из исходных словарей (имеются в виду ключи элементов). Значениями элементов в создаваемом словаре являются множества из значений соответствующих элементов в исходных словарях.

Глава 5
РАБОТА С ТЕКСТОМ

Какое-то загадочное явление природы.

Из м/ф «Приключения капитана Врунгеля»

В этой главе мы обсудим методы работы с *текстом*. В предыдущих главах мы часто использовали текстовые значения и научились выполнять простые операции с текстом. Вместе с тем, некоторые темы остались нерассмотренными. Поэтому мы попытаемся наверстать упущенное и отчасти повторим то, что уже знаем.

Текстовые литералы

Ты посмотри, как интересно! Похоже на вулкан.

Из м/ф «Приключения капитана Врунгеля»

Текстовый литерал представляет собой последовательность символов, заключенную в одинарные или двойные кавычки. Это самое общее правило. Дальше начинаются частности.

НА ЗАМЕТКУ

По умолчанию при отображении текстовых значений в области вывода обычно используются одинарные кавычки (если по контексту команды текст должен отображаться в кавычках), даже если исходный литерал создавался с использованием двойных кавычек.

Первый вопрос, который возникает, связан тем, как в самом текстовом литерале использовать двойные или одинарные кавычки (речь о ситуации, когда текст, формирующий текстовый литерал, содержит одинарные или двойные кавычки). Самый простой (но не всегда применимый) выход из ситуации состоит в следующем: если в тексте содержатся двойные

кавычки, то весь литерал заключается в одинарные кавычки. Если же текст содержит одинарные кавычки, то весь литерал можно заключить в двойные кавычки. Правда, такой вариант в силу разных причин тоже может быть неприемлемым. На этот случай существует другая стратегия: мы можем использовать обратную косую черту \ и после нее указать одинарные или двойные кавычки (в зависимости от потребностей). То есть мы можем использовать в текстовом литерале инструкции `\'` и `\"` для включения в литерал, соответственно, одинарных и двойных кавычек.

Сама по себе обратная косая черта \ используется для разбивки литерала на несколько строк в окне редактора кодов. Имеется в виду следующее: если текстовое значение слишком «длинное» для того, чтобы поместиться в одну строку, или в силу каких-то причин его нужно вводить в нескольких строках в окне редактора кодов, то в конце строки, в том месте, где выполняется перенос, ставится символ \. При отображении такого литерала в области вывода в соответствующем месте переноса строки не будет, а символ \ при этом не отображается. Обратная косая черта в литерале служит индикатором того, что перенос текстового значения выполнен осознанно.

НА ЗАМЕТКУ

Просто так разбить значение текстового литерала на несколько строк не получится — возникнет ошибка.

Обратная косая черта ставится в конце строки. После нее не должно быть никаких символов. Если после обратной косой черты какой-то символ все же поставить, то такая комбинация (обратная косая черта и символ после нее) будет интерпретироваться как управляющая инструкция.

Также обратная косая черта используется в *управляющих инструкциях*. Управляющая инструкция состоит из обратной косой черты и символа. Такие инструкции обозначают определенное действие или операцию, которые должны выполняться в процесс «отображения» инструкции — например, когда соответствующее текстовое значение передано аргументом функции `print()`. Скажем, инструкция `\n` в текстовом литерале означает, что в процессе отображения этого текста в области вывода с том месте, где размещена инструкция, выполняется переход к новой строке. Другими примером управляющей инструкции является инструкция табуляции `\t`. Выполнение этой инструкции сводится к тому, что курсор перемещается к ближайшей справа позиции табуляции.

ПОДРОБНОСТИ

Область вывода условно разбивается на последовательность позиций, в которых отображаются символы. Позиции табуляции — это равноудаленные позиции в этой последовательности. Обычно позициями табуляции являются 1-я, 9-я, 25-я и так далее (с интервалом, равным 8) позиции. В нашем случае интервал между позициями табуляции определяется настройками среды разработки.

НА ЗАМЕТКУ

Кроме инструкции перехода к новой строке `\n` и инструкции табуляции `\t`, есть и другие управляющие инструкции. Мы на них останавливаться не будем, поскольку не предполагается их использование. Кроме того, корректность применения этих управляющих инструкций достаточно сильно зависит от среды разработки и версии интерпретатора. Вообще, в некоторых версиях Python если интерпретатор «не узнает» управляющую инструкцию, то она обрабатывается как обычная последовательность символов. Однако следует понимать, что все равно такая ситуация является ошибочной, и нет гарантии, что в будущих версиях Python такой «либерализм» будет сохранен.

Если нам нужно в самом литерале использовать обратную косую черту (не как часть управляющей инструкции, а как самостоятельный символ), то используют инструкцию `\\`.

Достаточно удобный способ задавать текстовые литералы базируется на использовании трех пар двойных или одинарных кавычек. Такой текстовый литерал можно вводить (в окне редактора) в нескольких строках (без использования обратной черты в качестве переноса строки), и отображаться он будет так, как введен в окне редактора. Небольшой пример, в котором разными способами создаются текстовые литералы, представлен в листинге 5.1.

Листинг 5.1. Текстовые литералы

```
A="Язык 'Python' отличается от языка \"Java\"."
print(A)
B='Язык "Java" отличается от языка \'C++\'.'
print(B)
```

```
C="Серый\tЖелтый\tКрасный\nСиний\
\tБелый\tЗеленый"
print(C)
print("\\ Омар Хайям \\")
D="""Зачем копить добро в пустыне бытия?
  Кто вечно жил средь нас? Таких не видал я.
    Ведь жизнь нам в долг дана, и то — на срок недолгий,
      А то, что в долг дано, не собственность твоя."""
print(D)
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 5.1)

```
Язык 'Python' отличается от языка "Java".
Язык "Java" отличается от языка 'C++'.
Серый   Желтый  Красный
Синий   Белый   Зеленый
\ Омар Хайям \
Зачем копить добро в пустыне бытия?
  Кто вечно жил средь нас? Таких не видал я.
    Ведь жизнь нам в долг дана, и то — на срок недолгий,
      А то, что в долг дано, не собственность твоя.
```

В этой программе создается несколько текстовых значений, и затем эти значения отображаются в области вывода. В частности, мы встречаем примеры использования одинарных и двойных кавычек в текстовых значениях, которые записываются в переменные `A` и `B`. Значение, которое записывается в переменную `C`, содержит несколько инструкций табуляции `\t`, инструкцию перехода к новой строке `\n` и инструкцию переноса литерала в следующую строку `\`.

В команде `print("\\ Омар Хайям \\")` мы использовали текстовый литерал, содержащий двойную обратную косую черту `\\`. С помощью такой комбинации мы отображаем в области вывода одинарную обратную косую черту. Наконец, литерал, присваиваемый в качестве значения переменной `D`, заключается в тройные «двойные кавычки». Такой

литерал отображается так, как он введен в окне редактора, с учетом всех переносов и отступов.

Текстовые литералы можно создавать с использованием специальных префиксов. Префикс — это специальный символ (или символы), который указывается непосредственно перед текстовым литералом. Например, если используется префикс `r` или `R`, то такой литерал обрабатывается в режиме, в котором обратная косая черта `\` интерпретируется как обычный символ (созданный с помощью префикса `r` или `R` литерал называется *необработанным* или *сырым*). Скажем, выражение `"\n"` представляет сбой литерал, который состоит из инструкции перехода к новой строке `\n`. Эта инструкция интерпретируется как один символ. А вот выражение `r"\n"` представляет собой литерал, который состоит из двух символов: это символы `\` и `n`.

С использованием префикса `f` или `F` создаются *форматированные литералы*. Главное преимущество форматированных литералов состоит в том, что они могут содержать *поля замены*. Это специальные инструкции, определяющие, какие значения и в каком формате должны быть добавлены в текстовый литерал в соответствующем месте. Поле замены представляет собой инструкцию, заключенную в фигурные скобки. Внутри этих скобок указывается название переменной, значение которой вставляется в соответствующем месте в текстовом литерале. Также в них могут содержаться дополнительные коды, определяющие формат (вид и способ) представления значения в литерале. Детали лучше рассмотреть на конкретном примере. В листинге 5.2 представлена программа, в которой есть примеры использования литералов с префиксами.

Листинг 5.2. Использование префиксов в литералах

```
# Текстовый литерал без префикса:
A="\"Java\"\n\"Python\""
print(A)
print("Символов:", len(A))
# Текстовый литерал с префиксом:
B=r"\"Java\"\n\"Python\""
print(B)
print("Символов:", len(B))
```

```
# Переменная с текстовым значением:
name="Python"
# Текстовый литерал с префиксом:
C=f"Язык {name} — простой и понятный"
print(C)
C=f"Язык {name!r} — простой и понятный"
print(C)
# Переменная с числовым значением:
num=12.34567
# Текстовый литерал с префиксом:
txt=f"Число: {num:9.3f}"
print(txt)
txt=f"Число: {num:09.3f}"
print(txt)
# Новое числовое значение переменной:
num=42
# Формат для отображения целого числа:
txt=f"Число: {num:*>9d}"
print(txt)
# Формат для отображения шестнадцатеричного числа:
txt=f"Число: {num:#09x}"
print(txt)
txt=f"Число: {num:9x}"
print(txt)
txt=f"Число: {num:*<9x}"
print(txt)
# Формат для отображения восьмеричного числа:
txt=f"Число: {num:*^#09o}"
print(txt)
# Формат для отображения двоичного числа:
txt=f"Число: {num:#9b}"
print(txt)
```

Результат выполнения программы следующий.

Результат выполнения программы (из листинга 5.2)

```
"Java"
"Python"
Символов: 15
\"Java\"\n\"Python\"
Символов: 20
Язык Python — простой и понятный
Язык 'Python' — простой и понятный
Число:    12.346
Число: 00012.346
Число: *******42
Число: 0x000002a
Число:        2a
Число: 2a*******
Число: **0o52***
Число:  0b101010
```

Сначала в программе переменной `A` в качестве значения присваивается текстовый литерал `"\"Java\"\n\"Python\""`. Он содержит управляющие инструкции, в том числе и инструкцию переноса строки `\n`. При отображении значения переменной `A` получаем вполне ожидаемый результат. Переменной `B` в качестве значения присваивается литерал `r"\"Java\"\n\"Python\""` с префиксом `r` (именно префиксом `r` данный литерал отличается от предыдущего). Но при отображении значения переменной `B` получаем совершенно иной результат. На этот раз все управляющие инструкции интерпретируются как обычная последовательность символов. Более того, результаты выражений `len(A)` и `len(B)`, определяющие длину соответствующих текстовых значений, различны. Причина та же — в случае с переменной `B` управляющие инструкции обрабатываются как обычные последовательности символов.

Еще один пример литерала с префиксом (на этот раз использован префикс `f`) дается командой `C=f"Язык {name} — простой и понятный"`. При этом предварительно переменной `name` присвоено текстовое

значение "Python". Текстовое значение переменной C формируется так: в присваиваемый в качестве значения литерал в месте размещения инструкции {name} вставляется значение переменной name.

ПОДРОБНОСТИ

Вместо инструкции {name} вставляется то значение переменной name, которое она имеет на момент присваивания значения переменной C. Если впоследствии значение переменной name изменится, то на значение переменной C это уже никак не повлияет.

Это самый простой способ инкапсулировать (вставить) в текстовый литерал значение переменной. В общем случае, как отмечалось, поле замены может содержать инструкции (или коды) форматирования. Например, если вместо инструкции {name} использовать {name !r}, то результат будет несколько иным — в текстовом литерале значение переменной name будет заключено в кавычки. Инструкция !r после имени переменной name означает, что перед вставкой значения переменной используется специальное преобразование. В данном случае вызывается функция repr(), с помощью которой для переменной name вычисляется текстовое представление, и уже оно вставляется в литерал. В результате таких манипуляций значение «обрастает» кавычками.

ПОДРОБНОСТИ

Практически для каждого объекта предусмотрено или может быть предусмотрено текстовое представление, применимое для отображения в области вывода. Получить такое представление можно с помощью функции repr().

Кроме инструкции !r, после имени переменной в поле замены можно использовать инструкцию !s (в этом случае для выполнения преобразования вызывается функция str()) и!a (для выполнения преобразования вызывается функция ascii()). Во всех этих случаях получаем текстовое представление для значения переменной — разница лишь в алгоритмах получения этого преобразования и способе обработки некоторых символов.

Далее в программе представлены примеры отображения в разном формате числовых значений. Числовое значение записывается в переменную num (начальное значение переменной равно 12.34567). При этом переменной txt в качестве значения присваивается литерал f"Число: {num:9.3f}". В нем поле замены {num:9.3f} означает, что

в соответствующем месте должно быть вставлено значение переменной num. Под переменную выделяется поле шириной в 9 позиций, после запятой отображается 3 цифры. Об этом свидетельствует выражение 9.3 в поле замены. Символ f в поле замены означает, что отображается число в формате с плавающей точкой.

Поле замены, определенное инструкцией {num:09.3f}, означает, что, как и в предыдущем случае, под число в формате с плавающей точкой выделяется 9 позиций и после десятичной точки отображаются 3 цифры. Но на этот раз все «лишние» позиции, выделенные под число, будут заполнены нулями. Этот эффект достигается за счет цифры 0, указанной перед инструкцией 9.3f.

На следующем этапе переменной num присваивается целочисленное значение 42. Это значение отображается с использованием различных форматов. Так, поле замены в текстовом литерале, реализованное инструкцией {num:*>9d}, означает, что отображается целое число (символ d в инструкции), под которое выделяется 9 позиций. Символ > перед числом 9 означает, что выравнивание значения в выделенной под него области выполняется по правому краю. Символ * в инструкции, определяющей поле замены, означает, что «лишние» позиции, выделенные под число, будут заполнены символом «звездочка» *.

НА ЗАМЕТКУ

В качестве символа для заполнения «лишних» позиций не обязательно использовать «звездочку» — это может быть и другой символ.

Для выравнивания содержимого по левому краю в поле замены используют символ <. Если использовать символ ^, то выравнивание будет выполняться по центру. Также можно использовать символ =. В этом случае если число отображается со знаком, то между знаком и цифрами в представлении числа выполняется разрыв (так чтобы число занимало всю выделенную под него область), и этот разрыв заполняется соответствующим символом (указанным перед символом, определяющим способ выравнивания).

Инструкция {num:9x} определяет режим отображения значения переменной num в виде шестнадцатеричного числа (такой формат задается символом x) с выделением 9 позиций для отображения числа. По умолчанию значение в выделенной для отображения области выравнивается по правому краю. Все «лишние» позиции остаются незаполненными.

ПОДРОБНОСТИ

В шестнадцатеричном представлении целое число записывается в виде $a_n a_{n-1}...a_2 a_1 a_0$, причем параметры a_k ($k = 0,1,2,..., n$) могут принимать значения от 0 до 9 или это могут быть символы от A до F (так обозначаются числа соответственно от 10 до 15). В десятичную систему такое число переводится так: $a_n a_{n-1}...a_2 a_1 a_0 = a_0 16^0 + a_1 16^1 + a_2 16^2 + ... + a_{n-1} 16^{n-1} + a_n 16^n$. Аналогично определяются и прочие системы счисления. Например, в восьмеричной системе счисления в представлении числа $a_n a_{n-1}...a_2 a_1 a_0$ параметры a_k могут принимать значения от 0 до 7 включительно, а для перевода числа в десятичную систему используется выражение $a_n a_{n-1}...a_2 a_1 a_0 = a_0 8^0 + a_1 8^1 + a_2 8^2 + ... + a_{n-1} 8^{n-1} + a_n 8^n$. Аналогично, в двоичной системе параметры a_k могут принимать значения 0 или 1, а в десятичную систему перевод выполняется в соответствии с соотношением $a_n a_{n-1}...a_2 a_1 a_0 = a_0 2^0 + a_1 2^1 + a_2 2^2 + ... + a_{n-1} 2^{n-1} + a_n 2^n$.

Что касается числа 42 (в десятичной системе), поскольку $42 = 2 \cdot 16^1 + 10 \cdot 16^0$, а число 10 в десятичной системе соответствует символу A, то в шестнадцатеричной системе это число записывается как 2A. В восьмеричной системе это будет 52, поскольку $42 = 5 \cdot 8^1 + 2 \cdot 8^0$. В двоичной системе получаем код 101010, поскольку $42 = 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1$.

Поле замены заданное инструкцией `{num:#09x}` означает отображение значения переменой `num` в шестнадцатеричном виде с выделением `9` позиций под число и заполнением «лишних» позиций нулями. Поскольку в инструкции использован символ `#`, то перед шестнадцатеричным кодом отображается префикс `0x`.

НА ЗАМЕТКУ

Символ `#` определяет специальный режим для отображения числовых значений. Для каждого числового типа (целые числа, числа с плавающей точкой, комплексные числа) этот режим имеет свои особенности. Например, для целых чисел в двоичном, восьмеричном и шестнадцатеричном представлении данный режим означает, что перед собственно значением числа будет отображаться соответственно префикс `0b`, `0o` и `0x`.

Инструкцию `{num:*<9x}` следует понимать так: значение переменной `num` отображается в шестнадцатеричном формате, под число выделяется `9` позиций, выравнивание выполняется по левому краю (символ `<`), а «лишние» символы заполняются «звездочками» `*`.

Поле замены, заданное инструкцией `{num:*^#09o}`, означает, что число отображается в восьмеричном представлении (символ `o` в инструкции), под число выделяется `9` позиций, в представлении числа отображается префикс `0o` (символ `#` в инструкции), выравнивание выполняется по центру (символ `^` в инструкции), а «лишние» позиции заполняются «звездочкой» `*`.

Наконец, в соответствии с инструкцией `{num:#9b}` числовое значение отображается в двоичном представлении (символ `b` в инструкции), под значение выделяется поле шириной в `9` символов, а в представлении используется префикс `0b` (символ `#` в инструкции).

ПОДРОБНОСТИ

В общем случае инструкция, определяющая поле замены, может содержать следующие параметры.

- Первым указывается символ, которым заполняются «лишние» позиции, выделенные для отображения значения.
- Затем указывается символ (`<`, `>`, `^` или `=`), определяющий способ выравнивания в поле, выделенном для отображения значения.
- Можно задать режим отображения знака для числовых значений. Если указан «плюс» `+`, то знак будет отображаться и для положительных, и для отрицательных чисел. Если указан «минус» `-`, то знак будет отображаться только для отрицательных чисел (режим используется по умолчанию). Также можно указать пробел. В этом случае для положительных чисел перед числом делается отступ, а отрицательные числа отображаются со знаком.
- Следующим может указываться символ `#`. Как отмечалось ранее, в этом случае используется специальный режим отображения значений, и для каждого конкретного типа этот режим имеет свои особенности.
- Если для числовых значений указан `0`, то «лишние» позиции заполняются нулями.
- Следующей опцией является числовое значение, определяющее ширину поля, выделяемого для отображения значения. Это минимальная ширина поля — если значение не помещается в выделенном поле, то ширина поля автоматически увеличивается.
- Можно указать символ (запятая `,` или символ подчеркивания `_`), который будет использоваться для выделения тысячных разрядов.
- Через точку указывается точность отображения числового значения. Это количество цифр после десятичной точки, которые будут отображаться в представлении числа.

- С помощью специального символа задается тип отображаемого значения. Используются такие символы: `s` (соответствует тексту), `b` (двоичное представление для числа), `c` (означает символ — фактическое целочисленное значение интерпретируется как код символа в кодовой таблице), `d` или `n` (целое число), `o` (восьмеричное представление для числа), `x` или `X` (шестнадцатеричное представление для числа), `e` или `E` (экспоненциальное представление для действительного числа), `f` или `F` (число в формате с плавающей точкой), `g` или `G` (общий формат — способ отображения числового значения зависит от фактического значения числа), % (проценты).

Это самая общая конструкция. На практике много частных случаев, в которых перечисленные параметры встречаются в разных комбинациях. Здесь мы подобные ситуации рассматривать не будем, поскольку это не является нашей целью.

В контексте создания форматированных текстовых значений уместно вспомнить также метод `format()`. Метод вызывается из текстового значения, которое содержит поля замены (этот текст будем называть *строкой форматирования*). Аргументами методу передаются значения, которые вставляются в строку форматирования вместо полей замены. Тот текст, который получается в результате вставки, возвращается результатом метода.

В строке форматирования поля замены формируются фактически по тем же правилам, что и в форматированных текстовых литералах, описанных выше. Однако вместо названия переменных, значения которых вставляются в строку, обычно указываются целочисленные индексы (начиная с нуля), определяющие аргумент метода `format()`, предназначенный для вставки в строку форматирования. Например, результат выражения `"Число {0}, текст {1} и снова число {2}".format(123,"Python",321)` вычисляется так: в текстовую строку `"Число {0}, текст {1} и снова число {2}"` вместо инструкции `{0}` вставляется аргумент `123` метода `format()`, вместо инструкции `{1}` вставляется текст `"Python"`, а вместо инструкции `{2}` вставляется число `321`. Если, как в рассмотренном случае, аргументы в строку форматирования вставляются в том порядке, как они переданы методу `format()`, то индексы аргументов можно не указывать. Поэтому вместо приведенной выше инструкции мы могли бы использовать более простой вариант: `"Число {}, текст {} и снова число {}".format(123,"Python",321)` (результат будет таким же). Небольшой

пример, в котором для создания форматированных текстовых значений используется метод `format()`, представлен в листинге 5.3.

Листинг 5.3. Создание форматированного текста

```
A="Число {}, текст {} и снова число {}"
txt=A.format(123,"Python",321)
print(txt)
txt="Число {0} — это {0: b} или {0: x}".format(42)
print(txt)
txt="Код: {0:05d}, символ: {0:*^5c}".format(65)
print(txt)
txt="Число: {:_>+20.3E}".format(123.468)
print(txt)
B="{0:_{2}{1}s}"
num=6
for k in range(1, num+1):
    print(B.format("*", k,">"), end="")
    print(" "*(2*(num-k)), end="")
    print(B.format("*", k,"<"))
```

Каким будет результат выполнения программы, показано ниже.

Результат выполнения программы (из листинга 5.3)

```
Число 123, текст Python и снова число 321
Число 42 — это 101010 или 2a
Код: 00065, символ: **A**
Число: _________+1.235E+02
*          *
 *        *
_          _
 _*      *_
__        __
  _*    *_
___      ___
   _*  *_
____    ____
     **
_____  _____
```

В начале программы командой A="Число {}, текст {} и снова число {}" в переменную A записывается текстовый литерал с тремя пустыми фигурными скобками. Затем выполняется команда txt=A.format(123,"Python",321), в которой из переменной A вызывается метод format(). У метода три аргумента. Значение первого аргумента вставляется вместо первой пары фигурных скобок, второй аргумент замещает вторую пару фигурных скобок, а третий аргумент замещает третью пару фигурных скобок. Результат проверяем с помощью команды print(txt).

Командой txt="Число {0} — это {0: b} или {0: x}".format(42) создается текстовое значение, которое формируется следующим образом: в трех местах в строку форматирования вставляется значение единственного целочисленного аргумента метода format(). Но вставка выполняется в разных форматах: обычное, не отформатированное значение (инструкция {0}), число в двоичном представлении (инструкция {0: b}) и число в шестнадцатеричном представлении (инструкция {0: x}).

Еще один пример отображения одного и того же значения в разных форматах дается командой txt="Код: {0:05d}, символ: {0:*^5c}".format(65). В данном случае целочисленное значение 65 отображается как целое число с выделением 5 позиций и заполнением «лишних» позиций нулями (инструкция {0:05d}) и как символ, под отображение которого выделяется 5 позиций, «лишние» позиции заполняются «звездочками» * и выравнивание выполняется по центру (инструкция {0:*^5c}). В последнем случае число преобразуется в символ так: в кодовой таблице определяется символ, код которого равен данному числу (в нашем случае числу 65 соответствует символ 'A').

Инструкция {:_>+20.3E} не содержит индекс аргумента метода format(), поэтому при выполнении команды txt="Число: {:_>+20.3E}".format(123.468) в соответствующее место будет вставляться единственный аргумент 123.468 метода format(). При этом используется такой формат: используется экспоненциальное представление для числа (символ E), под число выделяется поле шириной в 20 позиций, после десятичной точки отображается 3 цифры, отображается знак числа (символ +), выравнивание выполняется по правому краю (символ >), а «лишние» позиции заполняются символом подчеркивания _.

ПОДРОБНОСТИ

В экспоненциальном представлении действительное число задается с помощью мантиссы и показателя степени. При отображении числа в таком формате мантисса и показатель степени разделяются символом `e` или `E`. Значение числа определяется как произведение мантиссы на десять в соответствующей степени. Например, выражение `1.5e2` соответствует числу `150`, а выражение `2.3E-2` соответствует числу `0.023`.

Экспоненциальную нотацию можно использовать для определения литералов, являющихся действительными числами.

Последний блок команд в программе требует дополнительных пояснений. В нем командой `B="{0:_{2}{1}s}"` в переменную `B` записывается текст, содержащий инструкцию форматирования `{0:_{2}{1}s}`, а также выполняется команда `num=6`. После этого запускается оператор цикла, в котором переменная `k` пробегает значения от `1` до значения переменной `num` включительно. За каждый цикл выполняется три команды. Командой `print(B.format("*", k,">"), end="")` отображается текст, созданный инструкцией `B.format("*", k,">")`. Текст получается следующим образом. В строку форматирования вместо инструкции `{0:_{2}{1}s}` вставляется значение первого аргумента метода `format()`. Речь о значении `"*"`, которое вставляется как текст (символ `s` в инструкции форматирования). При этом сама инструкция форматирования содержит внутренние блоки из фигурных скобок. Эти блоки определяют параметры форматирования. Так, вместо инструкции `{2}` вставляется значение третьего аргумента метода `format()`. Это символ `"<"`. Он определяет способ выравнивания по левому краю значения, которое вставляется вместо инструкции `{0:_{2}{1}s}`. Вместо инструкции `{1}` вставляется значение переменной `k`. Этот параметр определяет ширину поля, выделенного для вставки значения. Получается, что за каждый цикл ширина этого поля увеличивается на единицу. При этом «лишние» символы заполняются символом подчеркивания `_`, указанным в инструкции форматирования.

Командой `print(" "*(2*(num-k)), end="")` отображается текст, который формируется выражением `" "*(2*(num-k))`. В этом выражении текст умножается на число. Результатом такого выражения является текст, который формируется повторением умножаемого текстового значения. Количество повторений определяется числом, на которое умножается текст. Результатом выражения `" "*(2*(num-k))` является

текст, который состоит из пробелов, причем количество этих пробелов определяется значением выражения `2*(num-k)`. Таким образом, за каждый цикл количество пробелов в тексте уменьшается на два. Наконец, командой `print(B.format("*", k,"<"))` отображается символ «звездочка» `*`, ширина выеденного поля определяется значением переменной `k`, «лишние» позиции заполняются символом подчеркивания `_`, а выравнивание выполняется по левому краю.

Основные операции с текстом

Простите, вы кого хотели обмануть?

Из м/ф «Приключения капитана Врунгеля»

Далее мы рассмотрим основные операции, которые могут выполняться с текстом. Здесь в первую очередь следует отметить, что текст во многом похож на список. Элементами являются отдельные символы. Главное отличие текста от списка состоит в том, что в тексте нельзя заменить один символ на другой. В подобных ситуациях обычно создается новый текст и присваивается в качестве значения исходной переменной. В результате создается иллюзия, что текст изменился (хотя на самом деле это просто другой текст).

Как мы уже знаем, количество символов в тексте можно узнать с помощью функции `len()`, которой в качестве аргумента передается текстовое значение. Для объединения (конкатенации) двух текстовых значений используют оператор `+`. Если `A` и `B` — переменные с текстовыми значениями, то результатом выражения `A+B` является текст, который получается объединением значений переменных `A` и `B`. В случае если речь идет об объединении двух текстовых литералов, можно воспользоваться такие приемом: если указать через пробел два текстовых литерала, то в результате получим новый литерал, который получается объединением исходных. Скажем, результатом выражения `"A" "B"` является текст `"AB"`.

Как отмечалось выше, если умножить текст на целое число или целое число умножить на текст, то в результате получим новое текстовое значение, которое получается повторением того текста, который умножался на число. Числовой множитель определяет количество повторений текста. Например, значением выражения `3*"A"` является текст `"AAA"`.

НА ЗАМЕТКУ

Имеет смысл напомнить, что для преобразования некоторых значений в текст может использоваться функция `str()`. Например, если передать этой функции в качестве аргумента целое число, то в результате получим текстовое представление для этого числа. Так, результатом выражения `str(123)` является текст `"123"`.

Если нам нужно по текстовому представлению для числа получить само число, то можно использовать, например, функции `float()` (для действительных чисел) и `int()` (для целых чисел). Так, результатом выражения `float("12.3")` является число `12.3`, а значением выражения `int("123")` является число `123`.

ПОДРОБНОСТИ

Во многих языках программирования есть специальный символьный тип. Значением переменной символьного типа может быть отдельный символ (буква). В Python как такого символьного типа нет. Отдельный символ реализуется как текст, состоящий из одного символа. Есть две достаточно важные операции, которые имеют отношение к обработке именно одного символа. Во-первых, это получение по символу его кода в кодовой таблице. Такая операция выполняется с помощью функции `ord()`. Аргументом передается символ (текст, содержащий один символ), а результатом является код символа в кодовой таблице. Например, результатом выражения `ord("A")` является число `65`. Это код большой английской буквы `"A"` в кодовой таблице ASCII. Все буквы в кодовой таблице следуют в том порядке, как они размещены в алфавите. Поэтому код буквы `"B"` равен `66`, код буквы `"C"` равен `67` и так далее. Код маленькой буквы больше кода соответствующей большой буквы на `32` (код буквы `"a"` равен `97`, код буквы `"b"` равен `98`, код буквы `"c"` равен `99`, и так далее). Само число `32` является кодом такого символа, как пробел. В этом смысле можно утверждать, что между маленькой и большой буквой — пробел.

Обратная операция — получение символа на основе кода. Эта задача решается с помощью функции `chr()`. В качестве аргумента функции передается код символа, а результатом возвращается текст, содержащий этот символ. Так, результатом выражения `chr(65)` является текст `"A"`.

Как и списки, текстовые значения можно индексировать и, в частности, выполнять срезы. Текстовое значение можно использовать в качестве перебираемого множества в операторе цикла `for`. В этом случае

последовательно перебираются символы, формирующие текст (причем в том порядке, как они размещены в тексте). Собственно, этих операций часто вполне достаточно для решения многих задач. В качестве иллюстрации рассмотрим программу в листинге 5.4.

Листинг 5.4. Базовые операции с текстом

```
# Исходный текст:
txt="Hello Python"
print(txt)
# Текст в обратном порядке:
A=txt[::-1]
print(A)
# Первое слово в тексте:
B=txt[:5]
print(B)
# Последнее слово в тексте:
C=txt[6:]
print(C)
# Переменная с текстовым значением:
new_txt=""
# Переменная с целочисленным значением:
delta=ord("a")-ord("A")
# Перебор символов в тексте:
for s in txt:
    # Если буква в диапазоне от "a" до "z":
    if(ord(s)>=ord("a") and ord(s)<=ord("z")):
        s=chr(ord(s)-delta)
    # Добавление символа к тексту:
    new_txt+=s
# Текст из больших букв:
print(new_txt)
```

Результат выполнения программы ниже.

Результат выполнения программы (из листинга 5.4)

```
Hello Python
nohtyP olleH
Hello
Python
HELLO PYTHON
```

В качестве базового мы используем текстовое значение, которое определяется командой `txt="Hello  Python"`. После этого для данного текстового значения выполняется несколько срезов. Так, командой `A=txt[::-1]` создается срез, который содержит все символы из исходного текста, но в обратном порядке.

НА ЗАМЕТКУ

Третий параметр `-1` в квадратных скобках определяет приращение по индексу при получении среза. Первые два параметра не указаны, и это в данном случае означает, что в срез включаются все символы с последнего и до первого.

Командой `B=txt[:5]` создается срез, который состоит из символов, начиная с первого (с индексом `0`) и до символа с индексом `4` включительно (это первое слово в исходном тексте). Командой `C=txt[6:]` создается срез из символов, начиная с символа с индексом `6` и до последнего символа в тексте (это второе — оно же последнее слово в тексте).

Кроме выполнения срезов, в программе в «ручном» режиме выполняется замена в исходном тексте маленьких букв на большие. Точнее, на основе исходного текста мы создаем новый, в котором все буквы большие. Выполняется эта операция следующим образом. Сначала создается переменная `new_txt` с пустым текстовым литералом `""` в качестве начального значения. Также мы используем переменную `delta` со значением, которое вычисляется выражением `ord("a")-ord("A")`. Это разность кодов английских букв `"a"` и `"A"`.

НА ЗАМЕТКУ

Если используется кодовая таблица ASCII, то разница кодов букв `"a"` и `"A"` равна `32`. Мы для надежности вычисляем это значение напрямую.

> Также стоит отметить, что для перевода маленьких букв в большие существует специальный метод `upper()`.

Перебор символов в тексте осуществляется с помощью оператора цикла `for`, в котором переменная `s` последовательно принимает значения символов из текста `txt`. Если прочитанный символ является маленькой буквой из диапазона от `"a"` до `"z"`, то от кода прочитанного символа отнимается значение, записанное в переменную `delta`, и полученное числовое значение преобразуется в символ (текст, содержащий один символ). Все эти операции реализуются следующим образом. Попадание считанного символа в нужный диапазон определяется как условие `ord(s)>=ord("a") and ord(s)<=ord("z")`, состоящее в том, что код прочитанного символа не меньше кода буквы `"a"` и не больше кода буквы `"z"`. Если условие истинное, то выполняется команда `s=chr(ord(s)-delta)`. Это как раз операция, связанная с изменением кода символа. Новый вычисленный код передается в качестве аргумента функции `chr()`. Вне зависимости от того, изменялось значение переменной `s` или нет, командой `new_txt+=s` к текущему значению переменной `new_txt` дописывается текущее содержимое переменной `s`.

Методы для работы с текстом

> Только не это... Только не это, только не это, Шеф!
>
> *Из м/ф «Приключения капитана Врунгеля»*

Помимо основных операций, которые выполняются с помощью операторов и индексирования текстовых значений, существует достаточно много специальных методов, позволяющих выполнять всевозможные операции с текстом. С некоторыми из них мы познакомимся далее.

Существует группа методов, которые используются для изменения регистра символов.

- Метод `upper()` позволяет на основе текста, из которого вызывается метод, получить текст, в котором все буквы большие.
- Если нужно получить текст, состоящий из маленьких букв, используют метод `lower()`.

- Метод `swapcase()` позволяет создать текст, в котором, по сравнению с исходным, большие буквы заменены на маленькие, а маленькие — на большие.
- Результатом вызова метода `title()` является текст, в котором каждое слово начинается с большой буквы. Все прочие буквы — маленькие.
- В результате вызова метода `capitalize()` получаем текст, в котором первое слово начинается с большой буквы. Все прочие буквы — маленькие.

Небольшой пример, в котором используются эти методы, представлен в листинге 5.5.

Листинг 5.5. Изменение регистра символов

```
txt="Язык PYTHON проще, чем язык JAVA!"
print(txt)
print(txt.upper())
print(txt.lower())
print(txt.swapcase())
print(txt.title())
print(txt.capitalize())
```

Результат выполнения программы выглядит следующим образом.

Результат выполнения программы (из листинга 5.5)

```
Язык PYTHON проще, чем язык JAVA!
ЯЗЫК PYTHON ПРОЩЕ, ЧЕМ ЯЗЫК JAVA!
язык python проще, чем язык java!
яЗЫК python ПРОЩЕ, ЧЕМ ЯЗЫК java!
Язык Python Проще, Чем Язык Java!
Язык python проще, чем язык java!
```

Программа простая, и комментариев, хочется верить, не требует.

Группа методов позволяет проверить текст на соответствие тем или иным критериям (такие методы начинаются со слова *is*). Результатом такие методы возвращают значение `True` или `False` в зависимости от того, выполняется критерий или нет. Вот некоторые из этих методов.

- Метод `isalnum()` в качестве результата возвращает значение `True`, если текст непустой и состоит из букв и/или чисел. В противном случае возвращается значение `False`.
- Метод `isalpha()` возвращает значение `True`, если текст непустой и состоит из букв. В противном случае возвращается значение `False`.
- Метод `isascii()` возвращает значение `True`, если текст пустой или состоит из символов кодовой таблицы ASCII. В противном случае возвращается значение `False`.
- Метод `isdecimal()` возвращает значение `True`, если текст непустой и состоит из десятичных цифр. В противном случае возвращается значение `False`.
- Метод `isdigit()` возвращает значение `True`, если текст непустой и состоит из цифр. В противном случае возвращается значение `False`.
- Метод `isidentifier()` в качестве результата возвращает значение `True`, если текст содержит название зарегистрированного в языке идентификатора или ключевого слова. В противном случае возвращается значение `False`.
- Метод `islower()` возвращает значение `True`, если текст непустой и состоит из маленьких букв. В противном случае возвращается значение `False`.
- Метод `isnumeric()` возвращает значение `True`, если текст непустой и состоит из числовых символов. В противном случае возвращается значение `False`.
- Метод `isprintable()` возвращает значение `True`, если текст пустой или состоит из печатных (тех, которые отображаются в области вывода) символов. В противном случае возвращается значение `False`.
- Метод `isspace()` возвращает значение `True`, если текст непустой и состоит из пробелов. В противном случае возвращается значение `False`.
- Метод `istitle()` возвращает значение `True`, если текст непустой и каждое слово в тексте начинается с большой буквы. В противном случае возвращается значение `False`.
- Метод `isupper()` возвращает значение `True`, если текст непустой и состоит из больших букв. В противном случае возвращается значение `False`.

НА ЗАМЕТКУ

Различие между методами `isdigit()`, `isnumeric()` и `isdecimal()` в основном касается того, как они «реагируют» на достаточно «экзотические» случаи. Желающие могут изучить этот вопрос самостоятельно.

Следующая группа методов предназначена для поиска и замены в тексте символов и подстрок.

НА ЗАМЕТКУ

Еще раз подчеркнем, что символ — это на самом деле текст, состоящий из одного символа.

Во всех случаях, когда речь идет о замене символов и подстрок в тексте, на самом деле имеется в виду, что на основе исходного текста создается новое текстовое значение.

- Для поиска символа или подстроки в тексте используются методы `find()`, `rfind()`, `index()` и `rindex()`. Искомая подстрока указывается аргументом метода. Методы `find()` и `index()` возвращают в качестве результата индекс первого вхождения подстроки в текст. Если подстроки в тексте нет, то метод `find()` возвращает значение `-1`, а метод `index()` генерирует исключение класса `ValueError`. Методам `find()` и `index()` кроме первого аргумента (подстрока для поиска) можно передать второй и третий аргументы. Они определяют срез, в котором будет выполняться поиск подстроки. Методы `rfind()` и `rindex()` отличаются от соответственно методов `find()` и `index()` тем, что выполняется поиск не первого, а последнего хождения подстроки в текст.
- Метод `count()` используется для определения количества вхождений подстроки или символа (первый аргумент метода) в текст, из которого вызывается метод. Методу можно передать еще два аргумента для определения среза, в котором будет выполняться поиск подстроки.
- Метод `endswith()` позволяет определить, заканчивается ли текст, из которого вызывается метод, подстрокой, переданной аргументом методу. Передав еще два аргумента методу, можно определить срез, для которого выполняется проверка (на предмет наличия в конце данной подстроки).

- Метод `startswith()` позволяет определить, начинается ли текст, из которого вызывается метод, подстрокой, переданной аргументом методу. Передав еще два аргумента методу, можно определить срез, для которого выполняется проверка (на предмет наличия в начале данной подстроки).
- Метод `expandtabs()` позволяет на основе исходного текста получить текст, в котором все инструкции табуляции заменены на соответствующее количество пробелов. В качестве аргумента методу можно передать числовое значение, определяющее отступ между соседними позициями табуляции (по умолчанию это значение равно `8`).
- Методы `strip()`, `lstrip()` и `rstrip()` позволяют удалять начальные и/или конечные символы в тексте (точнее, создается новый текст, у которого, по сравнению с исходным, удалены начальные и/или конечные символы). В качестве аргумента методам можно передавать текстовую строку с символами, предназначенными для удаления. Если аргумент методу не передан, то по умолчанию удаляются пробелы. Методом `strip()` удаляются символы в начале и конце текста, методом `lstrip()` удаляются символы в начале текста, методом `rstrip()` символы удаляются в конце текста.
- Методом `replace()` выполняется замена в тексте одной подстроки на другую подстроку. Замена выполняется в тексте, из которого вызывается метод. В результате создается новая текстовая строка. Заменяемая подстрока указывается первым аргументом метода. Строка, на которую выполняется замена, — второй аргумент метода. Можно указать и третий аргумент, который будет в таком случае определять количество выполняемых замен.

Далее мы рассмотрим небольшие программы с примерами того, как могут использоваться некоторые из перечисленных методов. В листинге 5.6 представлена программа, при выполнении которой пользователь вводит текст, а затем в этом тексте подсчитывается количество определенных символов (при этом используется метод `count()`).

Листинг 5.6. Подсчет символов в тексте

```
txt=input("Введите текст: ")
symb=input("Какую букву найти? ")
num=txt.count(symb)
```

```
if num==0:
    print("Такой буквы в тексте нет!")
else:
    print(f"В тексте {num} букв(ы) '{symb}'")
```

Так может выглядеть результат выполнения программы (здесь и далее жирным шрифтом выделено значение, которое вводит пользователь).

Результат выполнения программы (из листинга 5.6)

```
Введите текст: Программировать нужно правильно
Какую букву найти? р
В тексте 4 букв(ы) 'р'
```

Или таким.

Результат выполнения программы (из листинга 5.6)

```
Введите текст: Программировать нужно правильно
Какую букву найти? ы
Такой буквы в тексте нет!
```

В данном случае все просто: пользователь вводит текст и символ (хотя формально это может быть еще один текст), а затем с помощью метода `count()` выполняется подсчет количества вхождений символа в текст.

Похожая на предыдущую программа, в которой не просто выполняется подсчет количества символов, а определяются еще и позиции этих символов (для этого используется метод `find()`), представлена в листинге 5.7.

Листинг 5.7. Определение позиций символа в тексте

```
txt=input("Введите текст: ")
symb=input("Какую букву найти? ")
num=txt.find(symb)
L=[]
while num!= -1:
```

```
    L.append(num)
    num=txt.find(symb, num+1)
if len(L)==0:
    print("Такой буквы в тексте нет!")
else:
    print(f"Позиции буквы '{symb}' в тексте: {L}")
```

В зависимости от того, что вводит пользователь, результат выполнения программы может быть разным, например таким.

Результат выполнения программы (из листинга 5.7)

```
Введите текст: Программировать нужно правильно
Какую букву найти? р
Позиции буквы 'р' в тексте: [1, 4, 9, 23]
```

Или таким.

Результат выполнения программы (из листинга 5.7)

```
Введите текст: Программировать нужно правильно
Какую букву найти? ы
Такой буквы в тексте нет!
```

После того как в переменные `txt` и `symb` записывается, соответственно, текст и символ для поиска в тексте, командой `num=txt.find(symb)` вычисляется индекс первого вхождения символа в текст. Если такого символа в тексте нет, метод `find()` возвращает значение `-1`. Также мы создаем пустой список `L`. В этот список будут заноситься индексы вхождения символа в текст. После этого запускается оператор цикла `while`, который выполняется, пока значение переменной `num` отлично от `-1` (признак того, что символ в тексте не найден). В теле оператора цикла командой `L.append(num)` в список добавляется текущее значение для индекса позиции, на которой символ найден в тексте. После этого командой `num=txt.find(symb, num+1)` вычисляется индекс следующего вхождения символа в текст. В этой команде метод `find()` вызывается со вторым аргументом, равным `num+1`. Если на данный момент в переменную `num` записан индекс вхождения символа в текст,

то дальнейший поиск символа будет начинаться со следующей позиции. Поскольку третий аргумент методу `find()` не передан, то поиск будет выполняться до конца текстовой строки. Если символ найден не будет, то переменная `num` получит значение `-1` и оператор цикла завершит выполнение.

Если при выполнении команды `num=txt.find(symb)` перед оператором цикла окажется, что символа в тексте нет, то, как отмечалось выше, переменная `num` получит значение `-1` и оператор цикла выполняться не будет — точнее, при первой же проверке условия оно окажется ложным и команды в теле оператора цикла выполняться не будут. В результате список `L` останется пустым. Поэтому после оператора цикла в условном операторе проверяется условие `len(L)==0`. Условие истинно, если в списке нет элементов. В таком случае отображается сообщение о том, что в тексте искомого символа нет. Если условие ложно, то отображается сообщение с содержимым списка `L` — а это индексы позиций в тексте, на которых находится искомый символ.

НА ЗАМЕТКУ

Мы говорим о символе, хотя на самом деле это может быть подстрока, которая содержит больше чем одну букву/символ.

Еще одна очень простая программа, представленная в листинге 5.8, иллюстрирует использование метода `replace()` для замены подстрок в тексте.

Листинг 5.8. Замена подстрок в тексте

```
txt="Мы изучаем язык Python"
print(txt)
A=txt.replace(" ","_*_")
print(A)
B=txt.replace(" ","\n")
print(B)
C=txt.replace(" "," не ",1).replace("Python","Java")
print(C)
D=txt.replace("язык ","")
print(D)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 5.8)

```
Мы изучаем язык Python
Мы_*_изучаем_*_язык_*_Python
Мы
изучаем
язык
Python
Мы не изучаем язык Java
Мы изучаем Python
```

В программе команда `txt="Мы изучаем язык Python"` создает базовый текст, на основе которого путем замены подстрок затем создаются новые текстовые строки. Так, командой `A=txt.replace(" ","_*_")` создается новая строка, в которой, по сравнению с исходной, пробелы заменяются на подстроку `"_*_"`.

При выполнении команды `B=txt.replace(" ","\n")` создается строка, в которой пробелы заменяются инструкциями перехода к новой строке.

Команда `C=txt.replace(" "," не ",1).replace("Python", "Java")` выполняется так. Сначала на основе текста, записанного в переменную `txt`, создается новый текст. Он получается заменой пробела на подстроку `" не "`. Причем, поскольку методу `replace()` передан третий аргумент `1`, такая замена выполняется только один раз, то есть заменяется только первый пробел в тексте. Из того текста, который получился в результате, снова вызывается метод `replace()` с аргументами `"Python"` и `"Java"` (слово `"Python"` заменяется на `"Java"`). Результат записывается в переменную `C`.

Наконец, при выполнении команды `D=txt.replace("язык ","")` создается новый текст, который получается из исходного заменой подстроки `"язык "` на пустой текст `""` — то есть фактически речь идет об удалении подстроки `"язык "`.

Кроме описанных выше операций имеется несколько методов для формирования текста на основе отдельных блоков и для разбивки текста на блоки. Вот методы, которые могут быть полезны в этом случае.

- Метод `join()` позволяет создать строку на основе итерируемой последовательности (например, списка). Текст получается объединением элементов последовательности, переданной аргументом методу. В качестве разделителя между элементами выступает текст, из которого вызывается метод.
- Метод `partition()` позволяет разбить исходный текст (из которого вызывается метод) на три части. В качестве аргумента методу передается текст, который служит разделителем при разбивке исходного текста на блоки. Точкой разбивки служит то место в тексте, где разделитель встречается первый раз. Результатом метод возвращает кортеж, который состоит из трех текстовых строк: текст до разделителя, разделитель, текст после разделителя. Если разделителя в тексте нет, то первым элементом кортежа является исходный текст, а также две пустые текстовые строки. Метод `rpartition()` аналогичен методу `partition()`, но только разбивка выполняется в месте, где разделитель последний раз встречается в тексте. Если разделитель не встречается, то в качестве результата возвращается кортеж, состоящий из двух пустых текстовых строк и исходного текста.
- Метод `split()` в качестве результата возвращает список слов в тексте, из которого вызывается метод. По умолчанию словами считаются блоки текста, разделенные пробелами. В качестве аргумента методу можно передать подстроку, которая будет интерпретироваться как разделитель между словами. Также можно указать максимальное количество разбивок в тексте (при этом максимальное количество слов на единицу больше). Метод `rsplit()` аналогичен методу `split()`, но только разбивка на слова выполняется справа налево.
- Метод `splitlines()` возвращает список подстрок, на которые разбивается исходный текст. Разделителем при разбивке текста является инструкция перехода к новой строке (фактически метод разбивает исходный текст на блоки, которые при отображении текста в области вывода отображаются в разных строках).

В листинге 5.9 представлена программа, в которой используются методы `join()`, `split()` и `splitlines()`.

Листинг 5.9. Формирование и разбивка теста

```
A=["Alpha","Bravo","Charlie"]
print("Список:", A)
```

```
B=", ".join(A)
print("Текст:", B)
C=B.split(", ")
print("Снова список:", C)
txt="""Прошли года
И в свете лет
Есть мудрость
А вот счастья нет"""
print(txt)
D=txt.splitlines()
print(D)
```

Ниже показано, как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 5.9)

```
Список: ['Alpha', 'Bravo', 'Charlie']
Текст: Alpha, Bravo, Charlie
Снова список: ['Alpha', 'Bravo', 'Charlie']
Прошли года
И в свете лет
Есть мудрость
А вот счастья нет
['Прошли года', 'И в свете лет', 'Есть мудрость', 'А вот счастья нет']
```

В этой простой программе на основе списка `A` из трех текстовых элементов `"Alpha"`, `"Bravo"` и `"Charlie"` создается текст. Мы используем команду `B=", ".join(A)`. Текст `", "`, из которого вызывается метод `join()`, служит разделителем при формировании текстового значения. Обратная операция (разбивка текста на блоки) выполняется командой `C=B.split(", ")`. Здесь текст `", "`, который служит разделителем, передается аргументом методу `split()`. Наконец, команда `D=txt.splitlines()` использована для разбивки текстового значения из переменной `txt` на блоки, которые соответствуют строкам текста (имеются в виду строки, отображаемые в области вывода).

Есть несколько методов, позволяющих создавать текстовые строки, которые определенным образом выравниваются внутри выделенного для их отображения поля. Это методы `center()`, `ljust()` и `rjust()`. Метод `center()` результатом возвращает текстовую строку, выравненную по центру поля, выделенного для отображения текста. Ширина поля (в символах) указывается аргументом метода. Текст, который выравнивается в поле, совпадает с текстом, из которого вызывается метод. Вторым аргументом можно указать символ, которым будут заполняться «лишние» позиции в поле, выделенном для отображения текста (по умолчанию это пробел). Методы `ljust()` и `rjust()` аналогичны методу `center()`, но методом `ljust()` выравнивание выполняется по левому краю, а методом `rjust()` текст выравнивается по правому краю.

Примеры использования этих методов представлены в программе в листинге 5.10.

Листинг 5.10. Выравнивание текста

```
txt="PYTHON"
num=20
A=txt.ljust(num,"_")
B=txt.center(num)
C=txt.rjust(num,"*")
print("|", A,"|")
print("|", B,"|")
print("|", C,"|")
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 5.10)

```
| PYTHON_____________ |
|        PYTHON       |
| **************PYTHON |
```

Программа исключительно простая, и хочется верить, что комментировать ее нет необходимости.

Примеры работы с текстом

> Ой, что было, Шеф, что было! Это было так интересно.
>
> *Из м/ф «Приключения капитана Врунгеля»*

Далее мы рассмотрим некоторые программы, в которых выполняются операции с текстовыми значениями. Начнем с программы, в которой на основе исходного текстового значения формируется новая текстовая строка. По сравнению с исходным текстом в созданной строке выполнена попарная замена символов: первый символ в тексте меняется местами со вторым символом, третий символ меняется местами с четвертым, и так далее. Рассмотрим код в листинге 5.11.

Листинг 5.11. Попарный обмен символами

```
# Исходный текст:
txt=input("Введите текст: ")
# Новый текст:
new_txt=""
# Индекс символа:
num=0
# Перебор символов:
while num<len(txt)-1:
    # Добавление символов в текст:
    new_txt+=txt[num+1]+txt[num]
    # Новое значение индекса:
    num+=2
# Если в исходном тексте остался символ:
if num<len(txt):
    new_txt+=txt[num]
# Отображение результата:
print("Результат:", new_txt)
```

Ниже показано, как может выглядеть результат выполнения программы (жирным шрифтом выделен текст, который вводится пользователем).

Результат выполнения программы (из листинга 5.11)

```
Введите текст: абракадабра
Результат: бааракадрба
```

Сначала командой `txt=input("Введите текст: ")` считывается текст, который вводит пользователь. Начальное значение переменной `new_txt` — пустая текстовая строка `""`. Также мы используем переменную `num` с начальным нулевым значением для перебора символов в тексте.

Символы перебираются с помощью оператора цикла `while`. В нем проверяется условие `num<len(txt)-1`. Следует учесть, что мы за один цикл обращаемся к двум соседним символам (с индексами `num` и `num+1`), и оба они не должны выходить за пределы текстового значения. В теле оператора цикла командой `new_txt+=txt[num+1]+txt[num]` к текущему текстовому значению переменной `new_txt` дописываются символы из исходного текста с индексами `num+1` и `num` (то есть здесь мы меняем порядок следования символов по сравнению с исходным текстом). После этого командой `num+=2` значение переменной `num` увеличивается на `2` (поскольку символы обрабатываются парами).

В принципе, после завершения оператора цикла новое текстовое значение вполне может оказаться сформированным. Но если в исходном тексте было нечетное количество символов, то последний символ в операторе цикла не обрабатывается. Если так, то мы его просто дописываем в конец сформированной строки. Операция выполняется с использованием условного оператора, в котором проверяется условие `num<len(txt)`. Истинность условия означает, что на момент его проверки переменная `num` содержит значение последнего символа в тексте, и этот символ в операторе цикла не обрабатывался. Поэтому командой `new_txt+=txt[num]` он дописывается в текстовую строку.

НА ЗАМЕТКУ

Если в исходном тексте четное количество символов, то после завершения оператора цикла значение переменной `num` будет на единицу больше индекса последнего символа в тексте. Если в исходном тексте нечетное количество символов, то после завершения оператора цикла значение переменной `num` будет равно индексу последнего символа в тексте. Этим обстоятельством мы и воспользовались.

Для проверки результата используем команду `print("Результат:", new_txt)`.

В следующей программе введенный пользователем текст будет шифроваться. Для шифрования мы используем алгоритм, в соответствии с которым каждая буква в тексте заменяется на следующую (а последняя буква алфавита заменяется на первую). Для простоты будем полагать, что исходный текст может содержать только кириллические буквы.

НА ЗАМЕТКУ

Речь о так называемом шифре Цезаря. При шифровании текста каждая буква заменяется другой, смещенной на несколько позиций вправо или влево. Это один из самых простых способов шифрования текста.

Интересующая нас программа представлена в листинге 5.12.

Листинг 5.12. Шифрование текста

```
# Исходный текст:
txt=input("Ваш текст: ")
# Переменные:
new_txt=""
m=ord("а")
n=ord("я")
M=ord("А")
N=ord("Я")
# Создание шифра:
for s in txt:
    k=ord(s)
    if (k>=m and k<n) or (k>=M and k<N):
        s=chr(k+1)
    elif k==n:
        s=chr(m)
    elif k==N:
        s=chr(M)
    new_txt+=s
# Проверка результата:
print("Шифр:", new_txt)
```

Как может выглядеть результат выполнения программы, показано ниже (жирным шрифтом выделен текст, введенный пользователем).

Результат выполнения программы (из листинга 5.12)

```
Ваш текст: Я сегодня — на работе!
Шифр: А тждпеоа — об сбвпуж!
```

Введенный пользователем текст записывается в переменную `txt`. Для удобства в отдельные переменные мы записываем коды первых и последних маленьких и больших букв из алфавита. Шифрованный текст записывается в переменную `new_txt`. В процессе создания шрифта переменная `s` пробегает значения символов из введенного пользователем текста. Если прочитанный символ — буква (код `k` символа попадает с соответствующий диапазон значений), то командой `s=chr(k+1)` символ заменяется на следующий. Отдельно рассматриваются случаи, когда прочитанный символ — последняя буква в алфавите.

В листинге 5.13 представлена программа, в которой для введенного пользователем текста определяется средняя длина слова (в символах).

Листинг 5.13. Средняя длина слова

```
# Исходный текст:
txt=input("Введите текст: ")
# Все буквы маленькие:
txt=txt.lower()
# Результат преобразования:
print(txt)
# Разбивка текста на подстроки:
L=txt.split(" ")
# Результат разбивки:
print(L)
# Переменные:
s=0
n=0
# Перебор элементов списка:
for k in range(len(L)):
    # Отбрасываем небуквенные символы:
    w=L[k].strip(".,:;-!?")
```

```
    # Если текст непустой:
    if len(w)!=0:
       # Отображение слова и его длины:
       print(w.ljust(12),"-", len(w))
       # Сумма букв в словах:
       s+=len(w)
       # Количество слов:
       n+=1
# Среднее значение:
s/=n
# Результат вычислений:
print("Средняя длина:", s)
```

Результат выполнения программы может быть таким (жирным шрифтом выделен текст, введенный пользователем).

Результат выполнения программы (из листинга 5.13)

```
Введите текст: Один, два — дальше три? Да.
один, два — дальше три? да.
['один,', 'два', '-', 'дальше', 'три?', 'да.']
один         — 4
два          — 3
дальше       — 6
три          — 3
да           — 2
Средняя длина: 3.6
```

Введенный пользователем текст записывается в переменную `txt`. Исключительно из эстетических соображений все буквы в тексте мы переводим в нижний регистр (делаем их маленькими), для чего используем команду `txt=txt.lower()`. Командой `L=txt.split(" ")` текст разбивается на блоки. Разделителем служит пробел. В принципе, такие блоки можно интерпретировать как «слова». Но если во введенном тексте были знаки препинания, то они останутся в «словах». Мы выполняем минимальную очистку «слов» от лишних символов, для чего запускается оператор цикла

`for` и переменная `k` перебирает значения индексов элементов списка `L`. Командой `w=L[k].strip(".,:;-!?")` в подстроке `L[k]`, играющей роль «слова», отбрасываются небуквенные символы (которые содержатся в текстовой строке, переданной аргументом методу `strip()`), а результат записывается в переменную `w`. При этом может оказаться, что переменная `w` будет содержать пустой текст. Мы пустой текст как «слово» не рассматриваем. Поэтому с помощью условного оператора отслеживаем данную ситуацию. Если текст в переменной `w` содержит символы (условие `len(w)!=0` истинно), командой `print(w.ljust(12),"-", len(w))` отображается «слово» и его длина, командой `s+=len(w)` количество символов в слове прибавляется к текущему значению переменной `s` (общее количество букв во всех словах), а значение переменной `n` (общее количество слов) с помощью команды `n+=1` увеличивается на единицу. После завершения оператора цикла общее количество букв в словах делится на общее количество слов (команда `s/=n`). Результат отображается командой `print("Средняя длина:", s)`.

Резюме

> Прекратите панику, Фукс. И слушайте меня внимательно.
>
> *Из м/ф «Приключения капитана Врунгеля»*

- Текстовые литералы заключаются в одинарные или двойные кавычки. Литерал в тройных двойных или одинарных кавычках отображается в области вывода так, как он введен в редакторе кода, с учетом всех отступов и переносов.
- Текстовый литерал в окне редактора кода можно разбивать на несколько строк. В таком случае в конце строки последним символом указывается обратная косая черта `\`, которая служит индикатором переноса литерала. При отображении такого литерала в области вывода в соответствующем месте разрыв не выполняется.
- Литерал может содержать специальные управляющие инструкции, которые начинаются с обратной косой черты, как, например, инструкция перехода к новой строке `\n` и инструкция выполнения табуляции `\t`. Такие инструкции интерпретируются как один символ. Инструкции `\\`, `\"` и `\'` используются для вставки в текстовый литерал соответственно обратной косой черты, двойных кавычек и одинарных кавычек.

- При создании литералов можно использовать префиксы. Префикс `r` или `R` позволяет создать литерал, в котором управляющие инструкции интерпретируются как последовательности обычных символов. Префикс `f` или `F` позволяет создавать литералы, содержащие поля замен. Поле замены определяет значение, которое вставляется в текстовый литерал, а также способ его отображения этого значения. Также для создания форматированных текстовых значений можно использовать метод `format()`.
- Текстовые значения можно складывать с помощью оператора `+`. В этом случае выполняется конкатенация (объединение) текстовых значений (создается новое текстовое значение). При умножении текста на число (или числа на текст) создается новый текст, который получается повторением исходного (умножаемого) текста, а количество повторений определяется значением числового множителя.
- Как и в случае со списками, текстовые значения можно индексировать, а также выполнять срезы. При этом следует учесть, что изменить значение того или иного символа в тексте невозможно — в таких случаях на основе исходного текста создается новое текстовое значение.
- Для работы с текстовыми значениями предусмотрены специальные методы, которые позволяют выполнять самые разнообразные операции, включая, кроме прочего, поиск символов в тексте, замену символов и подстрок (в режиме создания нового текстового значения), выравнивание текста, изменение регистра символов и так далее.

Задания для самостоятельной работы

> Будьте внимательны. Помните о строжайшей секретности этой операции.
>
> *Из м/ф «Приключения капитана Врунгеля»*

1. Напишите программу, в которой с использованием текстовых литералов (например, используя текстовый литерал, заключенный в тройные кавычки) символом «звездочка» `*` в области вывода отображается фраза `"Hello Python"`.

2. Напишите программу, в которой с использованием операторов цикла и форматированных литералов, символом «звездочка» `*` в области вывода отображается буква `"A"`.

3. Напишите программу, в которой пользователю предлагается ввести текст, а затем в этом тексте, без применения специальных методов (а именно, не используя метод `swapcase()`), все большие буквы меняются на маленькие, а маленькие — на большие.

4. Напишите программу, в которой на основе введенного пользователем текста создается новая текстовая строка. В этой строке по сравнению с исходной символы меняются местами «через один»: первый символ меняется местами с третьим, четвертый символ меняется местами с шестым, седьмой меняется местами с девятым и так далее.

5. Напишите программу, в которой пользователь вводит два текстовых значения, и на их основе создается новый текст. В этот новый текст поочередно включаются буквы из текстов, введенных пользователем. Когда один из текстов заканчивается, в качестве символов из этого текста используется «звездочка» `*`.

6. Напишите программу для расшифровки текста, зашифрованного в соответствии с таким алгоритмом: каждая буква заменяется на следующую, а последняя буква в алфавите заменяется на первую.

7. Напишите программу для шифрования и дешифрования текста. Текст шифруется так: каждая буква заменяется на ту, что размещена от нее на две позиции влево. Вторая буква в алфавите заменяется на последнюю. Первая буква в алфавите заменяется на предпоследнюю.

8. Напишите программу, в которой шифруется и дешифруется введенный пользователем текст. При шифровании каждая буква заменяется на следующую (а последняя — на первую), но только эта операция отдельно выполняется для гласных букв и для согласных. Для этого нужно сформировать список гласных и согласных букв и шифрование и дешифрование выполнять на основе этих списков.

9. Напишите программу, в которой на основе текста, введенного пользователем, создается новый текст, в котором по сравнению с исходным удалено самое длинное и самое короткое слово. Если таких слов несколько, то удаляется первое из них. Под словами подразумевать блоки текста, разделенные пробелами.

10. Напишите программу, в которой на основе текста, введенного пользователем, создается новый текст. По сравнению с исходным в нем слова расположены в обратном порядке. Под словами подразумевать блоки текста, разделенные пробелами.

Глава 6
ФУНКЦИИ

Для чего живет человек на земле? Скажите.

Из к/ф «Формула любви»

В самом начале книги мы очень кратко познакомились с тем, как создаются пользовательские функции. Кроме этого, мы постоянно использовали встроенные библиотечные функции для решения различных задач. Теперь пришло время побольше узнать о функциях.

В этой главе рассматривается ряд тем, имеющих непосредственное отношение к созданию и использованию функций.

Объявление и вызов функции

Какая разница? Объявление перепишем! А что у вас с рукой?

Из к/ф «Бриллиантовая рука»

Напомним, что *функцией* называется блок программного кода, у которого есть имя и который можно выполнить, указав в правильном контексте имя функции (этот процесс называется вызовом функции). Прежде чем функцию вызывать, ее, естественно, следует описать — то есть создать тот блок кода, который будет выполняться при вызове функции. Мы сталкивались с объявлением функции, но, думается, имеет смысл напомнить, как это делается.

В самом простом случае при объявлении функции указывается ключевое слово `def`, после которого следует имя функции, в круглых скобках перечисляются аргументы функции, затем ставится двоеточие и собственно блок, который содержит описание функции. То есть конструкция такая:

```
def имя(аргументы):
    # Описание функции
```

Если у функции нет аргументов, то после имени функции указываются пустые круглые скобки.

Функция может возвращать значение, а может и не возвращать. В случае если функция возвращает значение, мы используем инструкцию `return`, после которой указываем возвращаемое значение.

НА ЗАМЕТКУ

Выполнение инструкции `return` в теле функции приводит к тому, что выполнение кода функции завершается. Если при этом после инструкции `return` указано некоторое значение, то оно становится результатом функции.

Также стоит отметить, что результат функции не обязательно возвращается `return`-инструкцией. Например, есть так называемые функции-генераторы, в теле которых для формирования результата используется инструкция `yield`. Функции-генераторы будут рассмотрены ниже.

Если функция возвращает значение, то инструкцию вызова функции можно отождествлять с этим значением. Если функция не возвращает значения, то при вызове функции просто выполняется последовательность команд.

НА ЗАМЕТКУ

Стоит обратить внимание вот на какое обстоятельство. Если функция возвращает результат, то тип результата определяется в процессе выполнения функции. Другими словами, в принципе, в разных обстоятельствах (например, если функции передаются разные аргументы) функция может возвращать не просто разные значения, а значения разных типов. Такая особенность функций Python имеет последствия, и ее следует принимать в расчет при работе с функциями.

Сам факт объявления функции не означает, что код функции будет выполняться. Для выполнения кода функции, ее, как отмечалось ранее, следует вызвать: указать название функции и после ее имени в круглых скобках перечислить аргументы, которые ей передаются. В процессе выполнения соответствующей команды выполняются команды из тела

функции, причем для аргументов подставляются те значения, которые фактически передаются функции при вызове.

> **НА ЗАМЕТКУ**
>
> Интерпретатор анализирует программный код построчно, сверху вниз. Поэтому прежде, чем использовать функцию в программе (вызывать), ее сначала следует описать. То есть правило такое: сначала функцию описываем, а затем вызываем.

Далее мы рассмотрим небольшой пример, в котором используются функции. Программа представлена в листинге 6.1.

Листинг 6.1. Создание и использование функций

```
# Функция без аргументов не возвращает результат:
def next_day():
    txt=input("Какой сегодня день недели? ")
    txt=txt.lower().strip()
    if txt=="понедельник":
       new_txt="вторник"
    elif txt=="вторник":
       new_txt="среда"
    elif txt=="среда":
       new_txt="четверг"
    elif txt=="четверг":
       new_txt="пятница"
    elif txt=="пятница":
       new_txt="суббота"
    elif txt=="суббота":
       new_txt="воскресенье"
    elif txt=="воскресенье":
       new_txt="понедельник"
    else:
       print("Нет такого дня недели!")
       return
```

```
    print(f"Завтра — {new_txt}")
# Функция без аргумента возвращает результат:
def get_name():
    name=input("Добрый день! Как Вас зовут? ")
    if name.strip(" .,:;!?_")=="":
       name="Мистер Икс"
    return name
# Функция без аргументов не возвращает результат:
def hello():
    name=get_name()
    print(f"Приятно познакомиться, {name}!")
    next_day()
# Вызов функции:
hello()
```

Программа простая, но мы ее все же прокомментируем. В программе описывается несколько функций. Функция `next_day()` описана без аргументов и предполагается, что она не возвращает результат. Командой `txt=input("Какой сегодня день недели? ")` в теле функции отображается сообщение с просьбой указать день недели. Значение, введенное пользователем, записывается в переменную `txt`. Мы «уточняем» это значение с помощью команды `txt=txt.lower().strip()`, которой введенный пользователем текст переводится в нижний регистр (все буквы маленькие), а затем в начале и конце этого текста отбрасываются все пробелы. Это дает пользователю небольшую свободу в способе ввода названия дня недели. Далее в игру вступает блок из вложенных условных операторов, с помощью которых перебираются возможные названия дней недели и для каждого из вариантов определяется следующий день недели. Название следующего дня недели записывается в переменную `new_txt`.

За случай, когда пользователь ввел некорректное название для дня недели, «отвечает» `else`-блок. В нем командой `print("Нет такого дня недели!")` отображается сообщение и неприятной ситуации, а командой `return` завершается выполнение функции. Поэтому если окажется, что «сработает» `else`-блок, то до выполнения команды `print(f"Завтра — {new_txt}")` дело не дойдет. Эта команда выполняется только в случае, если пользователь ввел приемлемое название для дня недели.

Функция `get_name()` не имеет аргументов, но при этом возвращает результат. Предполагается, что это имя, которое введет пользователь. В функции предусмотрена минимальная защита от некорректных названий: в условном операторе проверяется условие `name.strip(".,:;!?_")==""`. Оно истинно, если после отбрасывания пробелов и знаков препинания в начале и конце введенного пользователем текста получается пустая текстовая строка. В таком случае командой `name= "Мистер Икс"` переменной `name` присваивается новое значение. Именно значение переменной `name` возвращается в качестве результата функции (команда `return name`).

У функции `hello()` нет аргументов, и она не возвращает результат. Особенность данной функции в том, что в ее теле вызываются другие определенные ранее функции. В частности, командой `name=get_name()` в переменную name записывается введенное пользователем имя. Оно используется в команде `print(f"Приятно познакомиться, {name}!")`. Для определения дня недели вызывается функция `next_day()`.

Все, что было описано выше, касается описания функций. А вызывается в программе всего одна функция `hello()`. Именно в результате ее вызова вызываются другие функции.

НА ЗАМЕТКУ

Переменные, которые получают свои значения в теле функции, называются локальными. Они доступны только в теле функции. Поэтому если в двух разных функциях используются переменные с одинаковыми названиями, то на самом деле это разные переменные. Каждая из них доступна только в своей функции. Это же правило относится и к аргументам функций.

Помимо локальных, в функциях могут использоваться и глобальные переменные. Локальные и глобальные переменные обсуждаются далее в этой главе.

Результат выполнения программы может быть таким, как показано ниже (здесь и далее жирным шрифтом выделены значения, которые вводит пользователь).

Результат выполнения программы (из листинга 6.1)

```
Добрый день! Как Вас зовут? Кот Матроскин
Приятно познакомиться, Кот Матроскин!
```

```
Какой сегодня день недели? Воскресенье
Завтра — понедельник
```

Или таким.

Результат выполнения программы (из листинга 6.1)

```
Добрый день! Как Вас зовут? _,! …
Приятно познакомиться, Мистер Икс!
Какой сегодня день недели? Вторница
Нет такого дня недели!
```

Аргументами и результатом функций могут быть и более сложные конструкции, чем числа или текст. Небольшой пример, который служит иллюстрацией к этому утверждению, представлен в листинге 6.2.

Листинг 6.2. Функции для работы с данными разных типов

```
# Импорт функций для генерирования случайных чисел:
from random import *
# Функция для отображения содержимого списков,
# множеств, текста и словарей:
def show(L, symb):
   for s in L:
      print(symb, s, sep="", end="")
   print(symb)
# Исходные данные:
A=[1,2,3,4,5]         # Список
B={'A','B','C','D'}   # Множество
C="Python"            # Текст
D={"A":1,"B":2,"C":3} # Словарь
# Вызов функции:
show(A,"|")
show(B,"/")
show(C,"*")
```

```
show(D,"#")
# Функция для создания списка чисел:
def get_nums(n, state):
    if type(n)!=int:
       return []
    if state:
        L=list(2*(k+1) for k in range(n))
    else:
        L=list(2*k+1 for k in range(n))
    return L
# Вызов функции:
print(get_nums(10, True))
print(get_nums(8, False))
print(get_nums(12.5, True))
# Функция для создания множества случайных букв:
def get_symbs(n):
    if n>10 or n<1:
       num=10
    else:
       num=n
    S=set()
    Nmin=ord("A")
    Nmax=ord("Z")
    while len(S)<num:
       S.add(chr(randint(Nmin, Nmax)))
    return S
# Инициализация генератора случайных чисел:
seed(2019)
# Вызов функции:
print(get_symbs(7))
print(get_symbs(-5))
print(get_symbs(15))
```

Результат выполнения программы (с учетом того, что используется генерирование случайных чисел/символов) может быть таким, как показано ниже.

Результат выполнения программы (из листинга 6.2)

```
|1—2|3—4|5|
/D/A/C/B/
*P*y*t*h*o*n*
#A#B#C#
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
[1, 3, 5, 7, 9, 11, 13, 15]
[]
{'U', 'Z', 'P', 'H', 'K', 'E', 'F'}
{'X', 'T', 'O', 'U', 'Y', 'C', 'Z', 'J', 'H', 'N'}
{'B', 'L', 'U', 'I', 'S', 'V', 'W', 'P', 'K', 'G'}
```

В программе создается несколько простых функций. Функция `show()` предназначена для отображения содержимого списков, множеств, текста и других объектов, относящихся к множественным типам данных. У функции два аргумента. Мы предполагаем, что в качестве первого аргумента функции передается список, множество или другой объект, содержимое которого можно «перебирать». Второй аргумент должен быть текстовым — его мы используем в качестве разделителя при отображении содержимого первого аргумента. Работа функции проверяется на различных данных (среди которых список, множество, текст и словарь).

НА ЗАМЕТКУ

При вызове функции `print()` аргумент `end=""` означает, что в конце вместо перехода к новой строке «печатается» пустой текст. Аргумент `sep=""` определяет в качестве разделителя между отображаемыми значениями (первый и второй аргументы функции `print()`) пустую текстовую строку.

Также стоит заметить, что при отображении содержимого словаря отображаются только ключи элементов, но не их значения. Это связано с тем, что перебор содержимого словаря по умолчанию подразумевает перебор его ключей. Желающие могут подумать, как следует изменить код функции, чтобы для словарей отображались не ключи элементов, а их значения.

Функция `get_nums()` позволяет создавать списки из четных или нечетных натуральных чисел. У нее два аргумента: мы исходим из того, что первый определяет количество чисел в списке, а второй логический (как мы думаем) аргумент определяет из четных (значение второго аргумента `True`) или нечетных (значение второго аргумента `False`) чисел состоит список.

ПОДРОБНОСТИ

В теле функции в условном операторе проверяется тип первого аргумента (условие `type(n)!=int` истинно, если значение, на которое ссылается аргумент `n`, не относится к целочисленному типу `int`). Если это не целое число, то командой `return []` в качестве результата возвращается пустой список. В таком случае до выполнения прочих команд в теле функции дело не доходит. Но если первый аргумент целочисленный, то в зависимости от истинности или ложности второго аргумента формируется список `L` из четных/нечетных натуральных чисел и ссылка на этот список возвращается в качестве результата функции (команда `return L`).

Также мы создаем функцию для генерирования множества случайных символов (поскольку в основе генерирования случайных символов — генерирование случайных чисел, то мы в начале программы использовали соответствующую `import`-инструкцию). У функции `get_symbs()` всего один аргумент, определяющий количество элементов в множестве. Причем в самом начале выполнения функции мы «уточняем» размер множества: если переданный функции аргумент имеет значение больше `10` или меньше `1`, то будет генерироваться множество из `10` элементов.

Сначала создается пустое множество `S` (команда `S=set()`), а также определяются границы для кодов генерируемых символов (команды `Nmin=ord("A")` и `Nmax=ord("Z")`). В теле оператора цикла `while` генерируются целые числа, на основе которых создаются символы, и эти символы добавляются в множество (команда `S.add(chr(randint(Nmin, Nmax)))`). Процесс продолжается до тех пор, пока размер множества `S` меньше значения переменной `num` (условие `len(S)<num` в операторе цикла). После того как множество сформировано, оно возвращается как результат функции (команда `return S`).

Но даже эти простые примеры в полной мере не раскрывают красоту и гибкость таких объектов (именно объектов!), как функции. В качестве иллюстрации рассмотрим еще одну программу. Она представлена в листинге 6.3.

Листинг 6.3. Имя функции как переменная

```
# Функции:
def alpha():
    print("Это Alpha!")
def bravo():
    print("Это Bravo!")
def hello():
    print("А это Hello!")
# Переменная с целочисленным значением:
num=123
# Вызов функций и проверка значения переменной:
print("Сначала было так:")
alpha()
bravo()
hello()
print("num =", num)
# Изменение значений:
alpha, bravo=bravo, alpha
num=hello
hello=321
# Вызов "функций" и проверка значения "переменной":
print("А стало так:")
alpha()
bravo()
num()
print("hello =", hello)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 6.3)

```
Сначала было так:
Это Alpha!
Это Bravo!
```

```
А это Hello!
num = 123
А стало так:
Это Bravo!
Это Alpha!
А это Hello!
hello = 321
```

Попробуем разобраться, что же происходит в данном случае. Мы объявляем три однотипные функции `alpha()`, `bravo()` и `hello()`. У функций нет аргументов, они не возвращают результат. При вызове каждой из функций отображается сообщение соответствующего характера — просто чтобы по сообщению можно было идентифицировать функцию, которая была вызвана. Также мы создаем переменную `num` со значением `123`. После создания функций они вызываются, а еще проверяется значение переменной. Результат вполне ожидаемый. После этого выполняются некоторые манипуляции — в основном с названиями функций.

Сразу отметим: чтобы понять смысл происходящего, следует рассматривать названия функций как переменные (что и есть на самом деле). Описание функции означает создание специального объекта, а название функции — это переменная, которая ссылается на данный объект. Значение этой переменной можно изменить. Так происходит при выполнении команды `alpha, bravo=bravo, alpha`. В этом случае «переменные» `alpha` и `bravo` обмениваются «значениями». В итоге «переменная» `alpha` ссылается на объект, на который раньше ссылалась «переменная» `bravo`, а «переменная» `bravo` ссылается на объект, на который раньше ссылалась «переменная» `alpha`. В результате выполнения команды `num=hello` в переменную `num` записывается ссылка на объект, на который ссылается «переменная» `hello`. Как следствие, переменная `num` становится «функцией». А после выполнения команды `hello=321` «переменная» `hello` получает значение `321`, это больше не «функция». Последующая проверка показывает правоту наших рассуждений.

Именованные аргументы функции

Действуйте только по моим инструкциям.

Из м/ф «Приключения капитана Врунгеля»

Теперь мы более подробно обсудим аспекты, связанные с аргументами функции. Хотя на первый взгляд может показаться, что особой интриги здесь нет, тем не менее несколько принципиально важных тем обойти стороной было бы крайне неразумно.

Начнем с того, как аргументы передаются функции при вызове. И сразу рассмотрим пример, представленный в листинге 6.4.

Листинг 6.4. Передача аргументов по позиции и по имени

```
# Функция с тремя аргументами:
def show(first, second, third):
    print(f"[1] Первый аргумент — {first}")
    print(f"[2] Второй аргумент — {second}")
    print(f"[3] Третий аргумент — {third}")
# Вызов функции:
show(1,2,3)
show(second="B", third="C", first="A")
show(1, third=3, second=2)
```

Результат выполнения программы такой, как показано ниже.

Результат выполнения программы (из листинга 6.4)

```
[1] Первый аргумент — 1
[2] Второй аргумент — 2
[3] Третий аргумент — 3
[1] Первый аргумент — A
[2] Второй аргумент — B
[3] Третий аргумент — C
[1] Первый аргумент — 1
```

```
[2] Второй аргумент — 2
[3] Третий аргумент — 3
```

Программа очень простая. В ней описана всего одна функция `show()` с тремя аргументами, которые называются соответственно `first`, `second` и `third`. При вызове функции в области выводов отображаются значения аргументов (с поясняющим текстом). Но нас интересуют команды, которыми вызывается функция. Их три. Первая из них выглядит как `show(1,2,3)`. Это привычная для нас инструкция. В ней вызывается функция `show()`, а аргументы, которые ей передаются, указаны в круглых скобках через запятую. Это так называемая *позиционная* передача аргументов, или передача аргументов *по позиции*: порядок передачи аргументов при вызове функции соответствует порядку, в котором аргументы объявлялись в описании функции. Но есть иной способ передачи аргументов, когда для аргумента явно указывается название и значение. Такой способ называется передачей *по имени*, или передачей *по ключу* (а соответствующий аргумент иногда называют *именованным*). Пример такого способа передачи аргументов дает команда `show(second="B", third="C", first="A")`. Здесь для каждого аргумента указано название (то, которое было использовано в объявлении функции) и, через знак равенства, значение аргумента. При этом порядок аргументов произвольный.

Оба способа передачи аргументов можно совмещать. Другими словами, часть аргументов может передаваться по позиции, а часть — по имени. Однако здесь существует важное ограничение: сначала указываются аргументы, которые передаются по позиции, и уже затем можно указывать аргументы, которые передаются по имени. Как пример можно привести команду `show(1, third=3, second=2)`, в которой первый аргумент передается по позиции, а два других — по имени.

Механизм передачи аргументов

> За это вам наша искренняя сердечная благодарность.
>
> *Из к/ф «Формула любви»*

Еще один важный момент связан с механизмом передачи аргументов в функцию. Как и в предыдущем случае, мы сразу начнем с примера, который представлен в листинге 6.5.

Листинг 6.5. Механизм передачи аргументов

```
# Функции:
def shift(val):
    print("Функция shift()")
    print("Исходное значение:", val)
    val=["A","B","C"]
    print("Конечное значение:", val)
def change(val):
    print("Функция change()")
    print("Исходное значение:", val)
    if type(val)==list:
        for k in range(len(val)):
            val[k]+=1
    else:
        val+=1
    print("Конечное значение:", val)
# Переменная:
num=100
# Список:
L=[10,20,30]
# Вызов функций:
print(f"Переменная num={num}")
change(num)
print(f"Переменная num={num}")
print(f"Список L={L}")
shift(L)
print(f"Список L={L}")
change(L)
print(f"Список L={L}")
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 6.5)

```
Переменная num=100
Функция change()
Исходное значение: 100
Конечное значение: 101
Переменная num=100
Список L=[10, 20, 30]
Функция shift()
Исходное значение: [10, 20, 30]
Конечное значение: ['A', 'B', 'C']
Список L=[10, 20, 30]
Функция change()
Исходное значение: [10, 20, 30]
Конечное значение: [11, 21, 31]
Список L=[11, 21, 31]
```

Мы описываем в программе две функции `shift()` и `change()`. Объединяет их то, что при вызове этих функций выполняется попытка изменить значение аргумента, переданного функции. Но делается это по-разному. В функции `shift()` отображается значение аргумента `val`, затем выполняется команда `val=["A","B","C"]`, которой аргументу присваивается новое значение, после чего снова отображается значение аргумента `val`.

В функции `change()` все происходит несколько сложнее. В ней также отображается значение переданного функции аргумента `val`. Но дальше все зависит от типа аргумента. Если это список (истинно условие `type(val)==list` в условном операторе), то значение каждого элемента массива увеличивается на единицу. Если аргумент функции — не список, то значение этого элемента увеличивается на единицу. Перед завершением выполнения функции значение аргумента снова отображается в области вывода.

Кроме описания этих двух функций в программе создается переменная `num` со значением `100` и список `L=[10,20,30]`. Далее мы проверяем

значение переменной, передаем значение этой переменной функции `change()` и снова проверяем значение переменной `num`. Что мы могли бы ожидать? Мы могли бы ожидать, что значение переменной `num` после передачи аргументом в функцию `change()` увеличится на единицу. Более того, при повторной проверке значения аргумента в процессе выполнения функции `change()` мы действительно получаем значение, на единицу больше исходного. Но проверка значения функции `num` после вызова функции `change()` свидетельствует о том, что значение переменной не изменилось. Так в чем же дело?

Чтобы понять причину происходящего, следует учесть, что при передаче аргумента функции на самом деле в функцию передается не «оригинал», а техническая копия аргумента. Поэтому все операции в теле функции выполняются не с той переменной, которая указана аргументом при вызове функции, а с ее копией. Следовательно, когда в теле функции проверяется значение аргумента (после попытки увеличения значения аргумента), то оно оказывается новым. Но это новое значение копии. Значение исходной переменной не изменяется.

НА ЗАМЕТКУ

Механизм передачи аргументов, когда для аргумента создается копия, называется передачей аргумента по значению. Есть языки программирования, в которых аргументы можно передавать по ссылке, когда в функцию передается фактически указанная переменная. В языке Python аргументы передаются только по значению.

Нечто похожее происходит, когда мы вызываем функцию `shift()` и передаем ей аргументом список `L`. Для переменной `L` создается копия, и эта копия передается в функцию. Копия содержит такое же значение, что и переменная `L`. А переменная `L` в качестве значения содержит ссылку на список — фактически значением переменной `L` является адрес списка в памяти. Поэтому копия содержит тот же адрес и, значит, ссылается на точно тот же список. При первой проверке в теле функции мы получаем исходный список. После присваивания аргументу в качестве значения списка `["A","B","C"]` в аргумент записывается адрес этого нового списка, но исходная переменная `L` продолжает ссылаться на список `[1,2,3]`. Поэтому после завершения выполнения функции `shift()` значение переменной `L` не изменилось.

А вот если мы передаем переменную `L` в функцию `change()`, то после вызова функции оказывается, что элементы в списке, на который

ссылается переменная L, изменили значения. Почему? Ответ базируется все на том же механизме передачи аргумента в функцию. Для переменной L создается копия, и эта копия ссылается на тот же список, на который ссылается переменная L. Поэтому когда в теле функции мы пытаемся изменить значения элементов списка, на которые ссылается аргумент, мы на самом деле вносим изменения в тот же список, на который ссылается переменная L.

> **НА ЗАМЕТКУ**
>
> Из того, что аргументы передаются по значению, следует, что мы не можем в теле функции изменить значение переменной, переданной функции в качестве аргумента. В случае передачи функции `change()` списка `L` противоречия с приведенным утверждением нет, поскольку в этом случае мы не пытаемся изменить аргумент: мы изменяем список, на который аргумент ссылается.

Значения аргументов по умолчанию

> Вы получите то, что желали, согласно намеченным контурам.
>
> *Из к/ф «Формула любви»*

У аргументов функции могут быть *значения по умолчанию*. Значения по умолчанию используются в том случае, если при вызове функции аргумент явно не указан. Значение по умолчанию задается через оператор присваивания в описании функции после имени аргумента (то есть в формате `аргумент=значение`).

> **НА ЗАМЕТКУ**
>
> Может статься, что все аргументы функции имеют значения по умолчанию. Но можно описывать и функции, в которых значения по умолчанию имеют только некоторые аргументы. Если так, что сначала в описании функции перечисляются аргументы без значений по умолчанию, а затем — аргументы со значениями по умолчанию.

Пример, в котором используются функции со значениями по умолчанию для аргументов, представлен в листинге 6.6.

Листинг 6.6. Значения аргументов по умолчанию

```
# Функция со значениями аргументов по умолчанию:
def show(first, second="Bravo", third="Charlie"):
    print(f"[1] — {first}")
    print(f"[2] — {second}")
    print(f"[3] — {third}")
    print("-"*13)
# Вызов функции:
show("Alpha")
show("A","B","C")
show(10,20)
show(100, third=300)
show(third="третий", first="первый")
```

Ниже показано, каким будет результат выполнения программы.

Результат выполнения программы (из листинга 6.6)

```
[1] — Alpha
[2] — Bravo
[3] — Charlie
-------------
[1] — A
[2] — B
[3] — C
-------------
[1] — 10
[2] — 20
[3] — Charlie
-------------
[1] — 100
[2] — Bravo
[3] — 300
```

```
------------
[1] — первый
[2] — Bravo
[3] — третий
------------
```

Мы описали функцию `show()` с тремя аргументами, причем для второго и третьего аргументов предусмотрены значения по умолчанию. Функция простая: при ее вызове в области вывода отображаются значения аргументов, переданных функции.

После того как функция описана, она вызывается, причем каждый раз аргументы ей передаются по-разному. В команде `show("Alpha")` функция вызывается с одним аргументом (хотя на самом деле их должно быть три). Что это значит? Это значит, что код функции выполняется со значением `"Alpha"` для первого аргумента, а для второго и третьего аргументов используются значения по умолчанию.

В команде `show("A","B","C")` указаны значения для всех трех аргументов, поэтому код функции выполняется с явно заданными значениями аргументов.

Команда `show(10,20)` обрабатывается так: первый аргумент получает значение `10`, второй аргумент получает значение `20`, а для третьего аргумента используется значение по умолчанию.

Инструкция `show(100, third=300)` означает, что функция вызывается со значением `100` для первого аргумента, для третьего аргумента указано значение `300`, а для второго аргумента используется значение по умолчанию.

Наконец, инструкция `show(third="третий", first="первый")` означает, что третий аргумент получает значение `"третий"`, первый аргумент получает значение `"первый"`, а для второго аргумента используется значение по умолчанию.

НА ЗАМЕТКУ

Таким образом, аргументы функции могут иметь значения по умолчанию. При вызове функции аргументы можно передавать по позиции или/и по имени, причем некоторые аргументы могут отсутствовать (поскольку имеют значение по умолчанию). Все это создает

достаточно широкий спектр возможностей и комбинаций в случае вызова функции. В таких случаях важно помнить, что команда вызова функции должна быть такой, что по ней однозначно можно определить, какое значение получает каждый аргумент.

Использование значений по умолчанию для аргументов — эффективный механизм, который очень часто используется на практике. Вместе с тем, здесь тоже имеются свои тонкости. Как иллюстрацию рассмотрим пример в листинге 6.7.

Листинг 6.7. Список как значение по умолчанию

```
# Значение по умолчанию для аргумента — список:
def show(val=[0,1,2]):
    for k in range(len(val)):
       val[k]+=1;
    print(val)
# Вызов функции:
show()
show()
show()
```

В программе описана функция `show()` с одним аргументом (предполагается, что это список). Аргумент имеет значение по умолчанию (это список `[0,1,2]`), поэтому функцию можно вызывать без аргументов. При вызове функции значение каждого элемента списка увеличивается на единицу, и затем содержимое списка отображается в области вывода. Мы после того, как описали функцию, трижды вызываем ее без аргументов. На первый взгляд все просто, но результат выполнения этой программы может для кого-то оказаться немного неожиданным.

Результат выполнения программы (из листинга 6.7)

```
[1, 2, 3]
[2, 3, 4]
[3, 4, 5]
```

Видим, что при каждом следующем вызове элементы отображаемого списка увеличивают свое значение на единицу (хотя команда вызова

функции каждый раз одна и та же). В чем причина? Чтобы дать ответ, следует сначала ответить на вопрос о том, что является значением по умолчанию для аргумента функции. Формально это список, но список, как мы знаем, «запоминается» по адресу. Таким образом, значение по умолчанию для аргумента функции — это на самом деле адрес списка. Когда при первом вызове функции значения элементов списка изменяются, это влияет на содержимое списка, но не изменяет его адреса в памяти. Другими словами, там, где раньше находился список `[0,1,2]`, теперь будет находиться список `[1,2,3]`. Это физически тот же список, только его элементы изменились. Адрес этого списка по-прежнему является значением по умолчанию для аргумента функции `show()`. При следующем вызове функции элементы списка снова изменятся, но адрес списка останется тем же, и так далее.

Функции с произвольным количеством аргументов

Откуда столько набралось?

Из к/ф «Кин-дза-дза»

Если аргументы функции имеют значения по умолчанию, то функцию по факту можно вызывать с разным количеством аргументов. Но даже в таком случае на количество аргументов есть ограничение. Вместе с тем нередко возникает необходимость создать функцию, которой при вызове можно передавать любое количество аргументов. Например, это может быть функция, которая в качестве результата возвращает сумму числовых значений, переданных аргументами функции. Другой пример — функция, которая по текстовым аргументам формирует и возвращает текст (результат конкатенации аргументов). И таких примеров может быть много.

Создавать подобные функции достаточно просто. Главная «идеологическая» проблема здесь в том, что на момент описания функции количество аргументов неизвестно. Выход из ситуации простой — вся последовательность аргументов описывается как один аргумент, но перед этим аргументом ставится символ «звездочка» `*`. В теле функции такой аргумент обрабатывается как кортеж. Но когда функция вызывается, то нет необходимости организовывать аргументы в виде кортежа. Они просто передаются в функцию как обычные аргументы, в круглых скобках

после имени функции, разделенные запятыми. Очень скромный пример, в котором объявляется и используется несколько функций с произвольным количеством аргументов, представлен в листинге 6.8.

Листинг 6.8. Произвольное количество аргументов у функции

```
# Функции с произвольным количеством аргументов:
def sqr_sum(*n):
    s=0
    for a in n:
       s+=a*a
    return s
def get_sum(*n):
    s=0
    for a in n:
        if type(a)==int:
           s+=a
    return s
def get_text(*t):
    return " ".join(t)
# Вызов функций:
print("Сумма квадратов:", sqr_sum(1,3,5))
print("Сумма квадратов:", sqr_sum(2,4,6,8,10))
print("Сумма чисел:", get_sum(2,"A",4,"B",6))
print("Сумма чисел:", get_sum(1,[2,3],4))
print("Сумма чисел:", get_sum())
print("Текст:", get_text("Всем","привет"))
print("Текст:", get_text("A","B","C","D"))
```

В программе описывается несколько функций. Каждая из них может принимать произвольное количество аргументов при вызове. В частности, у нас есть функция `sqr_sum()` для вычисления суммы квадратов чисел (как мы предполагаем), преданных аргументами функции. Функция `get_sum()` вычисляет простую сумму чисел, но при этом в расчет принимаются только целочисленные аргументы функции. Аргументы прочих типов игнорируются. Наконец, функция `get_text()`

результатом возвращает текстовую строку, которая получается объединением (с использованием в качестве разделителя пробела) значений аргументов, переданных функции. Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 6.8)

```
Сумма квадратов: 35
Сумма квадратов: 220
Сумма чисел: 12
Сумма чисел: 5
Сумма чисел: 0
Текст: Всем привет
Текст: A B C D
```

Но еще раз подчеркнем, что самое важное в данном случае то, что функции могут вызываться с разным количеством аргументов (в том числе и без них).

ПОДРОБНОСТИ

Формально, когда мы вызываем функцию, которая может принимать произвольное количество аргументов, мы просто перечисляем аргументы в круглых скобках. Но технически аргументы в функцию передаются в виде кортежа. В этом несложно убедиться: достаточно создать функцию, которая принимает произвольное количество аргументов, и затем в теле функции разместить команду, которой в области вывода отображается значение аргумента функции. А можно просто проверить тип аргумента. Код функции мог бы выглядеть так:

```
def show(*val):
    print("Тип:", type(val))
    print("Аргумент:", val)
```

Желающие могут проверить функциональность такого кода и результат его выполнения.

Кроме аргумента со «звездочкой», у функции могут быть и другие аргументы. Скажем, если мы описываем функцию и у нее один

формальный аргумент со «звездочкой», то это означает, что при вызове функции ей может передаваться произвольное количество аргументов, в том числе функцию можно вызывать без аргументов. А если мы описываем функцию, у которой один аргумент обычный, а один — со «звездочкой», то при вызове функции хотя бы один аргумент ей следует передать. Также следует учесть, что некоторые аргументы могут иметь значения по умолчанию. То есть здесь возможно много интересных комбинаций. Небольшая иллюстрация к такому подходу представлена в листинге 6.9.

Листинг 6.9. Функция с разными аргументами

```
# Функция с разными аргументами:
def my_sum(n,*a, txt="Сумма чисел"):
    s=0
    for m in range(len(a)):
        s+=a[m]**n
    print(txt+":", s)
# Вызов функции:
my_sum(1,100,200,300)
my_sum(2,10,20,30, txt="Сумма квадратов")
```

Вот как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 6.9)

```
Сумма чисел: 600
Сумма квадратов: 1400
```

Мы описали функцию `my_sum()`, у которой первый аргумент — обычный (без «звездочки"), второй аргумент — со «звездочкой», а третий имеет значение по умолчанию. Функция предназначена для вычисления суммы чисел, которые «спрятаны» во втором аргументе (со «звездочкой»). Каждое слагаемое возводится в степень, определяемую первым аргументом функции. Поэтому если значение первого аргумента равно `1`, то вычисляется сумма чисел. Если значение первого аргумента равно `2`, то вычисляется сумма квадратов. Если первый аргумент равен `3` — сумма кубов, и так далее. Последний аргумент со значением по умолчанию определяет текст, который отображается при выводе результата вычислений.

Мы вызываем функцию `my_sum()` дважды. При выполнении команды `my_sum(1,100,200,300)` первый аргумент получает значение `1`, сумма вычисляется для чисел `100`, `200` и `300`, а отображаемый текст определяется значением по умолчанию для последнего аргумента функции. В команде `my_sum(2,10,20,30, txt="Сумма квадратов")` для последнего аргумента явно указано значение, причем этот аргумент передается по имени. Это принципиально, поскольку, укажи мы просто значение последнего аргумента, он бы интерпретировался как значение, включенное в аргумент со «звездочкой».

Локальные и глобальные переменные

Дикари, аж плакать хочется.

Из к/ф «Кин-дза-дза»

Как уже отмечалось ранее, переменные, которые мы создаем внутри функции, называются *локальными* и доступны только в теле функции. В отличие от этих переменных, есть переменные, которые создаются вне тела функции. Такие переменные называются *глобальными*. Важное правило состоит в том, что если в теле функции переменной присваивается значение, то такая переменная интерпретируется как локальная, даже если существует глобальная переменная с таким же именем. Если значение переменной только считывается, то используется глобальная переменная. Если мы хотим использовать в теле функции глобальную переменную (в том числе и присваивать ей значение), ее необходимо объявить с ключевым словом `global`. Программа в листинге 6.10 проясняет ситуацию.

Листинг 6.10. Локальные и глобальные переменные

```
# В функции используются глобальные
# и локальные переменные:
def myfunction():
    # Глобальные переменные:
    global A, B
    # Присваивание значений переменным:
    A="Альфа"
```

```
    B="Браво"
    D="Дельта"
    # Проверка значений:
    print("В функции: A =", A)
    print("В функции: B =", B)
    print("В функции: C =", C)
    print("В функции: D =", D)
# Глобальные переменные:
A="Alpha"
C="Charlie"
D="Delta"
# Проверка значений переменных:
print("До вызова функции: A =", A)
print("До вызова функции: C =", C)
print("До вызова функции: D =", D)
# Вызов функции:
myfunction()
# Проверка значений переменных:
print("После вызова функции: A =", A)
print("После вызова функции: B =", B)
print("После вызова функции: C =", C)
print("После вызова функции: D =", D)
```

Ниже результат выполнения программы.

Результат выполнения программы (из листинга 6.10)

```
о вызова функции: A = Alpha
До вызова функции: C = Charlie
До вызова функции: D = Delta
В функции: A = Альфа
В функции: B = Браво
В функции: C = Charlie
В функции: D = Дельта
```

```
После вызова функции: A = Альфа
После вызова функции: B = Браво
После вызова функции: C = Charlie
После вызова функции: D = Delta
```

Мы используем четыре переменные A, B, C и D и функцию `myfunction()`. Сначала присваиваются значения трем переменным, вызывается функция (в теле которой присваиваются значения трем переменным), и затем проверяются значения всех четырех переменных. У каждой из переменных своя «судьба», и мы должны все это проанализировать. Начнем с переменной A. В самом начале она получает значение `"Alpha"`. При вызове функции переменной присваивается значение `"Альфа"`. Важное обстоятельство связано с тем, что в теле функции переменная объявлена с инструкцией `global`. Поэтому в теле функции значение присваивается именно глобальной переменной. После завершения выполнения функции переменная A остается с тем значением, которое присвоено переменной в теле функции.

Переменная B также задекларирована в функции с ключевым словом `global`, поэтому переменная интерпретируется как глобальная. До вызова функции этой переменной значение не присваивалось. Первое присваивание значения переменной выполняется командой `B="Браво"` в теле функции. Но это глобальная переменная, поэтому после завершения выполнения функции переменная B продолжает свое существование со значением `"Браво"`.

До вызова функции командой `C="Charlie"` фактически создается переменная C. В теле функции как глобальная она не задекларирована. Тем не менее, поскольку переменной в теле функции значение не присваивается, то при проверке значения переменной в теле функции она интерпретируется как глобальная.

Наконец, переменная D. До вызова функции командой `D="Delta"` этой переменной присваивается значение, а затем в теле функции выполняется команда `D="Дельта"`. Но поскольку, в отличие от переменной A, переменная D как глобальная не зарегистрирована, то выполнение команды `D="Дельта"` означает создание локальной переменной D. Эта переменная «перекрывает» одноименную глобальную переменную с таким же именем. После завершения выполнения функции глобальная переменная D остается с исходным значением `"Delta"`.

Вложенные функции

Женщину вынули, автомат засунули.

Из к/ф «Кин-дза-дза»

В Python одну функцию можно описывать в другой функции. Такие функции называют *вложенными* или *внутренними*. Вложенная функция доступна внутри той функции, в которой она описана. Главная ее особенность состоит в том, что она имеет доступ к переменным во «внешней» функции. При этом «внешняя» функция доступа к локальным переменным вложенной функции не имеет.

НА ЗАМЕТКУ

Здесь уместно напомнить, что имя функции является переменной, которая ссылается на объект функции. В этом смысле объявление вложенной функции сродни созданию локальной переменной, только значением этой «переменной» является не число и не текст, а функция.

Мы рассмотрим небольшой пример, в котором используется вложенная функция. В частности, мы опишем функцию для вычисления суммы чисел, суммы квадратов этих чисел и суммы кубов этих же чисел. Причем сами вычисления будут выполняться с помощью вложенной функции. Программа представлена в листинге 6.11.

Листинг 6.11. Вложенная функция

```
# Функция с вложенной функцией:
def mysum(*a):
    # Список:
    txt=["чисел","квадратов","кубов"]
    # Вложенная функция:
    def calc(n):
        s=0
        for m in range(len(a)):
            s+=a[m]**n
        return s
```

```
    # Вызов вложенной функции:
    for k in range(len(txt)):
        print("Сумма", txt[k]+":", calc(k+1))
# Вызов функции:
mysum(1,3,5,7)
```

Так выглядит результат выполнения программы.

Результат выполнения программы (из листинга 6.11)

```
Сумма чисел: 16
Сумма квадратов: 84
Сумма кубов: 496
```

Функция `mysum()` описана с одним аргументом (он обозначен как `a`) со «звездочкой». Это означает, что при вызове функции ей можно передавать произвольное количество аргументов. В этой функции описана вложенная функция `calc()` с одним аргументом `n` (мы предполагаем, что это целое число). В теле этой функции перебираются значения, переданные аргументами внешней функции. Для этого использован оператор цикла `for` с индексной переменной `m`. За каждый цикл выполняется команда `s+=a[m]**n`, которой значение аргумента `a[m]` в степени `n` прибавляется к текущему значению переменной `s` (ее начальное значение равно нулю). По окончании вычислений значение переменной `s` возвращается в качестве результата вложенной функции.

НА ЗАМЕТКУ

Таким образом, при вызове вложенной функции `calc()` с некоторым аргументом `n` результатом возвращается сумма чисел, переданных в качестве аргументов внешней функции `mysum()`, возведенных в степень, определяемую аргументом функции `calc()` (то есть `n`).

Во внешней функции запускается оператор цикла, в котором индексная переменная `k` перебирает значения индексов локального списка `txt`. За каждый цикл выполняется команда `print("Сумма", txt[k]+":", calc(k+1))`, в которой вызывается вложенная функция `calc()`. На этом описание функции `mysum()` заканчивается. Также программа содержит команду `mysum(1,3,5,7)` вызова данной функции. Последствия, думается, вполне понятны и предсказуемы.

Лямбда-функции

Желтые штаны! Два раза «ку»!

Из к/ф «Кин-дза-дза»

Мы несколько раз подчеркивали, что название функции можно рассматривать как некоторую «переменную», которая ссылается на объект функции. Естественным образом возникает вопрос о том, можно ли как-то получить собственно функцию, не привязывая ее к какому-то конкретному имени? Такая возможность существует. Состоит она в том, чтобы использовать *анонимные функции*, или *лямбда-функции* (соответствующую конструкцию иногда еще называют *лямбда-выражением*).

Описывается лямбда-функция достаточно просто: указывается ключевое слово `lambda`, после которого перечисляются аргументы, ставится двоеточие и в той же строке — значение, которое возвращает функция. То есть шаблон описания лямбда-выражения такой:

```
lambda аргументы: результат
```

Если аргументов несколько, они перечисляются через запятую. Если аргументов нет, они просто пропускаются.

Выражение описанного выше типа создает объект для анонимной функции. Значением лямбда-выражения является ссылка на этот объект. Такую ссылку в качестве значения можно присвоить переменной (после чего эта переменная станет «функцией»). И вообще, лямбда-выражение в большинстве случаев может интерпретироваться как функция. Просто обычно мы обращаемся к функции по ее имени, а тут вместо имени используется более сложная конструкция. Поэтому при желании лямбда-функцию можно вызвать, не присваивая предварительно в качестве значения переменной. Примеры таких ситуаций представлены в программе в листинге 6.12.

Листинг 6.12. Использование лямбда-функций

```
num=10
# Функция на основе лямбда-выражения:
L=lambda n: 2*n+1
# Проверка результата:
print("Нечетные числа:")
```

```
for k in range(num):
    print(L(k), end=" ")
# Новое значение:
L=lambda n: 2**n
# Проверка результата:
print("\nСтепени двойки:")
for k in range(num):
    print(L(k), end=" ")
# Прямой вызов лямбда-функции:
print("\nКвадраты чисел:")
for k in range(num):
    print((lambda x: x*x)(k+1), end=" ")
# Обычная функция:
def calc(x, y):
    return x+y
# Использование функции в лямбда-выражении:
F=lambda x, y: calc(x, y)
# Переменной присваивается имя функции:
f=calc
# Имени функции присваивается лямбда-выражение:
calc=lambda x, y: x*y
# Проверка результата:
print("\nВызов F(3,5):", F(3,5))
print("Вызов f(3,5):", f(3,5))
print("Вызов calc(3,5):", calc(3,5))
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 6.12)

```
Нечетные числа:
1 3 5 7 9 11 13 15 17 19
Степени двойки:
1 2 4 8 16 32 64 128 256 512
```

```
Квадраты чисел:
1 4 9 16 25 36 49 64 81 100
Вызов F(3,5): 15
Вызов f(3,5): 8
Вызов calc(3,5): 15
```

В программе командой `L=lambda n: 2*n+1` лямбда-выражение `lambda n: 2*n+1` присваивается в качестве значения переменной `L`. Фактически в данном случае создается функция. В соответствии с лямбда-выражением, у этой функции один аргумент `n` и в качестве результата функция возвращает значение выражения `2*n+1` (формула для нечетного числа при условии, что `n` является числом целым).

После проверки работы вновь созданной функции командой `L=lambda n: 2**n` мы ее переопределяем. Теперь при заданном аргументе `n` результатом возвращается двойка в соответствующей степени.

Как отмечалось выше, лямбда-функцию можно вызывать напрямую, без присваивания лямбда-выражения в качестве значения переменной. В программе есть пример такого вызова: в операторе цикла использована инструкция `(lambda x: x*x)(k+1)`, которая означает, что функция, определяемая лямбда-выражением `lambda x: x*x` (вычисляется квадрат значения аргумента), вызывается с аргументом `k+1`.

Кроме этого, в программе создается обычная функция `calc()` с двумя аргументами, и в качестве результата функция возвращает сумму этих аргументов. Далее выполняется несколько манипуляций. Во-первых, командой `F=lambda x, y: calc(x, y)` фактически определяется функция. Функция определяется на основе лямбда-функции с двумя аргументами, причем в этой лямбда-функции вызывается функция `calc()`.

Во-вторых, выполняется команда `f=calc`, которой в переменную `f` записывается ссылка на объект функции, на который в данный момент ссылается идентификатор `calc`.

В-третьих, командой `calc=lambda x, y: x*y` идентификатору `calc` присваивается в качестве значения лямбда-выражение, определяющее функцию с двумя аргументами, которая результатом возвращает произведение этих аргументов.

После всех этих действий вычисляются значения выражений `F(3,5)`, `f(3,5)` и `calc(3,5)`. Каков будет результат? При вычислении выражения `F(3,5)` в качестве результата возвращается значение выражения `calc(3,5)`. А функция `calc()` на момент вызова определена так, что результатом является произведение аргументов. Поэтому и значение выражения `F(3,5)`, и значение выражения `calc(3,5)`, оба равны `15` (произведение чисел `3` и `5`). А вот значение выражения `f(3,5)` равно `8` (сумма чисел `3` и `5`). Почему так? Потому что на момент выполнения команды `f=calc` идентификатор ссылался на объект функции, которая в качестве результата возвращает сумму аргументов. Поэтому в переменную `f` будет записана та же ссылка, и факт, что идентификатор `calc` получает новое значение, на значении переменной `f` никак не сказывается.

Функция как аргумент и результат

— Астронавты! Которая тут цапа?
— Там... ржавая гайка, родной.
— У вас все тут ржавое.
— А эта самая ржавая.

Из к/ф «Кин-дза-дза»

Функция может передаваться в качестве аргумента другой функции, а также может возвращаться в качестве результата функции. При этом нередко используются вложенные функции или лямбда-функции. Чтобы понять технологию этих процессов, следует помнить, что функция представляет собой некий объект, доступ к которому реализуется через ссылку, которую можно присвоить в качестве значения переменной. Как это выглядит на практике, показано в программе в листинге 6.13.

Листинг 6.13. Функция как аргумент и результат

```
# Аргумент функции — функция (и два числа):
def display(f, a, b):
    for k in range(a, b+1):
        print("{0:4}".format(f(k)), end=" ")
    print()
# Результат функции — функция:
def mypow(n):
    return lambda x: x**n
```

```
# Аргументы функции — функции. Результат — функция:
def apply(f, h):
    def calc(x):
        return f(h(x))
    return calc
# Определение функций:
A=mypow(2)
B=mypow(3)
C=apply(lambda x: 2*x+1, lambda x: 2*x)
# Проверка результата:
print("x   ", end="")
display(lambda x: x,1,5)
print("A(x)", end="")
display(A,1,5)
print("B(x)", end="")
display(B,1,5)
print("C(x)", end="")
display(C,1,5)
# Определение функции:
F=lambda f: lambda x: f(f(x))
# Проверка результата:
print("F(x->x*x)(5): ", F(lambda x: x*x)(5))
print("F(x->2*x+1)(5):", F(lambda x: 2*x+1)(5))
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 6.13)

```
x      1    2    3    4    5
A(x)   1    4    9   16   25
B(x)   1    8   27   64  125
C(x)   5    9   13   17   21
F(x->x*x)(5):  625
F(x->2*x+1)(5): 23
```

Проанализируем программный код и результаты его выполнения. В первую очередь в программе объявляется несколько функций. У функции `display()` есть три аргумента (`f`, `a` и `b`). Мы предполагаем, что первым аргументом `f` является ссылка на функцию (имя функции), а два других аргумента (`a` и `b`) — целые числа (мы так хотим). При вызове функции `display()` на интервале значений, определяемой вторым и третьим аргументами, отображаются значения функции, переданной первым аргументом. Реализуется это просто: запускается оператор цикла, в котором индексная переменная `k` принимает значения от `a` до `b` включительно. При заданном значении `k` в области вывода отображается значение `f(k)` (то есть мы с первым аргументом `f` обращаемся так, как если бы это было имя функции).

У функции `mypow()` один аргумент `n`. Мы исходим из того, что это целое число. В качестве результата функции возвращается выражение `lambda x: x**n`. Это ссылка на функцию, которая для заданного аргумента `x` возвращает значение `x**n` (аргумент в степени, которая определяется аргументом функции `mypow()`). При выполнении команды `A=mypow(2)` в переменную `A` записывается ссылка на функцию, которая для заданного аргумента результатом возвращает аргумент в квадрате (то есть в степени `2`). Выполнение команды `B=mypow(3)` приводит к тому, что переменная `B` содержит ссылку на функцию, которая при вызове с некоторым аргументом возвращает значение аргумента в кубе (то есть в степени `3`).

В описании функции `apply()` есть два аргумента `f` и `h`, которые, в нашем понимании, должны быть функциями. В теле функции `apply()` описывается вложенная функция `calc()` с одним аргументом `x`. Результатом эта вложенная функция возвращает значение выражения `f(h(x))`. Ссылка на вложенную функцию `calc()` (инструкция `return calc`) возвращается в качестве результата функции `apply()`.

ПОДРОБНОСТИ

Таким образом, при вызове функции `apply()` ей в качестве аргумента передаются две функции, а результатом возвращается ссылка на функцию, результат которой вычисляется так: при заданном аргументе сначала вызывается функция, переданная вторым аргументом функции `apply()`, и полученное значение передается в качестве аргумента функции, переданной первым аргументом функции `apply()`.

Командой `C=apply(lambda x: 2*x+1, lambda x: 2*x)` определяется функция, которая для аргумента `x` возвращает в качестве результата выражение `4*x+1`.

ПОДРОБНОСТИ

С помощью лямбда-выражений аргументами функции `apply()` передаются функциональные зависимости f(x) = 2x + 1 и h(x) = 2x. Результатом возвращается функция p(x) = f(h(x)) = 2(2x) + 1 = 4x + 1.

Проверяем результат с помощью функции `display()`. Первым аргументом функции передается ссылка на ту функцию, значения которой нужно отображать. Это может быть переменная со ссылкой на функцию, название функции или лямбда-выражение. Еще два аргумента — диапазон изменения аргумента для отображения значений.

Пример определения функции, которая результатом возвращает функцию, реализован командой `F=lambda f: lambda x: f(f(x))`. Проанализируем это выражение. Значением переменой `F` присваивается в качестве значения лямбда-выражение, определяющее функцию, которая по аргументу `f` возвращает результатом выражение `lambda x: f(f(x))`. Значение этого выражения — это ссылка на функцию, которая для аргумента `x` возвращает значение `f(f(x))`. Поэтому при вызове «функции» `F()` в качестве аргумента передается ссылка на функцию `f()`, а результатом является ссылка на функцию, которая для аргумента `x` возвращает значение `f(f(x))`. Следовательно, результатом выражения `F(lambda x: x*x)` является ссылка на функцию, которая для аргумента `x` возвращает аргумент в четвертой степени. При значении аргумента `5` получаем результат `625`. Значением выражения `F(lambda x: 2*x+1)` является ссылка на функцию, которая для аргумента `x` возвращает значение `4*x+3`. Для значения аргумента `5` получаем результат `23`.

ПОДРОБНОСТИ

Если $f(x) = x^2$, то $f(f(x)) = (x^2)^2 = x^4$. Если f(x) = 2x + 1, то f(f(x)) = 2(2x + 1) + 1 = 4x + 3.

Рекурсия

Странная, если не сказать больше.

Из к/ф «Бриллиантовая рука»

Есть особый подход в создании функций, который базируется на *рекурсии*. Рекурсия — это способ описания функции, когда функция вызывает сама себя. Как это делается на практике, показано в программе в листинге 6.14.

Листинг 6.14. Использование рекурсии

```
# Функции с рекурсивным вызовом.
# Функция для вычисления суммы чисел:
def mysum(n):
    if n==0:
        return 0;
    else:
        return n+mysum(n-1)
# Функция для вычисления чисел Фибоначчи:
def fib(n):
    if n==1 or n==2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
# Функция для инверсного отображения текста/списка:
def show(txt):
    if len(txt)==0:
        print("|")
    else:
        print("|", txt[-1], end="", sep="")
        show(txt[:-1])
# Вызов функций:
print("Сумма чисел:")
for k in range(12):
```

```
    print(mysum(k), end=" ")
print("\nЧисла Фибоначчи:")
for k in range(15):
    print(fib(k+1), end=" ")
print("\nИнверсия текста:")
show("Hello Python")
print("Инверсия списка:")
show([1,2,3,4,5])
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 6.14)

```
Сумма чисел:
0 1 3 6 10 15 21 28 36 45 55 66
Числа Фибоначчи:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
Инверсия текста:
|n|o|h|t|y|P| |o|l|l|e|H|
Инверсия списка:
|5—4|3—2|1|
```

В программе с использованием рекурсии определены три функции: функция для вычисления суммы чисел `mysum()`, функция для вычисления чисел Фибоначчи `fib()` и функция `show()` для отображения текста в инверсном порядке.

Функция `mysum()` определена так: если значение аргумента `n` равно нулю, то результатом функции возвращается значение `0`. В противном случае результатом функции возвращается значение выражения `n+mysum(n-1)`, в котором функция `mysum()` вызывает сама себя, но с уменьшенным на единицу аргументом.

ПОДРОБНОСТИ

Если функция `mysum()` вызывается с аргументом `0`, то результат равен `0`. Если аргумент больше нуля, то функция рекурсивно вызывает сама себя. Допустим, функция вызвана с аргументом `5`. Тогда

результат вызова вычисляется как сумма чисел `5` и значения, возвращаемого функцией при вызове с аргументом `4`. При вычислении этого выражения начинает вычисляться сумма числа `4` и значения выражения, полученного при вызове функции с аргументом `3`. И так далее. Процесс будет продолжаться до тех пор, пока функция `mysum()` не будет вызвана со значением аргумента `0`. Аналогичным образом выполняются и прочие функции с рекурсией.

Функция `fib()` для вычисления числа Фибоначчи по его номеру `n` (аргумент функции) описана так, что при значении аргумента, равном `1` или `2` в качестве результата функция возвращает значение `1` (первые два числа в последовательности Фибоначчи равны единице). В противном случае результатом функции возвращается выражение `fib(n-1)+fib(n-2)` (очередное число в последовательности равно сумме двух предыдущих).

Мы предполагаем, что функция `show()` аргументом получает текстовое значение или, например, список для инверсного отображения. Выполняется функция так: если длина текста или списка равна нулю, то выполняется команда `print("|")`. В противном случае командой `print("|", txt[-1], end="", sep="")` отображается значение последнего символа в тексте или элемента в списке. Затем командой `show(txt[:-1])` функция вызывает сама себя, но переданный функции аргумент не содержит последнего элемента (значение выражения `txt[:-1]` — это срез от первого до предпоследнего элемента включительно). Таким образом, длина последовательности, передаваемой аргументом при каждом новом вызове функции, уменьшается на единицу, пока функция не будет вызвана с пустым текстом или списком в качестве аргумента. На этом цепочка рекурсивных вызовов прекратится.

Помимо описания функций с рекурсией, программа содержит примеры вызова этих функций. Там все просто, и комментарии, хочется верить, не нужны.

НА ЗАМЕТКУ

Откровенно говоря, для решения рассмотренных выше задач использовать рекурсию не было совершенно никакой необходимости. Оправданием служит лишь то, что нас интересовал не результат, а процесс.

Декораторы функций

> Все эти десять лет он искусно маскировался под порядочного человека. Я ему не верю.
>
> *Из к/ф «Бриллиантовая рука»*

При работе с функциями могут использоваться *декораторы*. Декоратор — это некоторая инструкция, которая размещается перед описанием функции и изменяет ее поведение. Чтобы понять природу этого механизма, мы сначала рассмотрим некоторое преобразование, а затем выясним, как оно реализуется на практике.

Итак, для «приготовления» нашего блюда нам понадобится две функции, причем одна из них должна быть особенной: эта функция в качестве аргумента должна принимать функцию и в качестве результата она должна возвращать тоже функцию. Для большей конкретики предположим, что функция `A()` такова, что в качестве аргумента ей передается функция, и результатом является тоже функция. Поэтому если `h()` есть некоторая функция, то выражение `A(h)` результатом возвращает ссылку на другую функцию. Преобразование, которое нас интересует, имеет вид `h=A(h)`. Суть его состоит в том, что на основе некоторой функции `h()` путем преобразования, которое задается функцией `A()`, мы создаем новую функцию, и ссылку на объект этой новой функции записываем в идентификатор `h` (то есть фактически переопределяем функцию `h()`). В принципе, такую процедуру можно было бы организовать стандартными средствами. Однако есть и более простой способ, который как раз и базируется на использовании декоратора. Состоит он в том, что перед описанием функции, которую мы хотим подвергнуть преобразованию, указывается инструкция, которая состоит из символа `@` и названия функции, с помощью которой выполняется преобразование. Например, если мы хотим применить преобразование к функции `h()` и использовать для этого преобразования функцию `A()`, то перед описанием функции `h()` следует разместить инструкцию `@A`. При этом функция `h()` будет определена не так, как она описана, а с учетом преобразования, выполняемого функцией `A()`. Как это выглядит на практике, иллюстрирует программа в листинге 6.15.

Листинг 6.15. Декораторы функций

```
# Функции для использования в декораторах:
def A(h):
```

```
    return lambda x: h(x)*h(7-x)
def B(h):
    return lambda x, y: h(x, y)+h(y, x)
def C(h):
    return lambda x: h(x,10-x)
# Функции с декоратором:
@A
def f(x):
    return 2*x-1
@B
def F(x, y):
    return (8-x)*(y+1)
@C
def H(x, y):
    return x*y
# Проверка результата:
print("f(3) =", f(3))
print("F(5,7) =", F(5,7))
print("H(6) =", H(6))
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 6.15)

```
f(3) = 35
F(5,7) = 30
H(6) = 24
```

В этой программе мы описываем три функции `A()`, `B()` и `C()`, предназначенные для использования в декораторах. Объединяет эти функции то, что у них один аргумент-функция, и в качестве результата они возвращают функцию. В частности, функция `A()` определена так, что если ей передается аргументом ссылка на некоторую функцию `h()`, то результатом возвращается ссылка на функцию, которая для заданного аргумента `x` возвращает выражение `h(x)*h(7-x)`. Эта функция

определяется лямбда-выражением `lambda x: h(x)*h(7-x)`. Причем здесь предполагается, что функции `h()` передается один аргумент.

Функция `A()` используется в программе, когда перед описанием функции `f()` размещается декоратор `@A`. Вся конструкция выглядит так:

```
@A
def f(x):
    return 2*x-1
```

Здесь формально функция `f()` определена так, что для аргумента `x` возвращается значение `2*x+1`. Но поскольку мы использовали декоратор, то на самом деле результат вызова функции `f()` будет иным. Результатом преобразования `A(f)` является функция, которая для аргумента `x` возвращает выражение `f(x)*f(7-x)`, причем значение выражения `f(x)` вычисляется как `2*x-1`, а значение выражения `f(7-x)` вычисляется как `2*(7-x)-1` (что эквивалентно выражению `13—2*x`). Таким образом, при вызове функции `f()` с аргументом `x` результат вычисляется как `(2*x-1)*(13—2*x)`. При значении аргумента, равном `3`, функция `f()` возвращает значение `35`.

Функция `B()` также предназначена для использования в декораторе. Она определена таким образом, что при передаче ей в качестве аргумента ссылки на некоторую функцию `h()` (функция с двумя аргументами) результатом возвращается ссылка на функцию, которая при вызове с аргументами `x` и `y` возвращает значение `h(x, y)+h(y, x)`. Функция, ссылка на которую возвращается в качестве результата функции `B()`, определяется лямбда-выражением `lambda x, y: h(x, y)+h(y, x)`.

Функцию `B()` мы используем в декораторе в описании функции `F()`. Соответствующая конструкция выглядит следующим образом:

```
@B
def F(x, y):
    return (8-x)*(y+1)
```

Учитывая способ определения функции `B()` и то, как описана функция `F()`, несложно сообразить, что при вызове функции `F()` с аргументами `x` и `y` результат будет вычисляться как значение выражения `F(x, y)+F(y, x)`. Причем здесь `F(x, y)` вычисляется как `(8-x)*(y+1)`, а `F(y, x)` вычисляется как `(8-y)*(x+1)`. Таким образом, окончательно результат

вызова функции `F()` с аргументами `x` и `y` такой, как если бы вычислялся выражением `(8-x)*(y+1)+(8-y)*(x+1)`. При значениях аргументов `5` и `7` получаем результат `30`.

Наконец, функция `C()` описана так, что при передаче ей аргументом ссылки на функцию `h()` результатом возвращается ссылка на функцию, определяемую лямбда-выражением `lambda x: h(x,10-x)`. Важно здесь то, что функция `h()`, как предполагается, имеет два аргумента. А вот результатом является функция с одним аргументом. И эта функция-результат такая, что при передаче ей в качестве аргумента значения `x` результат вычисляется вызовом функции `h()`, причем первым аргументом функции передается значение `x`, а вторым аргументом передается значение `10-x`.

Декоратор `@C` используется в описании функции `H()`:

```
@C
def H(x, y):
    return x*y
```

Исходная версия функции `H()` описана с двумя аргументами `x` и `y`, и ее результат вычисляется как произведение значений аргументов `x*y`. Но из-за наличия декоратора эта функция «теряет» один аргумент. Что же происходит при вызове функции `H()` с аргументом `x`? В соответствии с описанием функции `C()` должно вычисляться выражение `H(x,10-x)`, которое означает, в силу описания функции `H()`, вычисление произведения `x*(10-x)`. При значении аргумента `6` в результате получаем число `24`.

Проверка результатов вызова функций подтверждает справедливость наших рассуждений.

Функции-генераторы

Легким движением руки брюки превращаются...

Из к/ф «Бриллиантовая рука»

Мы уже упоминали, что существуют специальные функции, которые называют *функциями-генераторами*. Функции-генераторы позволяют создавать особые итерируемые объекты, которые можно затем использовать в операторах цикла или создавать на их основе списки и другие последовательности.

НА ЗАМЕТКУ

В этом разделе под итерируемым объектом мы будем подразумевать именно тот итерируемый объект, который создается при вызове функции-генератора.

ПОДРОБНОСТИ

Концепция функций-генераторов тесно связана с объектно-ориентированным программированием. Принципы объектно-ориентированного программирования мы будем обсуждать, но немного позже. Пока же отметим, что функция-генератор возвращает в качестве результата определенный объект. Главное свойство этого объекта состоит в том, что его содержимое можно «перебирать», подобно тому, как перебирается содержимое списка. С такими объектами мы уже на самом деле сталкивались — например, когда использовали функцию `range()`. В данном случае речь идет о том, что мы сами можем описать функцию, подобную функции `range()`.

С формальной точки зрения главное отличие функции-генератора от обычной функции состоит в том, что вместо ключевого слова `return` мы используем инструкцию `yield`. Причем речь не идет о простой замене одной инструкции на другую. Вся «идеология» создания функции-генератора отличается от того, что мы делали ранее.

Концепция такая: в процессе выполнения функции-генератора необходимо сформировать набор значений, которые затем будут возвращаться при переборе содержимого объекта, возвращаемого функцией-генератором в качестве результата. При этом нет необходимости создавать сам объект — он автоматически создается и возвращается как результат функции. То есть наша задача сводится лишь к тому, чтобы указать, что следует включить в итерируемый объект. Именно для этих целей и служит ключевое слово `yield`. Значение, включаемое в итерируемый объект, указывается после инструкции `yield`. В листинге 16.6 представлена программа, в которой создаются и используются функции-генераторы.

Листинг 6.16. Функции-генераторы

```
# Функции-генераторы:
def names():
    yield "Дядя Федор"
    yield "Пес Шарик"
```

```
    yield "Кот Матроскин"
def colors():
    L=["Красный","Желтый","Зеленый","Синий"]
    for clr in L:
        yield clr
def myrange(n):
    for k in range(n):
        yield 2*k+1
# Использование функций-генераторов:
print("Они из Простоквашино:")
for name in names():
    print(name)
print(list(names()))
R=colors()
print("Цветовой спектр:")
for r in R:
    print(r, end=" ")
print("\nЕще одна попытка…")
for r in R:
    print(r, end=" ")
print("Ничего нет? Это нормально.")
print("Нечетные числа:")
print(list(myrange(10)))
print(tuple(myrange(10)))
N=myrange(8)
A=list(N)
print("A =", A)
B=list(N)
print("B =", B)
for num in myrange(8):
    print(num, end=" ")
print()
```

Ниже представлен результат выполнения программы.

Результат выполнения программы (из листинга 6.16)

```
Они из Простоквашино:
Дядя Федор
Пес Шарик
Кот Матроскин
['Дядя Федор', 'Пес Шарик', 'Кот Матроскин']
Цветовой спектр:
Красный Желтый Зеленый Синий
Еще одна попытка…
Ничего нет? Это нормально.
Нечетные числа:
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
(1, 3, 5, 7, 9, 11, 13, 15, 17, 19)
A = [1, 3, 5, 7, 9, 11, 13, 15]
B = []
1 3 5 7 9 11 13 15
```

В программе используется три функции-генератора. Проанализируем код каждой из них.

В функции `names()` последовательно выполняются команды `yield "Дядя Федор"`, `yield "Пес Шарик"` и `yield "Кот Матроскин"`. Поэтому при вызове функции `names()` результатом возвращается объект, который «содержит» элементы `"Дядя Федор"`, `"Пес Шарик"` и `"Кот Матроскин"` (извлекаются в той последовательности, как они добавлялись в итерируемый объект).

Похожим образом реализуется функция-генератор `colors()`, но на этот раз текстовые значения (названия `"Красный"`, `"Желтый"`, `"Зеленый"` и `"Синий"`) записываются в список `L`, после чего в операторе цикла переменная `clr` последовательно принимает значения из списка `L` и командой `yield clr` соответствующее значение добавляется в итерируемый объект.

Наконец, функция-генератор `myrange()` описана с одним аргументом `n`. В теле функции запускается оператор цикла `for`, в котором переменная

k пробегает значения от 0 до n-1 включительно. Для каждого значения k выполняется команда yield 2*k+1. Таким образом, итерируемый объект «заполняется» последовательностью нечетных чисел, и количество этих чисел определяется аргументом n функции-генератора.

Далее программа содержит примеры использования функций-генераторов. Так, мы используем инструкцию вызова функции-генератора names() в операторе цикла. Еще с помощью данной функции создается список (инструкция list(names()). И здесь может сложиться впечатление, что такого же (или похожего) результата мы могли бы добиться, если бы функция names() была обычной и возвращала бы в качестве результата список ["Дядя Федор","Пес Шарик","Кот Матроскин"]. Но это не так. Между итерируемым объектом и списком, помимо «формальных» отличий, есть одно фундаментальное: содержимое списка можно перебирать много раз, а вот итерируемый объект — «одноразовый». Его «перебрать» можно только один раз. После этого объект к использованию фактически не пригоден. И в программе есть иллюстрация к этому. А именно, командой R=colors() в переменную R записывается результат вызова функции-генератора colors(). Таким образом, переменная R содержит ссылку на итерируемый объект. Поэтому мы, например, можем использовать эту переменную в операторе цикла, как до этого использовали инструкцию вызова функции-генератора. Но если мы после завершения оператора цикла попытаемся повторить такую же процедуру, ни одна итерация выполнена не будет. Причина как раз в том, что итерируемый объект, на который ссылается переменная R, уже «израсходован». Использовать его повторно не получится.

НА ЗАМЕТКУ

Если бы мы в операторе цикла вместо переменной R использовали инструкцию вызова функции-генератора colors(), то таких операторов цикла можно было бы использовать сколько угодно. Причина в том, что каждый раз при вызове функции colors() создается новый итерируемый объект. При этом переменная R ссылается на один и тот же объект, созданный при присваивании переменной значения.

При вызове функции-генератора myrange() ей передается целочисленный аргумент, определяющий количество нечетных чисел в итерируемом объекте. Результат вызова функции-генератора можно использовать в операторе цикла, при создании списка (инструкция list(myrange(10))) или, например, кортежа (инструкция tuple(myrange(10))).

При этом если мы, как и в случае выше, присваиваем результат вызова функции-генератора переменной (команда `N=myrange(8)`), то итерируемый объект, на который ссылается переменная `N`, можно использовать единожды: список командой `A=list(N)` мы создаем, а вот список, созданный командой `B=list(N)`, оказывается пустым. Причина описана выше.

Аннотации и документирование в функциях

— Ей, может, что-нибудь надо?
— Что ей надо, я тебе потом скажу!

Из к/ф «Бриллиантовая рука»

Как уже многократно отмечалось, функция реализуется в виде специального объекта (если точнее, то объекта класса `function`). Это имеет последствия: у функции есть некоторые свойства, обусловленные как раз с тем, что в основе всей этой конструкции находится объект.

НА ЗАМЕТКУ

Классы и объекты мы будем обсуждать в главах, посвященных объектно-ориентированному программированию. Сейчас же речь пойдет лишь о тех аспектах, которые имеют непосредственное отношение к практическому использованию функций.

В частности, в описании функции сразу после строки с названием функции можно размесить текст. Этот текст называется текстом *документирования*. При выполнении кода функции он игнорируется. Но впоследствии его можно «прочитать», указав после имени функции (без круглых скобок) через точку поле `__doc__` (начинается и заканчивается двойным символом подчеркивания). Поэтому если в описании функции разместить начальный текст с кратким описанием функции, то в последующем при необходимости мы сможем использовать это описание. В целом механизм достаточно удобный. В листинге 6.17 представлена программа, в которой используется документирование функции.

Листинг 6.17. Использование документирования

```
# Функции с документированием:
def show(txt):
    "Это функция show() с одним аргументом."
```

```
    print("Единственный аргумент:", txt)
def display(a, b):
    "Это функция display() с двумя аргументами."
    print("[1] Первый аргумент:", a)
    print("[2] Второй аргумент:", b)
# Функция без документирования:
def hello():
    print("Всем привет!")
# Вызов функций и проверка документирования:
print(show.__doc__)
show("A")
print(display.__doc__)
display("B","C")
# Переменная ссылается на функцию:
f=show
# Вызов функций и проверка документирования:
print(f.__doc__)
f("D")
# Новый текст документирования для функции:
display.__doc__="Новый текст для display()"
# Проверка результата:
print(display.__doc__)
display("E","F")
# Создается документирование для функции:
hello.__doc__="Функция hello()"
# Проверка результата:
print(hello.__doc__)
hello()
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 6.17)

```
Это функция show() с одним аргументом.
Единственный аргумент: A
```

```
Это функция display() с двумя аргументами.
[1] Первый аргумент: B
[2] Второй аргумент: C
Это функция show() с одним аргументом.
Единственный аргумент: D
Новый текст для display()
[1] Первый аргумент: E
[2] Второй аргумент: F
Функция hello()
Всем привет!
```

Код достаточно простой и в комментариях не нуждается. Хочется лишь обратить внимание, на два обстоятельства. Во-первых, текст документирования «привязан» к объекту функции, а не к переменной, которая ссылается на объект. Примером служит ситуация, когда в переменную `f` записывается ссылка на функцию `show()`. Во-вторых, текст документирования можно изменить (или создать) в процессе работы с функцией. Для этого достаточно полю `__doc__` присвоить соответствующее значение.

Также при работе с функциями существует возможность создавать специальные «пояснения» для аргументов функции и результата функции. Такие «пояснения» называются *аннотациями*.

Если аргументы функции имеют аннотации, то такая аннотация указывается в описании функции после имени соответствующего аргумента. Название аргумента и аннотация разделяются двоеточием. Если при этом аргумент имеет значение по умолчанию, то значение по умолчанию указывается после аннотации (через оператор присваивания). Общий синтаксис описания аргумента (с аннотацией и значением по умолчанию) выглядит так:

```
аргумент: аннотация=значение
```

Аннотация для результата функции указывается через стрелку -> после круглых скобок с аргументами (но перед двоеточием) в описании функции. Вся конструкция выглядит так:

```
def функция(аргументы)->аннотация:
    # Код функции
```

Наличие или отсутствие аннотаций (если в качестве последних не использованы команды) не влияет на результат выполнения функции. Получить значения для аннотаций функции можно с помощью поля `__annotations__` (начинается и заканчивается двойным пробелом). Значением поля `__annotations__` является словарь, ключи которого — это названия аргументов, а значения соответствующих элементов — это аннотации. Аннотация к результату функции запоминается с ключом `"return"`.

ПОДРОБНОСТИ

Если для аргумента или результата аннотация не предусмотрена, то соответствующий элемент словаря имеет значение `None`.

В листинге 6.18 представлена программа, в которой используются аннотации.

Листинг 6.18. Использование аннотаций

```
# Функция с аннотациями:
def show(txt:"Текст"="Функция show()")->"Результата нет":
    print(txt)
# Вызов функции:
show()
# Словарь аннотаций:
print(show.__annotations__)
# Аннотации:
for k in show.__annotations__:
    print(k,"-", show.__annotations__[k])
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 6.18)

```
Функция show()
{'txt': 'Текст', 'return': 'Результата нет'}
txt — Текст
return — Результата нет
```

Это достаточно простая программа, и хочется верить, что читатель сможет с ней разобраться. Но есть момент, на который стоит обратить внимание. Как и в любой словарь, мы можем добавить в него новый элемент или даже удалить элемент из словаря. В таком случае фактическое содержимое словаря не будет соответствовать «архитектуре» функции.

Резюме

Меня терзают смутные сомнения.

Из к/ф «Иван Васильевич меняет профессию»

- Функция представляет собой именованный блок программного кода, который можно вызвать по имени. При вызове функции могут передаваться аргументы. Функция может возвращать результат.
- Перед использованием (вызовом) функции ее необходимо описать. Описание функции начинается с ключевого слова `def`, после которого указывается имя функции и список аргументов (в круглых скобках), затем ставится двоеточие. Затем описывается тело функции (команды, которые выполняются при вызове функции). Для вызова функции указывают имя функции и в круглых скобках — аргументы, которые ей передаются.
- Функция реализуется в виде объекта. Идентификатор с названием функции содержит ссылку на объект функции.
- Аргументы функции могут иметь значения по умолчанию. Если при вызове соответствующий аргумент не указан, то используется значение по умолчанию. В описании функции значения аргументов по умолчанию указываются через оператор присваивания в описании аргументов.
- Аргументы в функцию можно передавать по позиции (порядок передачи аргументов соответствует порядку их описания в определении функции) и по имени (по ключу). В последнем случае указывается имя аргумента и, через оператор присваивания, — его значение.
- Аргументы в функцию передаются по значению: на самом деле передается копия переменной, фактически указанной в качестве аргумента функции. Как следствие, изменить значение аргумента в теле функции нельзя.

- Можно создавать функции с произвольным количеством аргументов. Такой набор аргументов описывается как один аргумент, перед названием которого указывается «звездочка» `*`. Обрабатывается подобный аргумент как кортеж.
- Переменные, которым значения присваиваются в теле функции, являются локальными и доступны только в теле функции. Чтобы использовать в функции глобальную переменную, она декларируется в теле функции с ключевым словом `global`.
- Одна функция может быть описана в другой функции. Такая функция называется вложенной. Она имеет доступ ко всем переменным внешней функции. Однако внешняя функция доступа к внутренним переменным вложенной функции не имеет.
- Можно создавать функции без имени (анонимные функции или лямбда-функции). Описание лямбда-функции начинается с ключевого слова `lambda`, затем перечисляются аргументы, ставится двоеточие и указывается значение, возвращаемое функцией. Значением такой инструкции (на основе ключевого слова `lambda`) является ссылка на объект функции.
- Функция может передаваться в качестве аргумента другой функции. В качестве результата функция может возвращать ссылку на другую функцию.
- При создании функций можно использовать рекурсию. В этом случае в теле функции содержится инструкция, которой функция вызывает сама себя.
- Для функций могут использоваться декораторы. В таком случае перед описанием функции указывается символ `@` и имя некоторой функции, которая аргументом принимает ссылку на функцию, и в качестве результата возвращается другая функция. Если так, то соответствующее преобразование применяется для описываемой функции.
- Функции-генераторы в качестве результата возвращают итерируемый объект. Элементы в этот итерируемый объект добавляются в процессе выполнения функции-генератора на основе инструкции с ключевым словом `yield`.
- Текстовая строка сразу после названия функции в ее описании является текстом документирования. Доступ к этому тексту можно получить с помощью поля `__doc__`. Для аргументов и результата

функции можно использовать аннотации. Для аргументов они указываются через двоеточие после имени аргумента, а для результата аннотация указывается через стрелку `->` после круглых скобок в заголовке функции. Значением поля `__annotations__` является словарь с названиями аргументов (ключи) и аннотациями для них (значения элементов). Аннотация для результата доступна по ключу `"return"`.

Задания для самостоятельной работы

— Я вам предлагаю маленький заговор.
— А большой нельзя?
— Маленький, с большими последствиями.
— Что надо делать? Я готова на все.

Из к/ф «31 июня»

1. Напишите программу, в которой создается функция с двумя аргументами, являющимися числовыми списками. Результатом является число, равное сумме попарных произведений элементов списков. Если в одном из списков элементов меньше, чем в другом, то недостающие элементы получают путем циклического повторения содержимого списка.

2. Напишите программу с функцией, аргументом которой передается числовой список, а результатом является еще один список, в который включены только нечетные числа из списка-аргумента.

3. Напишите программу, в которой описывается функция с произвольным количеством числовых аргументов, а результатом возвращается список из трех элементов: среднее значение аргументов, максимальное значение среди аргументов и минимальное значение среди аргументов.

4. Напишите программу, в которой создается функция с одним текстовым аргументом и произвольным количеством целочисленных аргументов. Результатом является текст, сформированный из букв первого текстового аргумента. Целочисленные аргументы определяют индексы букв, которые нужно включить в текст-результат.

5. Напишите программу с функцией, которая в качестве аргумента получает ссылку на функцию и два целых числа. Результатом функция возвращает наибольшее значение переданной первым аргументом функции

в целочисленных точках диапазона (границы которого определяются вторым и третьим аргументом).

6. Напишите программу с функцией, которая аргументами получает ссылку на функцию (например, `f()`) и целое число (например, `n`). Результатом является функция, которая вычисляет результат путем `n`-кратного применения функции `f()`.

7. Напишите программу, в которой методом рекурсии символы из текста, переданного аргументом функции, отображаются «через один»: то есть отображается первый, третий, пятый и так далее, символы.

8. Напишите программу, в которой методом рекурсии вычисляется сумма геометрической прогрессии: первое слагаемое равно единице, а каждое следующее получается из предыдущего умножением на определенное число (передается в качестве аргумента функции, также как и количество слагаемых в сумме).

9. Напишите программу, в которой используется функция-генератор, создающая итерируемый объект с названиями месяцев.

10. Напишите программу, в которой используется функция-генератор, создающая итерируемый объект со степенями двойки. Количество элементов определяется аргументом функции-генератора.

Глава 7
ФАЙЛЫ И ДАННЫЕ

Что бы мы делали без науки? Подумать страшно!

Из к/ф «31 июня»

В этой главе мы рассмотрим некоторые вопросы, связанные с использованием данных. В первую очередь это будут числовые данные. Но кроме них мы еще уделим внимание работе с датой и временем. Также мы узнаем, как информация считывается из файлов и как она в файлы заносится. А еще эта глава содержит примеры выполнения всевозможных операций, в том числе мы рассмотрим и вычислительные задачи.

Числовые данные

Очень убедительно. Мы подумаем, к кому это применить.

Из к/ф «31 июня»

Мы уже знаем, что целочисленные значения реализуются как данные типа `int`. Кроме целых чисел, в Python могут использоваться действительные числа. Они реализуются как данные типа `float`. Но кроме целых и действительных еще можно использовать комплексные числа, которые реализуются как значения типа `complex`.

Целые числа применялись нами многократно. Действительные числа тоже использовались. А вот с комплексными числами сталкиваться нам еще не приходилось. Тем не менее у нас остались поводы обсудить все — даже целые числа.

НА ЗАМЕТКУ

Кроме числовых типов `int`, `float` и `complex`, можно использовать типы `Fraction` и `Decimal`. Эти типы позволяют оперировать

соответственно с рациональными дробями и действительными числами фиксированной точности. Утилиты для работы с данными типа `Fraction` собраны в модуле `fractions`, а для использования утилит, предназначенных для работы с данными типа `Decimal`, подключается модуль `decimal`. Методы работы с типами `Fraction` и `Decimal` описываются далее.

Целые числа задаются с помощью литералов. Причем это могут быть литералы не только в десятичной системе счисления, также допускается использовать двоичную, восьмеричную и шестнадцатеричную системы.

ПОДРОБНОСТИ

Система счисления определяется количеством разных цифр (символов), используемых для позиционного представления числа. Например, в двоичной системе счисления для записи чисел используется две цифры (0 и 1). В восьмеричной системе счисления используется восемь цифр (от 0 до 7 включительно). В десятичной системе счисления числа записываются с помощью десяти цифр (от 0 до 9 включительно). Еще одна популярная система счисления — шестнадцатеричная. В ней для записи чисел используется шестнадцать символов (цифры от 0 до 9 и буквы от A до F, обозначающие числа от 10 до 15 соответственно).

Если основу системы счисления составляет число N (количество разных цифр/символов, используемых для записи числа), то число с позиционным представлением $a_n a_{n-1}...a_2 a_1 a_0$ в десятичную систему переводится по формуле $a_n a_{n-1}...a_2 a_1 a_0 = a_n N^n + a_{n-1} N^{n-1} + ... + a_2 N^2 + a_1 N^1 + a_0 N^0$, причем коэффициенты a_m (m = 0,1,2,..., n) могут принимать значения от 0 до N-1 включительно. Например, число 135 в восьмеричной системе равно десятичному числу 93, поскольку $135 = 1 \cdot 8^2 + 3 \cdot 8^1 + 5 \cdot 8^0 = 93$. Число A2B в шестнадцатеричной системе соответствует десятичному числу 2603 ($A2B = 10 \cdot 16^2 + 2 \cdot 16^1 + 11 \cdot 16^0 = 2603$). Вообще, чем больше основа системы счисления, тем «компактнее» получаются числа. Самые «длинные» числа — в двоичной системе: скажем, число 10011 в двоичной системе соответствует десятичному числу 19 ($10011 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$).

Литералы в двоичной системе начинаются с префикса `0b` (или `0B`). Для создания восьмеричных литералов используют префикс `0o` (или `0O`), а шестнадцатеричные литералы начинаются с префикса `0x` (или `0X`). В последнем случае для обозначения чисел от `10` до `15` используются

буквы от A до F (можно использовать как большие, так и маленькие буквы). Целочисленные литералы, заданные в системах счисления, отличных от десятичной, интерпретируются как целые числа с соответствующим десятичным значением. Переменные, которым в качестве значения присваиваются двоичные, восьмеричные или шестнадцатеричные значения могут использоваться в вычислениях как обычные переменные с целочисленным значением.

Чтобы по целочисленному значению, заданному в десятичной системе, получить бинарный, восьмеричный или шестнадцатеричный код данного числа, можно воспользоваться соответственно функциями `bin()`, `oct()` и `hex()`. В качестве аргумента каждой из функций передается десятичное число, а результатом является текст с представлением числа в соответствующей системе счисления. Если нам нужно выполнить обратную операцию — по текстовому представлению числа в одной из систем счисления получить числовое значение в десятичной системе, можно использовать функцию `int()`. Первым аргументом функции передается текст с кодом числа в соответствующей системе счисления, а второй аргумент определяет основание системы счисления, в которой представлено число.

ПОДРОБНОСТИ

Если текстовое представление для числа содержит префикс, определяющий систему счисления (`0b`, `0o` или `0x`), то соответствующее тестовое выражение также можно передать в качестве аргумента функции `eval()`. В результате вычисляется выражение, «спрятанное» в тексте. Если это выражение будет корректным целочисленным литералом, заданным в соответствующей системе счисления, то в итоге получим число. Например, результатом выражения `eval("0b101")` является десятичное число `5`, поскольку литерал `0b101` задает целое число с двоичным кодом `101`, что соответствует числу `5`.

Что касается использования разных систем счисления, то фактически в Python поддерживаются системы счисления с основанием от `2` до `36` включительно. В случае необходимости (если основанием системы счисления является число больше `10`) в качестве «цифр» используются буквы от `A` (значение `10`) до `Z` (значение `35`). Другими словами, с помощью функции `int()` на основе текстового представления для числа в произвольной системе счисления (с основанием от `2` до `36`) можно получить число. А для случаев, когда используется система счисления с основанием `2`, `8` или `16`, можно еще и задавать литералы.

В листинге 7.1 представлен небольшой пример, в котором выполняются операции с целыми числами, заданными в разных системах счисления.

Листинг 7.1. Целые числа в разных системах счисления

```
print("Числа заданы в разных системах:")
# Число 19 в двоичной системе:
A=0b10011
print("A =", A)
# Число 93 в восьмеричной системе:
B=0o135
print("B =", B)
# Число 2603 в шестнадцатеричной системе:
C=0xA2B
print("C =", C)
print("C-A-B =", C-A-B)
# Обратное преобразование:
print("Обратное преобразование:")
A=int("10011",2)
print(bin(A),"=", A)
B=int("0o135",8)
print(oct(B),"=", B)
C=int("0xA2B",16)
print(hex(C),"=", C)
D=int("ABC",20)
print("ABC =", D)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 7.1)

```
Числа заданы в разных системах:
A = 19
B = 93
```

```
C = 2603
C-A-B = 2491
Обратное преобразование:
0b10011 = 19
0o135 = 93
0xa2b = 2603
ABC = 4232
```

В данном случае программа простая: сначала нескольким переменным присваиваются целочисленные значения, причем они заданы в разных системах счисления. Но важно понимать, что речь идет лишь о способе представления числа. Переменные при этом получают обычные целочисленные значения, с которыми можно выполнять все арифметические операции. Также в программе есть примеры использования функций `bin()`, `oct()` и `hex()` для получения текстового представления числа в соответствующей системе счисления. Функция `int()` используется для получения на основе текстового представления числа (в определенной системе счисления) собственно числа.

ПОДРОБНОСТИ

Значение выражения `int("ABC",20)` вычисляется следующим образом. Текст `"ABC"` интерпретируется как число, заданное в системе счисления с основанием `20`. Буква `A` соответствует числу `10`, символ `B` соответствует числу `11`, символ `C` соответствует числу `12`. Поэтому получается $12 \cdot 20^0 + 11 \cdot 20^1 + 10 \cdot 20^2 = 4232$.

Что касается операций, которые могут выполняться с целыми (и не только целыми) числами, то с большинством из них мы уже сталкивались. Тем не менее в табл. 7.1 перечислены арифметические операторы, которые могут применяться к числовым значениям.

Табл. 7.1. Арифметические операторы

Оператор	Описание
+	Оператор *сложения*. Результат выражения вида `A+B` с числовыми операндами `A` и `B` вычисляется как сумма значений этих операндов. Также может использоваться как унарный оператор, определяющий положительный знак для числа (например, допустимым является выражение вида `+A`)

Табл. 7.1. Арифметические операторы. Продолжение

-	Оператор *вычитания*. Результат выражения вида `A-B` вычисляется как разность значений числовых операндов `A` и `B`. Кроме этого, используется как унарный оператор, определяющий отрицательные числа (например, в выражении вида `-A`)
*	Оператор *умножения*. Результат выражения вида `A*B` с числовыми операндами `A` и `B` вычисляется как произведение значений операндов `A` и `B`
/	Оператор *деления*. Результат выражения вида `A/B` вычисляется делением значения операнда `A` на значение операнда `B`
//	Оператор *целочисленного деления*. Результат выражения вида `A//B` вычисляется делением нацело значения операнда `A` на значение операнда `B`. А именно, вычисляется результат (обычного) деления значения операнда `A` на значение операнда `B`, а полученный результат округляется до ближайшего целочисленного значения (в направлении минус бесконечности)
%	Оператор вычисления *остатка* от целочисленного деления. Результатом выражения вида `A%B` является остаток от деления значения операнда `A` на значение операнда `B`. А именно, остаток от деления интерпретируется как значение выражения вида `A-(A//B)*B`
**	Оператор *возведения в степень*. Результатом выражения вида `A**B` является число, которое получается возведением значения операнда `A` в степень, определяемую значением операнда `B`

НА ЗАМЕТКУ

Для выполнения арифметических операций полезными могут быть некоторые функции. Например, с помощью функции `abs()` вычисляется модуль числа, переданного аргументом функции. Для возведения числа в степень может использоваться функция `pow()`. Аргументов у функции два: число и степень, в которую оно возводится. Функция `divmod()` результатом возвращает кортеж из двух элементов: результат целочисленного деления и остаток от деления чисел, переданных аргументами функции.
Также можно использовать математические функции из модуля `math`.

Если речь идет о целочисленных значениях, то нередко к таким значениям применяются *побитовые операции*. Это операции, определенные непосредственно на уровне побитового представления числа.

НА ЗАМЕТКУ

Числовые значения в памяти хранятся и обрабатываются в виде бинарных кодов. Другими словами, хотя мы в программах обычно

используем десятичные числа, на самом деле они «запоминаются» и обрабатываются в двоичной системе (ну, во всяком случае, мы так можем думать). При этом есть особенность, связанная с запоминанием отрицательных чисел. Поясним это на примере. Допустим, нам нужно закодировать число `-5`. Для числа `5` бинарный код имеет вид `101`. С отрицательным числом `-5` проблема связана с тем, как «кодировать» знак «минус». Дело в том, что под число выделяется определенное количество битов, и каждый бит может содержать `0` или `1`. Допустим, что число кодируется с помощью `32` битов. Тогда код числа `5` будет содержать `29` нулей в начале и `101` — в конце. А как закодировать число `-5`? Будем отталкиваться от того, что сумма чисел `5` и `-5` должна давать в результате `0`. То есть если мы найдем код, который в сумме с кодом числа `5` будет давать `0`, то этот код и будет кодом числа `-5`. Проделаем такую мысленную операцию: возьмем код числа `5` и заменим в нем `0` на `1`, а `1` на `0` (эта процедура называется побитовой инверсией). Получим код, у которого в начале `29` единиц, а в конце — `010`. Этот код сложим с кодом числа `5`. Несложно сообразить, что получим код, в котором `32` единицы. Если к этому результату прибавить `1`, получим код, у которого в начале `1`, и затем `32` нуля. Но поскольку для числа выделяется только `32` бита, то первая единица «теряется» и остается код из одних нулей. Это число `0`. Таким образом, если мы инвертируем код числа `5` и прибавим к результату единицу, получим код, который в сумме с кодом числа `5` дает `0`. Вывод — мы получили код числа `-5`.

Общее правило выглядит так: чтобы получить код отрицательного числа, нужно взять код противоположного положительного числа, заменить в нем нули на единицы и наоборот и прибавить к результату единицу.

Получается, что у отрицательных чисел старший бит будет единичным, а у положительных — нулевым. Поэтому обычно старший бит называют знаковым (он определяет знак числа).

Если у нас есть код числа с начальным нулем, то это положительное число. Для его перевода в десятичное представление используются стандартные формулы, которые приводились ранее. Если код числа начинается с единицы, то это отрицательное число. Чтобы понять, какое это число, следует инвертировать код, прибавить к нему единицу, полученное значение перевести в десятичную систему и «дописать» знак «минус».

Для выполнения операций на побитовом уровне есть специальные операторы, которые называются *побитовыми*. Побитовые операторы, используемые в языке Python, перечислены в табл. 7.2.

Табл. 7.2. Побитовые операторы

Оператор	Описание
`&`	Оператор *побитовое и*. Результатом выражения вида `A&B` с целочисленными операндами `A` и `B` является число. Его код получается попарным сравнением значений битов в операндах `A` и `B`. Если оба бита равны `1`, в результате получаем единичный бит. Если хотя бы один бит равен `0`, в результате получаем нулевой бит
`|`	Оператор *побитовое или*. Результатом выражения вида `A|B` с целочисленными операндами `A` и `B` является число, код которого получается попарным сравнением значений битов в операндах `A` и `B`. Если хотя бы один бит равен `1`, в результате получаем единичный бит. Если оба бита равны `0`, в результате получаем нулевой бит
`^`	Оператор *побитовое исключающее или*. Результатом выражения вида `A^B` с целочисленными операндами `A` и `B` является число. Его код получается попарным сравнением значений битов в операндах `A` и `B`. Если значения битов разные, то в результате получаем единичный бит. Если оба бита имеют одинаковые значения, в результате получаем нулевой бит
`<<`	Оператор побитового *сдвига влево*. Результатом выражения вида `A<<n` является число с кодом, который получается сдвигом влево кода числа `A` на количество позиций, определяемых значением операнда `n`. Младшие биты заполняются нулями. Старшие биты теряются
`>>`	Оператор побитового *сдвига вправо*. Результатом выражения вида `A>>n` является число с кодом, который получается сдвигом вправо кода числа `A` на количество позиций, определяемых значением операнда `n`. Старшие биты заполняются значением знакового бита. Младшие биты теряются
`~`	Оператор побитового *инвертирования*. Результатом выражения вида `~A` является число, код которого получается из кода числа `A` заменой нулей на единицы и единиц на нули

НА ЗАМЕТКУ

Стоит отметить, что сдвиг влево на одну позицию в бинарном представлении числа эквивалентен умножению числа на `2`. Сдвиг вправо на одну позицию в бинарном представлении числа эквивалентен целочисленному делению этого числа на `2`.

Также достоин внимания тот факт, что если `x` — некоторая переменная с целочисленным значением, то значением выражения `~x+1` является `-x`.

Пример, в котором используются побитовые операторы, представлен в листинге 7.2.

Листинг 7.2. Побитовые операции

```
# Переменные:
A=13
print("A =", A)
B=7
print("B =", B)
# Побитовые операции:
print("A&B =", A&B)
print("A|B =", A|B)
print("A^B =", A^B)
print("~A+1 =",~A+1)
# Побитовые сдвиги:
print("B>>1 =", B>>1)
print("B<<2 =", B<<2)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 7.2)

```
A = 13
B = 7
A&B = 5
A|B = 15
A^B = 10
~A+1 = -13
B>>1 = 3
B<<2 = 28
```

В программе мы используем переменные `A` и `B` со значениями `13` и `7`. В двоичном представлении они имеют коды `1101` и `0111` соответственно. При вычислении значения выражения `A&B` при попарном сравнении битов единичный бит получаем, только если оба исходных бита единичные. Поэтому результатом является код `0101`, который означает число `5`.

При вычислении выражения `A|B` при сравнении битов единичный бит получаем, если хотя бы один из битов единичный. Это дает нам код `1111`, определяющий число `15`.

Значением выражения `A^B` является число `10`. Получается результат следующим образом. При сравнении кодов `1101` и `0111` в соответствующей позиции имеем единицу, если сравниваемые биты имеют разные значения. Это дает код `1010`, то есть число `10`.

Поскольку значение переменой `A` равно `13`, то значением выражения `~A+1` (с учетом способа кодировки отрицательных чисел) является число `–13`.

При выполнении команды `B>>1` код `0111` смещается вправо на одну позицию, и мы получаем код `011` для числа `3`. При вычислении значения выражения `B<<2` код `0111` сдвигается влево на две позиции, и мы получаем код `011100` для числа `28`.

НА ЗАМЕТКУ

Для определения количества битов, необходимых для записи числа, можно использовать функцию `bit_length()`. Биты определяются без учета старших нулевых битов для положительных чисел и старших единичных битов для отрицательных чисел.

Действительные числа по умолчанию реализуются как значения типа `float`. Такое значение может быть получено или в результате вычислений, или введено как соответствующий литерал. Причем при вводе литерала действительная и дробная части отделяются точкой. Также можно использовать экспоненциальную нотацию, когда для числа указывается мантисса, и через символ `E` или `e`, показатель степени.

НА ЗАМЕТКУ

Если имеется текстовое представление для действительного числа, то получить на его основе само число можно с помощью функции `float()`.

Кроме значений типа `float`, иногда полезными могут оказаться типы `Fraction` (из модуля `fractions`) и `Decimal` (из модуля `decimal`). Тип `Fraction` позволяет выполнять операции с рациональными дробями.

НА ЗАМЕТКУ

По умолчанию все действительные значения запоминаются и обрабатываются в приближенном виде с определенной точностью. Например, если мы в качестве значения некоторой переменной присвоим значение `1/3`, то переменной на самом деле будет присвоено приближенное значение, которое получается в результате деления числа `1` на число `3`. Тип `Fraction` позволяет обрабатывать действительные значения, такие как `1/3`, не в приближенном, а точном виде.

Если действительное число реализуется как значение типа `Fraction`, то в действительности оно определяется двумя целыми числами — числителем и знаменателем дроби. Эти значения передаются в качестве аргументов функции `Fraction()` при создании числа. Если арифметические операции выполняются со значениями типа `Fraction`, то результатом будет значение того же типа. Фактически в данном случае речь идет о том, что операции с действительными числами выполняются «точно», без округления.

С помощью значений типа `Decimal` можно выполнять операции с числами, заданными в формате с плавающей точкой, но при этом сами операции будут выполняться без округления (с определенной точностью). Пример, в котором выполняются несложные операции с действительными числами, представлен в листинге 7.3.

Листинг 7.3. Действительные числа

```
from fractions import Fraction
from decimal import Decimal
print("Дробные значения:")
A=Fraction(2,5)
print("A =", A)
B=Fraction(3,7)
print("B =", B)
C=A+B
print("A+B =", C)
print("Действительные числа:")
X=2/5
```

```
print("X =", X)
D=X+B
print("X+B =", D)
print("Числа заданной точности:")
A=Decimal('1.01')
print("A =", A)
B=Decimal('2.02')
print("B =", B)
C=A+B
print("A+B =", C)
print("1.01+2.02 =",1.01+2.02)
```

Результат выполнения программы будет следующим.

Результат выполнения программы (из листинга 7.3)

```
Дробные значения:
A = 2/5
B = 3/7
A+B = 29/35
Действительные числа:
X = 0.4
X+B = 0.8285714285714285
Числа заданной точности:
A = 1.01
B = 2.02
A+B = 3.03
1.01+2.02 = 3.0300000000000002
```

Командами `A=Fraction(2,5)` и `B=Fraction(3,7)` мы двум переменным присваиваем значения, которые соответствуют рациональным дробям `2/5` и `3/7`. При вычислении выражения `C=A+B` получаем дробь `29/35` (легко проверить, что `2/5+3/7=29/35`). То есть результатом выражения `A+B` является значение типа `Fraction`. Но если хотя бы один из операндов является значением типа `float` (как, например, в выражении `D=X+B` с использованием переменной `X`, которой значение

присваивалось командой `X=2/5`), результат вычисляется как значение типа `float`.

Значения типа `Decimal` использованы в командах `A=Decimal('1.01')` и `B=Decimal('2.02')`. Здесь мы создаем значения типа `Decimal` на основе текстовых представлений для действительных чисел, и в результате соответствующие действительные числа будут точно такими, как указано в текстовых литералах. В результате вычисления выражения `C=A+B` получаем ожидаемое значение `3.03`. Для сравнения в программе приведен результат вычисления суммы чисел `1.01` и `2.02`, которые по умолчанию интерпретируются как значения типа `float`.

Как отмечалось выше, в Python существует возможность работать с комплексными числами. Соответствующие значения реализуются как данные типа `complex`. Создать комплексное число можно либо с помощью литерала, либо вызвав функцию `complex()` с аргументами, определяющими действительную и мнимую части комплексного числа.

ПОДРОБНОСТИ

Мнимой единицей (обычно обозначается как *i*) называется такое воображаемое число, которое, будучи возведенным в квадрат (умноженным на себя) дает значение –1. То есть по определению $i^2 = -1$.
Любое комплексное число z может быть представлено в виде $z = x + i \cdot y$, где *x* и y являются действительными числами (соответственно, действительная и мнимая части комплексного числа). Алгебраические операции (такие как сложение, вычитание, деление и умножение) комплексных чисел выполняются так же, как и для действительных чисел, но с учетом того обстоятельства, что $i^2 = -1$.
Если число $z = x + i \cdot y$, то комплексно-спряженным к этому числу называется число $z^* = x - i \cdot y$, которое получается заменой в исходном числе i на -i. Модулем комплексного числа называется величина $|z| = \sqrt{z \cdot z^*} = \sqrt{x^2 + y^2}$.
Также комплексное число может быть представлено в тригонометрической форме $z = |z| \cdot \exp(i \cdot \varphi) = \cos(\varphi) + i \cdot \sin(\varphi)$ следует, что $\cos(\varphi) = x / |z|$ и $\sin(\varphi) = y / |z|$.

В литералах, определяющих комплексные числа, мнимая часть задается с помощью суффикса `j` или `J`. Например литерал `1j` определяет мнимую единицу, а литерал `3+4j` соответствует комплексному числу *3 + 4i*. Такое же число получим в результате выполнения инструкции `complex(3,4)`. Пример, в котором иллюстрируется работа с комплексными числами, представлен в листинге 7.4.

Листинг 7.4. Комплексные числа

```
# Комплексные числа:
A=3+4j
print("A =", A)
print("Re(A) =", A.real)
print("Im(A) =", A.imag)
print("|A| =", abs(A))
B= -1+1j
print("B =", B)
C=complex(0,1)
print("C =", C)
# Арифметические операции:
print("A+B =", A+B)
print("A*C =", A*C)
print("A/B =", A/B)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 7.4)

```
A = (3+4j)
Re(A) = 3.0
Im(A) = 4.0
|A| = 5.0
B = (-1+1j)
C = 1j
A+B = (2+5j)
A*C = (-4+3j)
A/B = (0.5—3.5j)
```

В данном случае создается несколько комплексных чисел и с ними выполняются наипростейшие арифметические операции. С математической точки зрения, как отмечалось выше, все эти действия выполняются так же, как с действительными числами, но только следует учитывать, что мнимая единица в квадрате дает значение `–1`. Для одного из чисел

вычисляется действительная и мнимая части (использованы атрибуты `real` и `imag`, соответственно), а также модуль числа (корень квадратный из суммы квадратов действительной и мнимой части). В последнем случае использована функция `abs()`.

Логические значения

> Мы не будем полагаться на случай. Мы пойдем простым логическим ходом.
>
> *Из к/ф «Ирония судьбы, или С легким паром»*

Вообще логический тип `bool` подразумевает, что соответствующая переменная может принимать только одно из двух значений: `True` (истина) или `False` (ложь). Вместе с тем практически любой объект может быть проверен на предмет «истинности». Это означает, что в качестве логического значения могут использоваться (например, в условном операторе или операторе цикла) данные не только логического типа. При этом соответствующий объект, указанный вместо логического значения, будет интерпретироваться как истинный (значение `True`) или ложный (значение `False`) в зависимости от своих характеристик. Для числовых значений (всех типов) правило такое: нулевое значение интерпретируется как `False`, а отличное от нуля значение интерпретируется как `True`. Пустые последовательности (к которым относятся и текстовые строки) интерпретируются как значения `False`, а если последовательность содержат элементы (текст содержит символы), то такая конструкция интерпретируется как значение `True`.

> **НА ЗАМЕТКУ**
>
> Вообще для объектов за «истинность» или «ложность» отвечает специальный метод `__bool__()`. Все эти вопросы мы рассмотрим немного позже, когда познакомимся с классами и объектами.

При работе с логическими значениями нередко используется оператор *логическое и* `and` и оператор *логическое или* `or`. Вообще, если `A` и `B` являются логическими значениями, то значением выражения вида `A and B` является значение `True`, если оба операнда равны `True`. Если хотя бы один из них равен `False`, то значением выражения `A and B` является значение `False`. Значением выражения вида `A or B` является значение

True, если хотя бы один из операндов имеет значение True. Если значения обоих операндов равны False, тогда результатом выражения A or B будет значение False.

> **НА ЗАМЕТКУ**
>
> Есть еще унарный оператор логического отрицания not. Результатом выражения not A является значение True, если у операнда A значение False. Если значение операнда A равно False, то результатом выражения not A будет значение True.

Вместе с тем не все так просто, как кажется на первый взгляд. Стоит отметить несколько моментов. Во-первых, если при вычислении выражения вида A and B при проверке значения операнда A окажется, что это значение False, то результат от значения операнда B уже не зависит — значением выражения A and B будет False. Во-вторых, если при проверке операнда A в выражении вида A or B получим значение True, то результатом выражения A or B будет True вне зависимости от значения операнда B. И, в-третьих, учтем, что операнды A и B не обязательно должны быть логическими значениями. Из всего этого следует, что при вычислении выражений на основе операторов and и or могут быть сюрпризы. Чтобы эти сюрпризы не стали неприятными, следует учесть правила, по которым вычисляются соответствующие выражения. В описанных далее правилах подразумеваются, что операнды A и B не обязательно являются логическими значениями.

- При вычислении выражения вида A and B сначала проверяется значение операнда A. Если это значение интерпретируется как True, то тогда в качестве значения выражения возвращается значение операнда B. Если при проверке значения операнда A оно интерпретируется как False, то в качестве результата всего выражения возвращается значение операнда A.
- При вычислении выражения вида A or B если значение операнда A интерпретируется как True, то в качестве значения всего выражения возвращается значение операнда A. Если значение операнда A интерпретируется как False, то результатом выражения является значение операнда B.

Пример, в котором иллюстрируются особенности использования логических операторов, представлен в листинге 7.5.

Листинг 7.5. Логические операторы

```
A=123
print("A =", A)
B="Python"
print("B =", B)
C=[]
print("C =", C)
D=0
print("D =", D)
print("A or B:", A or B)
print("A and B:", A and B)
print("C or D:", C or D)
print("C and D:", C and D)
```

Ниже показано, каким будет результат выполнения программы.

Результат выполнения программы (из листинга 7.5)

```
A = 123
B = Python
C = []
D = 0
A or B: 123
A and B: Python
C or D: 0
C and D: []
```

Пример простой, но результат все же прокомментируем. Так, результатом выражения `A or B` является значение переменной `A`, поскольку число `123` отлично от нуля и интерпретируется как `True`. А вот результатом выражения `A and B` является значение переменной `B`.

При проверке значения первого операнда `C` в выражении `C or D` это значение (пустой список `[]`) интерпретируется как ложное, поэтому значением выражения является значение операнда `D`. В силу той же причины

(пустой список интерпретируется как значение `False`) результатом выражения `C and D` является значение операнда `C`.

Дата и время

В настоящее время каждый имеет свое право.

Из к/ф «Собачье сердце»

Дата и время реализуются с помощью специальных объектов. Классы и объекты мы будем рассматривать позже. Но в силу практической важности данного вопроса работу с датой и временем рассмотрим отдельно.

Вообще дата и время обычно реализуются по относительно простой и «прямолинейной» схеме. Определенная дата выбирается за начало отсчета, а все прочие даты (и моменты времени) отсчитываются (например, в миллисекундах) от этой «опорной» даты. Получается, что любая дата или момент времени определяется некоторым числом (обычно целым). И вопрос лишь сводится к тому, как это число привести к виду, понятному для человека. Вот, собственно, и все. Именно для выполнения такого преобразования используются специальные типы данных и соответствующие им объекты.

В Python для работы с датой и временем есть довольно много утилит. Мы рассмотрим те из них, что содержатся в модуле `datetime`.

НА ЗАМЕТКУ

Далее нам предстоит познакомиться с некоторыми объектами разных типов. Если понятие «тип» применяется по отношению к объекту, то на самом деле имеется в виду класс. Другими словами, класс аналогичен типу данных, но только это «сложный» тип. Объекты соответствующего класса не только содержат определенные данные, но еще и «умеют» обрабатывать эти данные с помощью методов (функции, связанные с объектом). Сами объекты можно интерпретировать как контейнеры, которые содержат в себе другие переменные (которые называются полями или атрибутами объекта), а также функции (называются методами), предназначенные для работы с данными, «спрятанными» в объекте.

Даже если сейчас все сказанное не очень понятно — не страшно. Главное усвоить, что есть определенные синтаксические конструкции, позволяющие работать с датой и временем, и есть правила

(или даже алгоритмы), определяющие, как эти конструкции использовать.

Также стоит заметить, что помимо модуля `datetime` для работы с датой и временем предназначены модули `time` и `calendar`.

В модуле `datetime`, среди прочего, есть класс `time`. Объекты класса `time` предназначены для реализации информации об определенном моменте времени. При создании объекта после ключевого слова `time` в круглых скобках указываются параметры момента времени: часы, минуты, секунды (можно указать еще и микросекунды, а также другие параметры).

НА ЗАМЕТКУ

Одна секунда содержит миллион микросекунд.

У объекта класса `time` есть поля `hour`, `minute`, `second` и `microsecond`, которые позволяют узнать, соответственно, часы, минуты, секунды и микросекунды для момента времени, реализованного с помощью соответствующего объекта.

НА ЗАМЕТКУ

При обращении к полю объекта используется точечный синтаксис: указывается имя объекта и, через точку, название поля.

Например, инструкцией `time(13,35,20)` создается объект класса `time`, который соответствует моменту времени `13` часов, `35` минут и `20` секунд. Если мы используем инструкцию `mytime=time(13,35,20)`, то ссылка на созданный объект будет записана в переменную `mytime`. Значением выражения `mytime.hour` будет число `13` (часы), значением выражения `mytime.minute` будет число `35` (минуты), а значением выражения `mytime.second` будет число `20` (минуты).

Объект для реализации момента времени можно создавать и по-другому. Например, если нужно создать новый объект на основе уже существующего, то можно воспользоваться методом `relpace()`, который вызывается из исходного объекта. В качестве аргументов методу передаются новые значения для часов, минут, секунд (микросекунд). Эти значения будут использованы в новом объекте. Если для какого-то параметра значение не указать, то будет использовано соответствующее значение

из исходного объекта (из которого вызывается метод). Скажем, если используется команда `newtime=mytime.replace(15, second=45)`, то на основе объекта, на который ссылается переменная `mytime`, создается новый объект класса `time` и ссылка на него записывается в переменную `newtime`. При этом значение `13` для часов меняется на значение `15` (такой первый аргумент передан методу `replace()`), значение `20` для секунд меняется на значение `45` (этот аргумент методу `replace()` передан по ключу), а значение для минут остается неизменным. В результате получаем объект, соответствующий моменту времени `15` часов, `35` минут и `45` секунд.

Также можно создать новый объект класса `time` с помощью статического метода `fromisoformat()` (метод вызывается из класса `time`). В качестве аргумента методу передается текстовая строка с указанием момента времени, для которого создается объект. Так, в результате выполнения инструкции `mytime=time.fromisoformat("12:34:56")` для момента времени `12` часов `34` минуты и `56` секунд создается объект класса `time` и ссылка на него записывается в переменную `mytime`.

НА ЗАМЕТКУ

Получить текстовую строку с информацией о моменте времени можно с помощью метода `isoformat()`, который вызывается из объекта класса `time`, для которого нужно получить текстовое представление.

Программа, в которой иллюстрируются способы создания объекта `time`, представлена в листинге 7.6.

Листинг 7.6. Объект для реализации момента времени

```
# Импорт класса из модуля:
from datetime import time
# Объект для реализации момента времени:
mytime=time(13,35,20)
# Проверка результата:
print("Время:", mytime)
# Использование полей объекта:
print("Часы:", mytime.hour)
print("Минуты:", mytime.minute)
```

```
print("Секунды:", mytime.second)
# Создание нового объекта на основе существующего:
newtime=mytime.replace(15, second=45)
# Проверка результата:
print("Время:", newtime)
# Создание нового объекта:
mytime=time.fromisoformat("12:34:56")
# Проверка результата:
print("Время:", mytime)
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 7.6)

```
Время: 13:35:20
Часы: 13
Минуты: 35
Секунды: 20
Время: 15:35:45
Время: 12:34:56
```

Класс `date` предназначен для создания объектов, через которые реализуются даты. Объект можно создать, указав после ключевого слова `date` в круглых скобках год, месяц и число для даты, реализуемой создаваемым объектом (все параметры задаются в числовом виде).

ПОДРОБНОСТИ

Когда мы создаем объект определенного класса, указывая имя этого класса и набор параметров, то на самом деле в этом случае вызывается специальный метод, который называется конструктором. В результате вызова конструктора и создается объект.

Например с помощью инструкции `date(2019,10,22)` создается объект класса `date`, который соответствует 22 октября 2019 года. Поля `year`, `month` и `day` позволяют узнать год, месяц и число, соответствующие дате (реализованной в объекте, для которого запрашиваются поля). Определить день недели по дате можно с помощью методов `weekday()`

и `isoweekday()`. Методы вызываются из объекта класса `date` и в качестве результата возвращают число, определяющее день недели для даты, реализованной объектом. Разница между методами в том, что метод `weekday()` возвращает значение `0` для понедельника, `1` для вторника и так до воскресенья (значение `6`), а метод `isoweekday()` для понедельника возвращает значение `1`, для вторника возвращает значение `2` и так до воскресенья (значение `7`).

Объект для даты можно создавать с помощью метода `replace()`. Метод вызывается из объекта даты, а результатом возвращает новый объект класса `date`. Аргументами методу передаются значения, которые при создании нового объекта нужно заменить в исходном объекте. Например, инструкцией `myday.replace(1985, day=15)` на основе объекта `myday` создается новый объект, соответствующий `1985` году, `15` числу, а месяц такой же, как в объекте `myday`. Также объект даты можно создать с помощью метода `fromisoformat()`, который вызывается из класса `date`, а в качестве аргумента методу передается текстовое представление для даты. Например, командой `date.fromisoformat("1998-08-12")` создается объект класса `date`, который соответствует 12 августа 1998 года.

Еще один важный метод `today()`, который вызывается из класса `date`, позволяет создать объект, соответствующий текущей дате (то есть дате, установленной на компьютере на момент вызова метода). Небольшая программа, в которой используются упомянутые выше подходы, представлена в листинге 7.7.

Листинг 7.7. Объект для реализации даты

```
# Импорт класса из модуля:
from datetime import date
# Объект для реализации даты:
myday=date(2019,10,22)
# Проверка результата:
print("Первая дата:", myday)
# Использование полей объекта:
print("Год:", myday.year)
print("Месяц:", myday.month)
print("Число:", myday.day)
```

```
# Определение дня недели:
print("День недели:", myday.weekday())
print("День недели:", myday.isoweekday())
# Создание нового объекта на основе существующего:
newday=myday.replace(1985, day=15)
# Проверка результата:
print("Вторая дата:", newday)
# Создание нового объекта:
newday=date.fromisoformat("1998-08-12")
# Проверка результата:
print("Новая дата:", newday)
# Объект для текущей даты:
thisday=date.today()
# Проверка результата:
print("Сегодня:", thisday)
# Разность дат:
delta=myday-thisday
# Проверка результата:
print("До первой даты:", delta)
```

Ниже показано, как будет выглядеть результат выполнения программы.

Результат выполнения программы (из листинга 7.7)

```
Первая дата: 2019-10-22
Год: 2019
Месяц: 10
Число: 22
День недели: 1
День недели: 2
Вторая дата: 1985-10-15
Новая дата: 1998-08-12
Сегодня: 2019-06-14
До первой даты: 130 days, 0:00:00
```

В этой программе есть один примечательный момент: в команде `delta=myday-thisday` вычисляется разность двух дат (разность двух объектов класса `date`). Результатом является объект класса `timedelta` (из модуля `datetime`), который содержит информацию об интервале времени между двумя датами.

ПОДРОБНОСТИ

Объект класса `timedelta` предназначен для реализации интервала времени. Такой объект можно создать, указав в круглых скобках после ключевого слова `timedelta` значения для дней, секунд, микросекунд, миллисекунд, минут, часов и недель (соответственно, параметры `days`, `seconds`, `microseconds`, `milliseconds`, `minutes`, `hours` и `weeks`), которые формируют интервал времени. При этом у объекта класса `timedelta` есть поля `days`, `seconds` и `microseconds`, позволяющие получить значение для полного количества дней в интервале, а также секунд (часы и минуты переводятся в секунды) и микросекунд (часы, минуты и секунды переводятся в микросекунды). С помощью метода `total_seconds()` можно узнать длительность всего интервала времени (включая и дни), выраженную в секундах.

Объект класса `datetime` позволяет сохранять информацию о дате и моменте времени (это то, что объекты классов `date` и `time` позволяют делать по отдельности). При создании объекта класса `datetime` в круглых скобках после названия класса указываются параметры, определяющие дату и момент времени — речь о параметрах `year`, `month`, `day`, `hour`, `minute`, `second` и `microsecond`. Одноименные поля используются для считывания соответствующих значений.

Создать объект класса `datetime` можно на основе уже существующего объекта с помощью метода `replace()` (в качестве аргументов методу передаются новые значения для параметров создаваемого объекта), а также с помощью метода `fromisoformat()`, передав аргументом методу текстовое представление для даты и времени. Помимо этого создать объект класса `datetime` можно на основе объектов классов `date` и `time`. Для этого достаточно данные объекты передать аргументами методу `combine()`. Для выполнения обратной операции (получения на основе объекта класса `datetime` объектов классов `date` и `time`) можно воспользоваться методами `date()` и `time()`.

НА ЗАМЕТКУ

У объектов класса `datetime` много методов таких же, как у объектов класса `date`. Например, с помощью методов `weekday()` и `isoweekday()` можно определить день недели, соответствующий дате, реализованной объектом класса `datetime`.

Получить объект класса `datetime`, соответствующий текущей дате и времени, можно с помощью методов `today()` и `now()` (в последнем случае можно передать аргумент, определяющий часовой пояс).

Небольшой пример использования объектов класса `datetime` представлен в листинге 7.8.

Листинг 7.8. Объект для реализации даты и времени

```
# Импорт класса из модуля:
from datetime import datetime
# Объект для реализации даты и времени:
md=datetime(2019,10,22,13,27,45)
# Проверка результата:
print("Дата и время:", md)
# Использование полей объекта:
print("Год:", md.year)
print("Месяц:", md.month)
print("Число:", md.day)
print("Часы:", md.hour)
print("Минуты:", md.minute)
print("Секунды:", md.second)
# Определение дня недели:
print("День недели:", md.weekday())
print("День недели:", md.isoweekday())
# Дата:
d=md.date()
# Проверка результата:
print("Дата:", d)
```

```
# Время:
t=md.time()
# Проверка результата:
print("Время:", t)
# Создание нового объекта на основе существующего:
nd=md.replace(1985, day=3, second=15)
# Проверка результата:
print("Дата и время:", nd)
# Создание нового объекта:
nd=datetime.fromisoformat("1998-08-12 11:25:36")
# Проверка результата:
print("Новая дата и время:", nd)
# Объект для текущей даты и времени:
td=datetime.today()
# Проверка результата:
print("Сегодня и сейчас:", td)
# Разность дат:
delta=md-td
# Проверка результата:
print("Интервал времени:", delta)
print("Дни:", delta.days)
print("Секунды:", delta.seconds)
print("Интервал в секундах:", delta.total_seconds())
```

Результат выполнения программы следующий.

Результат выполнения программы (из листинга 7.8)

```
Дата и время: 2019-10-22 13:27:45
Год: 2019
Месяц: 10
Число: 22
Часы: 13
Минуты: 27
```

```
Секунды: 45
День недели: 1
День недели: 2
Дата: 2019-10-22
Время: 13:27:45
Дата и время: 1985-10-03 13:27:15
Новая дата и время: 1998-08-12 11:25:36
Сегодня и сейчас: 2019-06-15 22:04:46.116228
Интервал времени: 128 days, 15:22:58.883772
Дни: 128
Секунды: 55378
Интервал в секундах: 11114578.883772
```

В процессе работы с интервалом времени (соответствующий объект класса `timedelta` вычисляется как разность объектов класса `datetime`) секунды выражаются в формате числа с плавающей точкой, в котором дробная часть определяет микросекунды.

> **НА ЗАМЕТКУ**
>
> Как отмечалось ранее, для работы с датой и временем, кроме модуля `datetime`, используются модули `time` и `calendar`. Например, в модуле `time` есть функция `sleep()`, которая позволяет приостановить выполнение программы (время в секундах, на которое выполняется пауза в выполнении программы, указывается аргументом функции).

Работа с файлами

> Если мы допустим беспорядок в документации, потомки нам этого не простят.
>
> *Из к/ф «Гостья из будущего»*

В контексте работы с файлами обычно решается две задачи: считывание информации из файла и запись информации в файл. В Python и та и другая задача решаются достаточно просто. Но прежде чем что-то сделать с файлом, его нужно открыть. Для этого существует специальная

функция `open()`. Первым обязательным аргументом функции при вызове указывается название открываемого файла (под названием имеется в виду текстовая строка с названием и полным путем к файлу). Кроме этого необязательного параметра могут указываться и дополнительные опции, которые, кроме прочего, определяют режим доступа к файлу (их мы обсудим позже).

В качестве результата (в случае удачного открытия файла) функция `open()` возвращает ссылку на файловый объект.

ПОДРОБНОСТИ

Файловый объект — это специальный объект, через который реализуется доступ к файлу. Такой объект автоматически создается при вызове функции `open()` и «привязан» к определенному файлу (указанному в качестве аргумента при вызове функции `open()`). В случае если файл открыть не удалось, генерируется исключение класса `OSError`. Такого рода ошибки можно перехватывать и обрабатывать с помощью системы обработки ошибок. Система обработки исключений более детально описывается в одной из следующих глав.

Созданный в результате открытия файла файловый объект имеет специальные методы для выполнения операций с файлом. Среди этих методов есть методы `read()` и `write()`, предназначенные соответственно для считывания содержимого файла и записи информации в файл. Очень простой пример, в котором из заранее созданного файла считывается информация, представлен в листинге 7.9.

Листинг 7.9. Считывание содержимого текстового файла

```
# Открываем текстовый файл для чтения:
mf=open("D:\\Books\\Python\\poetry.txt")
# Считывается содержимое файла:
txt=mf.read()
print("Содержимое файла:")
# Отображение содержимого файла:
print(txt)
# Закрываем файл:
mf.close()
print("Файл закрыт…")
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 7.9)

```
Содержимое файла:
Чтоб мудро жизнь прожить, знать надобно немало.
Два важных правила запомни для начала.
Ты лучше голодай, чем что попало есть,
И лучше будь один, чем вместе с кем попало.
Омар Хайям
Файл закрыт…
```

Предполагается, что перед началом выполнения программы в каталоге `D:\Books\Python` размещается текстовый файл `poetry.txt` с содержимым, как на рис. 7.1.

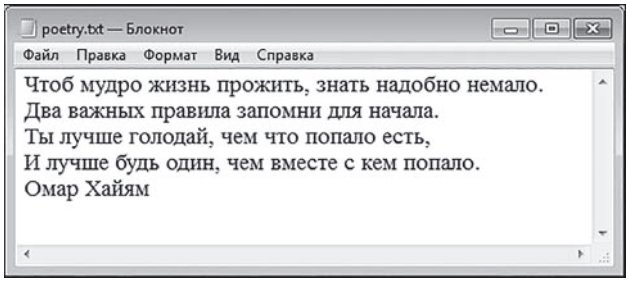


Рис. 7.1. Содержимое считываемого в программе текстового файла

Командой `mf=open("D:\\Books\\Python\\poetry.txt")` этот файл открывается, а ссылка на объект файла записывается в переменную `mf`. Далее командой `txt=mf.read()` считывается содержимое файла, и ссылка на прочитанный текст запоминается в переменной `txt`. После этого прочитанный текст отображается в области вывода, а командой `mf.close()` закрывается файл. Эта операция необходима для того, чтобы гарантировать корректное завершение процесса ввода/вывода данных, связанного с файлом.

В данном случае мы считали полностью содержимое файла. Нередко информацию нужно считывать блоками. Например, текст из файла можно считывать построчно. Для этого можно задействовать оператор цикла, в котором перебирается содержимое текстового файла, а «элементом»

при этом являются отдельные строки. Как все может выглядеть на практике, показано в листинге 7.10.

Листинг 7.10. Построчное считывание содержимого файла

```
# Открываем текстовый файл для чтения:
mf=open("D:\\Books\\Python\\poetry.txt")
# Переменная для нумерации строк:
k=1
# Построчное считывание файла:
print("Построчное считывание файла")
for L in mf:
    # Отображение номера строки и самой строки:
    print("["+str(k)+"]", L, end="")
    # Новое значение для номера строки:
    k+=1
# Закрываем файл:
mf.close()
print("\nФайл закрыт…")
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 7.10)

```
Построчное считывание файла
[1] Чтоб мудро жизнь прожить, знать надобно немало.
[2] Два важных правила запомни для начала.
[3] Ты лучше голодай, чем что попало есть,
[4] И лучше будь один, чем вместе с кем попало.
[5] Омар Хайям
Файл закрыт…
```

В данном случае ссылка на объект файла записывается в переменную `mf`, а в операторе цикла переменная `L` перебирает «содержимое» объекта `mf`. Это означает, что значение переменной `L` на каждой итерации совпадает с очередной строкой из файла.

ПОДРОБНОСТИ

Один из способов просмотра содержимого файла подразумевает использование конструкции `with`. Программа, аналогичная рассмотренной выше, могла бы иметь следующий код:

```
k=1
print("Построчное считывание файла")
with open("D:\\Books\\Python\\poetry.txt") as fm:
    for L in fm:
        print("["+str(k)+"]", L, end="")
        k+=1
print("\nФайл закрыт…")
```

Результат ее выполнения такой же, как и в предыдущем случае. Фактически команду `mf=open("D:\\Books\\Python\\poetry.txt")` мы заменили на инструкцию `with open("D:\\Books\\Python\\poetry.txt") as fm`. При этом по завершении работы файл закрывается автоматически.

Прочитать отдельную строку можно с помощью метода `readline()`. В случае, если достигнут конец файла, метод вернет в результате пустую строку.

ПОДРОБНОСТИ

Если файл содержит пустую строку, то при ее считывании в качестве результата возвращается текст `"\n"`.

Небольшая вариация на тему предыдущего примера, но на этот раз с использованием метода `readline()`, представлена в листинге 7.11.

Листинг 7.11. Еще один способ построчного считывания

```
# Открываем текстовый файл для чтения:
mf=open("D:\\Books\\Python\\poetry.txt")
# Переменная для нумерации строк:
k=1
# Построчное считывание файла:
print("Построчное считывание файла")
```

```
L=mf.readline()
while L!="":
    # Отображение номера строки:
    print("["+str(k)+"] ", end="")
    # Построчное отображение символов строки:
    for s in L:
        # Замена пробела на подчеркивание:
        if s==' ':
            s='_'
        print(s, end="")
    # Новое значение для номера строки:
    k+=1
    # Считывание новой строки:
    L=mf.readline()
# Закрываем файл:
mf.close()
print("\nФайл закрыт…")
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 7.11)

```
Построчное считывание файла
[1] Чтоб_мудро_жизнь_прожить,_знать_надобно_немало.
[2] Два_важных_правила_запомни_для_начала.
[3] Ты_лучше_голодай,_чем_что_попало_есть,
[4] И_лучше_будь_один,_чем_вместе_с_кем_попало.
[5] Омар_Хайям
Файл закрыт…
```

В этом примере строки из файла считываются командой `L=mf.readline()` до тех пор, пока истинно условие `L!=""` (считана не пустая строка). В строке перебираются символы, и пробелы заменяются на символ подчеркивания (для этого используется условный оператор).

Возможности Python не ограничены работой с текстовыми файлами, а информацию из файлов можно не только считывать, но еще и записывать в файлы. В таком случае при открытии файла с помощью функции `open()` следует указать параметр, определяющий режим работы с файлом. Символьные значения, определяющие режим доступа к файлу, перечислены в табл. 7.3.

Табл. 7.3. Символьные значения для определения режима доступа к файлу

Символ	Режим доступа
`'r'`	Режим чтения содержимого файла. Используется по умолчанию, если режим доступа явно не указан
`'w'`	Режим записи в файл (с предварительной очисткой файла)
`'x'`	Режим создания файла с возможность записи данных в файл. Если файл уже существует, возникает ошибка
`'a'`	Режим записи в файл. Если файл существует, то новое содержимое дописывается в конец файла
`'b'`	Режим доступа к бинарному файлу
`'t'`	Режим доступа к текстовому файлу. Используется по умолчанию, если явно режим доступа не задан
`'+'`	Символ используется для формирования шаблона, определяющего доступ к файлу в режиме обновления, когда возможно чтение и запись. Например, `'w+b'` (или `'w+t'`) означает режим, при котором бинарный (или текстовый) файл открывается для чтения и записи с предварительной очисткой содержимого файла. Инструкция `'r+b'` (или `'r+t'`) означает режим доступа к бинарному (или текстовому) файлу для чтения и записи без предварительной очистки содержимого файла

Можно использовать не только отдельные символы, но и их комбинации. Например, `"rb"` означает открытие для чтения бинарного файла, а `"wt"` означает режим доступа к текстовому файлу для записи с предварительной очисткой содержимого файла.

НА ЗАМЕТКУ

Инструкция `"rb"` является эквивалентом инструкции `"b"`, поскольку по умолчанию используется режим доступа для чтения из файла. По этой же причине инструкция `"t"` означает открытие текстового файла для чтения.

Для лучшего понимания ситуации поясним разницу между некоторыми режимами. Как отмечалось выше, режим `"wt"` означает

доступ к текстовому файлу только для записи, причем после открытия файла выполняется его очистка (содержимое файла удаляется). Режим "w+t" означает, что доступ к текстовому файлу осуществляется и для чтения, и для записи, но после открытия выполняется очистка файла. Режим "r+t" означает, что файл открывается для чтения и записи, но очистка файла после открытия не выполняется, а запись данных по умолчанию начинается с начала файла (то есть новое содержимое записывается «поверх» уже существующего). В режиме "a+t" текстовый файл открывается для чтения и записи, но новые данные дописываются в конец файла. Режим "rt" означает открытие текстового файла только для чтения, а в режиме "at" текстовый файл открывается только для записи, причем новые данные дописываются в конец файла (исходное содержимое файла не удаляется).

Также стоит отметить, что вместо комбинаций вида "w+t" можно использовать выражения вида "wt+".

Пример, в котором выполняется запись данных в файл, представлен в листинге 7.12.

Листинг 7.12. Запись данных в файл

```
# Считывается текст:
txt=input("Введите текст: ")
# Файл открывается для записи:
mf=open("D:\\Books\\Python\\mytext.txt",'w')
# Текст записывается в файл:
mf.write(txt)
# Закрывается файл:
mf.close()
# Сообщение о завершении копирования:
print("Текст записан в файл")
```

Программа простая: пользователю предлагается ввести текст, этот текст считывается, а затем записывается в файл mytext.txt, размещенный в каталоге D:\Books\Python. Файл открывается (создается объект файла) командой mf=open("D:\\Books\\Python\\mytext.txt",'w'). Вторым аргументом функции open() указан символ 'w'. Это означает, что файл добавляется для записи. Если такого файла в каталоге не существует, то он будет создан. Если файл уже имеется, то его

прежнее содержимое будет удалено. Для записи текста из переменной `txt` в файл используем команду `mf.write(txt)`. Метод `write()` вызывается из объекта файла, в который записываются данные, а записываемый в файл текст передается методу в качестве аргумента.

Результат выполнения программы может быть таким (жирным шрифтом выделен текст, который вводит пользователь).

Результат выполнения программы (из листинга 7.12)

```
Введите текст: Изучаем язык Python
Текст записан в файл
```

Но главные последствия выполнения программы проявляются не в области вывода. На рис. 7.2 показано содержимое файла `mytext.txt` после выполнения программы.

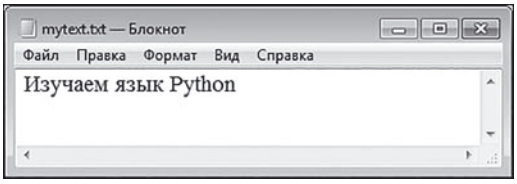


Рис. 7.2. Введенный пользователем текст записывается в файл *mytext.txt*

Как и ожидалось, введенный пользователем текст записан в файл. Немного более сложный пример, связанный с чтением данных из файла и записью данных в файл, представлен в листинге 7.13.

Листинг 7.13. Чтение и запись данных

```
# Имя файла:
name="D:\\Books\\Python\\mydata.txt"
# Текст для записи в файл:
txt="Python"
print("Текст для записи в файл:", txt)
# Файл открывается для чтения и записи:
mf=open(name,'w+t')
# Текст записывается в файл:
mf.write(txt)
```

```
# Переходим в начало файла:
mf.seek(0)
# Первый символ в файле:
print(mf.tell(),"->", mf.read(1))
# Переходим в конец файла:
mf.seek(0,2)
# Позиция с последним символом в файле:
num=mf.tell()-1
# Переходим на позицию с последним символом:
mf.seek(num)
# Последний символ в файле:
print(mf.tell(),"->", mf.read(1))
# Возвращаемся в начало файла:
mf.seek(0)
# Три первых символа:
print("Три символа:", mf.read(3))
# Закрывается файл:
mf.close()
print("Программа завершила выполнение")
```

Результат выполнения программы будет таким.

Результат выполнения программы (из листинга 7.13)

```
Текст для записи в файл: Python
0 -> P
5 -> n
Три символа: Pyt
Программа завершила выполнение
```

В программе для чтения и записи открывается файл `mydata.txt` (использована команда `mf=open(name,'w+t')`). Полный путь к файлу записан в переменную `name`. В файл записывается текст из переменной `txt`. Командой `mf.seek(0)` переходим в начало файла.

ПОДРОБНОСТИ

Мы использовали метод `seek()`. Метод позволяет определить место в файле для считывания или записи значения. Он вызывается из объекта файла и в общем случае в качестве аргументов ему передаются два целых числа. Второй аргумент определяет «опорную» позицию, по отношению к которой определяется место перехода. Началу файла соответствует значение `0`, текущей позиции в файле соответствует значение `1`, концу файла соответствует значение `2`. Первый аргумент задает смещение относительно этой «опорной» позиции. При этом для бинарных файлов «единицей измерения» являются байты, а для текстовых файлов — символы. Если второй аргумент не указан, то по умолчанию смещение выполняется по отношению к началу файла. Для текстовых файлов по отношению к текущей позиции и концу файла может выполняться только нулевой сдвиг.

Узнать текущую позицию можно с помощью метода `tell()`. Методу `read()` при вызове передается целочисленный аргумент, который для текстового файла определяет количество считываемых символов (для бинарного файла — количество считываемых байтов).

Командой `print(mf.tell(),"->", mf.read(1))` отображается текущая позиция в файле и символ на этой позиции. Для определения позиции последнего символа с помощью команды `mf.seek(0,2)` переходим в конец файла, после чего командой `num=mf.tell()-1` вычисляем и записываем в переменную `num` значение позиции последнего символа в файле. Переходим на эту позицию с помощью команды `mf.seek(num)`, и командой `print(mf.tell(),"->", mf.read(1))` отображаем позицию последнего символа и сам символ. Затем возвращаемся в начало файла (команда `mf.seek(0)`) и командой `print("Три символа:", mf.read(3))` отображаем три первых символа в файле. После этого командой `mf.close()` закрываем файл.

Еще одна небольшая программа, в которой копируется файл, представлена в листинге 7.14.

Листинг 7.14. Копирование файла

```
print("Начинается копирование файла")
# Контролируемый код:
try:
    # Бинарный файл открывается для чтения:
    A=open("D:\\Books\\animal.jpg","rb")
```

```
    # Создается бинарный файл:
    B=open("D:\\Books\\Python\\bear.jpg","xb")
    # Содержимое первого файла считывается
    # и записывается во второй файл:
    B.write(A.read())
    # Файлы закрываются:
    A.close()
    B.close()
    print("Копирование прошло успешно")
# Если второй файл уже существует:
except FileExistsError:
    print("Ошибка: такой файл уже существует")
# Все прочие ошибки:
except:
    print("Ошибка доступа к файлу")
print("Программа завершила выполнение")
```

Программа простая и в плане кода, и в плане назначения. При запуске программы на выполнение файл `animal.jpg` из каталога `D:\Books` копируется в каталог `D:\Books\Python`, причем копия будет называться `bear.jpg`. Важное условие состоит в том, что до копирования в каталоге `D:\Books\Python` файла с названием `bear.jpg` быть не должно. Реализуется все достаточно просто. Командой `A=open("D:\\Books\\animal.jpg","rb")` копируемый файл открывается как бинарный для чтения. А командой `B=open("D:\\Books\\Python\\bear.jpg","xb")` создается конечный бинарный файл. После этого командой `B.write(A.read())` содержимое первого файла считывается (инструкция `A.read()`) и записывается во второй файл. Затем оба файла закрываются (команды `A.close()` и `B.close()`).

В программе используется обработка ошибок. Дело в том, что, поскольку при открытии второго файла вторым аргументом функции `open()` указано значение `"xb"`, файл не просто открывается, а создается. Это подразумевает, что такого файла в соответствующем каталоге нет. В противном случае генерируется исключение класса `FileExistsError`. Чтобы его перехватить и обработать, мы команды, связанные с открытием файлов и выполнением манипуляций с ними,

размещаем в `try`-блоке. После этого блока указаны два `except`-блока. В первом из них после ключевого слова `except` указано название класса ошибки `FileExistsError`, которая обрабатывается в этом блоке. Второй `except`-блок, поскольку тип обрабатываемой ошибки для него не указан, обрабатывает все ошибки, которые могут возникнуть, за исключением ошибок класса `FileExistsError`, обрабатываемых в первом `except`-блоке.

Если при выполнении команд в `try`-блоке ошибки не возникают, то команды в `except`-блоках игнорируются. Если же ошибка возникает, то выполнение `try`-блока прекращается. Если возникла ошибка класса `FileExistsError`, то выполняется первый `except`-блок. Если возникла ошибка иного типа, то выполняется второй `except`-блок.

Ниже показано, как может выглядеть результат выполнения программы в случае, если файл успешно копируется.

Результат выполнения программы (из листинга 7.14)

```
Начинается копирование файла
Копирование прошло успешно
Программа завершила выполнение
```

Если в конечном каталоге на момент запуска программы уже существует файл `bear.jpg`, то результат выполнения программы будет таким.

Результат выполнения программы (из листинга 7.14)

```
Начинается копирование файла
Ошибка: такой файл уже существует
Программа завершила выполнение
```

Если же ошибка, например, связана с тем, что копируемый файл отсутствует, то результат выполнения программы будет следующим.

Результат выполнения программы (из листинга 7.14)

```
Начинается копирование файла
Ошибка доступа к файлу
Программа завершила выполнение
```

Вообще система обработки исключений — очень эффективный механизм, который успешно используется в том числе и при работе с файлами, и пренебрегать им не следует.

НА ЗАМЕТКУ

Напомним, что первое знакомство с обработкой исключений состоялось во второй главе. Более подробно обработка исключений обсуждается в одной из следующих глав.

Резюме

Это великая победа дедуктивного метода.

Из к/ф «Гостья из будущего»

- Целые числа реализуются как значения типа `int`. Действительные числа реализуются как значения типа `float`. Для работы с комплексными числами предназначен тип `complex`. Помимо этого для выполнения операций с рациональными дробями используют тип `Fraction` из модуля `fractions`. Для выполнения операций с действительными числами фиксированной точности используют тип `Decimal` из модуля `decimal`.
- Целочисленные значения можно задавать, кроме десятичной системы, в двоичной (литералы начинаются с префикса `0b` или `0B`), восьмеричной (литералы начинаются с префикса `0o` или `0O`) и шестнадцатеричной (литералы начинаются с префикса `0x` или `0X`). Также есть возможность использовать системы счисления со значением основания до `36` включительно.
- При вводе действительных чисел их можно задавать в явном виде (с целой и дробной частью) или в формате с мантиссой и показателем степени (при этом для разделения мантиссы и показателя степени используются символы `e` или `E`).
- Для выполнения операций с числовыми значениями используют арифметические операторы, специальные функции, а также побитовые операторы — для выполнения операций с целыми числами.
- Логические значения (всего два варианта — `True` или `False`) относятся типу `bool`. Однако в тех местах, где должны быть логические

значения (например, в условном операторе и операторе цикла) можно использовать и объекты других типов: ненулевые числовые значения интерпретируется как `True`, а нулевые — как `False`. Пустые строки и последовательности интерпретируются как значения `False`, а непустые — как `True`. С логическими (и не только) значениями можно использовать логические операторы `and`, `or` и `not`. Соответствующие выражения вычисляются по определенным правилам.

- Для работы с датой и временем существует несколько модулей (`datetime`, `time`, `calendar`) с соответствующими утилитами. В модуле `datetime` реализованы классы `datetime`, `date`, `time`, `timedelta`, которые используются в процессе работы с датой и временем.
- Для считывания информации из файлов и записи информации в файл создается специальный объект файла. В этом случае используется функция `open()`. При создании объекта файла указывается полный путь к файлу и определяется режим доступа к файлу. Считывание данных из файла и запись данных в файл выполняется с помощью специальных методов (в том числе методов `read()`, `readline()`, `write()` и `writeline()`). По завершении работы с файлом он закрывается (с помощью метода `close()`).

Задания для самостоятельной работы

Отрицательный результат — это тоже результат.

Из к/ф «Гостья из будущего»

1. Напишите программу, в которой пользователь вводит основание для системы счисления и число (в десятичной системе), а программа отображает это число в соответствующей системе счисления.

2. Напишите программу, в которой пользователь вводит целое число и номер бита, значение которого нужно определить в программе.

3. Напишите программу, в которой пользователь вводит целое число, а программа отображает сумму значений всех битов в бинарном представлении этого числа.

4. Напишите программу, в которой пользователь вводит целое число, а программа переводит его в восьмеричную систему, меняет порядок

следования цифр в представлении числа и результат отображает в области вывода.

5. Напишите программу, в которой пользователь вводит две рациональные дроби, а программа вычисляет сумму, произведение, разность и частное этих дробей, среди полученных значений находит наибольшее и наименьшее и отображает результат вычислений.

6. Напишите программу, в которой пользователь вводит два комплексных числа, а программа вычисляет сумму, разность, произведение и частное этих чисел. Среди полученных значений необходимо определить то, у которого наибольший и наименьший модуль.

7. Напишите программу, в которой пользователь вводит две даты, а программа определяет количество полных дней между этими датами.

8. Напишите программу, в которой пользователь вводит момент времени, а программа определяет интервал между текущим моментом и моментом времени, который указал пользователь.

9. Напишите программу, в которой пользователь вводит имя текстового файла, а программа отображает содержимое этого файла, а также создает копию этого файла с пронумерованными строками.

10. Напишите программу, в которой создается текстовый файл. Имя файла вводится пользователем. Текст для файла вводится пользователем. При записи текста в файл все маленькие буквы заменяются на большие.

Глава 8
КЛАССЫ И ОБЪЕКТЫ

История, леденящая кровь. Под маской овцы скрывался лев!

Из к/ф «Покровские ворота»

В этой главе мы начинаем знакомство с принципами *объектно-ориентированного программирования* (сокращенно ООП). В основе этого подхода лежит использование *классов* и *объектов*. Поэтому в данной главе мы фактически будем обсуждать методы и подходы, посредством которых классы и объекты реализуются в языке Python.

НА ЗАМЕТКУ

По сравнению с такими «китами», как языки C++, C# и Java, реализация классов и объектов в Python выполняется довольно своеобразно. То есть парадигма объектно-ориентированного программирования нетривиальна сама по себе, а в Python она нетривиальна в особенности.

Концепция классов и объектов

Мастера не мудрствуют.

Из к/ф «Покровские ворота»

Начнем с понятия объекта — пожалуй, главного понятия в ООП. До этого мы в основном имели дело с переменными (хотя объекты на самом деле тоже встречались часто). По большому счету, переменная предоставляет нам доступ к определенным данным, или к определенному значению. Кроме этого, для работы с переменными мы использовали функции. Функция, как мы знаем, это определенный набор команд, которые можно выполнять, вызвав функцию. При вызове функции в качестве

аргументов ей можно передавать переменные. Таком образом, функция при вызове может обрабатывать данные, которые «спрятаны» в переменных. В этой схеме важно то, что у нас есть отдельно данные, и отдельно есть код для обработки данных. В самой программе мы «соединяем» данные и код, который их обрабатывает. В этом нет ничего плохого, но такой подход работает, если программа не очень большая и данных для обработки не очень много. Можно провести аналогию с кухней в ресторане с одним общим залом, в котором много холодильников для хранения продуктов (аналог данных), несколько плит и много поваров (аналог функций). Там есть шеф-повар (аналог программы), который получает заказы и раздает задания поварам. Если кухня не очень большая, то все это чудесно функционирует. Но если увеличивать размеры кухни, количество персонала и количество холодильников и плит, то в какой-то момент может наступить хаос. Одному шеф-повару будет крайне сложно уследить за протекающими процессами. Неизбежно возникнут конфликты из-за доступа к холодильникам, нестыковки в приготовлении блюд и много других мелких неприятностей. Причем это не проблема профессиональной подготовки поваров или функциональности оборудования. Это проблема организации производства. Ведь шеф-повар, даже если он очень профессиональный, все же человек, а человеку свойственно ошибаться. И при увеличении масштабов «производства» риск ошибки растет.

С программами такая же ситуация. Хоть они и выполняются компьютером, но пишут их люди. Если данных много и функций для их обработки тоже много, то отследить логику выполнения программы может быть весьма проблематично. Поэтому нужны альтернативные подходы в организации программы. Какие? Если вернуться к аналогии с кухней, то можно было бы организовать несколько отделов, каждый из них специализировался бы на каких-то определенных блюдах: например, кондитерский отдел, или мясной отдел. У каждого такого отдела был бы свой набор холодильников, свои плиты для готовки блюд, собственный персонал и собственный шеф, контролирующий работу только своего отдела. Тогда шеф-повару достаточно было бы раздавать задания шефам отделов, а уже они определяли бы задания для своих подчиненных. Опять же, автоматически снимается проблема некорректного доступа к ресурсам вроде холодильников и плит. То есть имеются свои преимущества, хотя организационная структура в этом случае более сложная и затратная. Но общий эффект — положительный.

Отделы, описанные выше, являются аналогами объектов. Объект — это набор данных (в общем случае разного типа), а также функции,

предназначенные для их обработки. Подобно тому, как процесс приготовления еды сводится к совместной работе разных отделов, так работа программы реализуется в виде взаимодействия разных объектов. Поэтому с практической точки зрения все, что нам нужно сделать, это научиться создавать объекты и реализовать их «взаимодействие» в программе.

Если мы решили организовать какой-то отдел, нам понадобится определенное описание того, как он должен выглядеть: что он должен делать, сколько там должно быть человек, какие продукты потребуются для нормального функционирования отдела. Другими словами, нам нужен план организации отдела. Без этого нельзя. Более того, если у нас такой план имеется, то в соответствии с этим планом мы сможем создать несколько однотипных отделов (такое иногда тоже необходимо). Так вот, если отдел — это аналог объекта, то план, на основе которого создается отдел, это аналог класса.

Можно привести и другой пример. Допустим, мы хотим построить дом. При правильном подходе перед тем, как начать строительство, следует разработать план дома. В этом плане должно быть описано, что, как и где размещено, тип материалов, из которых все это будет создаваться, и многое другое. План дома — аналог класса. Но даже если у нас есть план, это еще не означает, что у нас есть дом. Дом нужно строить (в соответствии с планом). Более того, мы, используя один и тот же план, можем создать не один, а много домов — например, целую улицу из однотипных домов. Однотипными они будут потому, что создавались на основе одного и того же плана. Но при этом каждый дом физически уникальный. У каждого будет свой жилец, свой адрес и своя «судьба». Дом, как несложно догадаться, — аналог объекта.

Итог наших рассуждений такой.

- В программе можно использовать объекты, которые объединяют в себе данные разного типа и функции, предназначенные для работы с этими данными.
- Объекты создаются на основе классов. Класс представляет собой шаблон, в соответствии с которым реализуются объекты.

НА ЗАМЕТКУ

Про классы можно думать как про некоторые (довольно специфичные) типы данных. Также важное обстоятельство связано с тем, что классы в Python сами по себе реализуются с помощью объектов. Но этот «тонкий» момент мы обсудим немного позже.

Описание классов и создание объектов

Молчать, неопознанный объект.

Из к/ф «Гостья из будущего»

Для успешной работы с классами и объектами нам необходимо научиться решать две задачи: описывать классы и создавать на их основе объекты. Эти и займемся.

Шаблон описания класса выглядит достаточно просто: после ключевого слова `class` указывается название класса, ставится двоеточие и далее следует блок с собственно описанием класса (как именно он описывается — вопрос отдельный и индивидуальный). Как выглядит общая конструкция с описанием класса, показано ниже.

```
class имя:
    # Описание класса
```

В самом простом случае класс вообще не содержит описания. Если так, то блок описания класса состоит из ключевого слова `pass`.

Для создания объекта на основе класса переменной в качестве значения присваивается инструкция, которая состоит из названия класса и пустых круглых скобок.

> **НА ЗАМЕТКУ**
>
> На самом деле скобки не обязательно пустые, но пока мы рассматриваем самые простые случаи.

Небольшой пример, в котором описывается класс и на его основе создается два объекта, представлен в листинге 8.1.

Листинг 8.1. Знакомство с классами и объектами

```
# Описание класса:
class MyClass:
    pass
# Создание объектов на основе класса:
A=MyClass()
```

```
B=MyClass()
# Объекты:
print("Объект A:", A)
print("Объект B:", B)
# Тип объектов:
print("Класс объекта A:", type(A).__name__)
print("Класс объекта B:", type(B).__name__)
# Сравнение объектов:
print("A==B:", A==B)
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 8.1)

```
Объект A: <__main__.MyClass object at 0x02174FB0>
Объект B: <__main__.MyClass object at 0x021AA890>
Класс объекта A: MyClass
Класс объекта B: MyClass
A==B: False
```

В этой программе мы описываем класс `MyClass` с пустым телом. На основе класса командами `A=MyClass()` и `B=MyClass()` создаются два объекта, ссылки на которые записываются в переменные `A` и `B`.

ПОДРОБНОСТИ

При создании объекта в памяти под него выделяется место. Если инструкция создания объекта (в данном случае это `MyClass()`) присваивается в качестве значения переменной, то такая переменная будет ссылаться на объект: фактически значением переменной является не сам объект, а его адрес (ссылка). При этом если мы обращаемся к переменной, то внешне иллюзия такая, как если бы значением был объект.

При попытке «напечатать» объект получаем сообщение, в котором содержится информация о классе объекта и фактически адрес, по которому объект записан. Также тип объекта можно узнать с помощью функции `type()`. Аргументом функции передается переменная, которая

ссылается на объект. Результатом является объект, определяющий тип значения, на которое ссылается переменная. Если для этого объекта запросить поле `__name__`, получим название класса (в данном случае это `MyClass`).

ПОДРОБНОСТИ

У объектов есть поля и методы (которые также называются атрибутами объекта). Среди полей и методов есть «особые», или специальные, поля и методы. Они начинаются и заканчиваются двойным символом подчеркивания. К таким специальным полям относится и поле `__name__` (два символа подчеркивания, идентификатор `name` и еще два символа подчеркивания).

Показательный момент связан с попыткой сравнения объектов (команда `A==B`). Хотя мы создали два совершенно одинаковых объекта, в результате их сравнения на предмет равенства получаем значение `False`. Причина в том, что в данном случае сравниваются не сами объекты, а значения переменных, которые ссылаются на объекты. А, как отмечалось выше, значениями переменных являются адреса объектов, а не сами объекты. И хотя объекты одинаковые, но адреса у них разные — подобно тому, как два одинаковых дома на одной улице имеют разные адреса.

Поскольку мы в классе `MyClass` ничего не описали, то объекты, которые были созданы на основе класса, не содержат никаких полезных данных и методов для их обработки. От таких объектов пользы мало.

НА ЗАМЕТКУ

Объект можно сравнить с коробкой, в которой сидят котята (это методы), и у них есть игрушки (поля, или данные). Котята могут играть с игрушками (методы обрабатывают данные). Какие в коробке должны быть котята и что они могут делать с игрушками, определяется в классе. Мы для использования объектов использовали пустой класс — в нем ничего не описано. Поэтому объекты, которые мы создали, напоминают пустые коробки — в них нет ни котят, ни игрушек. Хотя, как мы узнаем далее, совсем пустыми эти коробки назвать тоже нельзя.

Поэтому на следующем этапе мы опишем класс с полезными методами, и, как следствие, созданные на основе такого класса объекты будут немного более функциональными, чем в рассмотренном выше примере.

Прежде, чем приступить к рассмотрению примера, сделаем несколько пояснений. Во-первых, метод в классе описывается точно так же, как описывается обычная функция. Принципиальная разница лишь в том, что функция описывается «сама по себе», а метод описывается в теле класса. Формально можно утверждать, что метод — это описанная в теле класса функция. Но к внешним эффектам все не сводится. Если для вызова функции достаточно указать ее имя и передать ей в случае необходимости аргументы, то метод вызывается из объекта с использованием точечного синтаксиса: указывается имя объекта, ставится точка, и затем имя метода с круглыми скобками (в которых могут быть аргументы, предназначенные для передачи методу). Пикантность ситуации в том, что формально при таком вызове метода объект, из которого вызывается метод, на самом деле автоматически передается в качестве аргумента этому методу. Поэтому когда метод описывается в классе, его первый аргумент по умолчанию означает объект, из которого будет вызываться метод. Такой аргумент, в общем-то, можно называть как угодно (в разумных пределах), но общая договоренность состоит в том, что первый аргумент метода называется `self`.

Другими словами, если мы описываем в классе метод, который планируем вызывать из объекта (а на самом деле есть и другие варианты), то в таком методе первый аргумент будет называться `self` и обозначает он объект, из которого метод вызывается. У метода могут быть и другие аргументы — они фактически передаются методу при вызове. Первый аргумент передавать в метод при вызове не нужно — в качестве этого первого аргумента используется объект, из которого вызывается метод.

Во-вторых, у объекта кроме методов могут быть и поля. Поле — это фактически переменная, ассоциированная с объектом. Проще говоря, поле — это переменная, существующая внутри объекта.

> **НА ЗАМЕТКУ**
>
> Если метод — это функция, связанная с объектом, то поле — это переменная, связанная с объектом.

Мы уже знаем, что переменная в Python появляется тогда, когда мы ей присваиваем значение. Нечто похожее происходит и при работе с полями: поле у объекта появляется, когда мы полю присваиваем значение. Где и как это можно сделать? На самом деле вариантов довольно много. Но один из них — присвоить значение полю при вызове метода. Пример, в котором показано, как это можно сделать, представлен в листинге 8.2.

Листинг 8.2. Знакомство с полями и методами

```
# Описание класса:
class MyClass:
    # Метод для присваивания значения полю:
    def set(self, n):
        self.number=n
    # Метод для отображения значения поля:
    def show(self):
        print("Поле number =", self.number)
# Создание объектов:
A=MyClass()
B=MyClass()
# Присваивание значений полям объектов:
A.set(100)
B.set(200)
# Проверка значений полей объектов:
A.show()
B.show()
# Присваивание значений полям объектов:
A.number=123
B.number=321
# Проверка значений поле объектов:
A.show()
B.show()
```

Ниже показано, как будет выглядеть результат выполнения программы.

Результат выполнения программы (из листинга 8.2)

```
Поле number = 100
Поле number = 200
Поле number = 123
Поле number = 321
```

Анализ программы начнем с кода класса `MyClass`, в котором есть два метода. Метод `set()` описан с двумя аргументами (`self` и `n`). Первый аргумент `self` обозначает объект, из которого будет вызываться метод. Второй аргумент `n` соответствует значению, которое будет передаваться методу при вызове. В теле метода выполняется всего одна команда `self.number=n`. Как понимать эту команду? Все просто. Полю `number` объекта, из которого будет вызываться метод, должно быть присвоено такое же значение, как и у переменной `n`, переданной аргументом методу.

У метода `show()` всего один аргумент `self`, отождествляемый с объектом, из которого вызывается метод. При вызове метода выполняется команда `print("Поле number =", self.number)`, отображающая значение поля `number` объекта, из которого вызывается метод.

НА ЗАМЕТКУ

Если придерживаться аналогии о том, что объект — это коробка с котятами и игрушками, то метод `set()` соответствует котенку, которому можно дать в лапы игрушку, а он ее положит в коробку. А метод `show()` соответствует котенку, который может нам эту игрушку показать.

Объекты на основе класса `MyClass` создаются командами `A=MyClass()` и `B=MyClass()`. У каждого из этих объектов есть методы `set()` и `show()`. В результате выполнения команд `A.set(100)` и `B.set(200)` полю `number` объекта `A` присваивается значение `100`, а полю `number` объекта `B` присваивается значение `200`.

ПОДРОБНОСТИ

При выполнении команды `A.set(100)` код метода `set()` выполняется с использованием вместо аргумента `self` ссылки на объект из переменной `A`, а вместо аргумента `n` используется значение `100`. При выполнении команды `B.set(200)` аргумент `self` заменяется на `B`, а аргумент `n` заменяется на `200`.

НА ЗАМЕТКУ

Если быть более точным, то при первом вызове метода `set()` поле `number` соответствующего объекта не просто получает значение, но и создается.

Для проверки значений полей объектов используем команды `A.show()` и `B.show()`. Стоит отметить, что обращаться к полям объектов можно «напрямую», без вызова методов. Например, вполне законными являются команды `A.number=123` и `B.number=321`, которыми полям `number` объектов `A` и `B` присваиваются новые значение.

НА ЗАМЕТКУ

Как и в случае с методами, обращение к полям объекта выполняется с использованием точечного синтаксиса: указывается объект, точка и название поля. Так, инструкция `A.number` означает, что речь идет о поле `number` объекта `A`, а инструкция `B.number` означает обращение к полю `number` объекта `B`. При этом важно понимать, что хотя поля имеют одинаковые названия, но это разные переменные, поскольку относятся они к разным объектам.

Хочется еще раз подчеркнуть, что поле у объекта появляется только после того, как мы ему первый раз присваиваем значение. Небольшая иллюстрация к сказанному представлена в листинге 8.3.

Листинг 8.3. Явное создание поля объекта

```
# Описание класса:
class MyClass:
    def set(self, n):
        self.number=n
    def show(self):
        print("Поле number =", self.number)
# Создание объекта:
obj=MyClass()
# Проверка наличия поля:
print("Наличие поля number:", hasattr(obj,"number"))
try:
    # Проверка значения поля:
    obj.show()
except AttributeError:
    print("Поля number у объекта нет!")
# Присваивание значения полю:
```

```
obj.number=123
# Проверка наличия поля:
print("Наличие поля number:", hasattr(obj,"number"))
# Проверка значения поля:
obj.show()
# Новое значение поля:
obj.set(321)
# Проверка значения поля:
obj.show()
```

Результат выполнения программы будет следующим.

Результат выполнения программы (из листинга 8.3)

```
Наличие поля number: False
Поля number у объекта нет!
Наличие поля number: True
Поле number = 123
Поле number = 321
```

Класс `MyClass` мы используем точно такой, как и в предыдущем примере. Изменились операции, выполняемые после создания объекта `obj` (использована команда `obj=MyClass()`). Собственно создание объекта не означает, что у него есть поле `number`. Для проверки наличия у объекта такого поля используем инструкцию `hasattr(obj,"number")`, в которой первым аргументом функции `hasattr()` передается ссылка на проверяемый на наличие поля объект, а вторым аргументом функции передается текст с названием поля. Если бы у объекта `obj` было такое поле, мы получили бы значение `True`. Значение `False` означает, что такого поля у объекта нет.

НА ЗАМЕТКУ

С помощью функции `hasattr()` можно проверять объект на наличие не только полей, но и методов. Принцип тот же: первым аргументом функции передается ссылка на объект, а вторым аргументом передается текст с названием метода.

Вызов метода `show()` из объекта `obj` подтверждает отсутствие у объекта поля `number`: при выполнении команды `obj.show()` генерируется исключение класса `AttributeError` (у объекта отсутствует запрашиваемый атрибут). Но поскольку команда размещена в `try`-блоке, то ошибка перехватывается и обрабатывается в `except`-блоке (в результате выполняется команда `print("Поля number у объекта нет!")`).

Ситуация меняется после выполнения команды `obj.number=123`, которой не просто полю `number` присваивается значение. Данной командой поле фактически создается. Проверка наличия поля с помощью инструкции `hasattr(obj,"number")` теперь дает значение `True`. Без проблем выполняется и команда `obj.show()`. Как и раньше, значение полю можно присвоить также с помощью метода `set()` (как это делается командой `obj.set(321)`).

Конструкторы и деструкторы

> — Какая гадость.
> — Это не гадость. Это последние достижения современной науки.
>
> *Из к/ф «31 июня»*

В классах можно определять *специальные методы*. Их довольно много, и каждый имеет особое назначение. Узнать специальный метод просто: название специального метода начинается и заканчивается с двойного символа подчеркивания. Сейчас мы познакомимся с двумя такими методами: *конструктором* и *деструктором*. Конструктор автоматически вызывается при создании объекта. Его удобно использовать, когда при создании объекта необходимо сразу выполнить определенные настройки или задать значения для полей объекта. Деструктор автоматически вызывается при удалении объекта из памяти. Конструктор — это метод `__init__()`. Деструктор — метод `__del__()`. Как данные методы можно использовать на практике, показано в программе в листинге 8.4.

Листинг 8.4. Знакомство с конструктором и деструктором

```
# Описание класса:
class MyClass:
    # Конструктор:
    def __init__(self, n="Белый"):
```

```
        self.name=n;
        print("Создан объект:", self.name)
    # Деструктор:
    def __del__(self):
        print("Удален объект:", self.name)
# Функция:
def create(n):
    obj=MyClass(n)
# Создание объектов:
A=MyClass()
B=MyClass("Красный")
C=MyClass("Синий")
# Вызов функции:
create("Желтый")
# Полю присваивается новое значение:
C.name="Зеленый"
# Удаление объектов:
del A
del B
del C
```

В результате выполнения программы получаем следующее.

Результат выполнения программы (из листинга 8.4)

```
Создан объект: Белый
Создан объект: Красный
Создан объект: Синий
Создан объект: Желтый
Удален объект: Желтый
Удален объект: Белый
Удален объект: Красный
Удален объект: Зеленый
```

Класс `MyClass` содержит описание конструктора и деструктора. У конструктора два аргумента, причем второй имеет значение по умолчанию. В теле конструктора командой `self.name=n` полю `name` объекта, из которого вызывается метод (в данном случае это объект, который создается) присваивается значение второго аргумента конструктора. После этого командой `print("Создан объект:", self.name)` отображается сообщение о создании объекта, а также отображается значение поля `name` этого объекта.

ПОДРОБНОСТИ

Конструктор автоматически вызывается при выполнении команды, которой создается объект. При этом следует исходить из того, что конструктор вызывается из объекта (который создается). Проблемы здесь нет — сначала создается объект, и затем сразу из уже созданного объекта вызывается конструктор. Как бы там ни было, но первый аргумент конструктора — это ссылка на объект, из которого вызывается конструктор. Все остальные аргументы, указанные в конструкторе, передаются конструктору в явном виде в команде, которой создается объект. Эти аргументы указываются в круглых скобках после названия класса.

В теле деструктора выполняется всего одна команда `print("Удален объект:", self.name)`, которой отображается сообщение об удалении объекта (с указанием значения поля `name` удаляемого объекта).

ПОДРОБНОСТИ

Деструктор также вызывается автоматически непосредственно перед удалением объекта. Деструктор вызывается из объекта, предназначенного для удаления, поэтому у деструктора один аргумент, обозначающий удаляемый объект.

Кроме класса, в программе описана функция `create()` с одним аргументом (мы предполагаем, что это текст). В теле функции командой `obj=MyClass(n)` создается объект класса `MyClass`, причем в качестве аргумента конструктору передается то значение, которое было передано функции при вызове.

Мы последовательно создаем три объекта. В команде `A=MyClass()` конструктору аргументы не передаются, поэтому в качестве второго аргумента конструктор используется значение по умолчанию `"Белый"`.

При создании этого объекта отображается сообщение, в котором поле `name` объекта указано как такое, что имеет значение `"Белый"`.

В командах `B=MyClass("Красный")` и `C=MyClass("Синий")` конструктору передается аргумент, определяющий значение поля `name` соответствующего объекта. При выполнении этих команд появляются сообщения о создании объектов.

При выполнении команды `create("Желтый")` сначала появляется сообщение о создании объекта с полем `"Желтый"`, а затем — сообщение об удалении этого объекта. Причина появления первого из этих сообщений понятна: при вызове функции `create()` создается объект класса `MyClass`. Но это объект локальный. Он существует, пока выполняется функция. Когда функция завершает выполнение, объект удаляется из памяти. Деструктор, который при этом вызывается, отображает сообщение об удалении объекта.

Командой `C.name="Зеленый"` полю `name` объекта `C` присваивается новое значение (именно оно будет отображаться при удалении объекта). Для удаления объектов использованы инструкции `del A`, `del B` и `del C`. При выполнении каждой из них вызывается деструктор, в результате чего в области вывода отображаются сообщения об удалении объектов.

Объект реализации класса

> Хотите обмануть мага? Боже, какая детская непосредственность. Я же вижу вас насквозь.
>
> *Из к/ф «31 июня»*

Прежде чем продолжить наше знакомство с принципами ООП и подходами, используемыми при работе с объектами, нам необходимо внести ясность в один принципиальный вопрос. Глобально касается он того, как объект соотносится с классом, на основе которого он создан. И здесь не все так просто, как может показаться на первый взгляд.

Мы уже знаем, что класс — это шаблон, на основе которого создается объект. Но истина в том, что сам класс реализуется с помощью объекта. Чтобы не путать этот объект с объектами, которые создаются на основе класса, будем называть его *объектом реализации класса* (или объектом, посредством которого реализуется класс).

НА ЗАМЕТКУ

Если вернуться к аналогии с домами (объекты), которые создавались на основе плана (класс), то сам план должен быть как-то записан. Допустим, это чертеж, выполненный на ватмане. Тогда такой ватман с чертежом — аналог объекта, на основе которого реализуется класс. Здесь важно понимать, что дом, который создается на основе плана, и ватман, на котором содержится план, — совершенно разные объекты. Точно так же следует отличать объекты, которые создаются на основе класса, и объект, с помощью которого реализован класс. Это разные объекты и по своей природе, и по своим характеристикам.

Вообще, в этом нет ничего удивительного. Например, функции реализуются как объекты, а название функции является ссылкой на такой объект. Аналогично, класс реализуется на основе некоторого объекта, а название класса содержит ссылку на этот объект. И чтобы легче было это понять, рассмотрим небольшой пример, представленный в листинге 8.5.

Листинг 8.5. Объект реализации класса

```
# Первый класс:
class Alpha:
    pass
# Второй класс:
class Bravo:
    pass
# Переменной присваивается имя класса:
MyClass=Alpha
# Создание объекта:
A=MyClass()
# Переменной присваивается имя класса:
MyClass=Bravo
# Создание объекта:
B=MyClass()
# Присваивание классов:
Alpha=Bravo
# Создание объекта:
```

```
C=Alpha()
# Получение ссылки на объект реализации класса:
MyClass=A.__class__
# Создание объекта:
D=MyClass()
# Проверка типа объектов:
print("Объект A:", type(A).__name__)
print("Объект B:", type(B).__name__)
print("Объект C:", type(C).__name__)
print("Объект D:", type(D).__name__)
# Изменение названия классов:
MyClass.__name__="First"
Bravo.__name__="Second"
# Проверка типа объектов:
print("Объект A:", type(A).__name__)
print("Объект B:", type(B).__name__)
print("Объект C:", type(C).__name__)
print("Объект D:", type(D).__name__)
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 8.5)

```
Объект A: Alpha
Объект B: Bravo
Объект C: Bravo
Объект D: Alpha
Объект A: First
Объект B: Second
Объект C: Second
Объект D: First
```

Код простой, но результат его выполнения требует пояснений. Чем и займемся. Итак, в программе описано два класса `Alpha` и `Bravo`. Классы

«пустые» — описание каждого из них состоит из ключевого слова pass. Дальше начинается сюрприз: выполняется команда MyClass=Alpha, которой переменной MyClass в качестве значения присваивается имя класса. Здесь как раз уместно вспомнить, что класс реализуется через объект, а имя класса содержит ссылку на этот объект. Поэтому в результате выполнения команды MyClass=Alpha переменная MyClass будет ссылаться на тот же объект, на который ссылается идентификатор Alpha. Поэтому когда выполняется команда A=MyClass(), то на самом деле создается объект класса Alpha, и ссылка на этот объект записывается в переменную A. Затем выполняется команда MyClass=Bravo. В результате переменная MyClass ссылается на объект, посредством которого реализован класс Bravo. Как следствие, выполнение команды B=MyClass() приводит к созданию объекта класса Bravo, а ссылка на этот объект записывается в переменную B. Но после этого ситуация усугубляется, поскольку выполняется команда Alpha=Bravo. Что происходит? Теперь идентификатор Alpha (а на самом деле это переменная) будет ссылаться на объект, через который реализован класс Bravo. Поэтому при выполнении команды C=Alpha() создается не объект класса Alpha, а объект класса Bravo. А что же с объектом, на который раньше ссылался идентификатор Alpha? Он не пропал. Просто переменная Alpha на него больше не ссылается. Но сам объект существует. К нему можно получить доступ. Для этого обращаемся к специальному полю __class__ (в начале и в конце по два символа подчеркивания) объекта A, который создавался самым первым на основе класса Alpha. Значение поля — ссылка на объект, посредством которого был реализован класс, послуживший, в свою очередь, основой для создания объекта A. Понятно, что это класс Alpha. Таким образом, после выполнения команды MyClass=A.__class__ переменная MyClass ссылается на объект, через который реализуется класс Alpha (хотя одноименная переменная на этот класс уже не ссылается). Создавая объект командой D=MyClass(), мы получаем объект класса Alpha. Для определения классов, на основе которых созданы объекты A, B, C и D используем функцию type() и поле __name__.

В этой истории остался один нюанс: переменная Alpha ссылается на объект, через который реализуется класс Bravo, а исходный класс, который ассоциировался с переменной Alpha, свое название не изменил — во всяком случае, когда мы проверяем это название с помощью функции type() и поля __name__. У сложившейся странной, на первый взгляд, ситуации в действительности простое объяснение. Дело в том, что, когда интерпретатор обрабатывает блок с описанием класса,

создается объект, через который этот класс реализуется. Ссылка на объект записывается в идентификатор, указанный в качестве названия класса. И у этого объекта есть поле `__name__`. Это поле фактически и содержит название класса. По умолчанию в поле заносится текст с названием идентификатора, указанного в качестве имени класса. Для класса `Alpha` это `"Alpha"`, а для класса `Bravo` это `"Bravo"`. Эти названия хранятся в объектах, через которые реализуются классы, и какие бы манипуляции мы ни выполняли с идентификаторами `Alpha` и `Bravo`, значения полей `__name__` в соответствующих объектах не меняется. Но, в принципе, такие изменения можно внести (другой вопрос — зачем это делать). Так, в программе использованы команды `MyClass.__name__="First"` и `Bravo.__name__="Second"`. В первом случае полю `__name__` объекта, на который ссылается переменная `MyClass`, присваивается значение `"First"`. Но это объект, через который реализовался класс `Alpha`. Поэтому все объекты, которые были созданы на основе этого класса, теперь станут объектами класса `First`. Аналогично, полю `__name__` объекта, на который ссылается переменная `Bravo`, присвоено значение `"Second"`. Речь об объекте, через который реализуется класс `Bravo`. И хотя переменная `Bravo` по-прежнему ссылается на этот объект, класс меняет название на `Second`. Проверка типа объектов `A`, `B`, `C` и `D` подтверждает наши выводы: объекты класса `Alpha` стали объектами класса `First`, а объекты класса `Bravo` стали объектами класса `Second`.

НА ЗАМЕТКУ

Особо стоит подчеркнуть, что это исключительно «декоративные» изменения, связанные с изменением значения поля `__name__` объекта, через который реализуется класс. По сути, класс остался тем же самым. Просто мы изменили свойства объекта, посредством которого он реализован.

Тот факт, что класс сам реализуется с помощью объекта, наводит на некоторые размышления. В частности, возникает вопрос о том, можно ли задавать поля и методы для объекта собственно класса. И понятно, что если бы ответ был отрицательным, то вопрос этот мы бы не поднимали. Мы сразу приступим к рассмотрению примера, представленного в листинге 8.6.

Листинг 8.6. Поля и методы класса

```
# Описание класса:
class MyClass:
```

```
    # Поле класса:
    color="Красный"
    # Методы класса:
    def set(txt):
        MyClass.color=txt
    def show():
        print(MyClass.color)
# Вызов методов класса:
MyClass.show()
MyClass.set("Зеленый")
# Отображение значения поля класса:
print(MyClass.color)
# Новое значение для поля класса:
MyClass.color="Синий"
# Вызов метода класса:
MyClass.show()
# Создание объектов класса:
A=MyClass()
B=MyClass()
# Проверка значения поля:
print("Класс:", MyClass.color)
print("Объект A:", A.color)
print("Объект B:", B.color)
# Присваивание значения полю:
A.color="Белый"
# Проверка значения поля:
print("Класс:", MyClass.color)
print("Объект A:", A.color)
print("Объект B:", B.color)
# Присваивание значения полю:
MyClass.color="Желтый"
# Проверка значения поля:
```

```
print("Класс:", MyClass.color)
print("Объект A:", A.color)
print("Объект B:", B.color)
# Удаление поля из объекта A:
del A.color
# Проверка значения поля:
print("Класс:", MyClass.color)
print("Объект A:", A.color)
print("Объект B:", B.color)
```

В результате выполнения программы в области вывода появляются следующие сообщения.

Результат выполнения программы (из листинга 8.6)

```
Красный
Зеленый
Синий
Класс: Синий
Объект A: Синий
Объект B: Синий
Класс: Синий
Объект A: Белый
Объект B: Синий
Класс: Желтый
Объект A: Белый
Объект B: Желтый
Класс: Желтый
Объект A: Желтый
Объект B: Желтый
```

На этот раз класс `MyClass` описан немного необычно. А именно, в теле класса инструкцией `color="Красный"` создается поле с названием `color` и значением `"Красный"`. Это поле, которое относится непосредственно к классу, а более точно — к объекту, посредством которого

реализуется класс. Также мы описываем два метода. У метода `set()` один аргумент, который называется `txt`. В теле метода выполняется команда `MyClass.color=txt`, которой значение аргумента присваивается полю `color`. Здесь достойны внимания два момента. Во-первых, поскольку речь идет о поле класса, то при обращении к такому полю указывается имя класса, а затем, через точку — название поля. Во-вторых, мы описываем метод `set()` как метод класса — этот метод будет вызываться из класса, подобно тому, как выполняется обращение к полю класса `color`. При вызове метода мы будем указывать название класса и, через точку, название метода. В данном случае метод является обычной функцией, описанной в теле класса. Поэтому нет необходимости первым аргументом в описании метода указывать ссылку на объект, как мы это делали ранее. Единственный аргумент, описанный в методе `set()`, передается методу при вызове (мы предполагаем, что это текстовое значение).

Метод `show()` также является методом класса. Аргументов у метода нет. При вызове метода выполняется команда `print(MyClass.color)`, которой отображается значение поля `color`.

Программа содержит команды, которыми вызываются методы класса. Так, командой `MyClass.show()` из класса `MyClass` вызывается метод `show()`. В результате в области вывода отображается текущее значение поля `color` (на момент вызова метода это `"Красный"`). В результате выполнения команды `MyClass.set("Зеленый")` поле `color` получает значение `"Зеленый"`. Убеждаемся в этом с помощью команды `print(MyClass.color)`, в которой выполняется обращение к полю `color` напрямую, без вызова метода `show()`. Подобным образом можно не только прочитать значение поля, но и присвоить значение полю. Примером служит команда `MyClass.color="Синий"`. При выполнении команды `MyClass.show()` в области вывода отображается текущее значение `"Синий"` для поля `color`.

Важная часть программы связана с использованием объектов, которые создаются командами `A=MyClass()` и `B=MyClass()`. К полю `color` можно обращаться не только через класс `MyClass` (инструкция `MyClass.color`), но и через объекты `A` и `B` (инструкции `A.color` и `B.color` соответственно). Во всех трех случаях, как показывает проверка, получаем значение `"Синий"`. Но после выполнения команды `A.color="Белый"`, которой полю `color` объекта `A` присваивается значение `"Белый"`, значение выражения `A.color` равно `"Белый"`, а для выражений `MyClass.color` и `B.color` получаем значение `"Синий"`. Более того, после

выполнения команды MyClass.color="Желтый" значение выражения A.color по-прежнему равно "Белый", а для выражений MyClass.color и B.color получаем значение "Желтый". Далее, после удаления командой del A.color поля color из объекта A проверка показывает, что все три выражения (MyClass.color, A.color и B.color) дают одно и то же значение "Желтый". В чем причина? Ответ не очень сложный. Связан он с тем, что и объект, посредством которого реализуется класс, и объекты, которые создаются на основе класса, технически реализуются в виде словарей. Атрибуты (названия полей и методов) служат ключами элементов словаря, а значения этих атрибутов, соответственно, значения элементов словаря. Если мы обращаемся к какому-то атрибуту объекта, созданного на основе класса, то поиск этого атрибута сначала осуществляется в словаре, через который реализован объект. Если такого атрибута нет, то поиск атрибута продолжается в словаре того объекта, посредством которого реализован класс.

Перенося все сказанное на наш пример, получаем такую картину. В самом начале у объектов A и B своего поля color нет. Поэтому когда мы запрашиваем значение поля через эти объекты, то поиск поля сначала выполняется в соответствующем объекте, а затем — в классе MyClass. А там поле имеет значение "Синий". При обращении к полю через класс MyClass поиск поля сразу начинается в классе (то есть в объекте, посредством которого реализован класс). Ситуация меняется после того, как полю color объекта A впервые присваивается значение. В этом случае у объекта A появляется собственное поле color. Когда мы проверяем значение поля color объекта A, то поиск начинается с объекта A, и поскольку поле найдено, то на этом все и заканчивается: получаем значение "Белый", которое было присвоено полю.

Когда присваивается значение "Желтый" полю color класса MyClass, то на значении поля color объекта A это обстоятельство никак не сказывается. Но если мы запрашиваем значение поля color объекта B, то поскольку собственного поля color у объекта B нет, будет возвращаться значение поля color класса MyClass. Легко понять, что после удаления поля color из объекта A при обращении к этому полю в формате A.color поиск начнется в объекте A и затем продолжится в классе MyClass. В результате возвращается значение поля из класса.

Еще один важный момент, который следует затронуть, связан с методами, описанными в классе. Мы описывали методы, которые вызывались из объектов класса, и методы, которые вызывались из класса. И если

смотреть в корень, то легко заметить, что принципиальной разницы в описании таких методов нет. Действительно, в описании методов, предназначенных для вызова из объектов класса, первый аргумент обозначает объект, из которого будет вызываться метод. Но на самом деле мы можем описать метод класса с первым аргументом, который будет обозначать некоторый объект этого класса. Формально такие методы неразличимы. И эта неразличимость имеет последствия. Рассмотрим пример, представленный в листинге 8.7.

Листинг 8.7. Методы класса и объекта

```
# Описание класса:
class MyClass:
    # Конструктор:
    def __init__(self):
        self.value=123
        print("Создается объект:", self.value)
    # Деструктор:
    def __del__(self):
        print("Удаляется объект:", self.value)
    # Метод для присваивания значения полю:
    def set(self, n):
        self.value=n
    # Метод для отображения значения поля:
    def show(self):
        print("Поле объекта:", self.value)
# Создание объекта:
obj=MyClass()
# Вызов методов из объекта:
obj.show()
obj.set(100)
# Вызов методов из класса:
MyClass.show(obj)
MyClass.set(obj,200)
# Проверка значения поля:
```

```
obj.show()
# Явный вызов конструктора:
MyClass.__init__(obj)
# Явный вызов деструктора:
MyClass.__del__(obj)
# Проверка значения поля:
obj.show()
# Изменение значения поля:
obj.value=321
obj.show()
# Явный вызов конструктора через объект:
obj.__init__()
# Явный вызов деструктора через объект:
obj.__del__()
# Проверка значения поля:
obj.show()
```

Результат выполнения программы представлены ниже.

Результат выполнения программы (из листинга 8.7)

```
Создается объект: 123
Поле объекта: 123
Поле объекта: 100
Поле объекта: 200
Создается объект: 123
Удаляется объект: 123
Поле объекта: 123
Поле объекта: 321
Создается объект: 123
Удаляется объект: 123
Поле объекта: 123
```

Проанализируем код примера. Там описывается класс `MyClass`, и в этом классе есть конструктор, деструктор и два метода `set()` и `show()`, предназначенных соответственно для присваивания значения полю `value` объекта класса (из которого вызывается метод) и отображения значения поля. При вызове конструктора полю `value` присваивается значение `123` и отображается сообщение о создании объекта (и значения поля `value` объекта). При вызове деструктора отображается сообщение об удалении объекта и значение поля `value` объекта. Далее вся эта конструкция тестируется с помощью различных команд. В первую очередь на основе класса `MyClass` командой `obj=MyClass()` создается объект `obj`. Команды `obj.show()` и `obj.set(100)` в этом случае представляются понятными и знакомыми. Сюрпризом могут быть команды `MyClass.show(obj)` и `MyClass.set(obj,200)`. На самом деле это эквивалент команд `obj.show()` и `obj.set(200)`. Но методы вызываются не как методы объекта, а как методы класса. В таком случае первым аргументом передается явная ссылка на объект `obj`. Причем такой же «трюк» можно проделывать не только с обычными методами, но и со специальными — такими, например, как конструктор или деструктор. Более того, конструктор и деструктор можно вызывать как через класс (команды `MyClass.__init__(obj)` и `MyClass.__del__(obj)`) так и через объект (команды `obj.__init__()` и `obj.__del__()`).

НА ЗАМЕТКУ

Особо следует подчеркнуть, что сам по себе вызов конструктора не приводит к созданию объекта, равно как и вызов деструктора не приводит к удалению объекта. Конструктор — это метод, который автоматически вызывается сразу после того, как объект создан. А деструктор — метод, который автоматически вызывается непосредственно перед удалением объекта. Если мы сами явно вызываем конструктор или деструктор, то просто выполняется код этих методов, не более того.

Операции с атрибутами классов и объектов

Не мешайте работать, инвентаризуемый.

Из к/ф «Гостья из будущего»

Рассмотренные выше примеры и описанная схема «связи» между объектами класса и самим классом наводит на некоторые размышления. Еще раз подчеркнем, что связь между объектом и классом, на основе которого

объект создан, устанавливается на уровне последовательности поиска атрибутов: сначала запрашиваемый атрибут ищется в объекте, а если он там не найден — то в классе (объекте, на основе которого реализован класс). При этом и объекты, и классы технически хранятся в виде словарей. В словарь элементы можно добавлять, и можно удалять элементы из словаря. Если к тому же учесть, что методы представляют собой ссылки на объекты, через которые реализуются функции, то перспективы открываются впечатляющие. Далее мы рассмотрим некоторые манипуляции, допустимые при работе с атрибутами классов и объектов.

Сразу отметим, что получить доступ к словарю, посредством которого реализован объект, можно с помощью поля `__dict__` (по два подчеркивания в начале и конце). Если поле запрашивается для класса, то получаем словарь для объекта, посредством которого реализуется класс. Если поле запрашивается для объекта, то результатом получаем словарь, посредством которого реализован данный объект. Рассмотрим программу, представленную в листинге 8.8.

Листинг 8.8. Атрибуты классов и объектов

```
# Первый класс:
class Alpha:
    pass
# Второй класс:
class Bravo:
    name="Bravo"
    def display():
        print("Поле name:", Bravo.name)
    def show(self):
        print("Поле value:", self.value)
    def __init__(self):
        self.value=123
# Создание объектов:
A=Alpha()
B=Bravo()
# Атрибуты первого класса:
print("Класс Alpha")
```

```
n=1
for s in Alpha.__dict__:
    print("["+str(n)+"] "+s+":", Alpha.__dict__[s])
    n+=1
# Атрибуты второго класса:
print("Класс Bravo")
n=1
for s in Bravo.__dict__:
    print("["+str(n)+"] "+s+":", Bravo.__dict__[s])
    n+=1
# Атрибуты объектов:
print("Объект A:", A.__dict__)
print("Объект B:", B.__dict__)
# Вызов метода класса:
Bravo.display()
# Создание атрибута класса:
Alpha.display=Bravo.display
# Удаление атрибута класса:
del Bravo.display
# Вызов метода из объекта:
B.show()
# Создание атрибута класса:
A.color="Красный"
# Создание атрибута объекта:
B.show=lambda: print("Объект B:", B.value)
# Атрибуты первого класса:
print("Класс Alpha")
n=1
for s in Alpha.__dict__:
    print("["+str(n)+"] "+s+":", Alpha.__dict__[s])
    n+=1
# Атрибуты второго класса:
```

```
print("Класс Bravo")
n=1
for s in Bravo.__dict__:
    print("["+str(n)+"] "+s+":", Bravo.__dict__[s])
    n+=1
# Атрибуты объектов:
print("Объект A:", A.__dict__)
print("Объект B:", B.__dict__)
# Вызов методов:
Alpha.display()
Bravo.show(B)
B.show()
```

Ниже показано, каким будет результат выполнения программы.

Результат выполнения программы (из листинга 8.8)

```
Класс Alpha
[1] __module__: __main__
[2] __dict__: <attribute '__dict__' of 'Alpha' objects>
[3] __weakref__: <attribute '__weakref__' of 'Alpha' objects>
[4] __doc__: None
Класс Bravo
[1] __module__: __main__
[2] name: Bravo
[3] display: <function Bravo.display at 0x02646078>
[4] show: <function Bravo.show at 0x026460C0>
[5] __init__: <function Bravo.__init__ at 0x02646108>
[6] __dict__: <attribute '__dict__' of 'Bravo' objects>
[7] __weakref__: <attribute '__weakref__' of 'Bravo' objects>
[8] __doc__: None
Объект A: {}
Объект B: {'value': 123}
Поле name: Bravo
```

```
Поле value: 123
Класс Alpha
[1] __module__: __main__
[2] __dict__: <attribute '__dict__' of 'Alpha' objects>
[3] __weakref__: <attribute '__weakref__' of 'Alpha' objects>
[4] __doc__: None
[5] display: <function Bravo.display at 0x02646078>
Класс Bravo
[1] __module__: __main__
[2] name: Bravo
[3] show: <function Bravo.show at 0x026460C0>
[4] __init__: <function Bravo.__init__ at 0x02646108>
[5] __dict__: <attribute '__dict__' of 'Bravo' objects>
[6] __weakref__: <attribute '__weakref__' of 'Bravo' objects>
[7] __doc__: None
Объект A: {'color': 'Красный'}
Объект B: {'value': 123, 'show': <function <lambda> at 0x02646030>}
Поле name: Bravo
Поле value: 123
Объект B: 123
```

В программе создается два класса. В классе `Alpha` вообще ничего не описано, а в классе `Bravo` описано:

- поле `name` со значением `"Bravo"`;
- метод класса `display()` без аргументов, при вызове которого отображается значение поля `name` класса;
- метод `show()` с одним аргументом (ссылка на объект, из которого вызывается метод), при вызове метода отображается значение поля `value` объекта, из которого вызывается метод;
- конструктор, при вызове которого полю `value` создаваемого объекта присваивается значение `123`.

Командами `A=Alpha()` и `B=Bravo()` мы создаем два объекта. Далее мы проверяем содержимое словарей, через которые реализованы классы

Alpha, Bravo и объекты A и B. Для доступа к этим словарям используем соответственно инструкции Alpha.__dict__, Bravo.__dict__, A.__dict__ и B.__dict__. Причем при отображении содержимого словарей для классов мы используем оператор цикла и нумеруем атрибуты.

НА ЗАМЕТКУ

Оба класса содержат намного больше атрибутов, чем мы описали (а в классе Alpha мы вообще ничего не описывали). Это те служебные поля, которые доступны в классах по умолчанию. Среди них в том числе и поле __dict__. Для объектов отображаются только те атрибуты, которые заданы непосредственно для объекта. Поэтому у объекта A атрибутов нет, а у объекта B всего одно поле value со значением 123. Это поле создается при вызове конструктора в процессе создания объекта B.

При выполнении команды Bravo.display() из класса Bravo вызывается метод display(), который, ожидаемо, отображает значение поля name класса Bravo. После этого выполняется команда Alpha.display=Bravo.display. Как следствие у класса Alpha появляется атрибут display. Значением атрибута является адрес объекта (ссылка на объект), через который реализован метод display() из класса Bravo.

ПОДРОБНОСТИ

У класса Bravo есть метод display(). Методы, как и функции, реализуются посредством специальных объектов. Имя функции является ссылкой на такой объект. Аналогично, атрибут display класса Bravo содержит ссылку на объект, через который реализован соответствующий метод. Фактически в атрибуте display записан адрес этого объекта. Этот адрес копируется в атрибут display класса Alpha при выполнении команды Alpha.display=Bravo.display. Получается, что при выполнении команд Alpha.display() и Bravo.display() вызывается один и тот же метод. Причем этот метод содержит ссылку на поле name класса Bravo.

Даже после того, как командой del Bravo.display у класса Bravo удаляется атрибут display, одноименный атрибут класса Alpha по-прежнему ссылается на объект, посредством которого был реализован метод display() класса Bravo.

> **НА ЗАМЕТКУ**
>
> При вызове метода `display()` из класса `Alpha` (в формате `Alpha.display()`) отображается значение поля `name` класса `Bravo`. Причина в том, что код метода (в описании класса `Bravo`) содержит явную ссылку `Bravo.name`.

Команда `A.color="Красный"` добавляет в объект `A` атрибут `color` со значением `"Красный"`. При первом выполнении команды `B.show()`, в соответствии с тем, как в классе `Bravo` описан метод `show()`, отображается вспомогательный текст и значение поля `value` объекта `B`. Но ситуация меняется после выполнения команды `B.show=lambda: print("Объект B:", B.value)`. В данном случае в объект `B` добавляется атрибут `show`, и в качестве значения атрибуту присваивается лямбда-выражение `lambda: print("Объект B:", B.value)`. Оно соответствует функции, которая при вызове отображает значение поля `value` объекта `B` (ну и еще текст). Именно эта функция будет вызываться при выполнении команды `B.show()`. При этом «исходную» версию метода `show()` для объекта `B` можно вызвать командой `Bravo.show(B)`. Поясним, как все это работает.

При первом выполнении команды `B.show()` в объекте `B` начинается поиск атрибута `show`. Поскольку у объекта (на момент первого выполнения команды) такого атрибута нет, то поиск продолжается в классе `Bravo`. Соответственно, вызывается метод `show()` из класса `Bravo`, но только с учетом того обстоятельства, что методу передается первый аргумент (ссылка на объект `B`). После того как мы определили атрибут `show` для объекта `B`, и этот атрибут содержит адрес объекта, через который реализуется заданная лямбда-выражением функция, при выполнении команды `B.show()` вызывается именно эта функция. А вот в команде `Bravo.show(B)` мы явно вызываем метод `show()` из класса `Bravo` и в качестве аргумента методу передаем ссылку на объект `B`.

Копирование объектов

Вам трудно угодить. Но я все-таки попробую.

Из к/ф «Служебный роман»

Доступ к объектам мы получаем через переменные, которые ссылаются на объекты (а не содержат их в качестве значения). Ничего особенного здесь нет — мы с подобными ситуациями уже сталкивались (например,

при работе со списками). Тем не менее не будет лишним остановиться подробнее на некоторых важных аспектах. Рассмотрим пример, представленный в листинге 8.9.

Листинг 8.9. Копирование объектов

```
# Импорт функций:
from copy import *
# Описание класса:
class MyClass:
    pass
# Создание объекта:
A=MyClass()
# Поля объекта:
A.value=100
A.nums=[1,2,3]
# Присваивание ссылки на объект:
B=A
# Копия объекта:
C=copy(A)
# Полная копия объекта:
D=deepcopy(A)
print("Созданы объекты")
# Поля объектов:
print("A:", A.value,"и", A.nums)
print("B:", B.value,"и", B.nums)
print("C:", C.value,"и", C.nums)
print("D:", D.value,"и", D.nums)
print("A.value=200 и A.nums[1]=0")
# Новые значения для полей:
A.value=200
A.nums[1]=0
# Поля объектов:
print("A:", A.value,"и", A.nums)
```

```
print("B:", B.value,"и", B.nums)
print("C:", C.value,"и", C.nums)
print("D:", D.value,"и", D.nums)
print("Удаляется A")
# Удаление переменой:
del A
print("B.value=300 и B.nums[2]=4")
# Новые значения для полей:
B.value=300
B.nums[2]=4
# Поля объектов:
print("B:", B.value,"и", B.nums)
print("C:", C.value,"и", C.nums)
print("D:", D.value,"и", D.nums)
```

Ниже представлен результат выполнения программы.

Результат выполнения программы (из листинга 8.9)

```
Созданы объекты
A: 100 и [1, 2, 3]
B: 100 и [1, 2, 3]
C: 100 и [1, 2, 3]
D: 100 и [1, 2, 3]
A.value=200 и A.nums[1]=0
A: 200 и [1, 0, 3]
B: 200 и [1, 0, 3]
C: 100 и [1, 0, 3]
D: 100 и [1, 2, 3]
Удаляется A
B.value=300 и B.nums[2]=4
B: 300 и [1, 0, 4]
C: 100 и [1, 0, 4]
D: 100 и [1, 2, 3]
```

В этом примере мы используем функции `copy()` и `deepcopy()` из модуля `copy`, поэтому программа начинается соответствующей `import`-инструкцией. Класс `MyClass` описан с пустым телом. На основе этого класса создается объект `A`. Командами `A.value=100` и `A.nums=[1,2,3]` в объект `A` добавляется два поля: одно с целочисленным значением и еще одно является ссылкой на список. Далее выполняется несколько важных команд: `B=A`, `C=copy(A)` и `D=deepcopy(A)`. После этого мы проверяем значения полей `value` и `nums` для объектов, на которые ссылаются переменные `A`, `B`, `C` и `D`. На этом этапе все ожидаемо: во всех четырех случаях значения одни и те же. Далее командой `A.value=200` присваивается новое значение полю `value` объекта `A`, а командой `A.nums[1]=0` в списке, на который ссылается поле `nums` объекта `A`, изменяется значение второго элемента. Проверка показывает, что точно так же, как изменились значения полей объекта `A`, изменились и значения полей объекта, на который ссылается переменная `B`. Здесь нет ничего удивительного, поскольку переменные `A` и `B` ссылаются на один и тот же объект. Дело в том, что при выполнении команды `B=A` в переменную `B` записывается значение переменной `A`. Но значением переменной `A` является не объект, а адрес объекта. Этот адрес записывается в переменную `B`. В итоге обе переменные, `A` и `B`, содержат один и тот же адрес и ссылаются на один и тот же объект.

У объекта `C` не меняется значение поля `value`, но меняется значение второго элемента в списке `nums`. Переменная `C` ссылается на объект, который был создан как копия объекта `A`. Но это была обычная (или поверхностная) копия. Она создается путем копирования значений полей. Значением поля `value` является число. А поле `nums` в качестве значения содержит адрес списка `[1,2,3]`. При создании копии в объекте `C` поле `nums` будет иметь такое же значение, как поле `nums` в объекте `A`. Это адрес одного и того же списка. Проще говоря, поля `nums` в объектах `A` и `C` ссылаются на один и тот же список. Если меняется значение одного из элементов списка, то это отображается на каждом из объектов. Но объект `D` создавался как полная копия объекта `A`. Поэтому для поля `nums` копируется не адрес, а создается копия соответствующего списка. В результате изменения, вносимые в объект `A`, на объекте `D` не сказываются.

При выполнении команды `del A` удаляется переменная `A`. Она ссылается на объект. Если бы это была единственная переменная, которая ссылается на данный объект, то следом за переменной `A` был бы удален и объект, на который ссылается переменная. Но у нас есть переменная `B`, которая ссылается на тот же объект. Поэтому удаление переменной `A`

к удалению объекта не приводит. Изменяя значения полей через переменную B получаем результат, аналогичный предыдущему случаю. Объяснение такое же.

НА ЗАМЕТКУ

Желающие могут проделать такой эксперимент. Описываем класс с деструктором:

```
class MyClass:
    def __del__(self):
        print("Удален объект")
```

Затем командой `A=MyClass()` создаем объект. Если после этого выполняется команда `del A`, то в окне вывода появится сообщение `Удален объект`. А если перед командой `del A` разместить команду `B=A`, то сообщение не появится. Причина в том, что деструктор вызывается перед удалением объекта. В первом случае объект, на который ссылается переменная `A`, удаляется. Во втором — нет.

Документирование и декораторы

— Исключительное что-то.
— Совершенно с вами согласен.

Из к/ф «Собачье сердце»

Если в описании класса сразу после строки с ключевым словом `class` и названием класса указать текстовый литерал, то он будет определять *строку документирования* класса. Эта строка доступна через специальное свойство `__doc__` (по два символа подчеркивания в начале и конце). Фактически этот текст можно рассматривать как справку по классу. Небольшой пример представлен в листинге 8.10.

Листинг 8.10. Документирование в классах

```
# Первый класс:
class Alpha:
    "Это класс Alpha"
    pass
```

```
# Второй класс:
class Bravo:
    "Это класс Bravo"
    pass
# Информация о классах:
print(Alpha.__doc__)
print(Bravo.__doc__)
# Объекты классов:
A=Alpha()
B=Bravo()
# Изменение строки документирования:
Alpha.__doc__="Первый класс"
B.__class__.__doc__="Второй класс"
# Информация о классах:
print(A.__class__.__doc__)
print(B.__doc__)
```

Результат выполнения программы будет таким.

Результат выполнения программы (из листинга 8.10)

```
Это класс Alpha
Это класс Bravo
Первый класс
Второй класс
```

Мы описываем классы `Alpha` и `Bravo`, причем каждый из них содержит строку документирования. На основе классов создаются соответственно объекты `A` и `B`. Доступ к тексту документирования классов можно получить через название класса (с помощью ссылок `Alpha.__doc__` и `Bravo.__doc__`) или через объекты классов с помощью ссылок `A.__class__.__doc__` и `B.__class__.__doc__`. В последнем случае сначала с помощью специального поля `__class__` мы получаем ссылку на класс объекта, а затем уже используем поле `__doc__` для получения доступа к тексту документирования. Также можно воспользоваться более простыми ссылками вида `A.__doc__` и `B.__doc__`. Здесь мы сразу

обращаемся к атрибуту `__doc__` объекта. Но поскольку у объекта такого атрибута нет, то поиск будет продолжен в классе. Во всем остальном код простой и, хочется верить, комментариев не требует.

А еще для классов может использоваться *декоратор*. Это определенная инструкция, которая на основе описанного класса позволяет получить другой класс. Для этого нам понадобится функция, которая получает класс в качестве аргумента и результатом возвращает класс. На первый взгляд звучит довольно необычно. Но на самом деле все просто. Ведь класс реализуется посредством объекта. Поэтому все, что нужно сделать, — описать функцию, которой в качестве аргумента передается объект, описывающий класс, а в качестве результата возвращается еще один объект, описывающий другой класс. Поскольку эта ситуация довольно интересна сама по себе, рассмотрим небольшой вспомогательный пример в листинге 8.11.

Листинг 8.11. Класс как аргумент и результат функции

```
# Функция получает ссылку на класс в качестве аргумента
# и результатом возвращает ссылку на класс:
def F(Alpha):
    # Внутренний класс:
    class Bravo:
        value=Alpha()
    Bravo.__name__="My"+Alpha.__name__
    return Bravo
# Описание класса:
class Charlie:
    # Конструктор:
    def __init__(self):
        self.number=123
    # Метод для отображения значения поля:
    def show(self):
        print("Поле number:", self.number)
# Создание объекта на основе класса,
# полученного при вызове функции:
obj=F(Charlie)()
```

```
# Проверка результата:
obj.value.show()
print("Класс объекта obj:", obj.__class__.__name__)
print("Класс поля value:", obj.value.__class__.__name__)
```

Результат выполнения программы следующий.

Результат выполнения программы (из листинга 8.11)

```
Поле number: 123
Класс объекта obj: MyCharlie
Класс поля value: Charlie
```

В этой небольшой программе описывается функция `F()`. Мы предполагаем, что аргумент функции (обозначенный как `Alpha`) является именем класса. Технически это означает, что в качестве аргумента функция получает ссылку на объект, посредством которого реализуется класс. В теле функции описывается класс `Bravo` (поскольку он описан внутри функции, то этот класс будем называть *внутренним*). В теле класса есть инструкция `value=Alpha()`. Если учесть, что `Alpha` обозначает некоторый класс, то инструкция `Alpha()` означает создание объекта этого класса. Таким образом, команда `value=Alpha()` означает, что у внутреннего класса будет поле `value` и это поле будет содержать ссылку на объект класса `Alpha`. После объявления внутреннего класса в теле функции командой `Bravo.__name__="My"+Alpha.__name__` определяется значение специального поля `__name__` внутреннего класса: оно получается дописыванием префикса `"My"` к названию класса, переданного аргументом функции. Данное название получаем на основе специального поля `__name__` класса, переданного в качестве аргумента (инструкция `Alpha.__name__`). Наконец, инструкцией `return Bravo` ссылка на объект, посредством которого реализуется внутренний класс, возвращается в качестве результата функции.

НА ЗАМЕТКУ

Таким образом, в теле функции описывается внутренний класс, задаются его параметры, а затем результатом функция возвращает ссылку на объект, через который реализован внутренний класс. Поскольку имя класса является ссылкой на объект, посредством которого реализуется класс, можно утверждать, что функция возвращает класс в качестве результата.

В программе описывается обычный класс `Charlie` с конструктором, при вызове которого у объекта создается поле `number` со значением `123`. Также в классе описан метод `show()`, предназначенный для отображения значения поля `number`.

Самая интересная команда в программе имеет вид `obj=F(Charlie)()`. Понять ее просто, если учесть, что результатом инструкции `F(Charlie)` является ссылка на объект, через который реализуется класс. Фактически это то же, что и имя класса. Другими словами, инструкцию `F(Charlie)` можно воспринимать как имя класса. Тогда инструкция `F(Charlie)()` означает создание объекта этого класса. Получается, что при выполнении команды `obj=F(Charlie)()` вызывается функция `F()`, которая на основе класса `Charlie` создает новый класс, на основе этого нового класса создается объект, и ссылка на объект записывается в переменную `obj`.

После создания объекта мы используем команду `obj.value.show()`. С ее помощью из объекта класса `Charlie`, на который ссылается поле `value` объекта `obj`, вызывается метод `show()`. Отображается значение `123` для поля `number`. Узнать название класса, на основе которого создавался объект `obj`, можно с помощью инструкции `obj.__class__.__name__`. Поскольку при вызове функции `F()` в качестве аргумента ей передавался класс `Charlie`, то у созданного класса название `"MyCharlie"`. А класс объекта, на который ссылается поле `value` в объекте `obj` (точнее, это поле класса), определяется инструкцией `obj.value.__class__.__name__`. Понятно, что соответствующий объект относится к классу `Charlie`.

Теперь перейдем к рассмотрению декораторов для класса. Допустим, что `F()` — некоторая функция, которая аргументом получает ссылку на класс и результатом также возвращает ссылку на класс. То есть если `Alpha` — какой-то класс, то `F(Alpha)` — тоже некоторый класс. Декоратор класса — инструкция вида `@F`, которая размещается в строке перед описанием класса. Последствия такие. Если мы описываем класс `Charlie`, и перед ним указан декоратор `@F`, то в действительности получаем класс `F(Charlie)`.

> **НА ЗАМЕТКУ**
>
> Можно сказать и по-другому. Если перед описанием класса `Charlie` указан декоратор `@F`, то класс подвергается преобразованию, которое можно описать инструкцией `Charlie=F(Charlie)`.

Программа с использованием декоратора класса представлена в листинге 8.12. В ней мы использовали функцию, аналогичную той, что описывалась в предыдущем примере.

Листинг 8.12. Декоратор класса

```
# Функция для декоратора:
def F(Alpha):
    class Bravo:
        value=Alpha()
    Bravo.__name__="My"+Alpha.__name__
    return Bravo
# Класс с декоратором:
@F
class Charlie:
    def __init__(self):
        self.number=123
    def show(self):
        print("Поле number:", self.number)
# Создание объекта:
obj=Charlie()
# Проверка результата:
obj.value.show()
print("Класс объекта obj:", obj.__class__.__name__)
print("Класс поля value:", obj.value.__class__.__name__)
```

Результат получаем такой же, как и в предыдущем случае.

Результат выполнения программы (из листинга 8.12)

```
Поле number: 123
Класс объекта obj: MyCharlie
Класс поля value: Charlie
```

Изменения в программе, по сравнению с предыдущим примером, в том, что ранее мы создавали объект инструкцией вида `obj=F(Charlie)()`,

а теперь мы перед описанием класса `Charlie` разместили декоратор `@F`, а объект создаем инструкцией `obj=Charlie()`. Но по сути отличия минимальные. Ранее инструкцией `F(Charlie)` на основе класса `Charlie` вычислялся новый класс, и на основе этого класса создавался объект. Теперь же за счет декоратора `@F` класс `Charlie` переопределяется. За основу берется описание класса `Charlie`, и с помощью этого класса создается новый класс. Он создается в соответствии с инструкцией `F(Charlie)`. Ссылка на созданный класс записывается в идентификатор `Charlie` (фактически выполняется команда `Charlie=F(Charlie)`). Поэтому если ранее ссылка на новый класс давалась инструкцией `F(Charlie)`, то теперь ссылка на этот новый класс дается инструкцией `Charlie`.

Использование классов и объектов

> Делом надо заниматься серьезно или не заниматься им вообще.
>
> *Из к/ф «Служебный роман»*

Тема использования классов и объектов практически безгранична. Мы рассмотрим лишь несколько примеров, которые позволят составить некоторое представление о гибкости и эффективности объектно-ориентированного подхода.

Начнем с примера, в котором в зависимости от типа переданного конструктору аргумента у объекта создается то или иное поле. Программа представлена в листинге 8.13.

Листинг 8.13. Объекты с разными полями

```
# Описание класса:
class MyClass:
    # Конструктор:
    def __init__(self, val):
        # Если аргумент целочисленный:
        if type(val)==int:
            self.number=val
        # Если аргумент текстовый:
        elif type(val)==str:
```

```
            self.name=val
        # Если аргумент — действительное число:
        elif type(val)==float:
            self.value=val
        # Прочие случаи:
        else:
            self.data=val
    # Метод для отображения значения поля:
    def show(self):
        # Список с названиями полей:
        L=["number","name","value","data"]
        # Перебор названий полей:
        for s in L:
            # Если поле существует:
            if s in dir(self):
                # Отображение названия и значения поля:
                print(s,"=", self.__dict__[s])
                # Завершение оператора цикла:
                break
        # Если поле не найдено:
        else:
            print("Странный объект")
# Создание объектов и проверка полей:
A=MyClass(123)
A.show()
del A.number
A.show()
B=MyClass("Объект B")
B.show()
C=MyClass(2.5)
C.show()
D=MyClass([1,2,3])
D.show()
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 8.13)

```
number = 123
Странный объект
name = Объект B
value = 2.5
data = [1, 2, 3]
```

В классе `MyClass` описан конструктор и метод `show()`. Оба они достойны внимания.

Конструктор описан с двумя аргументами. Первый аргумент `self` традиционно обозначает создаваемый объект, а второй аргумент `val` задает значение, присваиваемое полю объекта. Название поля зависит от типа этого аргумента. Для проверки типа аргумента используем функцию `type()`. Если аргумент целочисленный (тип `int`), то поле объекта называется `number`. При текстовом аргументе (тип `str`) поле называется `name`. Для аргумента типа `float` поле называется `value`, а во всех прочих случаях поле называется `data`.

Метод `show()` предназначен для отображения значения поля объекта. Проблема в том, что нужно сначала определить, какое это поле. В теле метода в локальную переменную `L` записывается список `["number", "name","value","data"]` с названиями полей, которые мы планируем искать в объекте. Перебор полей выполняем с помощью оператора цикла `for`, в котором переменная `s` пробегает значения из списка `L`. За каждый цикл выполняется условный оператор, в котором проверяется условие `s in dir(self)`. Здесь мы воспользовались встроенной функцией `dir()`, которая в качестве результата возвращает список с названиями атрибутов объекта, переданного аргументом функции. Если в списке атрибутов искомое поле есть, то командой `print(s,"=", self.__dict__[s])` отображается название поля и его значение. Поскольку у нас в данном случае название поля «спрятано» в переменную `s`, то для получения значения поля мы используем инструкцию `self.__dict__[s]`. Что мы сделали? Мы с помощью служебного поля `__dict__` получили доступ к словарю с атрибутами объекта и их значениями, а затем для этого словаря по ключу элемента (значение переменной `s`) получили значение элемента. После этого выполняется инструкция `break`, которая завершает выполнение оператора цикла.

Для оператора цикла предусмотрен блок `else`, в котором выполняется команда `print("Странный объект")`. Это блок выполняется только в том случае, если работа оператора цикла завершается не вследствие выполнения инструкции `break`. А это возможно, только если при поиске поля в объекте оно там не найдено.

НА ЗАМЕТКУ

Таким образом, если хотя бы одно поле из списка `L` у объекта есть, то отображается название и значение поля (первого из найденных). Если ни одно поле не найдено, то отображается сообщение `"Странный объект"`.

После описания класса создается несколько объектов, причем каждый раз аргументом конструктору передаются значения разных типов. Для каждого из этих объектов вызывается метод `show()`, которым отображается значение поля. Также в программе есть пример ситуации, когда у объекта удаляется поле, после чего вызывается метод `show()`.

В следующем примере описана функция, при вызове которой создается объект. Название класса, на основе которого создается объект, а также поля и значения этих полей определяются аргументами функции. Рассмотрим программу, представленную в листинге 8.14.

Листинг 8.14. Функция для создания объектов

```
# Функция для создания объектов:
def create(fields, vals, name=None):
    # Если последний аргумент не текстовый:
    if type(name)!=str:
        name="MisterX"
    # Если первые два аргумента — не списки:
    if type(fields)!=list or type(vals)!=list:
        # Внутренний класс:
        class MyClass:
            # Метод:
            def show(self):
                print("Объект без полей")
                print("Класс", self.__class__.__name__)
```

```
    # Если первые два аргумента — списки:
    else:
        # Внутренний класс:
        class MyClass:
            # Конструктор:
            def __init__(self):
                k=0
                for f in fields:
                    self.__dict__[f]=vals[k]
                    k+=1
            # Метод:
            def show(self):
                print("Объект с полями:")
                for s in dir(self):
                    if not s.startswith("_") and s!="show":
                        print(s,"=", self.__dict__[s])
                print("Класс", self.__class__.__name__)
    # Название класса:
    MyClass.__name__=name
    # Результат функции:
    return MyClass()
# Создание объекта и проверка полей:
A=create(["red","green","blue"],[1,2,3],"MyColors")
A.show()
# Создание объекта и проверка полей:
B=create(["alpha","bravo"],["Alpha","Bravo"])
B.show()
# Создание объекта и проверка полей:
C=create(1,2,3)
C.show()
# Изменение значений полей объекта:
A.red=100
A.green=200
```

```
A.blue=300
A.show()
# Создание объекта и проверка полей:
D=A.__class__()
D.show()
```

Результат выполнения программы будет следующим.

Результат выполнения программы (из листинга 8.14)

```
Объект с полями:
blue = 3
green = 2
red = 1
Класс MyColors
Объект с полями:
alpha = Alpha
bravo = Bravo
Класс MisterX
Объект без полей
Класс MisterX
Объект с полями:
blue = 300
green = 200
red = 100
Класс MyColors
Объект с полями:
blue = 3
green = 2
red = 1
Класс MyColors
```

Пример простой, и в нем не проверяются многие потенциально опасные ситуации, но он все же показательный. Основу программы составляет

функция `create()`. Предполагается, что функции при вызове передаются три аргумента: `fields` и `vals` являются списками (одинаковых размеров, но это обстоятельство в программе не проверяется) соответственно с названиями полей и их значениями, а третий текстовый аргумент `name` определяет название класса, на основе которого будет создаваться объект.

ПОДРОБНОСТИ

Для аргумента `name` указано значение по умолчанию `None`. Это пустая ссылка, которая означает «отсутствующее» значение. В данном случае важно то, что даже если при вызове функции аргумент `name` не указать, то у него все равно будет значение (хоть и «отсутствующее»).

В теле функции в условном операторе проверяется условие `type(name)!=str`, истинное, если аргумент `name` не текстовый. В таком случае командой `name="MisterX"` мы переопределяем значение этого аргумента.

НА ЗАМЕТКУ

Получается, что если при вызове функции `create()` название класса не указать, то класс будет называться `MisterX`.

Условие `type(fields)!=list or type(vals)!=list` истинно, если хотя бы один из первых двух аргументов не является списком. Если так, то описывается внутренний класс `MyClass` с единственным методом `show()`, который при вызове отображает сообщение об отсутствии полей у объекта и название класса (вычисляется инструкцией `self.__class__.__name__`).

В случае если первые два аргумента — списки, также создается класс `MyClass`, но на этот раз немного другой. У него есть конструктор, в котором перебираются элементы первого списка с названиями полей, соответствующие поля создаются у объекта и им присваивается значение из второго списка.

Метод `show()` определен теперь таким образом, что для объекта отображаются названия полей и их значения. Список атрибутов получаем с помощью функции `dir()`. Но это будет список всех атрибутов, в том числе и служебных полей. Нас служебные поля не интересуют, поэтому отображаются только те поля, которые не начинаются с символа подчеркивания.

Также мы исключаем из списка отображаемых атрибутов сам метод `show()`. Условие, истинность которого необходима для отображения названия и значения поля, выглядит как `not s.startswith("_") and s!="show"`.

После завершения выполнения условного оператора командой `MyClass.__name__=name` задаем название для внутреннего класса, а командой `return MyClass()` создаем объект этого класса и возвращаем его в качестве результата функции.

НА ЗАМЕТКУ

Инструкцией `MyClass()` создается объект класса `MyClass`. Значением инструкции является ссылка на созданный объект. Ее можно записать в переменную. Но в данном случае в этом необходимости нет. Мы всю инструкцию указываем результатом функции. При вызове функции создается объект, а ссылка на него возвращается функцией как результат. Эту ссылку мы можем присвоить в качестве значения переменной, и такая переменная будет ссылаться на созданный функцией объект.

Программа содержит несколько команд, которыми создаются объекты. Содержимое объектов проверяем с помощью метода `show()`.

При выполнении команды `A=create(["red","green","blue"],[1,2,3],"MyColors")` создается объект с полями `red`, `green` и `blue` со значениями `1`, `2` и `3` соответственно. Класс, на основе которого создается объект, будет называться `MyColors`.

Команда `B=create(["alpha","bravo"],["Alpha","Bravo"])` означает, что объект будет иметь поля `alpha` и `bravo` со значениями `"Alpha"` и `"Bravo"`. Поскольку название для класса не указано, то по умолчанию класс будет называться `MisterX`.

Наконец, командой `C=create(1,2,3)` создается объект без полей (мы не добавляем в объект поля, но у него есть служебные поля), поскольку первые два аргумента не являются списками. В силу того, что третий аргумент не текстовый, класс также будет называться `MisterX`.

Показательной является последний блок программы. Там сначала меняются значения полей объекта `A`. После этого выполняется команда `D=A.__class__()`. Этой командой на основе класса, который использовался при создании объекта `A`, создается новый объект. Проверка

командой `D.show()` показывает, что у объекта `D` такие же поля с такими же значениями, что были у объекта `A` при создании. Как это происходит?

Инструкция `A.__class__` позволяет получить ссылку на класс, на основе которого создавался объект `A`. Инструкция `A.__class__()` означает, что на основе данного класса создается объект. В этом случае вызывается конструктор. В теле конструктора содержатся ссылки на списки, которые передавались функции `create()` при вызове, когда создавался объект `A`. Поэтому когда конструктор вызывается еще раз, он обращается к тем же спискам. Как следствие, объект `D` получается такой же, как объект `A`.

НА ЗАМЕТКУ

Желающие могут проделать такой эксперимент и немного изменить код программы. Например, вместо команды `A=create(["red", "green", "blue"],[1,2,3],"MyColors")` используем команду `A=create(["red","green","blue"], V,"MyColors")`, предварительно выполнив команду `V=[1,2,3]`. А перед выполнением команды `D=A.__class__()` выполнить команду `V[1]=123`. Попробуйте объяснить результат (поле `green` объекта `D` будет иметь значение `123`).

В следующей программе создается цепочка объектов. В такой цепочке каждый объект (за исключением последнего) ссылается на объект того же класса. Ссылка записывается в поле объекта. Таким образом, имея доступ к первому объекту, можем получить доступ к любому объекту в цепочке. Программа представлена в листинге 8.15.

Листинг 8.15. Цепочка объектов

```
# Класс:
class MyClass:
    # Конструктор:
    def __init__(self, name, n=1):
        self.name=name
        if n==1:
            self.next=None
        else:
            self.next=MyClass(self.name, n-1)
        self.set()
    # Деструктор:
```

```
    def __del__(self):
        print("Удаление:", self.code)
    # Метод для заполнения цепочки кодами:
    def set(self, num=1):
        self.code=self.name+"["+str(num)+"]"
        if self.next!=None:
            self.next.set(num+1)
     # Метод для отображения кодов объектов в цепочке:
    def show(self):
        print(self.code)
        if self.next!=None:
            self.next.show()
# Создание цепочки объектов:
print("Один объект:")
A=MyClass("Alpha")
A.show()
print("Цепочка объектов:")
B=MyClass("Bravo",5)
B.show()
print("Начиная с третьего объекта:")
B.next.next.show()
# Удаление объектов:
print("Удаление объектов:")
del A
del B
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 8.15)

```
Один объект:
Alpha[1]
Цепочка объектов:
Bravo[1]
```

```
Bravo[2]
Bravo[3]
Bravo[4]
Bravo[5]
Начиная с третьего объекта:
Bravo[3]
Bravo[4]
Bravo[5]
Удаление объектов:
Удаление: Alpha[1]
Удаление: Bravo[1]
Удаление: Bravo[2]
Удаление: Bravo[3]
Удаление: Bravo[4]
Удаление: Bravo[5]
```

Основу программы составляет класс `MyClass`. В классе описан конструктор с тремя аргументами. Предполагается, что второй аргумент `name` — текстовый. Он определяет значение одноименного поля объекта. Третий аргумент `n` (с единичным значением по умолчанию) определяет количество объектов в цепочке. Если значение `n` равно `1`, то командой `self.next=None` полю `next` текущего объекта присваивается пустое значение `None`. В противном случае командой `self.next=MyClass(self.name, n-1)` создается новый объект, и ссылка на него записывается в поле `next` текущего объекта. Стоит заметить, что в данном случае третий аргумент конструктора равен `n-1` (то есть на единицу меньше, чем соответствующее значение, переданное конструктору при создании текущего объекта). Если это новое значение отлично от единицы, то при создании нового объекта снова будет создаваться объект, и так далее по цепочке. Процесс продолжается, пока конструктор не будет вызван с единичным третьим аргументом. Другими словами, в данном случае рекурсивно вызывается конструктор, и количество таких вызовов равно значению третьего аргумента, переданного конструктору в самом начале.

После того как цепочка объектов создана, в конструкторе выполняется команда `self.set()`, которой из объекта вызывается метод `set()`. Метод нам понадобился для идентификации объектов. У каждого объекта

будет поле code, которое состоит из названия, записанного в поле name, и порядкового номера объекта. Метод set() описан со вторым аргументом num, имеющим единичное значение по умолчанию. Фактически это номер объекта в цепочке. При вызове метода из объекта он последовательно задает значение поля code для каждого объекта в цепочке. Делается это командой self.code=self.name+"["+str(num)+"]". После ее выполнения проверяется условие self.next!=None, истинное, если объект не является последним в цепочке. Если так, то командой self.next.set(num+1) метод set() вызывается для следующего объекта в цепочке, причем номер объекта увеличен на единицу (значение num+1).

ПОДРОБНОСТИ

Мы исходим из того, что значение поля next для последнего объекта в цепочке равно None. Честно говоря, достаточно зыбкий критерий (в реальной программе его легко нарушить).

Доступ к следующему объекту в цепочке можно получить через поле next. Например, в теле метода set() доступ к следующему объекту выполняется инструкцией self.next.

Еще один момент достоин внимания: код конструктора достаточно простой. Но простой код не означает, что алгоритм выполнения эффективный. В данном случае метод set() вызывается в конструкторе, причем вызывает сам себя во всей последующей цепочке объектов. А конструктор вызывается при создании каждого объекта в цепочке. Получается, что метод set() вызывается для последнего объекта. Затем метод set() вызывается из предпоследнего объекта, и по цепочке вызывается метод set() из последнего объекта. Затем метод set() вызывается из третьего с конца объекта, и снова по цепочке до последнего объекта. Последняя «итерация» реализуется, когда объект вызывается из первого объекта в цепочке. В этом случае получаем окончательное заполнение значений поля code объектов в цепочке. Убедиться в этом просто — достаточно добавить команду отображения сообщения о присваивании значения полю code (например, print(self.code)) сразу после команды self.code=self.name+"["+str(num)+"]". Желающие могут подумать, как можно было бы усовершенствовать этот код.

В классе есть метод show(), при вызове которого отображаются значения поля code для всех объектов в цепочке (начиная с объекта, из которого вызван метод). Схема использована такая же, как и в предыдущих случаях: отображается значение поля, и если объект не последний в цепочке, то метод вызывается из следующего объекта.

Помимо этого, в классе описан деструктор. При вызове деструктора отображается сообщение об удалении объекта и код удаляемого объекта.

При выполнении команды `A=MyClass("Alpha")` создается всего один объект (цепочка из одного объекта). Выполнение команды `B=MyClass("Bravo",5)` приводит к созданию цепочки из пяти объектов. Ссылка на первый объект в цепочке записана в переменную `B`. А вот инструкция `B.next.next` означает обращение к третьему объекту в цепочке (первый объект — это `B`, второй объект — это `B.next`, а третий объект — соответственно `B.next.next`).

Для удаления объектов использованы команды `del A` и `del B`. В последнем случае автоматически удаляется вся цепочка. Объяснение такое: формально команда `del B` удаляет переменную `B`. Поскольку это была единственная переменная в программе, которая ссылалась на первый объект в цепочке, то для программы этот объект «потерян» и он удаляется из памяти. Но у этого объекта было поле `next`, которое единственное ссылалось на второй объект в цепочке. Этот объект тоже теряется для программы и удаляется из памяти, и так по всей цепочке.

НА ЗАМЕТКУ

Можно провести небольшой эксперимент. Перед командой `del B` добавьте команду `C=B.next`. Теперь при выполнении команды `del B` будет удаляться только первый объект в цепочке. А если вместо команды `C=B.next` использовать команду `C=B.next.next`, то будут удалены два первых объекта в цепочке. В этом смысле объекты похожи на людей — и для тех, и для других важно, чтобы о них помнили.

Резюме

Независимые умы никогда не боялись банальностей.

Из к/ф «Покровские ворота»

- Объект позволяет объединить в одно целое данные и код, предназначенный для обработки данных. Объекты создаются на основе классов. Класс представляет собой шаблон, определяющий структуру и функциональность объектов, которые создаются на основе класса.

- Описание класса начинается с ключевого слова `class`, после которого указывается имя класса, двоеточие и блок с описанием класса. Если тело класса пустое, используется инструкция `pass`.
- Для создания объекта класса указывается имя класса и круглые скобки (пустые или с аргументами, которые передаются конструктору). В результате создается объект, ссылку на который можно записать в переменную.
- Объект может иметь поля и методы (атрибуты). При обращении к полям и методам объекта используется точечный синтаксис: указывается имя объекта, ставится точка и затем имя поля или метода (с круглыми скобками с аргументами или без).
- Классы реализуются с помощью специальных объектов. Имя класса является ссылкой на объект, через который реализуется класс. Технически и объекты, созданные на основе классов, и объекты, посредством которых реализуются классы, хранятся в виде словарей. Доступ к соответствующему словарю можно получить с помощью специального поля `__dict__`.
- При обращении к атрибуту объекта поиск атрибута выполняется в объекте, и если атрибут не найден, то продолжается в классе. Поля и методы можно добавлять в классы и объекты, а также можно удалять из классов и объектов. В первом случае соответствующему атрибуту присваивается значение, а для удаления атрибута можно использовать инструкцию `del`.
- Если метод вызывается из объекта, то первым аргументом ему неявно передается ссылка на объект. Поэтому соответствующие методы в описании содержат на один аргумент больше, чем им передается аргументов при вызове. Традиционно первый аргумент называется `self` и обозначает объект, из которого вызывается метод.
- Конструктор — специальный метод `__init__()`, который вызывается автоматически при создании объекта. Деструктор — специальный метод `__del__()`, который автоматически вызывается при удалении объекта.
- Есть специальные поля, которые позволяют получать полезную информацию о классах и объектах. Так, с помощью поля `__doc__` можно получить доступ к текстовой строке документирования (обычно содержит описание класса, а соответствующий текстовый литерал указывается сразу под первой строкой в инструкции

описания класса). С помощью поля `__class__` можно получить ссылку на класс, на основе которого создавался объект. Поле `__name__` позволяет узнать название этого класса.

- В переменных сохраняются ссылки на объекты. Поэтому присваивание переменных не приводит к копированию объектов. Для создания поверхностной копии объекта можно использовать функцию `copy()`, а полную копию объекта можно создать с помощью функции `deepcopy()`.
- Для классов можно применять декораторы. Декоратор представляет собой инструкцию, которая состоит из символа `@` и названия функции, которая в качестве аргумента получает класс и результатом возвращает класс. Такая инструкция размещается перед описанием класса и в результате класс переопределяется в соответствии с тем, как определена функция, использованная в декораторе.

Задания для самостоятельной работы

Чего не надо, того не сделают.

Из к/ф «Покровские ворота»

1. Напишите программу, в которой описывается класс со следующими характеристиками. У класса есть конструктор, которому (кроме ссылки на объект вызова) передаются два значения. Эти значения присваиваются полям объекта класса. В классе должен быть описан метод, при вызове которого отображаются значения полей класса. Проверьте функциональность класса, создав на его основе несколько объектов.

2. Напишите программу, в которой описан класс со следующими свойствами. В классе описан конструктор, которому в качестве аргументов (помимо первой ссылки на создаваемый объект) передаются текст и целое число, причем в произвольном порядке. Число и текст присваиваются как значения определенным полям. Если переданы два текстовых значения, то создается только текстовое поле со значением, получающимся объединением значений аргументов. Если аргументами переданы два числовых поля, то у объекта будет только поле с целочисленным значением, равным сумме значений аргументов. В иных случаях поля для объекта не создаются. Создать на основе класса объекты и проверить функциональность кода.

3. Напишите программу, в которой описан класс. У объектов класса должно быть поле, представляющее собой числовой список. Этот список формируется на основе списка, переданного конструктору в качестве аргумента. При этом из списка-аргумента в список-поле включаются только числовые элементы (элементы других типов игнорируются). Необходимо также описать метод, отображающий содержимое поля-списка, а также метод, вычисляющий среднее значение элементов поля-списка (сумма значений элементов, деленная на их количество).

4. Напишите программу, в которой описана функция, предназначенная для создания объектов. Функции при вызове передается список и текстовый аргумент. Текстовый аргумент определяет название класса, на основе которого создается объект. Текстовые элементы из списка определяют названия полей объекта (нетекстовые аргументы игнорируются). Значениями полей объекта являются натуральные числа.

5. Напишите программу, в которой описывается функция, предназначенная для создания объекта. Объект создается на основе уже существующего объекта, который передается функции в качестве аргумента. В создаваемый объект добавляются только те неслужебные поля из исходного объекта, которые имеют целочисленное значение.

6. Напишите программу, в которой описан класс и функция, предназначенная для создания списка из объектов. У объектов класса должно быть поле (предназначенное для записи целочисленных значений). При вызове функции аргументом ей передается целое число, определяющее количество объектов в списке. Поля объектов заполняются целыми нечетными числами.

7. Напишите программу, в которой описана функция. В качестве аргументов функции передаются два объекта одного и того же класса. У каждого объекта есть поле, представляющее собой список из целых чисел. В результате функция возвращает объект того же класса. Поле-список этого объекта получается суммированием соответствующих элементов из полей-списков объектов, переданных аргументами функции. Если в этих объектах списки разной длины, то недостающие элементы в списке заменяются нулями.

8. Напишите программу, в которой создается цепочка объектов. Для создания цепочки объектов предложите функцию, при вызове которой в качестве аргумента передается целое число, определяющее количество объектов в цепочке. Результатом функция должна возвращать ссылку на первый объект в цепочке.

9. Напишите программу, в которой создается цепочка объектов. Предложите метод или функцию, которые позволяют вставить новый объект в уже существующую цепочку, а также метод или функцию, которые позволяют удалить объект из цепочки (так, чтобы оставшиеся объекты образовали цепочку).

10. Напишите программу, в которой создается бинарное дерево объектов: в вершине структуры находится объект, который содержит ссылки на два объекта того же класса. Каждый из этих объектов содержит ссылки на два объекта, и так далее.

Глава 9
НАСЛЕДОВАНИЕ И СПЕЦИАЛЬНЫЕ МЕТОДЫ

Ну а это довесок к кошмару.

Из к/ф «Старики-разбойники»

В этой главе мы рассмотрим вопросы, связанные с наследованием и использованием специальных методов. Это две связанные темы, поскольку использование специальных методов базируется на наследовании. Наследование, в свою очередь — один из фундаментальных механизмов объектно-ориентированного подхода. Правда, в Python наследование реализуется специфически. И нам предстоит познакомиться с этой спецификой.

Знакомство с наследованием

Скажите, доктор Ватсон, вы понимаете всю важность моего открытия?

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Наследование — механизм, название которого очень точно отображает его суть. С помощью наследования новые классы можно создавать не на пустом месте, а на основе уже существующих классов. Мы будем использовать такую терминологию. Класс, на основе которого создается новый класс, будем называть *базовым*. Класс, который создается на основе базового класса, будем называть *производным*.

НА ЗАМЕТКУ

Базовый класс иногда еще называют суперклассом. Производный класс в некоторых случаях называют подклассом.

Зачем нужно наследование? Оно позволяет автоматически включать в производный класс атрибуты из базового класса. Это удобно, просто, эффективно и серьезно экономит время, сокращает объем программного кода и повышает его надежность.

НА ЗАМЕТКУ

Стоит заметить, что базовых классов может быть несколько. Другими словами, при создании производного класса «за основу» можно взять сразу несколько классов. Такой механизм называется множественным наследованием.

Кроме множественного, есть еще такое понятие, как многократное наследование. Это ситуация, когда на основе базового класса создается производный класс, а на основе этого производного класса создается еще один производный класс и так далее. То есть производный класс сам может быть базовым для другого класса.

Как мы уже знаем, при обращении к атрибуту объекта поиск этого атрибута сначала выполняется в объекте, а затем, если непосредственно у объекта такого атрибута нет, поиск продолжается в классе. Если в классе данного атрибута тоже нет, но у этого класса есть базовый класс, то будет выполняться поиск в базовом классе. В принципе, довольно удобно.

С формальной точки зрения создать производный класс достаточно просто. Производный класс описывается так же, как обычный класс, но в круглых скобках после имени производного класса указывается имя базового класса. Допустим, базовый класс называется `Alpha` и мы на основе этого класса хотим создать производный класс `Bravo`. Тогда для описания класса `Bravo` следует использовать такой шаблон:

```
class Bravo(Alpha):
    # Описание производного класса
```

Причем в теле производного класса мы можем описывать как новые атрибуты, так и обращаться к атрибутам (например, методам), описанным в базовом классе. Как иллюстрацию рассмотрим небольшой пример, представленный в листинге 9.1.

Листинг 9.1. Знакомство с наследованием

```
# Базовый класс:
class Alpha:
```

```
    # Конструктор:
    def __init__(self):
        self.set(100)
        print("Объект класса Alpha:", self.number)
    # Методы:
    def set(self, n):
        self.number=n
    def show(self):
        print(self.__class__.__name__, self.number)
# Производный класс:
class Bravo(Alpha):
    # Конструктор:
    def __init__(self):
        self.set(200)
        print("Объект класса Bravo:", self.number)
# Объект базового класса:
A=Alpha()
A.set(123)
A.show()
# Объект производного класса:
B=Bravo()
B.set(321)
B.show()
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.1)

```
Объект класса Alpha: 100
Alpha 123
Объект класса Bravo: 200
Bravo 321
```

В программе описывается класс `Alpha`, который мы используем как базовый для создания производного класса `Bravo`. Класс `Alpha` содержит

описание методов set() и show(). При вызове метода set() полю number объекта присваивается значение, переданное аргументом методу. Метод предназначен для отображения названия класса объекта, из которого вызывается метод (инструкция self.__class__.__name__), значение поля number объекта (инструкция self.number). Также у класса есть конструктор, в котором путем вызова метода set() полю number присваивается значение 100, а еще отображается сообщение о создании объекта класса Alpha и значение поля number.

Класс Bravo создается наследованием класса Alpha. Формально в классе Bravo описан только конструктор. Но благодаря наследованию мы можем обращаться в классе Bravo к методам set() и show(), описанным в классе Alpha, так, как если бы описание этих методов содержалось непосредственно в классе Bravo. Более того, у объектов класса Bravo соответствующие методы также имеются (они «извлекаются» из класса Alpha при обращении к соответствующему атрибуту).

Что касается конструктора класса Bravo, то в нем командой self.set(200) с помощью метода set() полю number объекта присваивается значение 200, а также отображается сообщение о создании объекта класса Bravo (и значение поля number).

После описания классов мы командой A=Alpha() создаем объект базового класса. Затем командой A.set(123) изменяется значение поля number объекта, а с помощью команды A.show() проверяется результат.

Объект производного класса создается командой B=Bravo(). У этого объекта есть как метод set() (инструкция B.set(321)), так и метод show() (инструкция B.show()).

НА ЗАМЕТКУ

Методы set() и show() есть у объекта B в том смысле, что из объекта B эти методы можно вызвать. Сами методы описаны в классе Alpha. При вызове метода из объекта сначала выполняется поиск метода в словаре, через который реализуется объект. Затем поиск выполняется в словаре, через который реализуется класс Bravo. А после этого — в словаре, посредством которого реализован класс Alpha.

В следующем примере используется три класса, которые последовательно наследуют друг друга (то есть имеет место многократное наследование). Программа представлена в листинге 9.2.

Листинг 9.2. Многократное наследование

```
# Первый класс:
class Alpha:
    code=123
    def alpha(self):
        print("Alpha:", self.code)
# Второй класс:
class Bravo(Alpha):
    def bravo(self):
        print("Bravo:", self.code)
# Третий класс:
class Charlie(Bravo):
    def charlie(self):
        print("Charlie:", self.code)
# Функция для отображения иерархии наследования:
def show(MyClass):
    print("Класс", MyClass.__name__, end=":\n")
    for s in MyClass.__mro__:
        print("<", s.__name__,">", end="", sep="")
    print()
# Иерархия наследования классов:
show(Alpha)
show(Bravo)
show(Charlie)
# Создание объектов:
A=Alpha()
B=Bravo()
C=Charlie()
# Вызов методов:
print("Объект A")
A.alpha()
print("Объект B")
```

```
B.alpha()
B.bravo()
print("Объект C")
C.alpha()
C.bravo()
C.charlie()
# Присваивание значения полю:
Bravo.code=321
print("Выполнена команда Bravo.code=321")
# Вызов методов:
print("Объект C")
C.alpha()
C.bravo()
C.charlie()
print("Объект A")
A.alpha()
```

Ниже показано, как будет выглядеть результат выполнения программы.

Результат выполнения программы (из листинга 9.2)

```
Класс Alpha:
<Alpha><object>
Класс Bravo:
<Bravo><Alpha><object>
Класс Charlie:
<Charlie><Bravo><Alpha><object>
Объект A
Alpha: 123
Объект B
Alpha: 123
Bravo: 123
Объект C
Alpha: 123
```

```
Bravo: 123
Charlie: 123
Выполнена команда Bravo.code=321
Объект C
Alpha: 321
Bravo: 321
Charlie: 321
Объект A
Alpha: 123
```

Класс `Alpha` описан с полем `code`, значение которого равно `123`. Кроме этого в классе описан метод `alpha()`, при вызове которого отображается значение поля `code` объекта и название класса `Alpha`.

Класс `Bravo` создается наследованием класса `Alpha`. Непосредственно в классе `Bravo` описан метод `bravo()`. При вызове метода отображается название класса `Bravo` и значение поля `code` объекта.

Наконец, класс `Charlie` наследует класс `Bravo` и содержит описание метода `charlie()`. Методом при вызове отображается название класса `Charlie` и значение поля `code` объекта, из которого вызывается метод.

Кроме трех классов, мы описываем функцию `show()`. Предполагается, что в результате вызова функции отображаются названия классов, которые (по цепочке наследования) являются базовыми для класса, переданного аргументом функции. В теле функции мы используем специальное поле `__mro__` для класса. Результатом поля является кортеж с теми классами, которые наследуются классом, для которого запрашивается поле. Мы хотим, чтобы при вызове функции отображались только названия классов. Для получения названий классов используем специальное поле `__name__`.

ПОДРОБНОСТИ

В операторе цикла переменная `s` пробегает значения из кортежа, который дается выражением `MyClass.__mro__` (через `MyClass` обозначен аргумент функции `show()`). Элементами кортежа `MyClass.__mro__` являются ссылки на объекты, посредством которых реализуются классы, наследуемые классом `MyClass` (причем класс `MyClass` сам входит в упомянутый кортеж). Таким образом,

переменная s является ссылкой на объект, посредством которого реализуется класс. Тогда название класса определяется инструкцией s.__name__.

Для определения иерархии наследования для классов Alpha, Bravo и Charlie используются команды show(Alpha), show(Bravo) и show(Charlie) соответственно. Сюрпризом может стать то обстоятельство, что во всех трех случаях цепочка наследования начинается с класса object, который мы нигде не описывали. Это библиотечный класс. Его особенность в том, что те классы, которые мы описываем как обычные (и которые не наследуют другие классы), автоматически являются производными классами от класса object. Фактически это класс, который находится в вершине иерархии наследования.

Командами A=Alpha(), B=Bravo() и C=Charlie() мы создаем по одному объекту для каждого из трех классов. Все три объекта имеют доступ к полю code и у всех трех объектов есть метод alpha(). У объектов B и C есть метод bravo(), а у объекта C есть еще и метод charlie().

ПОДРОБНОСТИ

Показательным является объект C, созданный на основе класса Charlie. У этого объекта нет собственного поля code, поэтому при обращении к полю поиск поля продолжается в классе Charlie. В нем такого поля тоже нет, и поле ищется в классе Bravo — базовом классе для класса Charlie. В классе Bravo поля нет, и поиск перемещается в класс Alpha, который является базовым для класса Bravo. Поскольку в классе Alpha такое поле описано, то именно оно и используется. Аналогичная ситуация и с методами.

Для объектов A и B ситуация похожая (с поправкой на цепочки наследования для соответствующих классов).

Ситуация немного меняется после выполнения команды Bravo.code=321, которой в класс Bravo добавляется поле code со значением 321. Теперь при обращении к полю code (напрямую или через методы) в объектах B и C используется значение именно поля code из класса Bravo. Причина в том, что при перемещении по цепочке наследования в поисках поля code именно в классе Bravo оно встречается впервые. Поэтому до использования поля code из класса Alpha дело не доходит. Но если мы обращаемся к полю code из объекта A, то будет использовано поле code из класса Alpha (поиск сначала выполняется непосредственно в объекте A, а затем в классе Alpha).

Множественное наследование

Это мелочи. Но нет ничего важнее мелочей!

и к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Как уже отмечалось ранее, у производного класса может быть сразу несколько базовых классов. В этом случае говорят о *множественном наследовании*. Технически такого типа наследование реализуется так же просто, как и наследование одного базового класса. Просто при множественном наследовании в круглых скобках после имени производного класса указывается не один, а сразу несколько базовых классов (названия классов разделяются запятыми). Например, если производный класс `Delta` создается на основе базовых классов `Alpha`, `Bravo` и `Charlie`, то шаблон описания производного класса будет выглядеть так:

```
class Delta(Alpha, Bravo, Charlie):
    # Описание производного класса
```

Если у производного класса несколько базовых классов, то поиск атрибута в объекте производного класса выполняется следующим образом.

- Сначала поиск атрибута выполняется непосредственно в объекте.
- Если в объекте атрибут не найден, поиск продолжается в объекте, посредством которого реализуется производный класс.
- Если в производном классе атрибут не найден, то поиск атрибута продолжается в базовых классах. Причем последовательность перебора базовых классов определяется тем, как базовые классы указаны в описании производного класса. В частности, для описанного выше шаблона сначала будет выполняться поиск в производном классе `Delta`, а затем в базовых классах `Alpha`, `Bravo` и `Charlie`.

В листинге 9.3 представлена программа, в которой используется множественное наследование.

Листинг 9.3. Множественное наследование

```
# Первый базовый класс:
class Alpha:
    def alpha(self):
```

```
        print("Класс Alpha")
# Второй базовый класс:
class Bravo:
    def bravo(self):
        print("Класс Bravo")
# Третий базовый класс:
class Charlie:
    def charlie(self):
        print("Класс Charlie")
# Производный класс:
class Delta(Alpha, Bravo, Charlie):
    pass
# Иерархия наследования:
print("Наследование:")
k=1
for s in Delta.__mro__:
    print("["+str(k)+"]", s.__name__)
    k+=1
# Объект производного класса:
obj=Delta()
# Вызов методов:
obj.alpha()
obj.bravo()
obj.charlie()
```

Результат выполнения программы будет таким.

Результат выполнения программы (из листинга 9.3)

```
Наследование:
[1] Delta
[2] Alpha
[3] Bravo
[4] Charlie
[5] object
```

```
Класс Alpha
Класс Bravo
Класс Charlie
```

Мы описываем три класса `Alpha`, `Bravo` и `Charlie`, которые используем как базовые при создании производного класса `Delta`. Каждый базовый класс имеет метод, при вызове которого отображается название соответствующего класса. На основе класса `Delta` создается объект `obj`. При вызове метода из объекта поиск выполняется сначала непосредственно в объекте, затем в классе `Delta`, после чего последовательно проверяются классы `Alpha`, `Bravo` и `Charlie`. Как и ранее, цепочку наследования для класса можно получить с помощью поля `__mro__`. Для класса `Delta` кортеж с наследуемыми классами получаем с помощью инструкции `Delta.__mro__`. Стоит заметить, что порядок следования классов в этом кортеже соответствует последовательности, в которой классы перебираются при поиске атрибутов.

Есть еще два немаловажных обстоятельства. Во-первых, в отличие от предыдущего примера, в данном случае классы `Alpha`, `Bravo` и `Charlie` независимы (а в предыдущем примере мы имели дело с цепочкой наследования). Во-вторых, каждый из классов `Alpha`, `Bravo` и `Charlie` является производным от класса `object`. При этом класс `object` в цепочке наследования класса `Delta` встречается только один раз. Вообще, ничего особенного здесь нет. Но эта ситуация наводит на некоторые размышления. Ведь одновременное использование множественного и многократного наследования может приводить к причудливым ситуациям (в том числе и к ошибкам). Рассмотрим пример в листинге 9.4.

Листинг 9.4. Иерархия наследования

```
# Описание классов:
class Alpha:
    pass
class Bravo:
    pass
class Charlie(Alpha, Bravo):
    pass
class Delta(Bravo):
    pass
```

```
class Echo(Charlie, Delta):
    pass
# Иерархия наследования:
k=1
for s in Echo.__mro__:
   print("["+str(k)+"]", s.__name__)
   k+=1
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.4)

```
[1] Echo
[2] Charlie
[3] Alpha
[4] Delta
[5] Bravo
[6] object
```

В программе описывается несколько классов. Поскольку нас интересует только цепочка наследования, то все классы пустые. В частности, у нас есть классы `Alpha` и `Bravo`, которые являются базовыми для класса `Charlie`. Класс `Delta` создается наследованием класса `Bravo`. Класс `Echo` создается на основе классов `Charlie` и `Delta`. Для лучшего восприятия ситуации схема наследования классов представлена на рис. 9.1.

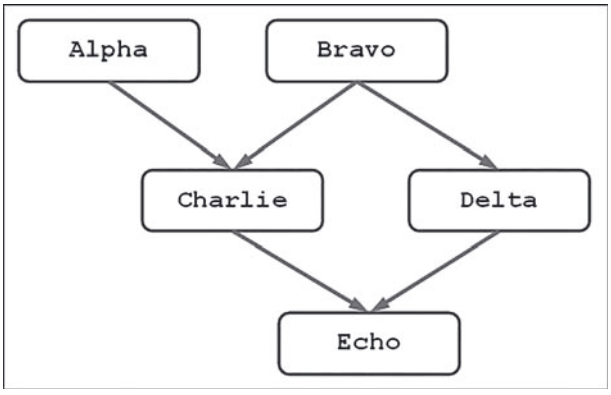


Рис. 9.1. Схема наследования классов. Стрелки означают наследование и направлены от базового класса к производному

Как мы уже знаем, последовательность, в которой в классе `Echo` наследуются прочие классы, можно узнать с помощью специального поля `__mro__`. В программе эта последовательность отображается с помощью оператора цикла, в котором перебирается кортеж `Echo.__mro__`.

Специфика ситуации в том, что класс `Bravo` наследуется в классе `Echo` дважды: через класс `Charlie` и через класс `Delta`. В таких случаях автоматически выполняется линеаризация цепочки наследования. Сам алгоритм обсуждать не будем, поскольку в данном случае это не так принципиально, да и он не очень простой. Выделим только некоторые базовые принципы, которые должны быть реализованы при формировании цепочки наследования. Она должна содержать каждый класс только один раз. Причем классы в цепочке наследования должны располагаться таким образом, чтобы базовый класс не просматривался ранее, чем производный класс, а также соблюдалась очередность просмотра базовых классов, если у производного класса их несколько. Далеко не всегда подобную цепочку возможно сформировать. Если цепочку сформировать не получается, то генерируется ошибка класса `TypeError`.

НА ЗАМЕТКУ

Например, если в рассмотренном примере сделать класс `Bravo` производным от класса `Alpha`, получим ошибку класса `TypeError`.

Переопределение методов при наследовании

> Меня не проведешь. Приемы сыщиков я вижу на пять футов вглубь.
>
> *Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»*

Итак, при наследовании производный класс получает доступ к полям и методам, описанным в базовых классах. Но иногда возникает ситуация, когда, например, унаследованный из базового класса метод следовало бы переопределить в производном классе. Скажем, в базовом классе имеется метод, который при вызове из объекта отображает значение поля этого объекта. А у объекта производного класса не одно, а два поля, значения которых следует отображать. Но унаследованный из базового класса метод про второе поле «ничего не знает». Какой выход

из ситуации? Можно в производном классе описать новый метод. Но это не очень хороший подход, поскольку если так делать каждый раз, когда нужен новый метод, то очень скоро у нас будет слишком много методов, выполняющих схожие операции. Более разумно — *переопределить* унаследованный метод. Делается это просто. В производном классе просто заново описывается метод с соответствующим именем. Как все это выглядит на практике, показано в программе в листинге 9.5.

Листинг 9.5. Переопределение методов

```
# Первый класс:
class Alpha:
    # Поле класса:
    code=123
    # Конструктор:
    def __init__(self, num):
        print("Конструктор № 1")
        self.number=num
        print("Создан объект")
        self.show()
    # Метод:
    def show(self):
        print("Метод № 1")
        print("Класс:", self.__class__.__name__)
        print("Код класса:", self.__class__.code)
        print("Поле number:", self.number)
# Второй класс:
class Bravo(Alpha):
    # Поле класса:
    code=456
# Третий класс:
class Charlie(Bravo):
    # Конструктор:
    def __init__(self, num, txt):
        print("Конструктор № 2")
```

```
        self.number=num
        self.name=txt
        print("Новый объект")
        self.show()
    # Переопределение метода:
    def show(self):
        print("Метод № 2")
        print("Класс:", self.__class__.__name__)
        print("Код класса:", self.__class__.code)
        print("Поле number:", self.number)
        print("Поле name:", self.name)
# Четвертый класс:
class Delta(Charlie):
    # Поле класса:
    code=789
# Создание объекта:
A=Alpha(100)
# Создание поля объекта:
A.code=321
print("После команды A.code=321")
# Вызов метода:
A.show()
# Создание объектов:
B=Bravo(200)
C=Charlie(300,"C")
D=Delta(400,"D")
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.5)

```
Конструктор № 1
Создан объект
Метод № 1
```

```
Класс: Alpha
Код класса: 123
Поле number: 100
После команды A.code=321
Метод № 1
Класс: Alpha
Код класса: 123
Поле number: 100
Конструктор № 1
Создан объект
Метод № 1
Класс: Bravo
Код класса: 456
Поле number: 200
Конструктор № 2
Новый объект
Метод № 2
Класс: Charlie
Код класса: 456
Поле number: 300
Поле name: C
Конструктор № 2
Новый объект
Метод № 2
Класс: Delta
Код класса: 789
Поле number: 400
Поле name: D
```

В программе описывается четыре класса, которые наследуют друг друга. В классе `Alpha` есть поле класса `code` со значением `123` и конструктор с двумя аргументами. При вызове конструктора отображаются сообщения, присваивается значение полю `number` объекта, а также вызывается

метод `show()`. Последний описан так, что при вызове отображается название класса объекта, из которого вызывается метод (использована инструкция `self.__class__.__name__`), значение поля `code` класса объекта, из которого вызывается метод (инструкция `self.__class__.code`), а также значение поля `number` объекта (инструкция `self.number`).

> **НА ЗАМЕТКУ**
>
> Стоит обратить внимание, что в данном случае выполняется именно обращение к полю класса, а не объекта. Если бы мы обращались к полю `code` в формате `self.code`, то это было бы обращение к полю объекта. Поиск поля сначала выполняется в объекте, затем в классе объекта, а после него — в базовых классах. Если мы используем инструкцию `self.__class__.code`, то теперь поле `code` запрашивается в классе, на основе которого создавался объект (из которого вызывается метод с данной инструкцией). Поиск поля будет выполняться сначала в классе объекта, а затем в базовых классах для данного класса.

Класс `Bravo` создается наследованием класса `Alpha`. В этом классе командой `code=456` определяется собственное поле `code` со значением `456`.

> **НА ЗАМЕТКУ**
>
> Важно то, что класс `Bravo` наследует поле `code` из класса `Alpha`, но командой `code=456` в классе `Bravo` определяется собственное поле `code`. Поле `code` класса `Alpha` осталось неизменным.

Больше ничего в классе `Bravo` не описано. Тем не менее из класса `Alpha` в классе `Bravo` наследуется не только метод `show()`, но и конструктор. Поэтому при создании объектов класса `Bravo` конструктору необходимо передавать аргумент (определяющий значение поля `number`).

Класс `Charlie` наследует класс `Bravo`. В классе `Charlie` по цепочке через класс `Bravo` наследуется метод `show()` и конструктор. Тем не менее мы переопределяем конструктор — теперь у него три аргумента (соответственно, при создании объекта конструктору передается два аргумента). Объект класса, кроме поля `number`, имеет еще и поле `name`. Также мы переопределяем метод `show()` (при вызове метода отображаются значения полей `number` и `name`).

НА ЗАМЕТКУ

Поле `code` в классе `Charlie` наследуется из класса `Bravo`.

Класс `Delta` создается наследованием класса `Charlie`. В классе всего одна инструкция `code=789`. Этой инструкцией в классе `Delta` создается собственное поле `code`. Конструктор и метод `show()` наследуются из класса `Charlie`.

Командой `A=Alpha(100)` создается объект класса `Alpha`. Значение поля `number` этого объекта равно `100`. Поле `code` равно `123`, и это поле класса `Alpha`. В последнем несложно убедиться: командой `A.code=321` создается поле `code` объекта `A`, значение поля равно `321`. Но при вызове метода `show()` из объекта `A` по-прежнему получаем для поля `code` значение `123`. Причина в том, что в методе `show()` запрашивается именно поле `code` класса, на основе которого создавался объект (из которого вызывается метод).

Также командами `B=Bravo(200)`, `C=Charlie(300,"C")` и `D=Delta(400,"D")` создаются объекты прочих классов. Поскольку в конструкторах каждый раз вызывается метод `show()`, мы в результате получаем последовательность сообщений в области вывода, которые позволяют понять, какой конструктор использовался и какая версия метода `show()` вызывалась.

ПОДРОБНОСТИ

Что касается класса `Bravo`, то в нем наследуется конструктор и метод `show()` из класса `Alpha`. При этом поле `code` используется собственное. В классе `Charlie` и конструктор, и метод `show()` переопределены (описаны заново). Именно они используются. Поле `code`, однако, наследуется из класса `Bravo`. В классе `Delta` конструктор и метод `show()` наследуются из класса `Charlie`, но поле `code` в классе `Delta` описано свое.

Вообще, идея переопределения методов достаточно проста и понятна. Вместе с тем могут быть не самые очевидные ситуации. Рассмотрим их на примерах. Обратимся к программе в листинге 9.6.

Листинг 9.6. Виртуальность методов

```
# Базовый класс:
class Alpha:
```

```
    # Методы:
    def display(self):
        print("Метод из Alpha")
        print("Поле code:", self.code)
    def show(self):
        self.display()
# Производный класс:
class Bravo(Alpha):
    # Переопределение метода:
    def display(self):
        print("Метод из Bravo")
        print("Поле name:", self.name)
# Создание объектов:
A=Alpha()
A.code=123
B=Bravo()
B.name="B"
# Вызов методов:
A.show()
B.show()
```

Результат выполнения программы будет таким.

Результат выполнения программы (из листинга 9.6)

```
Метод из Alpha
Поле code: 123
Метод из Bravo
Поле name: B
```

Программа достаточно простая. В базовом классе `Alpha` описан метод `display()`, при вызове которого отображается сообщение с названием класса `Alpha`, а также значение поля `code` объекта, из которого вызывается метод. В классе `Alpha` есть еще и метод `show()`, при вызове которого на самом деле вызывается метод `display()`.

На основе базового класса `Alpha` создается производный класс `Bravo`. В этом классе мы переопределяем метод `display()`. Теперь при вызове метода отображается сообщение с названием класса `Bravo` и значение поля `name` объекта, из которого метод вызывается. При этом метод `show()` в классе `Bravo` мы не переопределяем, и в классе `Bravo` данный метод наследуется из класса `Alpha`. Если вызывать метод `show()` из объекта класса `Bravo`, то вызываться будет метод `display()`. Но какая его версия? Та, что описана в том же классе, что и метод `show()`, или та, что описана в классе, на основе которого создавался объект? Результат выполнения программы показывает, что, хотя метод `show()` наследуется в классе `Bravo` из класса `Alpha`, в теле этого метода вызывается версия метода `display()`, описанная в классе `Bravo`. Таким образом, версия метода при вызове определяется объектом, из которого вызывается метод. Это свойство методов называется *виртуальностью*. В Python все методы виртуальные.

При переопределении методов часто задача состоит в том, чтобы доопределить унаследованный из базового класса метод. Другими словами, даже если метод переопределяется, желательно иметь доступ и к «старой» (унаследованной из базового класса) версии метода. Если так, то полезной будет функция `super()`, с помощью которой можно вызывать методы, описанные в базовом классе.

ПОДРОБНОСТИ

При вызове функции создается специальный (промежуточный) прокси-объект, через который выполняется вызов нужного метода из базового класса. Функция `super()` может вызываться без аргументов и с двумя аргументами: первый определяет название класса, по отношению к которому определяется базовый класс, а второй аргумент определяет ссылку на объект, из которого будет вызываться метод базового класса.

Также не следует забывать, что в случае необходимости вызвать версию метода из некоторого класса (базового или нет) мы можем явно указать название класса в инструкции вызова метода.

Небольшой пример, в котором используется инструкция `super()`, представлен в листинге 9.7.

Листинг 9.7. Использование инструкции `super()`

```
# Первый класс:
class Alpha:
```

```
    # Конструктор:
    def __init__(self, num):
        self.code=num
        print("Присвоено значение полю code")
    # Метод:
    def show(self):
        print("Поле code:", self.code)
# Второй класс:
class Bravo(Alpha):
    # Конструктор:
    def __init__(self, num, txt):
        # Вызов конструктора базового класса:
        super().__init__(num)
        self.name=txt
        print("Присвоено значение полю name")
    # Метод:
    def show(self):
        # Вызов метода из базового класса:
        super().show()
        print("Поле name:", self.name)
# Третий класс:
class Charlie(Bravo):
    # Конструктор:
    def __init__(self, num, txt, val):
        # Вызов конструктора базового класса:
        super().__init__(num, txt)
        self.value=val
        print("Присвоено значение полю value")
    # Метод:
    def show(self):
        # Вызов метода из базового класса:
        super().show()
```

```
        print("Поле value:", self.value)
# Создание объектов и вызов методов:
print("Объект A")
A=Alpha(100)
A.show()
print("Объект B")
B=Bravo(200,"B")
B.show()
print("Объект C")
C=Charlie(300,"C",[1,2,3])
C.show()
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.7)

```
Объект A
Присвоено значение полю code
Поле code: 100
Объект B
Присвоено значение полю code
Присвоено значение полю name
Поле code: 200
Поле name: B
Объект C
Присвоено значение полю code
Присвоено значение полю name
Присвоено значение полю value
Поле code: 300
Поле name: C
Поле value: [1, 2, 3]
```

Класс `Alpha` содержит конструктор, в котором присваивается значение полю `code` и отображается сообщение о создании соответствующего

поля. Метод `show()` позволяет отобразить значение поля `code` объекта, из которого вызывается метод.

Класс `Bravo` создается на основе класса `Alpha`. В нем также описан конструктор и переопределяется метод `show()`. Инструкция `super().__init__(num)` в теле конструктора означает, что вызывается конструктор из базового класса (в данном случае это класс `Alpha`). Инструкция `super().show()` в теле метода `show()` означает вызов версии этого метода из базового класса. Таким образом, при переопределении конструктора и метода мы не переписываем код заново, а фактически дополняем уже существующий.

Аналогично поступаем и при создании класса `Charlie`, наследующего класс `Bravo`. Как и в предыдущем случае, инструкция `super().__init__(num, txt)` означает вызов конструктора базового класса, но на этот раз это класс `Bravo`. Для вызова версии метода `show()`, описанной в базовом классе, использована инструкция `super().show()`.

На основе каждого из классов создается объект, и из этих объектов вызывается метод `show()`. Хочется верить, что результат выполнения программы комментариев не требует.

Еще один пример, в котором переопределяется метод, и вызываются его разные версии, представлен в листинге 9.8.

Листинг 9.8. Вызов разных версий метода

```
# Первый класс:
class Alpha:
    # Конструктор:
    def __init__(self, num):
        self.code=num
    # Метод:
    def show(self):
        print("Класс Alpha:", self.code)
# Второй класс:
class Bravo(Alpha):
    # Переопределение метода:
    def show(self):
```

```
        print("Класс Bravo:", self.code)
        super().show()
# Третий класс:
class Charlie(Alpha):
    # Переопределение метода:
    def show(self):
        print("Класс Charlie:", self.code)
        super(Charlie, self).show()
# Четвертый класс:
class Delta(Bravo, Charlie):
    # Переопределение метода:
    def show(self):
        print("Класс Delta:", self.code)
        super().show()
        Charlie.show(self)
        super(Bravo, self).show()
# Функция для отображения цепочки наследования:
def display(MyClass):
    print("Наследование для "+MyClass.__name__+":")
    k=1
    for s in MyClass.__mro__:
        print("["+str(k)+"]", s.__name__)
        k+=1
# Отображение цепочек наследования,
# создание объектов и вызов метода:
display(Alpha)
A=Alpha(100)
A.show()
display(Bravo)
B=Bravo(200)
B.show()
display(Charlie)
```

```
C=Charlie(300)
C.show()
display(Delta)
D=Delta(400)
D.show()
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.8)

```
Наследование для Alpha:
[1] Alpha
[2] object
Класс Alpha: 100
Наследование для Bravo:
[1] Bravo
[2] Alpha
[3] object
Класс Bravo: 200
Класс Alpha: 200
Наследование для Charlie:
[1] Charlie
[2] Alpha
[3] object
Класс Charlie: 300
Класс Alpha: 300
Наследование для Delta:
[1] Delta
[2] Bravo
[3] Charlie
[4] Alpha
[5] object
Класс Delta: 400
Класс Bravo: 400
```

```
Класс Charlie: 400
Класс Alpha: 400
Класс Charlie: 400
Класс Alpha: 400
Класс Charlie: 400
Класс Alpha: 400
```

Мы используем четыре класса: класс `Delta` наследует классы `Bravo` и `Charlie`, а они, в свою очередь, являются производными классами от класса `Alpha`.

НА ЗАМЕТКУ

Цепочка наследования классов для класса `Delta` выглядит так: `Delta`, `Bravo`, `Charlie`, `Alpha`, `object`.

В классе `Alpha` описан конструктор, который наследуется во всех прочих классах. При создании объекта каждого из классов конструктору передается аргумент. Он присваивается в качестве значения полю `code` объекта, который создается на основе соответствующего класса. Также в классе описан метод `show()`. При вызове метода отображается название класса `Alpha` и значение поля `code` объекта, из которого вызывается метод.

В классах `Bravo`, `Charlie` и `Delta` метод `show()` переопределяется, причем так, что вызываются разные версии метода из других классов. В классе `Bravo` метод `show()` отображает название класса `Bravo`, значение поля `code` объекта (из которого вызывается метод), но кроме этого командой `super().show()` вызывается версия метода, определенная в базовом классе. Причем поскольку мы используем множественное наследование, то какой это будет класс, зависит от цепочки наследования.

НА ЗАМЕТКУ

Если речь идет о многократном наследовании, когда у каждого производного класса только один базовый, мы имеем дело с линейной цепочкой наследования. В этом случае особой интриги нет: у каждого производного класса один базовый, и что это за класс, понятно из описания производного класса.

Если используется множественное наследование, то определить, в какой последовательности будут просматриваться классы, может

быть не очень просто. В некоторых случаях это невозможно (если так, то генерируется ошибка класса `TypeError`). Как выглядит цепочка наследования в случае использования множественного наследования можно узнать с помощью поля `__mro__`.

Класс `Charlie` наследует класс `Alpha`. Метод `show()` переопределяется так, что при вызове отображается название класса, значение поля `code` и выполняется команда `super(Charlie, self).show()`. Здесь функция `super()` вызывается с двумя аргументами, которые означают следующее: вызывается версия метода `show()` из класса, который является базовым для класса `Charlie`, и в качестве аргумента методу передается ссылка на объект `self`.

Класс `Delta` наследует классы `Bravo` и `Charlie`. Метод `show()` в классе `Delta` переопределен таким образом, что помимо отображения названия класса и значения поля `code`, еще выполняются команды `super().show()`, `Charlie.show(self)` и `super(Charlie, self).show()`. Команда `super().show()` означает вызов версии метода `show()` и базового класса. Инструкция `Charlie.show(self)` означает вызов из класса `Charlie` метода `show()` с аргументом `self` (фактически, вызов версии метода, описанной в классе `Charlie`). Команда `super(Bravo, self).show()` означает вызов версии метода из класса, базового для класса `Bravo`.

Помимо четырех классов, в программе описана функция `display()`. При вызове функции для класса, переданного аргументом функции, отображается цепочка наследования.

Далее для каждого из классов создается объект и вызывается метод `show()`. Некоторые случаи не очень тривиальные.

- При вызове метода из объекта класса `Alpha` просто появляется сообщение с названием класса и значением поля `code`.
- При вызове метода из объекта класса `Bravo` также появляется сообщение с названием класса, значением поля `code`, а также вследствие выполнения инструкции `super().show()` вызывается версия метода `show()` из класса `Alpha`, поскольку именно класс `Alpha` является базовым для класса `Bravo`.
- Похожая ситуация имеет место при вызове метода `show()` из объекта класса `Charlie`. Поскольку базовым для класса `Charlie` является класс `Alpha`, то в результате выполнения инструкции

`super(Charlie, self).show()` вызывается версия метода `show()`, описанная в классе `Alpha`.

- Сюрпризы появляются при вызове метода `show()` из объекта класса `Delta`. Чтобы понять результат, нужно принять в расчет цепочку наследования для класса `Delta`. А именно, сразу после класса `Delta` находится класс `Bravo`. Поэтому выполнение команда `super().show()` означает вызов метода `show()` и класса `Bravo`. В этой версии метода выполняется инструкция `super().show()`, которая ранее означала вызов метода `show()` из класса `Alpha`. Но это если бы мы вызывали метод из объекта класса `Bravo`. А сейчас метод вызывается из объекта класса `Delta`. А для класса `Delta` в цепочке наследования находится класс `Charlie`. Поэтому вызывается метод `show()` из этого класса. А этот метод, в свою очередь, вызывает версию метода из базового класса для класса `Charlie` (это класс `Alpha`). Далее командой `Charlie.show(self)` вызывается метод `show()` из класса `Charlie`, что влечет вызов метода `show()` из класса `Alpha`. Точно то же происходит при выполнении команды `super(Bravo, self).show()`, поскольку в этом случае вызывается метод из базового класса для класса `Bravo`. Здесь термин «базовый» следует понимать в том смысле, что это предыдущий класс в цепочке наследования. Таким классом (в цепочке наследования для класса `Delta`) является класс `Charlie`. Ситуация повторяется.

Приведение типов

> Ну что вы теперь на это скажете, мой дорогой психолог?
>
> *Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»*

Далее мы познакомимся с некоторыми специальными методами, использование которых значительно повышает гибкость программного кода. Такие методы достаточно легко узнать: их названия начинаются и заканчиваются двойными символами подчеркивания.

> **НА ЗАМЕТКУ**
>
> С двумя специальными методами мы уже сталкивались: это конструктор `__init__()` и деструктор `__del__()`.

Специальные методы обычно вызываются автоматически, если складываются «определенные обстоятельства» (хотя в принципе их можно вызывать и как обычные методы).

Есть группа методов, которые позволяют приводить объекты, созданные на основе описанных нами классов, к таким типам данных, как числа (целые, действительные и комплексные), логические значения или текст. Сразу начнем с небольшого примера, представленного в листинге 9.9.

Листинг 9.9. Приведение типов

```
# Класс со специальными методами:
class MyClass:
    # Конструктор:
    def __init__(self, val):
        self.value=val
    # Метод для приведения к текстовому типу:
    def __str__(self):
        return "Поле "+str(self.value)
    # Метод для приведения к логическому типу:
    def __bool__(self):
        if type(self.value)==int:
            return True
        else:
           return False
    # Метод для приведения к целочисленному типу:
    def __int__(self):
        if self:
            return self.value
        else:
            return 0
# Создание объектов и проверка методов:
print("Объект A:")
A=MyClass(100)
print(A)
```

```
print("Число", int(A))
print("A — 1 =", int(A)-1)
print("Объект B:")
B=MyClass("B")
print(B)
print("Число", int(B))
print("B + 1 =", int(B)+1)
```

Результат выполнения программы будет следующим.

Результат выполнения программы (из листинга 9.9)

```
Объект A:
Поле 100
Число 100
A — 1 = 99
Объект B:
Поле B
Число 0
B + 1 = 1
```

Мы описываем класс `MyClass`, у которого есть несколько специальных методов. Конструктор (специальный метод `__init__()`) описан так, что при создании объекта класса необходимо указать значение, которое будет присвоено полю `value` объекта.

Специальный метод `__str__()` вызывается в тех случаях, когда объект приводится к текстовому типу. Это может быть автоматическое приведение как в случае передачи объекта аргументом функции `print()`, так и явное приведение, когда объект передается в качестве аргумента функции `str()`. В нашем случае метод `__str__()` описан так, что результатом возвращается значение выражения `"Поле "+str(self.value)` (результат объединения текста `"Поле "` и текстового представления для поля `value` объекта, из которого вызывается метод).

Метод `__bool__()` предназначен для приведения объекта к логическому типу. Метод автоматически вызывается, если объект указан в качестве условия в операторе цикла или условном операторе. Также метод

вызывается, если объект передан аргументом функции `bool()`. В нашем случае реализовано такое правило приведения объекта к логическому типу: если поле `value` целочисленное, то объект «истинный» (метод `__bool__()` возвращает значение `True`). В противном случае объект «ложный» (метод `__bool__()` возвращает значение `False`).

Метод `__bool__()` неявно использован в описании специального метода `__int__()`, предназначенного для приведения объекта к целочисленному формату (метод вызывается в случае, если объект передается в качестве аргумента функции `int()`). Действительно, в условии в условном операторе `if` указана ссылка на объект `self`. В этом случае будет автоматически вызываться метод `__bool__()`. Условие будет истинным, если метод `__bool__()` возвращает результатом значение `True`, а это происходит, если поле `value` объекта целочисленное. В таком случае метод `__int__()` возвращает в качестве результата значение поля `value` объекта. В противном случае метод `__int__()` возвращает значение `0`.

В программе создается два объекта класса `MyClass`: у объекта `A` значение поля `value` равно `100`, а у объекта `B` значение поля `value` равно `"B"`.

Метод `__str__()` автоматически вызывается при выполнении команд `print(A)` и `print(B)`. Метод `__int__()` вызывается при вычислении значений выражений `int(A)` и `int(B)`.

НА ЗАМЕТКУ

Кроме рассмотренных методов `__str__()`, `__bool__()` и `__int__()`, есть и другие специальные методы, предназначенные для приведения объектов к разным типам. А именно, специальный метод `__float__()` предназначен для приведения объектов к формату числа с плавающей точкой (метод вызывается при вызове функции `float()`, аргументом которой передается объект). Метод `__complex__()` предназначен для приведения объекта к формату комплексного числа (метод вызывается при вызове функции `complex()` с аргументом-объектом).

Также стоит отметить, что есть несколько методов, полезных в плане определения способов преобразования объектов. Специальный метод `__len__()` вызывается в том случае, если вызывается функция `len()`, в качестве аргумента которой передан объект соответствующего класса. Метод `__index__()` вызывается в тех случаях, когда объект передается аргументом функциям `bin()`, `oct()` и `hex()`. Фактически методом `__index__()` возвращается

числовое значение, которое переводится в бинарное, восьмеричное или шестнадцатеричное представление. Оно и возвращается соответственно функциями `bin()`, `oct()` и `hex()`.
Также иногда полезно использовать специальный метод `__round__()`, который автоматически вызывается, если объект передается в качестве аргумента функции `round()`.

Перегрузка операторов

> Холмс, это исключено. Сразу видно, что вы мало читаете.
>
> *Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»*

В предыдущем примере для выполнения арифметических операций с участием объектов мы сначала с помощью функции `int()` явно выполняли приведение объектов к целочисленному формату. Вместе с тем существует возможность с помощью специальных методов «научить» программу выполнять арифметические (и некоторые другие) операции с объектами. В этом случае речь идет о *перегрузке операторов*.

Идея очень простая. Допустим, у нас есть некоторый объект и у него, например, имеется поле с целочисленным значением. Мы хотим добиться того, чтобы к объекту можно было прибавлять число и чтобы в результате это число прибавлялось к целочисленному полю объекта. Для реализации такого нехитрого плана достаточно в классе, на основе которого создается объект, описать специальный метод `__add__()`. По этой схеме можно определять и другие арифметические операции для объектов. Также можно перегружать побитовые операторы и операторы сравнения. Для начала рассмотрим небольшой пример, представленный в листинге 9.10.

Листинг 9.10. Знакомство с перегрузкой операторов

```
# Класс со специальными методами:
class MyClass:
    # Конструктор:
    def __init__(self, num):
        self.code=num
```

```
    # Метод для преобразования к текстовому формату:
    def __str__(self):
        return str(self.code)
    # Метод для перегрузки оператора сложения:
    def __add__(self, n):
        if type(n)==int:
            val=self.code+n
        else:
            val=0
        return MyClass(val)
# Создание объекта и проверка методов:
A=MyClass(100)
print("Объект A:", A)
# К объекту прибавляется число:
B=A+25
print("Объект B:", B)
# К объекту прибавляется текст:
C=A+"Hello"
print("Объект C:", C)
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.10)

```
Объект A: 100
Объект B: 125
Объект C: 0
```

Для удобства в классе `MyClass` описан конструктор и метод `__str__()` (используется для автоматического приведения объектов класса к текстовому формату — результатом возвращается текстовое представление для поля `code` объекта). Кроме этого, мы описали в классе еще и метод `__add__()`. У метода два аргумента. Первый традиционно обозначает объект, из которого вызывается метод. Второй аргумент метода соответствует прибавляемому к объекту значению.

ПОДРОБНОСТИ

Метод `__add__()` предназначен для обработки выражений вида «объект плюс что-то». Причем именно в таком порядке. Для обработки выражений вида «что-то плюс объект» нужно описать в классе объекта метод `__radd__()`.

Метод `__add__()` вызывается из объекта, к которому прибавляется «что-то». Ссылка на этот объект и есть первый аргумент метода. Второй аргумент метода — «что-то», что прибавляется к объекту.

Метод `__add__()` описан достаточно просто. В условном операторе проверяется тип второго аргумента (значение, прибавляемое к объекту). Если это целое число, то командой `val=self.code+n` вычисляется (и записывается в переменную `val`) сумма поля `code` объекта и этого числа. Если к объекту прибавляется не число, то командой `val=0` переменной `val` присваивается нулевое значение. После того как значение переменной `val` определено, командой `return MyClass(val)` в качестве результата метод `__add__()` возвращает ссылку на созданный объект, и поле `code` этого объекта определяется значением переменной `val`.

НА ЗАМЕТКУ

Таким образом, если мы к объекту класса `MyClass` прибавляем целое число, то в результате получаем ссылку на новый объект класса `MyClass`, и поле `code` этого объекта равно сумме поля `code` исходного объекта и второго целочисленного операнда. Если к объекту класса `MyClass` прибавляется не целочисленное значение (например, текст), то в качестве результата получаем новый объект класса `MyClass` с нулевым значением поля `code`. Исходный объект в любом случае не меняется.

Командой `A=MyClass(100)` создаем объект класса `MyClass` со значением `100` для поля `code`. При выполнении команды `B=A+25` создается новый объект, и ссылка на него записывается в переменную `B`. Поле `code` этого объекта равно `125` (сумма значения поля `code` объекта `A` и числа `25`).

В результате выполнения команды `C=A+"Hello"` также получаем новый объект, но поскольку к объекту `A` прибавлялось не целое число, а текст `"Hello"`, то поле `code` объекта `C` будет иметь нулевое значение.

НА ЗАМЕТКУ

Еще раз подчеркнем, что мы «научили» программу прибавлять к объекту некоторый операнд. Чтобы прибавлять к операнду объект, следует определить еще и метод `__radd__()`.

Следующий небольшой пример также иллюстрирует ситуацию, когда в программе используется перегрузка операторов. Рассмотрим программу в листинге 9.11.

Листинг 9.11. Перегрузка операторов

```
# Класс с перегрузкой операторов:
class Alpha:
    # Конструктор:
    def __init__(self, lst):
        self.vals=[]
        if type(lst)==list:
            for n in lst:
               self.vals.append(n)
    # Метод для приведения к текстовому формату:
    def __str__(self):
        return str(self.vals)
    # Унарный оператор "плюс":
    def __pos__(self):
        x=self.vals[0]
        del self.vals[0]
        self.vals.append(x)
        return self
    # Унарный оператор "минус":
    def __neg__(self):
        x=self.vals[-1]
        del self.vals[-1]
        self.vals.insert(0, x)
        return self
```

```
    # Умножение объекта на операнд:
    def __mul__(self, v):
        self.vals.append(v)
        return self
    # Умножение операнда на объект:
    def __rmul__(self, v):
        self.vals.insert(0, v)
        return self
    # Сокращенная форма операции умножения:
    def __imul__(self, v):
        return self*v
# Создание объекта:
A=Alpha([1,"A",2])
# Выполнение операций с объектом:
print(A)
print(+A)
print(-A)
print(A*3)
print(4*A)
A*="Alpha"
print(A)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 9.11)

```
[1, 'A', 2]
['A', 2, 1]
[1, 'A', 2]
[1, 'A', 2, 3]
[4, 1, 'A', 2, 3]
[4, 1, 'A', 2, 3, 'Alpha']
```

В рассматриваемой программе описывается класс `Alpha` с конструктором `__init__()` и методом `__str__()`. Мы исходим из того, что

у объекта класса должно быть поле vals, являющееся ссылкой на список. Конструктору при создании объекта передается список значений, и эти значения поэлементно копируются в список, на который ссылается поле vals. При приведении объекта к текстовому формату результатом возвращается текстовая строка с содержимым списка.

В программе на основе класса Alpha создается объект A с полем vals, ссылающимся на список [1,"A",2] (команда A=Alpha([1,"A",2])). С этим объектом выполняются различные операции. В частности, к объекту применяется операция «унарный плюс» (речь о выражении +A). Эта операция обрабатывается специальным методом __pos__().

ПОДРОБНОСТИ

Мы знаем, что сложение (например, A+B) — операция бинарная, в ней участвует два операнда (A и B). Но, кроме «бинарного плюса», есть еще и «унарный плюс». В арифметике обычно он не используется, но формально имеет право на существование. Речь о выражениях вида +A, когда перед числом (или объектом в общем случае) как признак «положительности» указывается знак «плюс».

Это же замечание относится и к оператору «минус». Есть бинарная операция вычитания (например, A-B), а есть унарная операция вида -A, когда оператор «минус» используется как признак отрицательного значения. И унарные, и бинарные операторы «плюс» и «минус» можно применять к объектам. Но обрабатываются эти операции разными специальными методами.

Метод __pos__() описан так, что в объекте, из которого вызывается метод, первый элемент в списке vals становится последним элементом. Для этого командой x=self.vals[0] запоминается значение первого элемента в списке, командой del self.vals[0] этот элемент удаляется, а командой self.vals.append(x) значение первого элемента добавляется в конец списка. Результатом метода возвращается ссылка на объект, из которого вызывается метод (инструкция return self). Поэтому в результате выполнения инструкции +A в объекте A первый элемент списка vals перемещается на последнюю позицию, причем значение выражения +A является ссылкой на тот же объект, на который ссылается переменная A. Поэтому при выполнении команды print(+A) отображается новое содержимое объекта A.

За выполнение операции -A «отвечает» метод __neg__(). Операция выполняется так, что последний элемент из списка vals перемещается

на первую позицию, а в качестве результата метод возвращает ссылку на объект, из которого вызывался.

Помимо перечисленных методов, мы описываем методы `__mul__()` (умножение с помощью оператора `*` объекта на операнд), `__rmul__()` (умножение с помощью оператора `*` операнда на объект), а также метод `__imul__()` (обрабатывается операция сокращенного умножения с помощью оператора `*=`).

При умножении объекта на операнд этот операнд дописывается в конец списка `vals` объекта, из которого вызывается метод `__mul__()`. При умножении операнда на объект значение операнда дописывается в начало списка `vals` объекта. В обоих случаях результатом возвращается ссылка на объект.

Метод `__imul__()` описан так, что при аргументах `self` (ссылка на объект, из которого вызывается метод) и `v` (операнд справа от оператора `*=`) результатом возвращается значение выражения `self*v`. Это выражение представляет собой произведение объекта на операнд. Такое выражение обрабатывается вызовом метода `__mul__()`. Поэтому получается, что метод `__imul__()` на самом деле вызывает (неявно) метод `__imul__()` для вычисления результата.

Таким образом, при вычислении вычисления выражения `A*3` в конец списка `vals` объекта `A` дописывается число `3`. Вычисление инструкции `4*A` приводит к тому, что число `4` дописывается в начало списка `vals` объекта `A`. Наконец, при выполнении команды `A*="Alpha"` текст `"Alpha"` дописывается в конец списка `vals`. Во всех случаях результатом соответствующего выражения является ссылка на объект `A`.

Вообще, в Python достаточно много операторов, которые можно перегружать. В табл. 9.1 перечислены методы, которыми перегружаются арифметические операторы.

Табл. 9.1. Методы для перегрузки арифметических операторов

Метод	Оператор	Операция
`__add__()`	`+`	`объект+операнд`
`__radd__()`	`+`	`операнд+объект`
`__iadd__()`	`+=`	`объект+=операнд`
`__sub__()`	`-`	`объект-операнд`

Метод	Оператор	Операция
`__rsub__()`	`-`	`операнд-объект`
`__isub__()`	`-=`	`объект-=операнд`
`__mul__()`	`*`	`объект*операнд`
`__rmul__()`	`*`	`операнд*объект`
`__imul__()`	`*=`	`объект*=операнд`
`__truediv__()`	`/`	`объект/операнд`
`__rtruediv__()`	`/`	`операнд/объект`
`__itruediv__()`	`/=`	`объект/=операнд`
`__floordiv__()`	`//`	`объект//операнд`
`__rfloordiv__()`	`//`	`операнд//объект`
`__ifloordiv__()`	`//=`	`объект//=операнд`
`__mod__()`	`%`	`объект%операнд`
`__rmod__()`	`%`	`операнд%объект`
`__imod__()`	`%=`	`объект%=операнд`
`__pow__()`	`**`	`объект**операнд`
`__rpow__()`	`**`	`операнд**объект`
`__ipow__()`	`**=`	`объект**=операнд`
`__neg__()`	`-`	`-объект`
`__pos__()`	`+`	`+объект`
`__abs__()`	`abs()`	`abs(объект)`

Кроме арифметических, можно перегружать побитовые операторы. Соответствующие методы перечислены в табл. 9.2.

Табл. 9.2. Методы для перегрузки побитовых операторов

Метод	Оператор	Операция
`__invert__()`	`~`	`~объект`
`__and__()`	`&`	`объект&операнд`
`__rand__()`	`&`	`операнд&объект`
`__iand__()`	`&=`	`объект&=операнд`
`__or__()`	`|`	`объект|операнд`

Табл. 9.2. Методы для перегрузки побитовых операторов. Продолжение

Метод	Оператор	Операция
`__ror__()`	`|`	`операнд|объект`
`__ior__()`	`|=`	`объект|=операнд`
`__xor__()`	`^`	`объект^операнд`
`__rxor__()`	`^`	`операнд^объект`
`__ixor__()`	`^=`	`объект^=операнд`
`__lshift__()`	`<<`	`объект<<операнд`
`__rlshift__()`	`<<`	`операнд<<объект`
`__ilshift__()`	`<<=`	`объект<<=операнд`
`__rshift__()`	`>>`	`объект>>операнд`
`__rrshift__()`	`>>`	`операнд>>объект`
`__irshift__()`	`>>=`	`объект>>=операнд`

В некоторых случаях полезно также перегружать операторы сравнения. Методы для перегрузки этих операторов перечислены в табл. 9.3.

Табл. 9.3. Методы для перегрузки операторов сравнения

Метод	Оператор	Операция
`__eq__()`	`==`	`объект==операнд` или `операнд==объект`
`__ne__()`	`!=`	`объект!=операнд` или `операнд!=объект`
`__lt__()`	`<`	`объект<операнд` или `операнд>объект`
`__gt__()`	`>`	`объект>операнд` или `операнд<объект`
`__le__()`	`<=`	`объект<=операнд` или `операнд>=объект`
`__ge__()`	`>=`	`объект>=операнд` или `операнд<=объект`
`__contains__()`	`in`	`операнд in объект`

В общем, операторы сравнения перегружаются так же, как и прочие операторы. Общепринято, что операторы сравнения в качестве результата возвращают логическое значение, но это не обязательно.

НА ЗАМЕТКУ

Если в классе определен метод `__eq__()`, но не описан метод `__ne__()`, то при выполнении команды вида `объект!=оператор` (или `оператор!=объект`) вызывается метод `__eq__()`, после чего к результату, возвращаемому этим оператором, применяется процедура логического отрицания.

Небольшая иллюстрация к перегрузке операторов сравнения представлена в программе в листинге 9.12.

Листинг 9.12. Перегрузка операторов сравнения

```
# Первый класс:
class Alpha:
    # Конструктор:
    def __init__(self, val):
        self.value=val
    # Метод для оператора "равно":
    def __eq__(self, val):
        print("Alpha: 'равно'")
        return self.value==val
    # Метод для оператора "не равно":
    def __ne__(self, val):
        print("Alpha: 'не равно'")
        return self.value!=val
    # Метод для оператора "меньше":
    def __lt__(self, val):
        print("Alpha: 'меньше'")
        return self.value<val
    # Метод для оператора "больше или равно":
    def __ge__(self, val):
        print("Alpha: 'больше или равно'")
        return self.value>=val
# Второй класс:
class Bravo:
```

```
    # Конструктор:
    def __init__(self, val):
        self.value=val
    # Метод для оператора "равно":
    def __eq__(self, val):
        print("Bravo: 'равно'")
        return self.value==val
# Создание объектов и выполнение сравнений:
A=Alpha(100)
print("Операции с объектом A")
print("[01] A==100:", A==100)
print("[02] A!=100:", A!=100)
print("[03] 200==A:",200==A)
print("[04] 200!=A:",200!=A)
print("[05] A<200:", A<200)
print("[06] 200>A:",200>A)
print("[07] A>=200:", A>=200)
print("[08] 100<=A:",100<=A)
B=Bravo(300)
print("Операции с объектом B")
print("[9] B==300:", B==300)
print("[10] B!=300:", B!=300)
print("[11] 400==B:",400==B)
print("Сравнение объектов A и B")
print("[12] A==B:", A==B)
print("[13] B!=A:", B!=A)
print("[14] A!=B:", A!=B)
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.12)

```
Операции с объектом A
Alpha: 'равно'
```

```
[01] A==100: True
Alpha: 'не равно'
[02] A!=100: False
Alpha: 'равно'
[03] 200==A: False
Alpha: 'не равно'
[04] 200!=A: True
Alpha: 'меньше'
[05] A<200: True
Alpha: 'меньше'
[06] 200>A: True
Alpha: 'больше или равно'
[07] A>=200: False
Alpha: 'больше или равно'
[08] 100<=A: True
Операции с объектом B
Bravo: 'равно'
[9] B==300: True
Bravo: 'равно'
[10] B!=300: False
Bravo: 'равно'
[11] 400==B: False
Сравнение объектов A и B
Alpha: 'равно'
Bravo: 'равно'
[12] A==B: False
Bravo: 'равно'
Alpha: 'равно'
[13] B!=A: True
Alpha: 'не равно'
Bravo: 'равно'
[14] A!=B: True
```

Код объемный, но в целом простой. Для удобства все специальные методы при вызове отображают сообщения, чтобы можно было отследить последовательность вызова методов. Также команды, в которых выполняется сравнение объектов и значений, пронумерованы. Комментариев заслуживает, пожалуй, лишь блок команд, связанных со сравнением объектов A и B.

Объект A создается на основе класса Alpha. В этом классе, помимо прочего, описаны методы __eq__() и __ne__(). Объект B создается на основе класса Bravo. В этом классе описан метод __eq__(), но не метод __ne__().

При вычислении инструкции A==B метод __eq__() вызывается из объекта A. В этом методе сравнивается значение поля value объекта A (а это целое число) со вторым операндом, которым является объект B. Для обработки такой инструкции из объекта B вызывается метод __eq__(), результат которого и дает результат всего выражения.

Для вычисления значения выражения B!=A, поскольку в классе Bravo метод __ne__() не определен, вызывается метод __eq__(). Поле value объекта B сравнивается, на предмет равенства, с объектом A. Для вычисления этого выражения из объекта A вызывается метод __eq__(). К полученному результату применяется процедура логического отрицания.

Несколько иная последовательность действий при вычислении выражения A!=B. В классе Alpha описан метод __ne__(), в результате чего сравниваются на предмет неравенства поле value объекта A и объект B. Поскольку для класса Bravo метод __ne__() не описан, то из объекта B вызывается метод __eq__(), выполняется сравнение на предмет равенства объекта B и поля value объекта A, а к результату применяется процедура логического отрицания.

Доступ к атрибутам

— Жаль, что мы не успели заметить его номер.
— Не мы, а вы, дорогой друг.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Есть группа специальных методов, которые автоматически вызываются при обращении к атрибутам объекта. В частности, при попытке прочитать значение атрибута вызывается метод __getattribute__().

Кроме ссылки на объект, атрибут которого запрашивается, методу также передается название запрашиваемого атрибута. Результат, возвращаемый методом, отождествляется со значением атрибута. Если запрашиваемого атрибута у объекта нет, автоматически вызывается метод `__getattr__()` (в качестве аргументов методу передается ссылка на объект и название запрашиваемого атрибута). Если этот метод в классе не описан, то все заканчивается генерированием исключения класса `AttributeError`.

ПОДРОБНОСТИ

Схема в общих чертах такая. При обращении к атрибуту объекта в формате точечного синтаксиса автоматически вызывается метод `__getattribute__()`. Метод результатом возвращает значение атрибута. Если запрашиваемого атрибута у объекта нет, то метод `__getattribute__()` генерирует исключение класса `AttributeError`. Это является сигналом для автоматического вызова метода `__getattr__()`. Поэтому в методе `__getattr__()` размещается код, предназначенный для выполнения в случае, если запрашивается несуществующий атрибут.

В листинге 9.13 представлена программа, в которой используются специальные методы `__getattribute__()` и `__getattr__()`.

Листинг 9.13. Считывание значения поля

```
# Класс со специальными методами:
class Alpha:
    # Метод для получения доступа к атрибуту:
    def __getattribute__(self, name):
        print("Alpha: запрос поля", name)
        return object.__getattribute__(self, name)
    # Метод вызывается если атрибута нет:
    def __getattr__(self, name):
        print("Нет такого поля!")
        return "Alpha: "+name
# Класс со специальным методом:
class Bravo:
    # Метод для получения доступа к атрибуту:
    def __getattribute__(self, name):
```

```
        print("Bravo: запрос поля", name)
        try:
            res=object.__getattribute__(self, name)
        except AttributeError:
            res="Bravo: нет поля "+name
        return res
# Создание объектов и обращение к полям:
print("Объект A класса Alpha")
A=Alpha()
A.value=123
print("Поле value:", A.value)
print("Еще раз:", object.__getattribute__(A,"value"))
A.value=321
print("Поле value:", A.value)
print(A.color)
print("Объект B класса Bravo")
B=Bravo()
B.mylist=[1,2,3]
print("Поле mylist:", B.mylist)
print("Еще раз:", object.__getattribute__(B,"mylist"))
B.mylist=["A","B","C"]
print("Поле mylist:", B.mylist)
print(B.myset)
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.13)

```
Объект A класса Alpha
Alpha: запрос поля value
Поле value: 123
Еще раз: 123
Alpha: запрос поля value
Поле value: 321
```

```
Alpha: запрос поля color
Нет такого поля!
Alpha: color
Объект B класса Bravo
Bravo: запрос поля mylist
Поле mylist: [1, 2, 3]
Еще раз: [1, 2, 3]
Bravo: запрос поля mylist
Поле mylist: ['A', 'B', 'C']
Bravo: запрос поля myset
Bravo: нет поля myset
```

В программе описывается два класса. В классе `Alpha` описан специальный метод `__getattribute__()` с аргументами `self` (ссылка на объект, из которого вызывается метод) и `name` (название атрибута). В теле метода командой `print("Alpha: запрос поля", name)` отображается сообщение с названием класса и названием запрашиваемого атрибута (поля). Результатом метода возвращается значение `object.__getattribute__(self, name)`. И здесь требуются пояснения. Наша задача состоит в том, чтобы вернуть в качестве результата значение атрибута с названием `name` из объекта `self`. Но проблема в том, что обратиться к какому-либо атрибуту объекта `self` в формате точечного синтаксиса в теле метода `__getattribute__()` мы не можем, поскольку для обработки соответствующего выражения будет вызываться метод `__getattribute__()`. Получится, что метод `__getattribute__()` рекурсивно вызывает сам себя, и рекурсия эта бесконечная. Поэтому мы воспользовались тем обстоятельством, что класс `object` находится в вершине иерархии наследования классов и у него есть свой метод `__getattribute__()`. Именно этот метод мы вызываем инструкцией `object.__getattribute__(self, name)`. Получается, что в версии метода `__getattribute__()`, описываемой для класса `Alpha`, вызывается версия метода `__getattribute__()` из класса `object`, и в этом случае рекурсии нет.

> **НА ЗАМЕТКУ**
>
> Такая же проблема решается и в описании метода `__getattribute__()` в классе `Bravo`.

Метод `__getattr__()` описан так, что при его вызове отображается сообщение об отсутствии поля, название класса и название запрашиваемого поля.

В классе `Bravo` метод `__getattr__()` не описывается. Но чтобы не было проблем при попытке обратиться к несуществующему атрибуту в методе `__getattribute__()` выполняется обработка исключения класса `AttributeError`: при возникновении соответствующей ошибки результатом метода `__getattribute__()` возвращается текст с названием класса и названием поля.

ПОДРОБНОСТИ

Напомним, что при обращении к атрибуту вызывается метод `__getattribute__()`, и если запрашиваемого атрибута у объекта нет, то генерируется исключение класса `AttributeError`. В этом случае автоматически вызывается метод `__getattr__()`. В классе `Bravo` мы описали метод `__getattribute__()` так, что исключение `AttributeError` обрабатывается в методе, поэтому метод `__getattr__()` вызываться не будет.

На основе классов `Alpha` и `Bravo` в программе создаются объекты `A` и `B`. Для объекта `A` создается поле `value` со значением `123`. При обращении к этому полю в инструкции `A.value` автоматически вызывается метод `__getattribute__()`.

НА ЗАМЕТКУ

Метод `__getattribute__()` только при считывании значения поля, но не при присваивании значения полю.

Если нам нужно прочитать значение поля «старым», традиционным способом, можно, например, воспользоваться командой `object.__getattribute__(A,"value")`, в которой явно вызывается версия метода `__getattribute__()` из объекта `object`.

При попытке прочитать значение несуществующего поля `color` у объекта `A` сначала вызывается метод `__getattribute__()`, описанный в классе `Alpha`, а затем автоматически вызывается `__getattr__()`, описанный в этом же классе.

ПОДРОБНОСТИ

При обращении к несуществующему полю ошибка класса `AttributeError` генерируется при вычислении выражения `object.__getattribute__(self, name)` в теле метода `__getattribute__()` из класса `Alpha`. Предыдущая команда (которой отображается сообщение) в теле метода успевает выполниться.

Похожие манипуляции проделываются и с объектом `B` класса `Bravo`. Поправку нужно лишь сделать на то, что при обращении к несуществующему полю метод `__getattribute__()` сам обрабатывает такую ситуацию.

Стоит заметить, что механизм определения режима доступа к полям (и методам) достаточно гибкий и красивый. В качестве еще одного небольшого примера рассмотрим программу, представленную в листинге 9.14.

Листинг 9.14. Обращение к несуществующим атрибутам

```
# Класс со специальным методом:
class Alpha:
    # Метод вызывается если атрибута нет:
    def __getattr__(self, name):
        return len(name)
# Класс со специальным методом:
class Bravo:
    # Обычный метод:
    def show(self, x):
        print("Метод show():", x)
    # Метод вызывается если атрибута нет:
    def __getattr__(self, name):
        if len(name)<5:
            return lambda x: print("Первый метод:", x)
        else:
            return lambda x: print("Второй метод:", x*x)
# Создание объектов и обращение к атрибутам:
print("Объект A класса Alpha")
```

```
A=Alpha()
A.value="Alpha"
print("Поле value:", A.value)
print("Поле color:", A.color)
print("Поле myattribute:", A.myattribute)
print("Объект B класса Bravo")
B=Bravo()
B.show(10)
B.hi(10)
B.display(10)
```

Результат выполнения программы будет следующим.

Результат выполнения программы (из листинга 9.14)

```
Объект A класса Alpha
Поле value: Alpha
Поле color: 5
Поле myattribute: 11
Объект B класса Bravo
Метод show(): 10
Первый метод: 10
Второй метод: 100
```

Как и в предыдущем примере, у нас есть два класса (`Alpha` и `Bravo`). В каждом из этих классов мы описываем метод `__getattr__()`, но не описываем метод `__getattribute__()`. Что это означает? Это означает, что в случае считывания значения существующего атрибута, все будет происходить как обычно, в «штатном режиме». А если запрашивается несуществующий атрибут, то результат запроса (возвращаемое значение) будет определяться результатом метода `__getattr__()`.

В классе `Alpha` метод `__getattr__()` определен так, что результатом возвращается количество символов в названии запрашиваемого атрибута.

В классе `Bravo` в зависимости количества символов в названии запрашиваемого атрибута результатом метод `__getattr__()` возвращает

лямбда-выражение, соответствующее функции с одним аргументом. При вызове функций отображается соответственно значение аргумента или квадрат значения аргумента.

Программа содержит примеры создания объектов на основе классов `Alpha` и `Bravo`. Для этих объектов запрашиваются существующие и несуществующие поля и методы.

При попытке присвоить атрибуту объекта значение автоматически вызывается метод `__setattr__()`, который можно переопределить в классе объекта. Аргументами методу, кроме ссылки на объект, передается название атрибута, которому присваивается значение, и собственно значение, присваиваемое атрибуту. Программа, в которой иллюстрируется работа с методом `__setattr__()`, представлена в листинге 9.15.

Листинг 9.15. Обработка присваивания значения полю

```
# Класс со специальным методом:
class Alpha:
    # Метод для присваивания значения атрибуту:
    def __setattr__(self, name, val):
        if name=="number" and type(val)!=int:
            res=0
        else:
            res=val
        self.__dict__[name]=res
# Создание объекта:
A=Alpha()
# Операции с полями объекта:
A.value="Объект A"
print("A.value =", A.value)
A.number=123
print("A.number =", A.number)
A.number="Hello"
print("A.number =", A.number)
A.value=321
print("A.value =", A.value)
```

```
A.__dict__["number"]="Python"
print("A.number =", A.number)
```

Ниже показано, как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 9.15)

```
A.value = Объект A
A.number = 123
A.number = 0
A.value = 321
A.number = Python
```

В классе `Alpha` описан специальный метод `__setattr__()` с аргументами `self` (ссылка на объект), `name` (название атрибута) и `val` (значение, присваиваемое атрибуту). В теле метода, в условном операторе проверяется условие `name=="number" and type(val)!=int`. Оно истинно в том случае, если значение присваивается полю `number`, и это значение не целочисленное. Если так, то переменной `res` присваивается значение `0`. В противном случае (если условие ложно) переменной `res` присваивается значение `val`. После выполнения условного оператора командой `self.__dict__[name]=res` значение переменной `res` присваивается атрибуту с названием, записанным в аргумент `name`.

ПОДРОБНОСТИ

В теле метода `__setattr__()` мы не можем присваивать значения полям объекта `self` в формате точечного синтаксиса: в этом случае для присваивания значения полю снова будет вызван метод `__setattr__()`. Получается бесконечный рекурсивный вызов метода. Отметим также, что как альтернативу к команде `self.__dict__[name]=res` мы могли бы использовать инструкцию `object.__setattr__(self, name, res)`, которой вызывается версия метода `__setattr__()` из класса `object`.

Что получилось в итоге? Если значение присваивается полю, отличному от `number`, то операция выполняется как обычно — значение присваивается соответствующему полю. Но если значение присваивается полю `number`, то возможны два варианта. Если присваивается целочисленное значение, то оно записывается в поле `number`. Если присваивается

значение не целочисленного типа, то полю number присваивается значение 0.

Объект A создается на основе класса Alpha. Полю value присваивается значение "Объект A". Полю number присваивается значение 123. Здесь все происходит как обычно. Но вот при выполнении команды A.number="Hello" поле number получает значение 0, поскольку присваиваемое полю значение "Hello" не является целочисленным. При этом полю value, кроме прочего, можно присвоить и целочисленное значение.

Программа содержит команду, показывающую, что наши ограничения на присваиваемое полю number значение можно обойти. Командой A.__dict__["number"]="Python" полю number присваивается текстовое значение "Python". Нам это удалось сделать, поскольку мы не обращались к полю number в формате точечного синтаксиса, а получили доступ к словарю объекта (с помощью специального поля __dict__) и присвоили значение элементу этого словаря, использовав название поля как ключ. Метод __setattr__() в этом случае не вызывается.

Еще один полезный специальный метод __delattr__() вызывается при попытке удаления атрибута объекта. Аргументами методу передаются ссылка на объект и название удаляемого атрибута. Вызывается метод в случае, если атрибут удаляется с помощью инструкции del. Программа, в которой показано, как можно переопределить данный метод, представлена в листинге 9.16.

Листинг 9.16. Обработка удаления атрибута

```
# Класс со специальными методами:
class Alpha:
    # Метод вызывается если атрибута нет:
    def __getattr__(self, name):
        return "такого атрибута нет"
    # Метод вызывается при удалении атрибута:
    def __delattr__(self, name):
        if name=="number":
            print("Удалять нельзя!")
        else:
```

```
            object.__delattr__(self, name)
# Создание объекта:
A=Alpha()
# Операции с полями объекта:
A.value="объект A"
print("value:", A.value)
del A.value
print("value:", A.value)
A.number=123
print("number:", A.number)
del A.number
print("number:", A.number)
del A.__dict__["number"]
print("number:", A.number)
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.16)

```
value: объект A
value: такого атрибута нет
number: 123
Удалять нельзя!
number: 123
number: такого атрибута нет
```

В классе `Alpha` описаны два специальных метода: кроме метода `__delattr__()` исключительно ради удобства мы описали еще и метод `__getattr__()`. Каждый раз, когда мы будем запрашивать значение отсутствующего поля, методом будет возвращаться текст `"такого атрибута нет"`.

Метод `__delattr__()` описан так: если выполняется попытка удалить атрибут `number`, выполняется команда `print("Удалять нельзя!")`. В противном случае (если удаляется другой атрибут) выполняется команда `object.__delattr__(self, name)`, которой собственно

и удаляется атрибут. В данном случае, чтобы не спровоцировать бесконечный рекурсивный вызов метода, мы вызываем версию метода `__delattr__()` из класса `object`.

> **НА ЗАМЕТКУ**
>
> Вместо команды `object.__delattr__(self, name)` можно было бы воспользоваться командой `del self.__dict__[name]`.

После создания на основе класса `Alpha` объекта `A` командой `A.value="объект A"` в этот объект добавляется поле `value`. Удаляем поле командой `del A.value`. Проверка показывает, что ничего необычного не происходит и результат ожидаем. Но вот удалить поле `number` командой `del A.number` не получается. Появляется сообщение о невозможности удаления, и поле остается в объекте со своим присвоенным ранее значением `123`. Тем не менее удалить поле можно. Например, с помощью команды `del A.__dict__["number"]`. В данном случае мы удаляем элемент из словаря, посредством которого реализуется объект `A`, минуя тем самым вызов метода `__delattr__()` из объекта `A`.

Индексирование объектов

> Пусть меня ждут в пирамиде Хеопса у восьмой мумии, третий коридор налево.
>
> *Из м/ф «Приключения капитана Врунгеля»*

Объекты можно индексировать — то есть после имени объекта в квадратных скобках можно указать индекс (не обязательно целочисленный), и при определенных обстоятельствах такая инструкция будет иметь смысл. Чтобы эти «определенные обстоятельства» сложились, в классе, на основе которого создается объект, описываются специальные методы `__setitem__()`, `__getitem__()` и `__delitem__()` (все или несколько). Метод `__setitem__()` вызывается в случае, если выполняется команда вида `объект[индекс]=значение`. Ссылка на `объект`, `индекс` и присваиваемое `значение` являются аргументами метода `__setitem__()`. Метод `__getitem__()` вызывается в случае, если вычисляется значение выражения вида `объект[индекс]`. Аргументами метода являются ссылка на `объект` и `индекс`. Наконец, с помощью метода `__delitem__()` обрабатываются инструкции вида

del объект[индекс]. Ссылка на объект и индекс являются аргументами метода. Программа, в которой использованы все три метода, представлена в листинге 9.17.

Листинг 9.17. Индексирование объектов

```
# Класс со специальными методами:
class Alpha:
    # Метод для считывания значения по индексу:
    def __getitem__(self, index):
        return self.value[index]
    # Метод для присваивания значения по индексу:
    def __setitem__(self, index, val):
        self.value[index]=val
    # Метод для удаления по индексу:
    def __delitem__(self, index):
        del self.value[index]
    # Метод для приведения к тестовому формату:
    def __str__(self):
        return str(self.value)
    # Метод для вычисления "длины" объекта:
    def __len__(self):
        return len(self.value)
# Создание объекта:
A=Alpha()
# Поле-список для объекта:
A.value=[100,200,300]
# Проверка содержимого объекта:
print(A)
# Операции с объектом с использованием индекса:
for k in range(len(A)):
    print(A[k], end=" ")
print()
A[1]="A"
```

```
print(A)
del A[0]
print(A)
```

Как будет выглядеть результат выполнения программы, показано ниже.

Результат выполнения программы (из листинга 9.17)

```
[100, 200, 300]
100 200 300
[100, 'A', 300]
['A', 300]
```

В программе реализуется довольно простая идея: у объекта есть поле `value`, являющееся ссылкой на список, и доступ к этому списку осуществляется с помощью индексирования объекта. Для этого мы описываем класс `Alpha`. В классе есть несколько специальных методов. Метод `__getitem__()` в качестве результата возвращает значение `self.value[index]`. Это элемент с индексом `index` в поле-списке `value` того объекта, из которого вызывается метод.

При вызове метода `__setitem__()` выполняется команда `self.value[index]=val`, которой значение присваивается элементу с индексом `index` в поле-списке `value`.

Наконец, при вызове метода `__delitem__()` командой `del self.value[index]` из списка, на который ссылается поле `value`, удаляется элемент с указанным индексом `index`.

Кроме этих методов, удобства ради мы также описываем метод `__str__()`, который вызывается при приведении объекта к текстовому формату. Мы метод описываем так, что в качестве результата он возвращает текстовое представление для списка, на который ссылается поле `value`.

Метод `__len__()` вызывается, когда объект передается аргументом функции `len()`. В качестве результата метод возвращает количество элементов в списке, на который ссылается поле `value` объекта.

После описания класса `Alpha` на его основе создается объект `A`. Командой `A.value=[100,200,300]` в этот объект добавляется поле `value`, которое ссылается на список `[100,200,300]`. Обращение к элементам

данного списка можно выполнять в обычном режиме, а можно напрямую индексировать объект A. Примеры таких обращений есть в программе. Например, инструкция A[k] означает обращение к элементу с индексом k списка value объекта A. Результатом выражения len(A) является количество элементов в списке, на который ссылается поле value объекта A. А выполнение команды del A[0] приводит к удалению из списка, на который ссылается поле value, элемента с индексом 0.

Вообще индексирование объектов может выполняться по-разному и с использованием различных подходов. Для иллюстрации, в листинге 9.18 представлена простая программа, в которой описывается класс, объекты которого при индексировании возвращают в качестве значения числа Фибоначчи.

Листинг 9.18. Индексирование объекта для вычисления чисел Фибоначчи

```
# Класс для вычислений чисел Фибоначчи:
class Fibs:
    # Метод вызывается при индексировании объекта:
    def __getitem__(self, n):
        a=1
        b=1
        for k in range(n-2):
            a, b=b, a+b
        return b
# Создание объекта:
F=Fibs()
# Вычисление чисел Фибоначчи:
for k in range(1,16):
    print(F[k], end=" ")
print()
```

При выполнении программы получаем такой результат.

Результат выполнения программы (из листинга 9.18)

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

В классе `Fibs` описывается всего один метод `__getitem__()`. В теле метода объявляются две переменные `a` и `b` с начальным значением `1`. После этого запускается оператор цикла `for`. Количество итераций на `2` меньше значения второго аргумента `n` метода. За каждый цикл выполняется команда `a, b=b, a+b`, которой переменной `a` присваивается значение переменой `b`, а переменной `b` в качестве значения присваивается `a+b`. В результате получается, что мы вычислили очередное число в последовательности Фибоначчи и записали его в переменную `b`, а предыдущее число записали в переменную `a`.

По завершении оператора цикла значение переменной `b` возвращается в качестве результата метода. Таким образом, если мы индексируем объект класса `Fibs`, то результатом соответствующего выражения будет число в последовательности Фибоначчи, порядковый номер которого определяется индексом объекта.

Командой `F=Fibs()` мы создаем объект `F` класса `Fibs`, а затем с помощью оператора цикла отображаем последовательность из `15` чисел Фибоначчи. При этом для вычисления числа с номером `k` используется выражение `F[k]`. Фактически мы создали видимость бесконечного списка с числами Фибоначчи, и изменить этот список не получится (просто потому, что на самом деле нет ни списка, ни элементов).

Вызов объекта

> Замечательная идея! Что ж она мне самому в голову не пришла?
>
> *Из к/ф «Ирония судьбы, или С легким паром»*

В предыдущем примере, когда мы индексировали объект, фактически речь шла о вызове метода, а индекс играл роль аргумента этого метода. Другими словами, объект использовался подобно тому, как мы используем функции или методы. Это подход, который на самом деле имеет свои преимущества, и для его реализации существуют специальные механизмы. Например, если описать в классе специальный метод `__call__()`, то объект такого класса можно будет вызывать как функцию: после имени объекта в круглых скобках указывается аргумент (или аргументы), и такое выражение может возвращать результат. Это именно то значение, которое возвращается методом `__call__()`, поскольку

он вызывается для обработки соответствующих инструкций. Пример использования данного метода представлен в листинге 9.19.

Листинг 9.19. Вызов объекта

```
# Класс со специальным методом:
class Alpha:
    def __call__(self, n):
        s=0
        for k in range(len(self.nums)):
           s+=self.nums[k]**n
        return s
# Класс со специальным методом:
class Bravo:
    def __call__(self, x, y):
       return self.num*x+self.val*y
# Создание объекта:
A=Alpha()
A.nums=[1,2,3]
# Вызов объекта:
print("A(1) =", A(1))
print("A(2) =", A(2))
# Создание объекта:
B=Bravo()
B.num=2
B.val=3
# Вызов объекта:
print("B(5,1) =", B(5,1))
print("B(3,4) =", B(3,4))
```

Результат выполнения программы будет таким.

Результат выполнения программы (из листинга 9.19)

```
A(1) = 6
```

```
A(2) = 14
B(5,1) = 13
B(3,4) = 18
```

Используется два класса `Alpha` и `Bravo`, в каждом из которых описан метод `__call__()` (код упрощен максимально, поэтому никакие дополнительные методы не описываются). В классе `Alpha` предполагается, что объект класса имеет поле `nums`, которое ссылается на список с числовыми элементами. При вызове метода `__call__()` (то есть при вызове объекта) в качестве результата возвращается сумма элементов списка, причем элементы при суммировании возводятся в степень, определяемую вторым аргументом метода `__call__()`.

ПОДРОБНОСТИ

Первым аргументом метода `__call__()` является ссылка на объект, из которого вызывается метод. Все прочие аргументы метода — это те аргументы, которые передаются объекту при вызове. В данном случае в классе `Alpha` метод `__call__()` описан с двумя аргументами. Это означает, что при вызове объекта ему передается один аргумент. В классе `Bravo` метод `__call__()` описан с тремя аргументами, поэтому при вызове объекта этого класса ему передается два аргумента.

В классе `Bravo` метод `__call__()` описан с тремя аргументами. Мы исходим из того, что у объекта класса `Bravo` есть числовые поля `num` и `val`. Результатом метод `__call__()` возвращает линейную комбинацию значений полей `num` и `val`, а коэффициентами, на которые умножаются значения полей, являются аргументы метода.

Далее создается объект `A` класса `Alpha` с полем `nums`, которое ссылается на список `[1,2,3]`. Результатом выражения `A(1)` является сумма значений элементов списка (при суммировании значения элементов возводятся в степень `1`). Значение выражения `A(2)` — это сумма квадратов значений элементов списка (при суммировании значения элементов возводятся в степень `2`).

На основе класса `Bravo` создается объект `B`. У этого объекта есть поле `num` со значением `2` и поле `val` со значением `3`. Значение выражения `B(5,1)` вычисляется как произведение чисел `2` и `5` плюс произведение чисел `3` и `1`. Соответственно, значение выражения `B(3,4)` вычисляется как произведение чисел `2` и `3` плюс произведение чисел `3` и `4`.

Итераторы

> Это экспонаты. Отходы, так сказать, магического производства.
>
> *Из к/ф «Чародеи»*

Есть такое понятие, как *итерируемый объект*. Итерируемый объект — это объект, содержимое которого можно в определенном смысле «перебирать». Хорошим примером итерируемых объектов являются списки и кортежи. Мы их использовали (и перебирали содержимое) многократно. Наряду с итерируемыми объектами есть *объекты-итераторы*, или просто *итераторы*. Итератор, как и итерируемый объект, можно «перебирать». Но методика перебора несколько иная. Обычно основой для создания итератора служит итерируемый объект. Но такой итерируемый объект играет роль «аргумента». Сам по себе итератор является объектом, и создается он на основе какого-то класса. Такие классы, на основе которых создаются итераторы, называются *классами-итераторами*. Если смотреть на ситуацию формально, то класс-итератор — это такой класс, в котором описаны специальные методы `__iter__()` и `__next__()`. Но за этим спрятана определенная «идеология», с которой имеет смысл хотя бы кратко познакомиться. А начнем мы с практической ситуации.

Если имеется итерируемый объект (например, список), то на его основе можно создать итератор. Для этого достаточно передать итерируемый объект аргументом функции `iter()`. Что можно делать с таким итератором? Его содержимое можно перебирать. Для этого достаточно передать итератор в качестве аргумента функции `next()`. Важно то, что каждый раз при вызове функции `next()` с аргументом-итератором получаем очередное значение, «спрятанное» в итераторе. Так продолжается до тех пор, пока не перебраны все значения из итератора. Если значения «закончились», то при попытке вызвать функцию `next()` генерируется исключение класса `StopIteration`. Собственно именно это обстоятельство служит признаком того, что итератор себя исчерпал. Небольшой пример по этому поводу представлен в листинге 9.20.

Листинг 9.20. Знакомство с итераторами

```
# Создание итератора:
vals=iter([100,"A",[1,2]])
# Перебор итератора:
```

```
try:
    print("Первое:", next(vals))
    print("Второе:", next(vals))
    print("Третье:", next(vals))
    print("Четвертое:", next(vals))
except StopIteration:
    print("Значений больше нет")
```

Ниже показан результат выполнения программы.

Результат выполнения программы (из листинга 9.20)

```
Первое: 100
Второе: A
Третье: [1, 2]
Значений больше нет
```

Программа исключительно простая. Командой `vals=iter([100,"A",[1,2]])` на основе списка `[100,"A",[1,2]]` создается итератор. Именно данный список определяет «содержимое» итератора. Ссылка на итератор записывается в переменную `vals`. Далее последовательно четыре раза вычисляется инструкция `next(vals)`. Каждый раз при вычислении этой инструкции возвращается очередное значение из списка `[100,"A",[1,2]]`, на основе которого создавался итератор. Но когда значение `next(vals)` вычисляется в четвертый раз, генерируется исключение класса `StopIteration`, которое в программе перехватывается и обрабатывается. На этом перебор итератора завершен.

НА ЗАМЕТКУ

Выполнить перебор итератора можно только один раз. В этом смысле итератор представляет собой «одноразовый» объект. Если нам нужно еще раз выполнить перебор, то придется создавать новый итератор.

Когда объект передается аргументом функции `iter()` при ее вызове, из данного объекта вызывается метод `__iter__()`. При вызове функции `next()` с аргументом-объектом из этого объекта вызывается метод `__next__()`. То есть чтобы на основе некоторого объекта создать

итератор, достаточно, чтобы у этого объекта был метод `__iter__()`. Результатом метод должен возвращать объект (итератор), у которого, в свою очередь, должен быть метод `__next__()`.

НА ЗАМЕТКУ

Вместо рассмотренного выше мы могли бы использовать такой код:

```
vals=[100,"A",[1,2]].__iter__()
try:
    print("Первое:", vals.__next__())
    print("Второе:", vals.__next__())
    print("Третье:", vals.__next__())
    print("Четвертое:", vals.__next__())
except StopIteration:
    print("Значений больше нет")
```

В этом коде вместо функций `iter()` и `next()` используются соответственно методы `__iter__()` и `__next__()`. Результат выполнения данного кода точно такой же, как и в рассмотренном выше примере.

Реализовать описанную схему не сложно. Пример представлен в листинге 9.21.

Листинг 9.21. Подготовка к созданию итератора

```
# Первый класс:
class Alpha:
    # Конструктор:
    def __init__(self,*vals):
        L=[]
        for v in vals:
           if type(v)==int:
               L.append(v)
        self.nums=L
    # Метод вызывается при вызове функции iter():
    def __iter__(self):
```

```
        return Bravo(self.nums)
# Второй класс:
class Bravo:
    # Конструктор:
    def __init__(self, nums):
        L=[]
        for n in nums:
            if n<10 and n>0:
                L.append(n)
        self.digits=L
        self.position= -1
    # Метод вызывается при вызове функции next():
    def __next__(self):
        self.position+=1
        if self.position<len(self.digits):
            return self.digits[self.position]
        else:
            raise StopIteration
# Создание объекта класса Alpha:
A=Alpha(2,"A",12,7,-3,"Hello",9,5,"Alpha")
# Создание объекта класса Bravo:
B=iter(A)
# Вызов функции next():
try:
    while True:
        print(next(B), end=" ")
except StopIteration:
    print()
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.21)

```
2 7 9 5
```

Прокомментируем программный код и поясним, какое отношение он имеет к созданию итератора. Итак, мы описываем класс `Alpha`, в котором есть конструктор. В качестве аргумента конструктору передается ссылка на объект и произвольное количество иных аргументов (совокупность этих аргументов обозначена как `*vals`). В теле конструктора создается пустой список `L`, а затем запускается оператор цикла, в котором перебираются переданные конструктору аргументы. Если тип аргумента целочисленный (условие `type(v)==int`), то соответствующее значение добавляется в список `L` (инструкция `L.append(v)`). После того как список `L` сформирован, командой `self.nums=L` ссылка на список записывается в поле `nums` объекта.

Также в классе `Alpha` описывается метод `__iter__()`. В качестве результата метод возвращает ссылку на объект класса `Bravo`, и аргументом конструктору при создании этого объекта передается ссылка на список, записанная в поле `nums`.

Класс `Bravo` также имеет конструктор. Мы предполагаем, что в качестве аргумента конструктору передается список `nums` с целочисленными значениями. На основе этого списка создается другой список `L`. В него добавляются только те элементы из списка `nums`, значение которых больше `0` и меньше `10` (то есть числа от `1` до `9` включительно).

После того как список `L` сформирован, ссылка на него записывается в поле `digits` объекта. Также в конструкторе создается поле `position` с начальным значением `-1`.

Метод `next()` описан следующим образом. Сначала текущее значение поля `position` увеличивается на `1`. Затем в условном операторе проверяется условие `self.position<len(self.digits)`. Его истинность означает, что значение поля `position` не превышает длину списка, на который ссылается поле `digits`. Если так, то результатом метод возвращает значение элемента из списка `digits` с индексом `position`. Если условие ложно, то инструкцией `raise StopIteration` генерируется исключение класса `StopIteration`.

> **НА ЗАМЕТКУ**
>
> Исключение можно сгенерировать искусственно. Для этого используется инструкция `raise`, после которой указывается класс генерируемого исключения (в данном случае это класс `StopIteration`).

Командой `A=Alpha(2,"A",12,7,-3,"Hello",9,5,"Alpha")` создается объекта `A` класса `Alpha`. На основе этого объекта командой `B=iter(A)` создается объекта класса `Bravo`, а ссылка на данный объект записывается в переменную `B`. У этого объекта есть метод `__next__()`, который вызывается при вызове функции `next()`, когда ей аргументом передается объект `B`. Мы запускаем формально бесконечный оператор цикла, помещенный в блок `try`. При этом в блоке `except` обрабатывается исключение класса `StopIteration`. В теле оператора цикла выполняется команда `print(next(B), end=" ")`. Здесь использована инструкция `next(B)`, которая при каждом очередном вызове возвращает очередное значение из поля-списка `digits` объекта `B`. Когда элементы перебраны все, генерируется исключение класса `StopIteration` (которое перехватывается и обрабатывается).

ПОДРОБНОСТИ

При создании объекта `A` его поле `nums` получает список `[2,12,7,-3,9,5]`. Это целочисленные значения из набора аргументов, переданных конструктору при создании объекта. При создании объекта `B` командой `B=iter(A)` из объекта `A` вызывается метод `__iter__()`, и созданный в итоге объект класса `Bravo` имеет поле `digits`, которое ссылается на список `[2,7,9,5]`. В него входят лишь те значения из списка `[2,12,7,-3,9,5]`, которые больше `0` и меньше `10`.

Объект `B` нельзя назвать полноценным итератором. Мы не сможем его использовать, например, в операторе цикла `for` в качестве итерируемой последовательности. Причина в том, что у объекта `B` нет специального метода `__iter__()`.

ПОДРОБНОСТИ

Если используется выражение вида `for элемент in объект`, то автоматически из `объекта` вызывается метод `__iter__()` и полученный в результате итератор перебирается в операторе цикла (последовательным вызовом метода `__next__()`).

Однако ситуацию легко исправить. Для этого достаточно «объединить» два класса в один. Другими словами, мы можем описать один класс, у которого будет и метод `__iter__()`, и метод `__next__()`. Реализация такого подхода представлена в программе в листинге 9.22.

Листинг 9.22. Создание итератора

```
# Класс итератора:
class MyClass:
    # Конструктор:
    def __init__(self,*vals):
        L=[]
        for v in vals:
           if type(v)==int:
               if v<10 and v>0:
                   L.append(v)
        self.digits=L
        self.position= -1
    # Метод вызывается при вызове функции iter():
    def __iter__(self):
        return self
    # Метод вызывается при вызове функции next():
    def __next__(self):
        self.position+=1
        if self.position<len(self.digits):
            return self.digits[self.position]
        else:
            raise StopIteration
# Создание итератора:
A=MyClass(2,"A",12,7,-3,"Hello",9,5,"Alpha")
# Вызов функции next():
try:
    while True:
        print(next(A), end=" ")
except StopIteration:
    print()
# Создание итератора:
B=MyClass(5,"B",1.2,11,-1,"Hi",8,4,"Bravo",3)
```

```
# Оператор цикла:
for s in B:
    print(s, end=" ")
print()
```

Результат выполнения программы будет следующим.

Результат выполнения программы (из листинга 9.22)

```
2 7 9 5
5 8 4 3
```

Мы теперь описываем только один класс (он называется `MyClass`). Конструктор у класса такой, что ему передается произвольное количество аргументов и на их основе формируется список из целых чисел, больших `0` и меньших `10`. Ссылка на список записывается в поле `digits`.

Поскольку у объекта класса `MyClass` также есть и метод `__iter__()`, и метод `__next__()`, то мы такой объект можем использовать как итератор. Это означает, что если из объекта будет вызываться метод `__iter__()`, то необходимости создавать новый объект нет — можно воспользоваться тем объектом, из которого вызывается метод. Поэтому метод `__iter__()` содержит всего одну инструкцию `return self`.

Метод `__next__()` описан фактически так же, как в предыдущем случае: при вызове метода перебираются (вызов за вызовом) элементы списка `digits`, а в конце генерируется исключение класса `StopIteration`.

После описания класса на его основе сначала создается объект `A`. Этот объект последовательно передается в качестве аргумента функции `next()`. После этого создается объект `B`, и этот объект используется в качестве итерируемой последовательности в операторе цикла `for`.

НА ЗАМЕТКУ

После того как мы один раз перебрали объекты-итераторы `A` и `B`, второй раз выполнить перебор не получится. Но в данном примере мы можем «восстановить» оба объекта. Для этого достаточно полю `position` соответствующего объекта присвоить значение `-1`. Желающие могут подумать о том, каковы будут последствия, если вместо значения `-1` полю присвоить, например, значение `1`.

Резюме

Ну зачем такие сложности?

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

- При наследовании класс создается на основе другого класса или классов. Исходные классы, на основе которых создается новый класс, называются базовыми. Класс, который называется на основе базовых классов, называется производным классом.
- В описании производного класса базовые классы указываются в круглых скобках после имени производного класса. Если базовых классов несколько, автоматически формируется цепочка наследования (получить такую цепочку можно с помощью специального поля `__mro__`). В вершине цепочки наследования находится класс `object`. При обращении к атрибуту объекта производного класса поиск атрибута выполняется последовательно по всей цепочке наследования.
- При наследовании можно переопределять методы. В этом случае в производном классе заново описывается метод, унаследованный из базового класса. Для доступа к версиям методов из базовых классов может использоваться инструкция `super()`.
- Существует группа специальных методов, которые позволяют определять для объектов такие операции, как приведение к определенному типу, выполнение арифметических операций, также можно перегружать побитовые операторы и операторы сравнения, определять режим доступа к атрибутам объектов, индексировать объекта и вызывать их как функции. Для этого достаточно описать соответствующий специальный метод в классе, на основе которого создается объект.
- Есть специальные объекты, которые называются итераторами. Такие объекты можно использовать в качестве итерируемых последовательностей, например, в операторе цикла. Итератор создается на основе класса, у которого должны быть описаны специальные методы `__iter__()` и `__next__()`.

Задания для самостоятельной работы

Ладно, все. Надо что-то делать. Давай-ка, может быть, сами изобретем.

Из к/ф «Чародеи»

1. Напишите программу, в которой создается цепочка наследования из трех классов. У объекта исходного класса имеется поле, и у каждого следующего класса добавляется по одному полю. Опишите методы, переопределяемые в производных классах, которые позволяют присваивать значения полям и отображать значения полей.

2. Напишите программу, в которой есть класс с переопределенными методами для приведения к разным типам. В частности, у объекта должны быть поля с целочисленным значением, текстом и действительным числовым значением. При приведении объекта к целочисленному, текстовому или действительному числовому типу возвращается значение соответствующего поля.

3. Напишите программу, в которой для объектов класса определена операция сложения. У каждого объекта есть поле-список, и при сложении объектов получается новый объект того же класса. Его поле-список получается объединением полей-списков исходных объектов.

4. Напишите программу, в которой для объектов предусмотрены операции сложения с числом, вычитания числа и вычитания из числа, а также умножения на число и деления на число. У объекта должно быть поле с числовым значением, и при выполнении указанных операций они должны выполняться с полем объекта.

5. Напишите программу, в которой для объектов класса предусмотрены операции сравнения. У каждого объекта есть поле-список с числовыми значениями. Операции сравнения выполняются так: объекты на предмет равенства проверяются по первому элементу в списках, на предмет «не равно» — по второму элементу в списках, «меньше» — по третьему элементу в списках, и так далее. Если соответствующего элемента в списке нет, используется нулевое значение.

6. Напишите программу, в которой для объектов класса предусмотрен специальный режим доступа к полям. В частности, у объекта должно быть поле-список, значением которому можно присваивать только

список. Из присваиваемого списка в поле-список копируются только текстовые значения. При считывании значения этого поля возвращается текстовая строка, содержащая только начальные буквы текстовых значений, которые входят в список.

7. Напишите программу с классом, объекты которого можно индексировать. В частности, у объекта должно быть два поля-списка с числами. При индексировании объекта возвращается сумма элементов из списков с соответствующим индексом. Если в каком-то списке нет такого элемента, он заменяется нулевым значением.

8. Напишите программу с классом, объекты которого можно вызывать. У объекта класса должно быть поле-список с числовыми значениями, а результатом метод возвращает полиномиальную сумму. В частности, если в списке содержатся числа a_0, a_1, ..., a_n и в качестве аргумента объекту при вызове передается значение x, то в качестве результата должно возвращаться значение $a_0 + a_1x + a_2x^2 + ... + a_nx^n$.

9. Напишите программу, в которой создается итератор, генерирующий нечетные натуральные числа. Количество генерируемых чисел определяется аргументом конструктора.

10. Напишите программу, в которой создается итератор, генерирующий числа Фибоначчи (первые два числа равны единице, а каждое следующее есть сумма двух предыдущих). Количество генерируемых чисел передается в качестве аргумента конструктору при создании итератора.

Глава 10
ОБРАБОТКА ИСКЛЮЧЕНИЙ И ПОТОКИ

— Что за вздор. Как вам это в голову взбрело?
— Да не взбрело бы, но факты, как говорится, упрямая вещь.

Из к/ф «Чародеи»

В этой главе обсуждаются в основном две темы: это *обработка исключительных ситуаций* и реализация *потоковой модели*. Обе достаточно важны для эффективной работы в Python и имеют непосредственное отношение к парадигме ООП.

Принципы обработки исключений

Наше повеление. Этот танец не вяжется с королевской честью. Мы запрещаем его на веки веков.

Из к/ф «31 июня»

С обработкой исключений мы уже много раз сталкивались. Правда, при этом возможности Python использовались минимально. Здесь мы кратко повторим то, что уже известно, и познакомимся с новыми приемами и подходами.

Идея, положенная в основу обработки исключений, проста: код, при выполнении которого может возникнуть ошибка, помещается в блок `try`. После `try`-блока в самом простом случае следует `except`-блок. Если при выполнении кода в `try`-блоке ошибка не возникает, то код в `except`-блоке игнорируется. Если при выполнении `try`-блока возникла ошибка, то выполнение команд в `try`-блоке прекращается и начинают выполняться команды в `except`-блоке. Это самая общая схема, которую мы уже использовали ранее. Дальше начинаются «подробности».

Во-первых, ошибки бывают разными. Если при выполнении кода происходит ошибка, то в соответствии с типом ошибки создается специальный объект (объект ошибки или объект исключения). Класс, на основе которого создается этот объект, можно отождествлять с типом ошибки. Существует целая иерархия классов ошибок, связанных наследованием. Кроме этого, можно создавать собственные классы ошибок.

Первая из возможностей, которые открываются перед нами в плане обработки ошибок, состоит в том, что мы можем по-разному обрабатывать ошибки разных типов. В этом случае для одного `try`-блока предусматривается несколько `except`-блоков, и в этих блоках после ключевого слова `except` указывается класс исключения, для перехвата и обработки которого предназначен данный блок. Шаблон конструкции `try-except` в этом случае такой:

```
try:
    # Контролируемый код
except класс_исключения:
    # Код обработки исключения
except класс_исключения:
    # Код обработки исключения
…
except класс_исключения:
    # Код обработки исключения
```

Выполняется соответствующий код следующим образом: если при выполнении кода в `try`-блоке ошибок не возникает, то все `except`-блоки игнорируются. Если ошибка произошла, то выполнение команд в `try`-блоке прекращается и начинается просмотр `except`-блоков. Просматриваются типы исключений, указанные в `except`-блоках. Если совпадение найдено (тип ошибки совпадает с классом исключения, указанного в `except`-блоке), то выполняется код соответствующего `except`-блока.

НА ЗАМЕТКУ

Если один и тот же блок предназначен для обработки исключений разных типов, то после ключевого слова `except` указывается кортеж, элементами которого являются классы обрабатываемых в блоке исключений.

Важное обстоятельство связано с тем, определенный `except`-блок обрабатывает не только исключения того класса, который указан в `except`-блоке, но и всех подклассов этого класса. Поэтому, в общем-то, важно иметь хотя бы общее представление об иерархии наследования классов исключений. Эта иерархия проиллюстрирована на рис. 10.1 (стрелки направлены от базовых классов к производным).

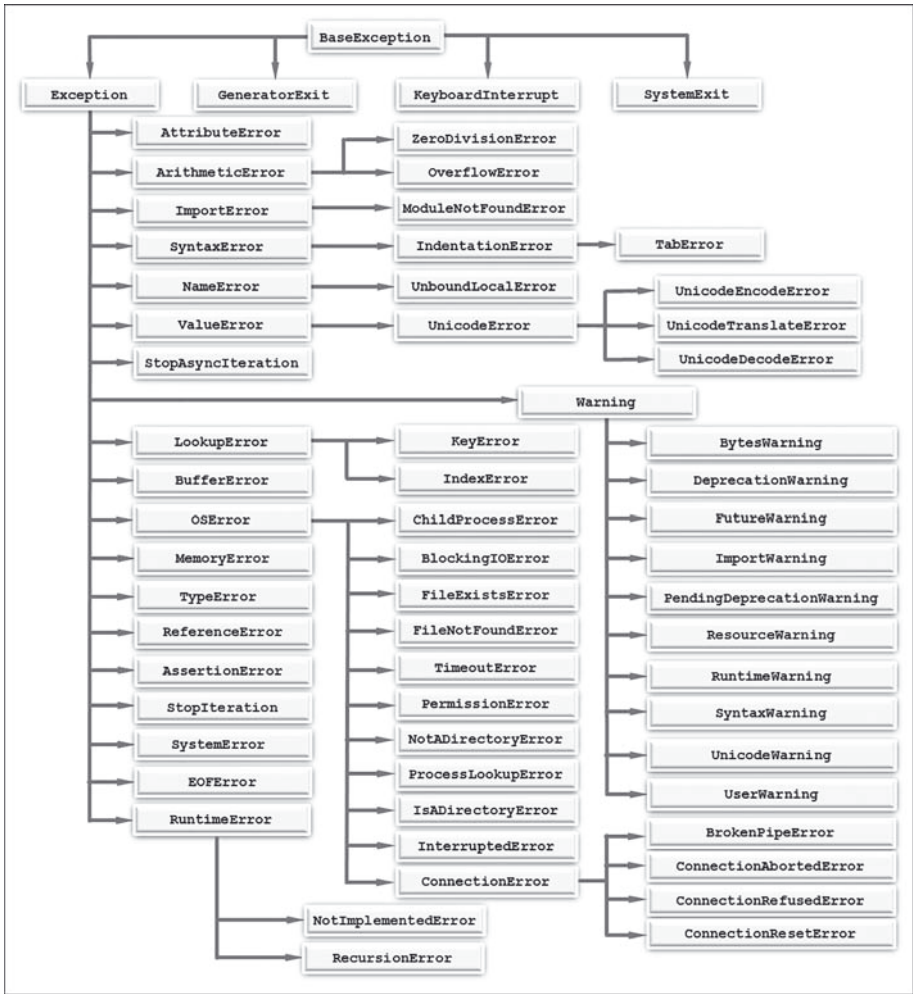


Рис. 10.1. Иерархия наследования классов исключений

В вершине этой иерархии находится класс `BaseException`. У класса есть четыре производных класса: `GeneratorExit` (событие, связанное с закрытием генератора или подпрограммы), `KeyboardInterrupt`

(возникает при нажатии пользователем кнопки прерывания — например, <Ctrl>+<C>), `SystemExit` (генерируется функцией `exit()` из модуля `sys`) и `Exception` (базовый класс для всех встроенных исключений, не связанных с отказом системы). Для нас интерес будет представлять именно класс `Exception`.

> **НА ЗАМЕТКУ**
>
> Как мы узнаем далее, можно создавать собственные классы исключений. Такие классы должны быть производными от класса `Exception`.

Фактически класс `Exception` является базовым для большинства классов исключений, с которыми на практике приходится иметь дело чаще всего. Некоторые из классов перечислены в табл. 10.1.

Табл. 10.1. Производные классы от класса `Exception`

Класс исключения	Описание
`ArithmeticError`	Базовый класс для исключений, которые генерируются при возникновении арифметических ошибок. Производные классы: `OverflowError`, `ZeroDivisionError`, `FloatingPointError` (больше не используется)
`AssertionError`	Исключение, связанное ошибкой при выполнении инструкции `assert` (используется для искусственного генерирования исключений)
`AttributeError`	Ошибка, связанная с доступом к атрибуту. Если атрибута у объекта нет совсем, то генерируется исключение класса `TypeError`
`BufferError`	Ошибка, связанная с доступом к буферу обмена
`ImportError`	Ошибка, связанная с импортом модуля. Имеет производный класс `ModuleNotFoundError`
`LookupError`	Базовый класс для исключений, связанных с ошибками при работе с клавишами или индексами. Производные классы: `IndexError` и `KeyError`
`MemoryError`	Ошибка, связанная с устранимыми ошибками в использовании памяти
`NameError`	Ошибка обращения к имени. Имеет производный класс `UnboundLocalError`

`OSError`	Базовый класс для системных ошибок. Производные классы: `BlockingIOError`, `ChildProcessError`, `ConnectionError` (имеет производные классы `BrokenPipeError`, `ConnectionAbortedError`, `ConnectionRefusedError`, `ConnectionResetError`), `FileExistsError`, `FileNotFoundError`, `InterruptedError`, `IsADirectoryError`, `NotADirectoryError`, `PermissionError`, `ProcessLookupError` и `TimeoutError`
`ReferenceError`	Ошибка слабой ссылки при доступе к атрибуту
`RuntimeError`	Класс для ошибок, которые нельзя отнести к другим категориям. Производные классы: `NotImplementedError` и `RecursionError`
`StopAsyncIteration`	Генерируется асинхронным итератором для прекращения итераций
`StopIteration`	Генерируется итератором для прекращения итераций
`SyntaxError`	Синтаксическая ошибка. Имеет производный класс `IndentationError`, у которого есть производный класс `TabError`
`SystemError`	Внешняя ошибка интерпретатора
`TypeError`	Исключение, связанное с использованием объекта некорректного типа
`ValueError`	Исключение генерируется, если аргумент функции или метода имеет корректный тип, но некорректное значение. Имеет производный класс `UnicodeError`, у которого есть подклассы `UnicodeDecodeError`, `UnicodeEncodeError` и `UnicodeTranslateError`
`Warning`	Базовый класс для исключений, которые являются предупреждениями (некритичными ошибками). Производные классы: `DeprecationWarning`, `PendingDeprecationWarning`, `RuntimeWarning`, `SyntaxWarning`, `UserWarning`, `FutureWarning`, `ImportWarning`, `UnicodeWarning`, `BytesWarning` и `ResourceWarning`

Некоторые из этих классов будут рассматриваться более подробно в примерах. Но прежде, чем перейти к рассмотрению примеров, отметим еще несколько возможностей, которые у нас имеются в процессе перехвата и обработки исключений.

Помимо нескольких except-блоков, в конструкции обработки исключений можно использовать блок finally. Код, размещенный в этом блоке, выполняется в любом случае: и если в try-блоке нет ошибок, и если при выполнении try-блока возникла ошибка.

> **НА ЗАМЕТКУ**
>
> Может показаться, что особого смысла в использовании блока finally нет, поскольку после обработки ошибки все равно выполняются команды после except-блоков. Полезность этого блока проявляется тогда, когда используются вложенные блоки try-except. Соответствующие примеры мы будем рассматривать.

Также можно использовать блок else. Команды else-блока выполняются только в том случае, если при выполнении команд из try-блока ошибок не было.

Мы отмечали, что при возникновении ошибки на основе класса соответствующего исключения создается специальный объект ошибки. Этот объект содержит информацию об ошибке и передается в except-блок для обработки. Другими словами, при обработке ошибки в except-блоке мы можем использовать объект ошибки. Шаблон описания конструкции try-except (с одним блоком except) в этом случае выглядит следующим образом:

```
try:
    # Контролируемый код
except класс as объект:
    # Код обработки ошибки
```

Проще говоря, после названия класса исключения указывается ключевое слово as и название переменной, через которую в except-блок передается объект ошибки. Вкратце это те возможности, которые связаны с использованием конструкции try-except. Теперь перейдем к рассмотрению конкретных ситуаций.

> **НА ЗАМЕТКУ**
>
> Две важные технологии, имеющие отношение к обработке исключений, — искусственное генерирование исключений и создание собственных классов исключений. Зачем и как все это используется, мы рассмотрим на примерах.

Обработка исключений разных типов

> Вы видите, мистер Холмс, как я ловко расставил сеть. И сеть, как вы видите, сужается.
>
> *Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»*

Для начала мы рассмотрим небольшой пример, в котором выполняется обработка исключения в зависимости от его типа. Программа представлена в листинге 10.1.

Листинг 10.1. Обработка исключений разных классов

```
# Список:
A=[10,20,30,40]
# Оператор цикла:
for k in [0,1,2,"три",4,3]:
    # Контролируемый код:
    try:
        print("A["+str(k)+"] = ", end="")
        A[k]/=(k-1)
        print(A[k])
    # Обработка исключений:
    except IndexError:
        print("нет такого элемента")
    except ZeroDivisionError:
        A[k]=0.0
        print(A[k],"- деление на ноль")
    except TypeError:
        print("неверный тип индекса")
```

Результат выполнения программы будет следующим.

Результат выполнения программы (из листинга 10.1)

```
A[0] = -10.0
A[1] = 0.0 — деление на ноль
```

```
A[2] = 30.0
A[три] = неверный тип индекса
A[4] = нет такого элемента
A[3] = 20.0
```

Программа достаточно простая. Сначала создается список `A=[10,20,30,40]`. Затем запускается оператор цикла `for`, в котором переменная `k` принимает значения из списка `[0,1,2,"три",4,3]`. Команды в теле оператора цикла размещены в `try`-блоке, а для перехвата и обработки исключений использованы три `except`-блока. В блоках обрабатываются исключения классов `IndexError` (индекс выходит за допустимые пределы), `ZeroDivisionError` (деление на ноль) и `TypeError` (указано значение некорректного типа).

Ошибки могут возникать при выполнении команды `A[k]/=(k-1)`. В принципе возможны такие неприятные ситуации. При значении `1` для переменной `k` выполняется попытка деления на ноль (исключение класса `ZeroDivisionError`). В этом случае выполняются команды `A[k]=0.0` и `print(A[k],"- деление на ноль")`. При значении `"три"` для переменной `k` возникает ошибка класса `TypeError`. В процессе ее обработки выполняется команда `print("неверный тип индекса")`. Наконец, при значении `4` для переменной `k` выполняется обращение к несуществующему элементу списка, вследствие чего генерируется исключение `IndexError`. Оно обрабатывается выполнением команды `print("нет такого элемента")`.

Использование объекта исключения

Многие меня уважают. Некоторые даже боятся.

Из к/ф «Служебный роман»

Мы можем не только использовать разные блоки для обработки исключений разных типов. Как отмечалось ранее, при возникновении ошибки на основе класса исключения создается объект, который можно использовать в процессе обработки. Как это делается на практике, показано в программе в листинге 10.2.

Листинг 10.2. Использование объекта исключения

```
# Список:
A=[10,20,30,40]
# Оператор цикла:
for k in [0,1,2,"три",4,3]:
    # Контролируемый код:
    try:
        print("* Значение элемента A["+str(k)+"]: ", end="")
        A[k]/=(k-1)
        print(A[k])
    # Обработка исключений:
    except (TypeError, IndexError) as error:
        print()
        print("Исключение класса", error.__class__.__name__)
        print(error.__doc__)
        print("Базовый класс:", error.__class__.__bases__[0].__name__)
    except ZeroDivisionError as error:
        print()
        print("Произошла ошибка деления на ноль")
        print("Цепочка наследования:")
        for s in error.__class__.__mro__:
            print(s.__name__)
        A[k]=0.0
        print("Элементу присвоено значение", A[k])
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 10.2)

```
* Значение элемента A[0]: -10.0
* Значение элемента A[1]:
Произошла ошибка деления на ноль
Цепочка наследования:
```

```
ZeroDivisionError
ArithmeticError
Exception
BaseException
object
Элементу присвоено значение 0.0
* Значение элемента A[2]: 30.0
* Значение элемента A[три]:
Исключение класса TypeError
Inappropriate argument type.
Базовый класс: Exception
* Значение элемента A[4]:
Исключение класса IndexError
Sequence index out of range.
Базовый класс: LookupError
* Значение элемента A[3]: 20.0
```

Данный пример является модификацией предыдущего примера. Мы только изменили способ обработки исключений (и внесли «декоративные» изменения в блок `try`). Теперь не три, а два `except`-блока, и в каждом из них явно используется объект исключения. Один блок предназначен для обработки исключений классов `TypeError` и `IndexError`, и один блок используется для обработки исключений класса `ZeroDivisionError`. В обоих блоках объект исключения обозначен как `error`.

Какую информацию мы получаем на основе объекта исключения? Во-первых, мы можем узнать класс возникшей ошибки, для чего используется специальное поле `__class__`. Для получения названия класса используем поле `__name__`. С помощью поля `__doc__` можно получить оригинальное (определенное классом исключения) описание ошибки. Также мы используем поля `__bases__` и `__mro__` при работе с классами исключений. Поле `__mro__` возвращает в качестве результата кортеж с классами из цепочки наследования. Поле `__bases__` возвращает кортеж непосредственных базовых классов данного класса. В нашем случае в таком кортеже один элемент, но для его «извлечения» из кортежа нам приходится использовать индексирование.

Вложенные блоки для обработки исключений

> Ведь это же настоящая тайна! Ты потом никогда себе не простишь!
>
> *Из к/ф «Гостья из будущего»*

Конструкции, предназначенные для обработки исключений, могут быть вложенными. Это ситуация, когда, например, внутри блока `try` размещается еще одна конструкция `try-except`. Здесь также уместно напомнить, что кроме `except`-блоков мы можем использовать еще и блок `finally`, команды которого выполняются в обязательном порядке. Использование этого блока наиболее показательно именно в случае, когда речь идет о вложенных блоках для обработки исключений.

НА ЗАМЕТКУ

Допустим, имеется `try`-блок, несколько `except`-блоков и блок `finally`. Если в блоке `try` ошибок не было, то `except`-блоки игнорируются, а выполняется блок `finally` и далее команды после этого блока. Если в `try`-блоке возникла ошибка, то она обрабатывается в одном из `except`-блоков, после чего выполняется блок `finally` и далее команды после этого блока. То есть даже если бы команды из блока `finally` просто размещались после конструкции `try-except`, алгоритм выполнения не изменился бы. Но это только в случае, если ошибки перехватываются в `except`-блоках. Если возникла ошибка, которая ни в одном `except`-блоке не обрабатывается, то наличие или отсутствие блока `finally` может иметь принципиальное значение.

Необходимость и полезность использования вложенных блоков для обработки исключений базируется на том обстоятельстве, что если в `try`-блоке возникает ошибка, которая не обрабатывается ни в одном `except`-блоке, то она передается для обработки во внешний блок обработки исключений. При этом если во внутреннем блоке обработки исключений имеется блок `finally`, то он выполняется перед передачей исключения во внешний блок.

Пример использования вложенных блоков для обработки исключений представлен в листинге 10.3.

Листинг 10.3. Вложенные блоки для обработки исключений

```
# Список:
A=[10,20,30,40]
# Оператор цикла:
for k in [0,1,2,"три",4,3]:
    # Внешний блок:
    try:
        print("* Значение элемента A["+str(k)+"]:")
        # Внутренний блок:
        try:
            A[k]/=(k-1)
            print(A[k])
        # Внутренний блок обработки:
        except ZeroDivisionError:
            print("Попытка деления на ноль")
            A[k]=0.0
            print("Новое значение", A[k])
        # Блок выполняется, если нет ошибки:
        else:
            print("Ошибки деления на ноль нет")
        # Блок выполняется всегда:
        finally:
            print("# Завершение внутреннего блока")
    # Внешний блок обработки:
    except:
        print("Что-то пошло не так")
print("Программа завершила выполнение")
```

Результат выполнения программы следующий.

Результат выполнения программы (из листинга 10.3)

```
* Значение элемента A[0]:
-10.0
```

```
Ошибки деления на ноль нет
# Завершение внутреннего блока
* Значение элемента A[1]:
Попытка деления на ноль
Новое значение 0.0
# Завершение внутреннего блока
* Значение элемента A[2]:
30.0
Ошибки деления на ноль нет
# Завершение внутреннего блока
* Значение элемента A[три]:
# Завершение внутреннего блока
Что-то пошло не так
* Значение элемента A[4]:
# Завершение внутреннего блока
Что-то пошло не так
* Значение элемента A[3]:
20.0
Ошибки деления на ноль нет
# Завершение внутреннего блока
Программа завершила выполнение
```

Программа представляет собой вариацию на тему из предыдущих примеров. Основная ее особенность в том, что мы используем вложенные блоки для обработки исключений. В частности, в теле оператора цикла размещена конструкция `try-except`, в блоке `try` которой имеется еще она конструкция для обработки исключений. В ней содержится блок `try`, блок `except`, а также блоки `else` и `finally`.

Внешний `try`-блок начинается инструкцией `print("* Значение элемента A["+str(k)+"]:")`. Во внутреннем `try`-блоке выполняется команда `A[k]/=(k-1)`. Как и ранее, в этом случае могут генерироваться исключения трех разных типов. Но обрабатывается во внутреннем блоке только исключение класса `ZeroDivisionError`. Поэтому возможны три ситуации:

- при выполнении команды `A[k]/=(k-1)` во внутреннем блоке `try` ошибки не возникли;
- возникла ошибка класса `ZeroDivisionError`;
- возникла ошибка класса, отличного от класса `ZeroDivisionError`.

Если ошибок не было, то выполняется блок `else` и блок `finally`. Если возникала ошибка класса `ZeroDivisionError`, то выполняется внутренний блок `except` и блок `finally` (блок `else` не выполняется). Наконец, если возникает ошибка класса `TypeError` или `IndexError`, то выполняется блок `finally` во внутреннем блоке обработки исключений, после чего исключение, поскольку оно не обрабатывается во внутреннем `except`-блоке, передается для обработки во внешний `except`-блок.

НА ЗАМЕТКУ

Во внешнем `except`-блоке класс исключения не указан. Поэтому в таком блоке обрабатываются исключения всех типов. При этом мы не используем объект исключения. Если бы была необходимость объект исключения использовать, то в `except`-блоке можно было бы указать класс `Exception` и ссылку на переменную, ссылающуюся на объект исключения. Поскольку класс `Exception` является базовым для большинства классов исключений, то в таком блоке перехватывались и обрабатывались бы исключения всех классов (которые являются производными от класса `Exception`).

Искусственное генерирование исключений

— А почему он роет на дороге?
— Да потому, что в других местах все уже перерыто и пересеяно.

Из к/ф «31 июня»

На первый взгляд это может показаться странным, но существует возможность генерировать исключения, что называется, вручную (или искусственно). О чем речь? Речь не о том, чтобы преднамеренно писать ошибочный код. Напротив, код будет совершенно корректным, но при его выполнении создается «иллюзия» того, что возникла ошибка. Благодаря тому, что в Python существует гибкая и эффективная система

обработки ошибок, подход, основанный на генерировании и обработке ошибок, находит самое широкое применение.

Есть разные форматы генерирования исключений. Мы рассмотрим самый простой. В этом случае исключение генерируются с помощью инструкции `raise`. После этой инструкции указывается объект исключения. Его можно создать самостоятельно либо воспользоваться уже готовым (который был создан, например, в результате возникновения реальной ошибки). Если после инструкции `raise` указать название класса исключения, то объект исключения будет создан автоматически. Помимо этого в `except`-блоках иногда используют инструкцию `raise` без указания объекта или класса исключения. В таком случае повторно генерируется обрабатываемое в `except`-блоке исключение.

В листинге 10.4 представлена несложная программа, в которой используется искусственное генерирование исключений.

Листинг 10.4. Искусственное генерирование исключений

```
print("Начинаем генерировать ошибки")
# Создание объекта исключения:
error=Exception("Первая ошибка")
# Первый блок контролируемого кода:
try:
    # Второй блок контролируемого кода:
    try:
        # Третий блок контролируемого кода:
        try:
            # Генерирование исключения:
            raise error
        # Обработка исключения для третьего блока:
        except:
            print(error)
            # Повторное генерирование исключения:
            raise
    # Обработка исключения для второго блока:
    except Exception as e:
```

```
        print("Повторная обработка:")
        print(e)
        # Внутренний блок контролируемого кода:
        try:
            # Генерирование исключения:
            raise ArithmeticError("Вторая ошибка")
        # Обработка исключения для внутреннего блока:
        except ArithmeticError as e:
            print(e)
        # Генерирование исключения:
        raise Warning
# Обработка исключения для первого блока:
except Warning:
    print("Еще одна ошибка")
print("Ошибок больше нет")
```

Результат выполнения программы будет таким.

Результат выполнения программы (из листинга 10.4)

```
Начинаем генерировать ошибки
Первая ошибка
Повторная обработка:
Первая ошибка
Вторая ошибка
Еще одна ошибка
Ошибок больше нет
```

Мы используем несколько вложенных блоков для обработки исключений. Сначала командой `error=Exception("Первая ошибка")` создается объект исключения класса `Exception`. Стоить заметить, что само по себе создание объекта исключения не означает генерирования ошибки. С одной стороны, это самый обычный объект. С другой стороны, мы можем его использовать при генерировании исключения, что мы

и делаем с помощью инструкции `raise error`. В процессе обработки исключения, при попытке «напечатать» объект исключения командой `print(error)` получаем описание, которое было указано при создании объекта ошибки.

ПОДРОБНОСТИ

При передаче ссылки на объект исключения аргументом в метод `print()` для объекта исключения автоматически вызывается метод `__str__()`, который и возвращает текстовую строку, отображаемую методом `print()`. Вообще при создании объекта исключения конструктору класса исключения можно передавать произвольное количество аргументов. Они в виде кортежа записываются в поле `args` объекта исключения. При вызове метода `__str__()` из объекта исключения отображается содержимое этого кортежа.

После выполнения инструкции `raise` в `except`-блоке исключение генерируется повторно и обрабатывается во внешнем `except`-блоке. Объект исключения в этом блоке обозначен как `e`, но на самом деле этот тот же объект, на который ссылается переменная `error`. Поэтому описание у объекта такое же, как и в предыдущем случае.

В этом же `except`-блоке есть своя конструкция `try-except`. В ней инструкцией `raise ArithmeticError("Вторая ошибка")` генерируется новая ошибка класса `ArithmeticError`.

НА ЗАМЕТКУ

В данном случае мы создаем объект исключения `ArithmeticError`, но ссылку на объект в переменную не записываем, а вместо этого инструкцию создания объекта размещаем сразу после инструкции `raise`.

Для обработки данной ошибки предусмотрен `except`-блок, в котором объект ошибки обозначен как `e`. Но теперь переменная `e` ссылается на объект ошибки класса `ArithmeticError`. При обработке ошибки инструкцией `raise Warning` генерируется исключение класса `Warning`. В данном случае мы указали только лишь класс исключения, а объект создается автоматически. Перехватывается и обрабатывается эта ошибка в `except`-блоке самого верхнего уровня.

Создание классов исключений

> Ален ноби, ностра алис! Что означает — ежели один человек построил, другой завсегда разобрать может.
>
> *Из к/ф «Формула любви»*

Благодаря наследованию мы можем создать класс на основе одного из классов исключений. Такой класс через систему наследования встраивается в иерархию классов исключений, объект данного класса применим для генерирования исключений, а сами исключения перехватываются и обрабатываются. То есть идея достаточно простая и прозрачная. В качестве базового допускается использовать, в общем-то, любой класс исключения, но есть настоятельная рекомендация, чтобы это был класс `Exception`. Мы будем придерживаться этой рекомендации.

Пример создания собственного класса для исключения и его использования (для генерирования исключения) содержится в программе в листинге 10.5.

Листинг 10.5. Создание классов исключений

```
# Класс исключения:
class MyError(Exception):
    # Конструктор:
    def __init__(self, code=0, msg="Исключение MyError"):
        self.code=code
        self.message=msg
    # Метод для приведения к текстовому формату:
    def __str__(self):
        txt=self.message+"\nКод ошибки: "+str(self.code)
        return txt
# Внешний блок контролируемого кода:
try:
    print("Создаем собственную ошибку")
    # Внутренний блок контролируемого кода:
    try:
```

```
        # Генерирование исключения:
        raise MyError(123)
    # Внутренний блок обработки исключения:
    except MyError as error:
        print(error)
        # Изменение параметров объекта ошибки:
        error.code=321
        error.message="Та же ошибка MyError"
        # Повторное генерирование исключения:
        raise
# Внешний блок обработки исключения:
except Exception as error:
    print(error)
```

Результат выполнения программы будет следующим.

Результат выполнения программы (из листинга 10.5)

```
Создаем собственную ошибку
Исключение MyError
Код ошибки: 123
Та же ошибка MyError
Код ошибки: 321
```

Класс исключения `MyError` создается наследованием класса `Exception`. В классе описан конструктор, который, кроме ссылки на объект исключения, получает еще два аргумента. Аргументы имеют значения по умолчанию — сделано это для того, чтобы можно при генерировании исключения указывать только имя класса (в этом случае вызывается конструктор без аргументов). При создании объекта исключения определяются значения полей `code` и `message` объекта исключения. Также в классе описан метод `__str__()`, благодаря чему объект исключения можно передавать, например, аргументом методу `print()`. При приведении объекта к текстовому формату возвращается текстовая строка с содержимым полей `message` и `code`.

В программе генерируется и обрабатывается исключение класса `MyError`. Первый раз для этого использована команда `raise MyError(123)`. Для обработки исключения ссылка на объект исключения записывается в переменную `error`, командами `error.code=321` и `error.message="Та же ошибка MyError"` изменяются значения полей объекта ошибки, после чего инструкцией `raise` исключение генерируется повторно (и обрабатывается во внешнем `except`-блоке).

Использование исключений

— Не ходи туда, там тебя ждут неприятности.
— Но как же туда не ходить? Они же ждут!

Из м/ф «Кто сказал мяу»

Далее мы рассмотрим несколько небольших примеров, в которых используется обработка исключительных ситуаций. В программе, представленной в листинге 10.6, вычисляется сумма натуральных чисел. Верхнюю границу суммы вводит пользователь. Собственно, процесс ввода и представляет наибольший интерес.

Листинг 10.6. Сумма натуральных чисел

```
# Оператор цикла с истинным условием:
while True:
    # Считывание значения:
    res=input("Введите натуральное число: ")
    # Контролируемый код:
    try:
        # Попытка преобразования в целое число:
        num=int(res)
        # Если число меньше единицы:
        if num<1:
            # Текст сообщения:
            msg="Число должно быть натуральным"
            # Генерирование исключения:
            raise ArithmeticError(msg)
```

```
    # Обработка исключений:
    except ArithmeticError as error:
        print(error)
    except:
        print("Ошибка ввода")
    # Если ошибок не было:
    else:
        # Завершение оператора цикла:
        break
# Вычисление суммы натуральных чисел:
print("Сумма от 1 до", num,"=", sum(range(num+1)))
```

Результат выполнения программы может быть таким (жирным шрифтом выделены значения, которые вводит пользователь).

Результат выполнения программы (из листинга 10.6)

```
Введите натуральное число: -3
Число должно быть натуральным
Введите натуральное число: пять
Ошибка ввода
Введите натуральное число: 6.7
Ошибка ввода
Введите натуральное число: [1,2,3]
Ошибка ввода
Введите натуральное число: 10
Сумма от 1 до 10 = 55
```

Основная часть программы относится к процессу ввода значения пользователем. Мы используем формально бесконечный оператор цикла, поскольку в качестве условия в операторе `while` указано логическое значение `True`. В теле оператора цикла командой `res=input("Введите натуральное число: ")` в переменную `res` считывается значение, введенное пользователем. В `try`-блоке выполняется попытка преобразовать это значение в целое число (команда `num=int(res)`). Здесь может возникнуть ошибка `ValueError` — если пользователь ввел значение,

которое не преобразуется к целочисленному формату. Если ошибка не возникла, то в условном операторе проверяется условие `num<1`. При его истинности командой `raise ArithmeticError(msg)` генерируется исключение класса `ArithmeticError`.

Для обработки ошибок мы используем два `except`-блока. Один обрабатывает ошибки класса `ArithmeticError`, а второй обрабатывает все остальные ошибки. Кроме этого, использован `else`-блок с инструкцией `break`, которой завершается выполнение оператора цикла.

Работает все это следующим образом. Если пользователь вводит целочисленное значение, то ошибки в `try`-блоке нет и выполняется `else`-блок, в результате чего завершается выполнение оператора цикла. Если пользователь ввел некорректное значение, то генерируется исключение класса `ValueError`. Это исключение обрабатывается во втором `except`-блоке (выполняется команда `print("Ошибка ввода")`) и оператор цикла продолжает работу (выводится новый запрос на ввод числа). Если пользователь ввел целочисленное значение, меньшее единицы, тогда генерируется исключение класса `ArithmeticError`. Исключение обрабатывается в первом `except`-блоке (выполняется команда `print(error)`, которой «печатается» объект исключения `error`). В результате будет отображаться тот текст, который был записан в переменную `msg` и передан в качестве аргумента конструктору при создании объекта исключения.

После того как значение переменной `num` определено, командой `print("Сумма от 1 до", num,"=", sum(range(num+1)))` вычисляется сумма чисел и отображается результат. Здесь собственно сумма вычисляется с помощью встроенной функции `sum()`.

> **НА ЗАМЕТКУ**
>
> Результатом выражения `range(num+1)` является итерируемый объект, соответствующий последовательности чисел от `0` до значения `num` включительно. Поэтому формально вычисляется сумма от `0` до `num`.

В следующем примере описывается функция, предназначенная для решения линейных уравнений вида $Ax = B$. С математической точки зрения это тривиальная задача: решением уравнения является $x = B / A$. Но это только если параметр A не равен нулю. Если же $A = 0$, то возможны два варианта: при $B \neq 0$ уравнение решений не имеет, а при $B = 0$

решением уравнения является любое число. В принципе, не составляет труда рассмотреть все эти варианты с помощью условных операторов. Но здесь мы сталкиваемся с «идеологической» проблемой: если параметр $A \neq 0$, то функция возвращает в качестве результата число. А если $A = 0$, то функция число в качестве результата не возвращает. Формально мы можем организовать функцию так, что будет возвращаться текстовое значение (с информацией о том, что решений нет или что решением является любое число). Но в таком случае тип результата функции потенциально менялся бы от вызова к вызову, а это не очень хорошо. Более грамотным является подход, при котором функция возвращает число, а если однозначного числового решения нет — генерирует исключение. Причем в случае разных исходов исключения относятся к разным классам. Это удобно, поскольку с помощью системы обработки исключений мы сможем их перехватить и обработать.

Рассмотрим программу, представленную в листинге 10.7.

Листинг 10.7. Решение линейного уравнения

```
# Функция для решения линейного уравнения:
def solve(A, B):
    # Контролируемый код:
    try:
        # Преобразование к числовому формату:
        a=float(A)
        b=float(B)
    # Обработка исключения:
    except:
        # Генерирование исключения:
        raise ValueError("Неверный формат данных")
    # Проверка значений коэффициентов:
    if a==0:
        if b!=0:
            # Генерирование исключения:
            raise ArithmeticError("Решений нет")
        else:
            # Генерирование исключения:
```

```
            raise Warning("Решение — любое число")
    # Результат функции:
    return b/a
# Использование функции для решения уравнений:
print("* Решаем линейные уравнения")
# Списки с коэффициентами для уравнений:
A=[2.5,2,"три",10,0,0.0]
B=[3.0,4,0,"пять",5,0]
# Вызов функции с разными аргументами:
for k in range(len(A)):
    a=A[k]
    b=B[k]
    print("[", k+1,"] Уравнение: ", a,"*x = ", b, sep="")
    # Контролируемый код:
    try:
        print("x =", solve(a, b))
    # Обработка исключений:
    except ValueError as error:
        print("Неприятная ситуация № 1")
        print(error)
    except ArithmeticError as error:
        print("Неприятная ситуация № 2")
        print(error)
    except Warning as error:
        print("Странная ситуация")
        print(error)
print("* Все уравнения решены")
```

Ниже представлен результат выполнения программы.

Результат выполнения программы (из листинга 10.7)

```
* Решаем линейные уравнения
[1] Уравнение: 2.5*x = 3.0
```

```
x = 1.2
[2] Уравнение: 2*x = 4
x = 2.0
[3] Уравнение: три*x = 0
Неприятная ситуация № 1
Неверный формат данных
[4] Уравнение: 10*x = пять
Неприятная ситуация № 1
Неверный формат данных
[5] Уравнение: 0*x = 5
Неприятная ситуация № 2
Решений нет
[6] Уравнение: 0.0*x = 0
Странная ситуация
Решение — любое число
* Все уравнения решены
```

Функция для решения линейного уравнения `solve()` описывается с двумя аргументами `A` и `B`, которые мы отождествляем с параметрами уравнения. В принципе, результат функции вычисляется как частное этих значений. Но предварительно выполняются некоторые проверки. Так, в `try`-блоке командами `a=float(A)` и `b=float(B)` выполняется попытка привести аргументы функции к формату чисел с плавающей точкой. На этом этапе может возникнуть ошибка класса `ValueError`. Мы ее перехватываем в `except`-блоке. Но обработка состоит в том, что командой `raise ValueError("Неверный формат данных")` генерируется новое исключение класса `ValueError`. На этот раз мы задаем текст, который будет описывать данную ошибку.

Далее, в случае, если обе переменные `a` и `b` имеют нулевые значения, командой `raise Warning("Решение — любое число")` генерируется исключение класса `Warning`. Если нулевое значение имеет только переменная `a`, то командой `raise ArithmeticError("Решений нет")` генерируется исключение класса `ArithmeticError`.

Если ни один из этих сценариев не реализовался, то командой `return b/a` возвращается числовое значение, являющееся решением уравнения.

НА ЗАМЕТКУ

Таким образом, если функции передать аргументы некорректного типа, то функция сгенерирует исключение класса `ValueError`. Если у уравнения нет решений, генерируется исключение класса `ArithmeticError`. Если решением уравнения является любое число, то функция генерирует исключение класса `Warning`. Если же у уравнения имеется единственное решение, то оно возвращается функцией в качестве результата.

Далее в программе создаются списки `A=[2.5,2,"три",10,0,0.0]` и `B=[3.0,4,0,"пять",5,0]`, элементы которых последовательно используются в качестве коэффициентов, определяющих уравнение. Мы задействовали оператор цикла `for`, благодаря чему тестируем работу функции `solve()` на разных наборах значений для коэффициентов уравнения (в том числе и некорректно заданных). При этом в операторе цикла выполняется перехват и обработка исключений разных типов, генерируемых функцией.

НА ЗАМЕТКУ

Обработка исключений разных типов выполняется довольно однообразно. Но нас в данном случае интересует сам подход, а не его частная реализация. Важно то, что функция генерирует в разных ситуациях разные исключения, и мы их можем перехватывать и обрабатывать. Стоит также заметить, что мы могли бы вместо использования встроенных классов исключений создать собственные классы исключений, которые генерировались бы функцией.

Следующий пример является скорее «экзотическим». Его основная задача — продемонстрировать гибкость и эффективность механизма генерирования и обработки исключений. Начнем с рассмотрения программного кода в листинге 10.8.

Листинг 10.8. Генерирование списка

```
# Класс исключения:
class MyError(Exception):
    # Конструктор:
    def __init__(self):
        self.values=[]
```

```
    # Метод обрабатывает операцию сложения:
    def __add__(self, val):
        self.values.append(val)
        return self
# Функция с рекурсивным вызовом генерирует исключение:
def getMyError(n):
    # Контролируемый код:
    try:
        if n<=1:
            # Генерирование исключения:
            raise MyError
        # Рекурсивный вызов функции:
        getMyError(n-1)
    # Обработка исключения:
    except MyError as error:
        # Генерирование исключения:
        raise error+n
# Функция для создания списка:
def getList(n):
    # Контролируемый код:
    try:
        # Вызов функции, генерирующей исключение:
        getMyError(n)
    # Обработка исключения:
    except MyError as error:
        # Результат функции:
        return error.values
# Создание списков:
A=getList(10)
print(A)
B=getList(7.5)
print(B)
```

Так выглядит результат выполнения программы.

Результат выполнения программы (из листинга 10.8)

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5]
```

В программе описывается класс исключения `MyError`, который наследует класс `Exception`. В классе описан конструктор, который при вызове создает у объекта исключения поле `values`, ссылающееся на пустой список. Также в классе описан специальный метод `__add__()`, который вызывается для обработки ситуации, когда к объекту исключения прибавляется некоторое значение (обозначено как `val`). В этом случае командой `self.values.append(val)` в конец списка, на который ссылается поле `values` объекта исключения, добавляется значение `val`. В качестве результата метод возвращает ссылку на объект исключения (инструкция `return self`).

Помимо класса `MyError` в программе описаны две функции. Функция `getMyError()` при вызове генерирует исключение класса `MyError`, но происходит это не очень тривиальным образом. В теле функции в блоке `try` при условии, что значение аргумента функции меньше или равно `1` (условие `n<=1`) командой `raise MyError` генерируется исключение класса `MyError`. Это исключение обрабатывается в `except`-блоке, но обработка состоит в выполнении команды `raise error+n`, где через `error` обозначен объект исключения, а через `n` обозначен аргумент функции `getMyError()`. Получается, что в процессе обработки исключения в поле-список `values` объекта исключения добавляется значение аргумента `n`. Затем снова генерируется исключение с этим уже измененным объектом. Все это происходит, напомним, если значение аргумента `n` не превышает `1`. Если данное условие не выполнено, то исключение не генерируется и командой `getMyError(n-1)` выполняется рекурсивный вызов функции с аргументом `n-1`.

Что происходит при вызове функции `getMyError()`? Это будет проще понять, если представить, что при рекурсивном вызове каждый раз вызывается новая функция. Мы их будем различать по значению аргумента. Итак, допустим, что вначале вызывается функция с аргументом `5`. Поскольку `5` больше `1`, то вызывается эта же функция с аргументом `4`. Далее по цепочке вызывается функция с аргументом `3`, с аргументом `2` и, наконец, с аргументом `1`. В последнем случае генерируется исключение

с объектом, поле values которого ссылается на пустой список. Функция, вызванная с аргументом 1, обрабатывает исключение: в поле-список values добавляется значение 1, и исключение генерируется повторно. Функция, вызванная с аргументом 1, это исключение уже не обрабатывает. Оно передается в функцию, вызвавшую данную (то есть функцию, вызванную с аргументом 2). Важно то, что функция с аргументом 1 вызывалась в try-блоке функции с аргументом 2. Поэтому исключение обрабатывается в except-блоке функции, вызванной с аргументом 2. В нем в список, на который ссылается поле values объекта исключения, добавляется значение 2, и исключение генерируется снова. Оно обрабатывается в функции, вызванной с аргументом 3. Так продолжается до тех пор, пока исключение не будет передано для обработки в функцию, вызванную с аргументом 5. При обработке в конец поля-списка values добавляется значение 5, и исключение снова генерируется. На этот раз его обрабатывать некому. Чтобы перехватить такое исключение, функцию с аргументом 5 следует вызывать в try-блоке. При этом объект исключения, выброшенный из функции, вызванной с аргументом 5, содержит список с числами от 1 до 5 включительно.

Функция getList() предназначена для создания списка. В качестве аргумента ей передается, как мы предполагаем, число (обозначено как n). В теле функции в try-блоке выполняется команда getMyError(n), что приводит к генерированию исключения класса MyError. В процессе обработки этого исключения командой return error.values в качестве результата функции возвращается список values из объекта исключения error.

НА ЗАМЕТКУ

Получается, что в функции вызывается функция getMyError(), это приводит к генерированию исключения, исключение перехватывается и список, на который ссылается поле values объекта исключения, возвращается в качестве результата функции. Данный список содержит (с конца в начало) значения n, n-1, n-2 и так далее до первого числа, которое не превышает 1.

Также в программе командами A=getList(10) и B=getList(7.5) мы создаем два списка и проверяем их содержимое.

НА ЗАМЕТКУ

Читателю предлагается попробовать объяснить принцип формирования этих списков.

Знакомство с потоками

> И мы с пути кривого ни разу не свернем, а надо будет — снова пойдем кривым путем.
>
> *Из к/ф «Айболит-66»*

Потоками называются блоки программного кода, которые выполняются одновременно. До этого мы имели дело с программами, в которых был всего один поток — это поток, в котором выполнялась программа. Такой поток называется *главным*. Фактически выполнение программы означает запуск на выполнение главного потока. А уже в этом главном потоке можно запускать и другие потоки, которые называются *дочерними*. Дочерние потоки выполняются одновременно с главным потоком, и в них тоже можно запускать дочерние потоки.

Перейдем к простому примеру. Допустим, у нас имеется две функции — назовем их `alpha()` и `bravo()`. Мы в программе вызываем сначала функцию `alpha()`, а затем функцию `bravo()`. Что происходит в этом случае? Сначала выполняется код функции `alpha()`, и только после этого будет выполняться код функции `bravo()`. Это обычная ситуация. Что изменится, если мы захотим вызвать функции `alpha()` и `bravo()` в разных потоках? Если так, то функции `alpha()` и `bravo()` будут выполняться одновременно.

НА ЗАМЕТКУ

Если у компьютера один процессор, то многопоточность реализуется за счет того, что процессор распределяет время выполнения между разными потоками. В итоге получается иллюзия того, что потоки выполняются одновременно. Отсюда понятно, что разбивка программы на несколько потоков далеко не всегда приводит к сокращению времени выполнения программы.

Таким образом, задача состоит в том, чтобы вызвать функцию `alpha()`, а затем, не дожидаясь завершения ее выполнения, вызвать функцию `bravo()`. Что для этого нужно? Как минимум необходимо иметь функции `alpha()` и `bravo()`. Функцию `alpha()` мы можем вызвать в главном потоке программы. А вот функция `bravo()` должна вызываться в дочернем потоке. Другим словами, функцию `alpha()` вызываем «как обычно», а функцию `bravo()` будем вызывать «в специальном режиме».

НА ЗАМЕТКУ

Есть два основных подхода относительно создания дочернего потока. Мы рассмотрим оба.

Основные утилиты для работы с потоками собраны в модуле `threading`. В частности, он содержит класс `Thread`. Если создать объект этого класса и вызвать из объекта метод `start()`, то в результате запускается на выполнение дочерний поток. В этом дочернем потоке будет выполняться программный код метода `run()` объекта, из которого вызывался метод `start()`. Метод `run()`, в свою очередь, описан так, что при вызове этого метода на самом деле вызывается функция, имя которой указано в качестве значения аргумента `target` конструктора класса `Thread`. Таким образом, если мы хотим, чтобы в потоке выполнялась функция `bravo()`, то при создании объекта класса `Thread` нам достаточно для аргумента с ключом `target` указать значение `bravo` (инструкция выглядит как `target=bravo`). Собственно и все. Как это выглядит на практике, показано в программе в листинге 10.9.

Листинг 10.9. Знакомство с потоками

```
# Импорт класса Thread:
from threading import Thread
# Импорт функции sleep():
from time import sleep
# Функция для вызова в главном потоке:
def alpha():
    for k in range(5):
        # Пауза в выполнении потока:
        sleep(1.5)
        print("[", k+1,"] Alpha", sep="")
# Функция для выполнения в дочернем потоке:
def bravo():
    for k in range(5):
        print("[", k+1,"] Bravo", sep="")
        # Пауза в выполнении потока:
        sleep(1)
```

```
# Создание объекта дочернего потока:
t=Thread(target=bravo)
# Запуск дочернего потока на выполнение:
t.start()
# Вызов функции в главном потоке:
alpha()
```

Сразу разберем программный код. С помощью двух `import`-инструкций мы включаем в программу класс `Thread` из модуля `threading` и функцию `sleep()` из модуля `time`. Функция `sleep()` нам нужна для того, чтобы делать паузы в выполнении потоков.

ПОДРОБНОСТИ

При вызове функции `sleep()` поток, в котором вызывается функция, приостанавливает выполнение (делает паузу). Время (в секундах), на которое выполняется приостановка в выполнении потока, указывается в качестве аргумента функции `sleep()`.
В данном случае мы делаем паузы в выполнении потоков для того, чтобы легче было проследить последовательность выполнения команд.

Функции `alpha()` и `bravo()` описаны однотипно: при вызове каждой из функций отображается по пять сообщений соответствующего содержания и номер сообщения. В функции `alpha()` между сообщениями выдерживается пауза в полторы секунды, а между сообщениями в функции `bravo()` выдерживается пауза в одну секунду.

НА ЗАМЕТКУ

В функции `alpha()` сначала делается пауза, а затем отображается сообщение. В функции `bravo()` сначала отображается сообщение, а затем делается пауза.

Командой `t=Thread(target=bravo)` создается объект `t` класса `Thread` (объект дочернего потока). Поскольку в качестве аргумента конструктору передана инструкция `target=bravo`, то при запуске потока будет выполняться код функции `bravo()`. Запускаем дочерний поток на выполнение командой `t.start()`. После этого в главном потоке вызываем функцию `alpha()`. Результат выполнения программы может быть таким.

Результат выполнения программы (из листинга 10.9)

```
[1] Bravo
[2] Bravo
[1] Alpha
[3] Bravo
[2] Alpha
[4] Bravo
[5] Bravo
[3] Alpha
[4] Alpha
[5] Alpha
```

Стоит заметить, что это лишь возможный вариант выполнения программы. Сообщения появляются с заметной паузой. А поскольку потоки не синхронизированы и одновременно отображают сообщения в области вывода, то сообщения разных потоков могут «накладываться» друг на друга.

НА ЗАМЕТКУ

Также важна последовательность, в которой мы запускаем поток с функцией `bravo()` и вызываем функцию `alpha()`. Если сначала в программе указать команду вызова функции `alpha()`, а затем поместить команду запуска дочернего потока, то сначала выполнится код функции `alpha()`, и только после этого будет запущен на выполнение дочерний поток.

В рассмотренном примере продолжительности пауз в выполнении потоков подобраны так, что дочерний поток заканчивается раньше главного. Нередко необходимо добиться того, чтобы один поток дождался завершения выполнения другого потока. Полезным в таком случае будет метод `join()`. Метод вызывается из объекта дочернего потока. Эффект следующий: поток, в котором размещается соответствующая команда (в нашем случае это главный поток), будет дожидаться завершения выполнения потока, из объекта которого вызван метод.

Стоит упомянуть еще одно обстоятельство: обычно в потоке необходимо вызывать функцию, которой следует передать аргументы. В качестве

аргумента с ключом `target` указывается имя вызываемой в потоке функции. Если этой функции при вызове в потоке следует передать аргументы, то их передают в виде кортежа по ключу `args` конструктору класса `Thread`.

НА ЗАМЕТКУ

На самом деле по ключу `target` можно передавать имя не только функции, но и вызываемого объекта — то есть объекта, для которого определен специальный метод `__call__()`.

Как иллюстрация в листинге 10.10 представлена модифицированная версия предыдущего примера.

Листинг 10.10. Вызов в потоке функции с аргументами

```
from threading import Thread
from time import sleep
# Функция с тремя аргументами:
def display(count, time, text):
    for k in range(count):
        # Пауза в выполнении потока:
        sleep(time)
        print("[", k+1,"] ", text, sep="")
print("Начинается выполнение программы")
# Создание объекта дочернего потока:
t=Thread(target=display, args=(5,2,"Дочерний поток"))
# Запуск дочернего потока на выполнение:
t.start()
# Вызов функции в главном потоке:
display(3,1.5,"Главный поток")
# Ожидание завершения дочернего потока:
t.join()
print("Программа завершила выполнение")
```

Результат выполнения программы может быть следующим.

Результат выполнения программы (из листинга 10.10)

```
Начинается выполнение программы
[1] Главный поток
[1] Дочерний поток
[2] Главный поток
[2] Дочерний поток
[3] Главный поток
[3] Дочерний поток
[4] Дочерний поток
[5] Дочерний поток
Программа завершила выполнение
```

В данном случае мы используем всего одну функцию `display()`, которую вызываем и в главном, и в дочернем потоках. У функции три аргумента, которые определяют количество отображаемых сообщений (аргумент `count`), продолжительность паузы (в секундах) между сообщениями (аргумент `time`), а также отображаемый в сообщении текст (аргумент `text`).

Для создания объекта дочернего потока использована команда `t=Thread(target=display, args=(5,2,"Дочерний поток"))`. Аргументы конструктора класса `Thread` означают, что в дочернем потоке будет вызываться функция `display()`, а аргументами ей будут передаваться значения `5`, `2` и `"Дочерний поток"` (отображается `5` сообщений с интервалом в `2` секунды).

После запуска на выполнение дочернего потока в главном потоке выполняется команда `display(3,1.5,"Главный поток")` (отображается `3` сообщения с интервалом в `1.5` секунды). По идее, главный поток должен завершить выполнение ранее, чем дочерний поток. Но поскольку в главном потоке есть команда `t.join()`, то следующая после нее команда `print("Программа завершила выполнение")` выполняется только после того, как дочерний поток завершит выполнение.

НА ЗАМЕТКУ

В некоторых случаях бывает необходимо определить, выполняется ли поток. Если так, то используют метод `is_alive()`. Метод вызывается из объекта потока и возвращает значение `True`, если

поток активный (выполняется). В противном случае методом возвращается значение `False`.

Количество активных на данный момент потоков можно узнать с помощью функции `active_count()`.

Функция `enumerate()` в качестве результата возвращает список со ссылками на объекты активных на данный момент потоков. Также полезными могут быть функции `current_thread()` и `main_thread()`, которые в качестве результата возвращают соответственно ссылку на текущий поток (в котором вызывается функция) и ссылку на главный поток.

Еще существует такое понятие, как демон-поток. Его особенность в том, что при завершении главного потока демон-поток завершается автоматически. Чтобы сделать поток демон-потоком, необходимо полю `daemon` объекта потока присвоить значение `True`. По умолчанию создаваемые потоки являются приоритетными: поле `daemon` объекта потока имеет значение `False`.

Код, предназначенный для выполнения в потоке, можно реализовать не только в виде функции, но и как вызываемый объект (то есть объект класса, в котором реализован специальный метод `__call__()`). В листинге 10.11 представлена программа, аналогичная предыдущему примеру, но на этот раз вместо функции для выполнения в потоке создается вызываемый объект.

Листинг 10.11. Использование вызываемого объекта

```
from threading import Thread
from time import sleep
# Класс для создания вызываемого объекта:
class MyClass:
    # Конструктор:
    def __init__(self, text):
        self.text=text
    # Метод вызывается при вызове объекта:
    def __call__(self, count, time):
        for k in range(count):
            sleep(time)
            print("[", k+1,"] ", self.text, sep="")
print("Начинается выполнение программы")
```

```
# Создание вызываемого объекта:
obj=MyClass("Дочерний поток")
# Создание объекта дочернего потока:
t=Thread(target=obj, kwargs={"time":2,"count":5})
# Запуск дочернего потока на выполнение:
t.start()
# Вызов анонимного объекта в главном потоке:
MyClass("Главный поток")(3,1.5)
# Ожидание завершения дочернего потока:
if t.is_alive():
    t.join()
print("Программа завершила выполнение")
```

Результат выполнения этой программы такой же, как и результат выполнения программы из листинга 10.10. Но сам код стоит проанализировать.

На этот раз мы описываем класс `MyClass`, у которого есть конструктор и метод `__call__()`. При вызове конструктора присваивается значение полю `text` объекта класса. Метод `__call__()` вызывается при вызове объекта. Предполагается, что при вызове объекта аргументами передаются два значения (соответственно, аргументы называются `count` и `time`). В теле метода выполняется оператор цикла, с помощью которого отображаются сообщения. Количество итераций определяется аргументом `count`, время задержки между сообщениями определяется аргументом `time`. В самом сообщении используется значение поля `text` объекта класса (дается инструкцией `self.text`).

Командой `obj=MyClass("Дочерний поток")` создается объект класса `MyClass`. Этот объект можно вызывать, передав два аргумента. Мы используем объект для вызова в дочернем потоке. Для создания объекта дочернего потока мы задействовали команду `t=Thread(target=obj, kwargs={"time":2,"count":5})`. Здесь в качестве значения аргумента `target` указана ссылка на объект `obj`. Этот объект, как отмечалось, будет вызываться в дочернем потоке. Ради разнообразия, значения аргументов, которые будут передаваться при вызове, мы определяем по ключу. Для этого для аргумента `kwargs` конструктора класса `Thread` значением указывается словарь `{"time":2,"count":5}` с названиями аргументов и их значениями.

Дочерний поток на выполнение запускается командой `t.start()`. Также в главном потоке выполняется вызов анонимного объекта (команда `MyClass("Главный поток")(3,1.5)`).

ПОДРОБНОСТИ

Инструкцией `MyClass("Главный поток")` создается объект класса `MyClass`. Поскольку ссылку на этот объект мы в переменную не записываем, то объект анонимный. Этот объект можно вызвать. Для этого ему передаются два аргумента, которые указываются в круглых скобках после инструкции `MyClass("Главный поток")`.

Перед выполнением команды `t.join()` сначала проверяется условие `t.is_alive()`, истинное, если дочерний поток выполняется.

НА ЗАМЕТКУ

Получается так: мы проверяем, завершил ли выполнение дочерний поток, и если он еще выполняется, то главный поток ожидает его завершения.

Еще один подход для реализации потоков состоит в том, чтобы путем наследования на основе класса `Thread` создать производный класс и переопределить в нем метод `run()`.

НА ЗАМЕТКУ

Нередко кроме переопределения метода `run()` переопределяют и конструктор класса `Thread`. Переопределять другие методы не рекомендуется.

В листинге 10.12 представлена программа, в которой для создания дочерних потоков используются объекты класса, производного от класса `Thread`.

Листинг 10.12. Создание потоков и наследование

```
from threading import Thread
from time import sleep
# Производный класс:
class MyThread(Thread):
```

```
    # Конструктор:
    def __init__(self, count, time, text):
        # Вызов конструктора базового класса:
        super().__init__()
        # Присваивание значений полям:
        self.count=count
        self.time=time
        self.text=text
    # Переопределение метода для выполнения в потоке:
    def run(self):
        for k in range(self.count):
            sleep(self.time)
            print("[", k+1,"] ", self.text, sep="")
print("Начинается выполнение программы")
# Создание объектов для дочерних потоков:
A=MyThread(5,2,"Alpha")
B=MyThread(3,1.5,"Bravo")
# Запуск дочерних потоков на выполнение:
A.start()
B.start()
# Ожидание завершения дочерних потоков:
if A.is_alive():
    A.join()
if B.is_alive():
    B.join()
print("Программа завершила выполнение")
```

Результат выполнения программы может быть таким, как показано ниже.

Результат выполнения программы (из листинга 10.12)

```
Начинается выполнение программы
[1] Bravo
```

```
[1] Alpha
[2] Bravo
[2] Alpha
[3] Bravo
[3] Alpha
[4] Alpha
[5] Alpha
Программа завершила выполнение
```

В программе описывается класс `MyThread`, наследующий класс `Thread`. В конструкторе класса сначала вызывается конструктор базового класса (команда `super().__init__()`), после чего командами `self.count=count`, `self.time=time` и `self.text=text` присваиваются значения полям объекта класса. Значения полей определяются аргументами конструктора.

НА ЗАМЕТКУ

Объект для потока планируется создавать на основе производного класса. У объекта производного класса будут поля, значения которых могут использоваться в процессы выполнения кода потока. Получается, что часть параметров, необходимых для выполнения кода потока, будут «спрятаны» в объекте потока.

Метод `run()` в классе `MyThread` переопределяется таким образом, что выполняется оператор цикла, количество итераций определяется значением поля `count` объекта потока (объекта, из которого вызывается метод `run()`). За каждую итерацию отображается сообщение со значением из поля `text`, а между сообщениями выполняется пауза, длительность которой определяется значением поля `time`.

В программе командами `A=MyThread(5,2,"Alpha")` и `B=MyThread(3,1.5,"Bravo")` создаются объекты для дочерних потоков, а командами `A.start()` и `B.start()` дочерние потоки запускаются на выполнение. В главном потоке ожидается завершение выполнения дочерних потоков, после чего завершает свое выполнение и главный поток.

Взаимодействие потоков

> Эх, погубят тебя слишком широкие возможности.
>
> *Из к/ф «Айболит-66»*

До этого мы рассматривали потоки, которые выполняются вне зависимости друг от друга. Но в общем случае потоки могут «взаимодействовать», в том числе при помощи использования общих ресурсов. В таких случаях потоки нужно *синхронизировать* — иначе могут возникнуть проблемы.

НА ЗАМЕТКУ

Почему потоки нужно синхронизировать, поясним на простом примере, не относящемся непосредственно к программированию. Представим ситуацию, когда трое парковых служащих высаживают деревья в парке. Один копает ямки, другой размещает дерево в ямке, а третий засыпает землей ямку с деревом. В данном случае служащие являются аналогом потоков, а парковая лужайка, на которой они высаживают деревья, — аналог общего ресурса.

Понятно, что при такой организации работы действия трех служащих должны быть скоординированы: например, если третий служащий начнет засыпать ямки быстрее, чем второй будет помещать туда саженец, результат получится не очень хорошим. Чтобы такого не происходило, нужно, чтобы служащие между собой общались в процессе работы. Еще один вариант — разбить лужайку на отдельные участки, и, пока один служащий выполняет на этом участке свои операции, другим следует запретить «вторгаться» на его территорию. Нечто похожее происходит и при синхронизации потоков.

Обычно синхронизация выполняется за счет ограничения доступа к общим ресурсам таким образом, что, пока один поток работает с ресурсом, доступ других потоков к этому ресурсу блокируется. В Python есть несколько способов для решения такой задачи (в том числе и вариант, когда потоки в процессе работы обмениваются «сообщениями»). Далее мы кратко рассмотрим некоторые из них.

Один из наиболее простых способов синхронизации процесса доступа к общим ресурсам базируется на использовании *объекта блокировки* — объекта, который создается на основе класса `Lock` из модуля

threading. Идея достаточно простая: создается объект блокировки, общий для всех потоков, и этот объект может находиться в одном из двух состояний: *заблокированном* и *разблокированном*. Синхронизация работы потоков (в части доступа к общим ресурсам) выполняется следующим образом. При попытке доступа к ресурсу поток переводит объект блокировки в заблокированное состояние. Далее выполняются некоторые операции с общим ресурсов, после чего поток переводит объект блокировки в разблокированное состояние. Ключевой момент здесь в том, что если поток пытается заблокировать объект блокировки, а он уже находится в заблокированном состоянии, то поток приостанавливает выполнение и ожидает, пока объект блокировки не будет разблокирован тем потоком, который его заблокировал.

Для перевода объекта блокировки в заблокированное состояние используют метод `acquire()`. Для разблокировки заблокированного объекта используется метод `release()`.

ПОДРОБНОСТИ

Если вызвать метод `release()` из разблокированного объекта (то есть если попытаться разблокировать уже разблокированный объект), будет сгенерировано исключение класса `RuntimeError`.

Как пример использования объекта блокировки мы рассмотрим небольшую программу, в которой два дочерних потока пытаются изменить значение одной общей глобальной переменной, созданной в главном потоке. Программа представлена в листинге 10.13.

Листинг 10.13. Блокировка ресурса

```
from threading import *
from time import sleep
# Функция для выполнения в потоке:
def calc(txt, op):
    # Глобальная переменная:
    global number
    # Оператор цикла:
    for k in range(3):
        # Блокировка ресурса:
```

```
        mylock.acquire()
        print(txt,": ресурс заблокирован", sep="")
        # Контролируемый код:
        try:
            # Считывание значения переменной:
            print(txt,"прочитано:", number)
            # Запоминается значение переменной:
            val=number
            # Пауза в выполнении потока:
            sleep(1)
            # Изменение значения переменной:
            if op:
                number=val+1
            else:
                number=val-1
            # Отображение значения переменной:
            print(txt," записано:", number)
        # Код выполняется всегда:
        finally:
            print(txt,": ресурс разблокирован", sep="")
            print("----------------------")
            # Разблокировка ресурса:
            mylock.release()
        # Пауза в выполнении потока:
        sleep(1)
# Начальное значение глобальной переменной:
number=0
# Объект блокировки:
mylock=Lock()
# Объекты дочерних потоков:
A=Thread(target=calc, args=["A", True])
B=Thread(target=calc, args=["B", False])
```

```
# Запуск дочерних потоков на выполнение:
A.start()
B.start()
# Ожидание завершения потоков:
A.join()
B.join()
```

Возможный результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 10.13)

```
A: ресурс заблокирован
A прочитано: 0
A  записано: 1
A: ресурс разблокирован
------------------------
B: ресурс заблокирован
B прочитано: 1
B  записано: 0
B: ресурс разблокирован
------------------------
A: ресурс заблокирован
A прочитано: 0
A  записано: 1
A: ресурс разблокирован
------------------------
B: ресурс заблокирован
B прочитано: 1
B  записано: 0
B: ресурс разблокирован
------------------------
A: ресурс заблокирован
A прочитано: 0
A  записано: 1
```

```
A: ресурс разблокирован
----------------------
B: ресурс заблокирован
B прочитано: 1
B  записано: 0
B: ресурс разблокирован
----------------------
```

Важное место в программе занимает функция `calc()`, которую мы используем для выполнения в дочерних потоках. Функция достаточно простая: при ее вызове считывается значение глобальной переменной `number`, отображается значение этой переменной, затем значение переменной изменяется, после чего отображается новое значение переменной `number`. При этом между выполнением операций делаются паузы. У функции два аргумента. Первый аргумент `txt` определяет текст, который будет отображаться в сообщениях. Аргумент `op` является, как мы предполагаем, логическим, и определяет, как изменяется значение переменной `number` (увеличивается на единицу или уменьшается на единицу).

В программе создается глобальная переменная `number` с начальным нулевым значением. Командой `mylock=Lock()` создается объект блокировки, а командами `A=Thread(target=calc, args=["A", True])` и `B=Thread(target=calc, args=["B", False])` создаются объекты дочерних потоков. Первый дочерний поток пытается увеличивать значение переменной `number`, а второй дочерний поток пытается уменьшать значение переменной `number`. В теле функции `calc()`, которая выполняется в каждом из потоков, перед считыванием значения переменной `number` выполняется команда `mylock.acquire()`, которой объект блокировки переводится в заблокированное состояние. Если этот объект заблокирован одним потоком, то второй поток его заблокировать не сможет и станет ожидать, пока объект блокировки будет разблокирован. Разблокировка объекта блокировки выполняется с помощью команды `mylock.release()`.

ПОДРОБНОСТИ

Поскольку в процессе выполнения операций с общими ресурсами теоретически могут возникать ошибки, то желательно удостовериться, что даже в таких случаях объект блокировки будет разблокирован. Поэтому команды, связанные с выполнением операций

с общими ресурсами, обычно помещаются в блок `try`, а команда разблокировки объекта блокировки помещается в блок `finally`. Но в принципе это не обязательно.

Общий эффект получается такой: если один поток начинает работу с переменной `number`, то другой поток будет ожидать завершения выполнения этих операций.

НА ЗАМЕТКУ

Желающие могут посмотреть, каким будет результат выполнения программы без использования объекта блокировки. Для этого в программном коде функции `calc()` достаточно закомментировать (поместить в начале команды символ `#`) команды `mylock.acquire()` и `mylock.release()`.

ПОДРОБНОСТИ

Кроме класса `Lock`, для создания объекта блокировки можно использовать и класс `RLock`. Принципиальная разница между объектами блокировки, созданными на основе этих классов, состоит в реакции на попытку повторного блокирования в одном и том же потоке. Например, рассмотрим такой код:

```
from threading import *
mylock=RLock()
mylock.acquire()
print("Первое сообщение")
mylock.acquire()
print("Второе сообщение")
mylock.release()
mylock.release()
```

Здесь на основе класса `RLock` создается объект блокировки `mylock`, объект переводится в заблокированное состояние (команда `mylock.acquire()`), отображается сообщение, после чего объект снова блокируется, и отображается еще одно сообщение. И только после этого выполняется разблокировка объекта (дважды, поскольку каждой `acquire`-инструкции должна соответствовать своя `release`-инструкция). В результате выполнения этого

кода отображаются оба сообщения. Но если вместо класса `RLock` использовать класс `Lock`, то программа «зависнет» после отображения первого сообщения. Причина в том, что повторный вызов метода `acquire()` из уже заблокированного объекта приводит к тому, что поток ожидает, пока объект будет разблокирован. В данном случае получится, что главный поток заблокировал сам себя. Если для создания объекта блокировки используется класс `RLock`, то поток сам себя не блокирует (при этом другие потоки блокируются).

Класс `Semaphore` позволяет создавать объекты блокировки, которые разрешают получать доступ к ресурсу определенному количеству потоков (с учетом «кратности» обращения к ресурсу). Например, имеется такой код:

```
from threading import *
mylock=Semaphore(5)
num=int(input("Введите целое число: "))
for k in range(num):
    mylock.acquire()
    print(k+1, end=" ")
for k in range(num):
    mylock.release()
```

Командой `mylock=Semaphore(5)` создается объект блокировки, позволяющий доступ к ресурсу пяти потокам. Поэтому если пользователь вводит число (значение переменной `num`) не больше `5`, то в результате выполнения кода будет отображаться последовательность из натуральных чисел до значения переменной `num` включительно. Если пользователь введет число больше `5`, то после отображения значения `5` программа «зависает». Причина в том, что блокировка объекта `mylock` к тому моменту уже выполнялась `5` раз, и поэтому при следующей попытке заблокировать объект поток (главный в данном случае) станет ждать, пока объект не будет разблокирован. Поскольку этого не происходит, программа не может завершить выполнение.

Как альтернативу к классу `Semaphore` можно использовать класс `BoundedSemaphore`. Разница между ними в том, что при вызове метода `release()` из незаблокированного `Semaphore`-объекта количество потоков, для которых открыт доступ, увеличивается на единицу. Если вызвать метод `release()` из незаблокированного `BoundedSemaphore`-объекта генерируется исключение класса `ValueError`.

Эффективное средство коммуникации между потоками реализуется с помощью объектов класса `Event`. Объект класса может находиться в одном из двух состояний: с установленным флагом (значение `True`) и с отмененным флагом (значение `False`). Узнать, установлен флаг или нет, можно с помощью метода `is_set()`. Чтобы установить флаг, вызывают метод `set()`. Отменить флаг можно с помощью метода `clear()`. Еще есть метод `wait()`. Этот метод блокирует выполнение потока (в котором вызван метод) до тех пор, пока не будет установлен флаг у объекта, из которого вызывался метод `wait()`. Как все это может выглядеть на практике, иллюстрирует программа в листинге 10.14.

Листинг 10.14. Взаимодействие потоков

```
from threading import *
from time import sleep
# Функция для выполнения в потоке:
def display(txt):
    A=[1,2]
    B=["A","B"]
    sleep(1)
    # Ожидание установки флага:
    myevent.wait()
    # Отмена установки флага:
    myevent.clear()
    for a in A:
        print("[", a,"] ", txt, sep="")
    # Установка флага:
    myevent.set()
    sleep(1)
    # Ожидание установки флага:
    myevent.wait()
    # Отмена установки флага:
    myevent.clear()
    for b in B:
        print("[", b,"] ", txt, sep="")
```

```
    # Установка флага:
    myevent.set()
# Создание объекта:
myevent=Event()
# Установка флага:
myevent.set()
# Объекты дочерних потоков:
F=Thread(target=display, args=["Первый"])
S=Thread(target=display, args=["Второй"])
# Запуск дочерних потоков:
F.start()
S.start()
# Ожидание завершения дочерних потоков:
F.join()
S.join()
```

Результат выполнения программы, скорее всего, будет таким.

Результат выполнения программы (из листинга 10.14)

```
[1] Первый
[2] Первый
[1] Второй
[2] Второй
[A] Первый
[B] Первый
[A] Второй
[B] Второй
```

Разберемся, что же происходит при выполнении программы. Все самое важное описано в функции `display()`, которая выполняется в двух дочерних потоках в программе. У функции один аргумент, который определяет текст, отображаемый в сообщениях. Функция отображает в общей сложности четыре сообщения. Начинается выполнение кода с создания двух списков, после чего делается секундная пауза. Команда

myevent.wait() означает, что поток, в котором вызвана функция, будет ожидать установки флага для объекта myevent класса Event. Как только флаг установлен, командой myevent.clear() он отменяется, и затем выполняется оператор цикла. После этого оператора цикла командой myevent.set() флаг снова устанавливается, делается секундная пауза, выполняется команда myevent.wait() (ожидание установки флага), команда myevent.clear() (отмена установки флага), выполняется оператор цикла (отображается два сообщения), затем командой myevent.set() снова устанавливается флаг. Если функция вызывается в дочернем потоке, то все это выглядит так: секундная пауза, ожидание установки флага для объекта myevent, отмена флага для этого объекта, два сообщения, установка флага, секундная пауза, ожидание установки флага, отмена флага, два сообщения, установка флага. Обратить внимание стоит на два момента.

- Отмена установки флага выполняется для того, чтобы просигнализировать другим потокам, что выполняется данный поток.
- После отображения двух сообщений флаг снова устанавливается, а поскольку дальше делается пауза, то в это время в работу может «вклиниться» другой поток (так, собственно, и происходит).

В программе командой myevent=Event() создается объект класса Event, командой myevent.set() для него устанавливается флаг. Объекты дочерних потоков создаются командами F=Thread(target=display, args=["Первый"]) и S=Thread (target=display, args=["Второй"]). Для запуска потоков на выполнение используем метод start(), а для ожидания завершения потоков использован метод join().

Что касается непосредственно результата выполнения программы, то стоит заметить, что, хотя дочерние потоки запускаются практически одновременно, сообщения они выводят поочередно, блоками по два сообщения.

НА ЗАМЕТКУ

Существуют и другие утилиты, используемые для синхронизации и управления потоками. Так, класс Condition позволяет, по сравнению с классом Event, создавать более «продвинутые» объекты для синхронизации работы потоков, в частности за счет использования метода notify() для коммуникации между потоками.
Объекты класса Barrier используются для синхронизации потоков в случае, если необходимо добиться скоординированного начала и завершения выполнения потоков.

Примеры использования потоков

Куда? Эй, куда же вы все-то разбежались?
Кто-нибудь, держите меня!

Из к/ф «Айболит-66»

Далее мы рассмотрим несколько небольших примеров, в которых используется многопоточное программирование. Начнем с программы, в которой вычисляется сумма целых чисел. Это простая задача, но решать ее мы будем не очень привычным образом: сумма будет вычисляться определенное время. Другими словами, вместо того, чтобы вычислить сумму из определенного количества слагаемых, мы запустим на выполнение оператор цикла, который будет выполняться определенное время. Для этого воспользуемся дочерним потоком. Программа представлена в листинге 10.15.

Листинг 10.15. Вычисление суммы целых чисел

```
from threading import *
from time import sleep
# Функция для вызова в потоке:
def mysum():
    # Глобальная переменная:
    global num
    # Добавка к сумме:
    k=1
    # Текст для отображения:
    txt=str(num)
    # Вычисление суммы:
    while myevent.is_set():
        # К сумме прибавляется слагаемое:
        num+=k
        # Новое значение для текста:
        txt+=" + "+str(k)
        # Отображение текущего значения суммы:
        print("[", k,"] ", txt," = ", num, sep="")
```

```
        # Добавка для следующей итерации:
        k+=1
        # Пауза в выполнении потока:
        sleep(0.3)
print("Сумма целых чисел")
# Создание объекта дочернего потока:
t=Thread(target=mysum)
# Начальное значение для суммы:
num=0
# Объект для синхронизации потоков:
myevent=Event()
# Установка флага:
myevent.set()
# Запуск дочернего потока на выполнение:
t.start()
# Пауза в выполнении главного потока:
sleep(2)
# Отмена флага:
myevent.clear()
# Ожидание завершения выполнения дочернего потока:
if t.is_alive():
    t.join()
# Результат вычислений:
print("Результат:", num)
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 10.15)

```
Сумма целых чисел
[1] 0 + 1 = 1
[2] 0 + 1 + 2 = 3
[3] 0 + 1 + 2 + 3 = 6
[4] 0 + 1 + 2 + 3 + 4 = 10
```

```
[5] 0 + 1 + 2 + 3 + 4 + 5 = 15
[6] 0 + 1 + 2 + 3 + 4 + 5 + 6 = 21
Результат: 21
```

Для записи результата вычислений мы используем глобальную переменную `num` (с начальным нулевым значением). Для вычисления суммы целых чисел предназначена функция `mysum()`, которая запускается в дочернем потоке. Код самой функции организован достаточно просто: выполняется оператор цикла `while`, в котором за каждую итерацию к текущему значению переменной `num` прибавляется очередное слагаемое. Для того чтобы операции не выполнялись слишком быстро, между итерациями выполняется пауза в `0.3` секунды. В качестве условия для продолжения работы оператора цикла указано выражение `myevent.is_set()`. Оно истинно, если установлен флаг объекта `myevent` класса `Event`, который создается в программе командой `myevent=Event()`. Командой `myevent.set()` для этого объекта устанавливается флаг. Объект дочернего потока создается инструкцией `t=Thread(target=mysum)`. После запуска дочернего потока на выполнение (команда `t.start()`) в выполнении главного потока делается пауза в `2` секунды (инструкция `sleep(2)`), после чего командой `myevent.clear()` отменяется флаг для объекта `myevent`. Теперь при очередной проверке условия в операторе цикла в функции `mysum()`, выполняемой в дочернем потоке, условие окажется ложным, и оператор цикла (а с ним функция и весь поток) завершит свое выполнение.

В следующей программе два дочерних потока заполняют список. Один из потоков заполняет список слева направо, а другой — справа налево. Программа представлена в листинге 10.16.

Листинг 10.16. Заполнение списка

```
from threading import Thread
from time import sleep
# Функции для выполнения в дочерних потоках:
def from_left():
    global first, last, L
    val=10
    while True:
```

```
        if first<last:
            L[first]=val
            val+=10
            first+=1
            sleep(0.3)
        else:
            break
def from_right():
    global first, last, L
    val="A"
    while True:
        if first<last:
            L[last]=val
            val=chr(ord(val)+1)
            last-=1
            sleep(0.3)
        else:
            break
# Размер списка:
size=11
# Создание списка:
L=["*" for k in range(size)]
# Начальный и конечный индексы:
first=0
last=len(L)-1
print("Список до заполнения:")
print(L)
# Создание объектов дочерних потоков:
A=Thread(target=from_left)
B=Thread(target=from_right)
# Запуск потоков на выполнение:
A.start()
```

```
B.start()
# Ожидание завершения потоков:
A.join()
B.join()
# Результат заполнения списка:
print("Список после заполнения:")
print(L)
```

Результат выполнения программы такой.

Результат выполнения программы (из листинга 10.16)

```
Список до заполнения:
['*', '*', '*', '*', '*', '*', '*', '*', '*', '*', '*']
Список после заполнения:
[10, 20, 30, 40, 50, '*', 'E', 'D', 'C', 'B', 'A']
```

Мы описываем две функции (`from_left()` и `from_right()`), предназначенные для заполнения списка. Обе функции используют для заполнения глобальный список `L` и два индекса `first` и `last`, в которые записаны значения граничных элементов, предназначенных для заполнения. Заполнение продолжается до тех пор, пока значение индекса элемента слева меньше значения индекса элемента справа. В остальном код достаточно простой, поэтому хочется верить, особых комментариев не требует.

НА ЗАМЕТКУ

Заполнение слева направо выполняется числовыми значениями, а справа налево список заполняется буквами. Для вычисления кода буквы используется функция `ord()`, а для получения на основе кода символа собственно символа используется функция `chr()`.

Еще одна программа, которую мы рассмотрим, связана с созданием большого количества дочерних потоков. В этом случае используется автоматически генерируемый список, элементами которого являются объекты дочерних потоков. Программа представлена в листинге 10.17.

Листинг 10.17. Список с объектами дочерних потоков

```
from threading import *
from time import sleep
# Функция для вычисления суммы:
def mysum(n, N):
    res=0
    for k in range(N+1):
        res+=k**n
        sleep(0.1)
    return res
# Функция для выполнения в дочернем потоке:
def display(n, N):
    # Блокировка объекта блокировки:
    mylock.acquire()
    # Получение ссылки на текущий поток:
    t=current_thread()
    # Отображение имени потока:
    print("Поток:", t.name)
    print("Слагаемых:", N)
    print("Степень:", n)
    # Результат вычислений:
    print("Сумма:", mysum(n, N))
    print("--------------")
    # Разблокирование объекта блокировки:
    mylock.release()
# Создание объекта блокировки:
mylock=Lock()
# Список с названиями потоков:
names=["Alpha","Bravo","Charlie","Delta"]
# Создание списка с объектами потоков:
T=[Thread(target=display, args=[k+1,10], name=names[k]) for k in range(len(names))]
# Запуск потоков на выполнение:
```

```
for t in T:
    t.start()
# Ожидание завершения выполнения потоков:
for t in T:
    t.join()
```

Результат выполнения программы будет следующим:

Результат выполнения программы (из листинга 10.17)

```
Поток: Alpha
Слагаемых: 10
Степень: 1
Сумма: 55
--------------
Поток: Bravo
Слагаемых: 10
Степень: 2
Сумма: 385
--------------
Поток: Charlie
Слагаемых: 10
Степень: 3
Сумма: 3025
--------------
Поток: Delta
Слагаемых: 10
Степень: 4
Сумма: 25333
--------------
```

В программе запускается несколько дочерних потоков. В каждом из потоков вычисляется сумма целых чисел, возведенных в целочисленную степень. Для вычисления таких сумм используется функция `mysum()`, аргументами которой передаются степень, в которую возводятся

слагаемые, и количество слагаемых в сумме. Для большей наглядности в процессе вычисления суммы делаются небольшие паузы.

Для выполнения в потоках предназначена функция `display()`. В теле этой функции командой `mylock.acquire()` объект блокировки (он нам нужен для того, чтобы синхронизировать процесс отображения сообщений потоками) переводится в заблокированное состояние. Командой `t=current_thread()` в переменную `t` записывается ссылка на текущий поток (тот поток, в котором выполняется код функции). У каждого потока есть стандартный атрибут `name`, определяющий название потока (мы используем инструкцию `t.name`). При выполнении потока отображается название потока, степень, в которую возводятся слагаемые, количество слагаемых и значение соответствующей суммы. После отображения сообщений отменяется блокировка для объекта блокировки (команда `mylock.release()`).

В программе командой `mylock=Lock()` создается объект блокировки. Названия для потоков оформляются в виде списка `names`. Список с объектами потоков создается командой `T=[Thread(target=display, args=[k+1,10], name=names[k]) for k in range(len(names))]`. Здесь собственно объект потока определяется выражением `Thread(target=display, args=[k+1,10], name=names[k])`. Конструктору класса `Thread` при этом, кроме прочего, передается еще и аргумент `name`, который задает имя потока. Для запуска потоков на выполнение используется оператор цикла, в котором перебирается список с объектами дочерних потоков. Аналогично в главном потоке выполняется переход в режим ожидания завершения выполнения дочерних потоков.

Резюме

У меня есть мысль, и я ее думаю.

Из м/ф «38 попугаев»

- При возникновении ошибки (исключения) в процессе выполнения программы автоматически создается объект исключения. Объект содержит информацию о возникшей ошибке. Каждому типу ошибки соответствует определенный класс (класс исключения). Эти классы образуют иерархию наследования.

- Ошибки, возникающие в процессе выполнения программы, можно перехватывать и обрабатывать. Для этого используется конструкция `try-except`. Контролируемый код (код, при выполнении которого может возникнуть ошибка) помещается в блок `try`. Код для обработки ошибок помещается в блоке `except`. Если при выполнении `try`-блока ошибки не возникли, то `except`-блок игнорируется. Если при выполнении `try`-блока возникала ошибка, то выполнение этого блока прекращается и начинают выполняться команды в `except`-блоке.

- В общем случае может одновременно использоваться несколько `except`-блоков, каждый из которых предназначен для обработки исключений определенного класса. Класс исключения указывается после ключевого слова `except` в соответствующем блоке. Такой блок перехватывает исключения не только указанного класса, но и всех его производных классов. Если `except`-блок предназначен для обработки исключений нескольких типов, то такие типы указываются в `except`-блоке в виде кортежа.

- В процессе обработки исключений можно использовать объект исключения. Переменная, в которую записывается на объект исключения, указывается через ключевое слово `as` после названия класса исключения в описании `except`-блока.

- В конструкции обработки исключений, кроме блоков `try` и `except`, можно использовать боки `else` и `finally`. Команды в `else`-блоке выполняются только в том случае, если в `try`-блоке не возникли ошибки. Команды в `finally`-блоке выполняются в любом случае.

- Для искусственного генерирования исключений используется инструкция `raise`. После этой инструкции указывается название класса генерируемого исключения или сразу объект исключения. Инструкцию `raise` также можно использовать без указания класса исключения или объекта в `except`-блоке для повторного генерирования обрабатываемого в блоке исключения.

- Можно создавать собственные классы для исключений. Обычно такие классы создаются наследованием класса `Exception`. Благодаря этому созданные классы встраиваются в общую иерархию классов исключений, а исключения данных классов могут генерироваться, перехватываться и обрабатываться в программе.

- Потоками называются блоки программного кода, которые выполняются одновременно. Выполнение программы начинается

с выполнения главного потока, в котором можно создавать дочерние потоки.

- Объект дочернего потока создается на основе класса `Thread` (из модуля `threading`) или на основе класса, наследующего класс `Thread`. В качестве аргумента конструктору класса `Thread`, кроме прочего, передается ссылка на функцию, выполняемую в дочернем потоке (имя функции указывается в качестве значения аргумента `target`). Аргументы для функции передаются посредством аргумента `args` или `kwargs`. Имя потока задается с помощью аргумента `name`. Для запуска потока на выполнение из объекта потока вызывается метод `start()`.
- Вместо функции при создании объекта потока можно использовать вызываемый объект. Если объект потока создается на основе класса, производного от класса `Thread`, то в этом производном классе переопределяется метод `run()`, который автоматически вызывается при запуске потока на выполнение.
- Есть группа методов, позволяющих выполнять различные операции с потоками. Например, метод `join()` используется в случае, если необходимо дождаться выполнения потока (из объекта которого вызывается метод), а метод `is_alive()` позволяет проверить, является ли активным поток.
- Для синхронизации работы потоков используются такие классы, как `Lock`, `RLock`, `Semaphore`, `Event` и ряд других (все из модуля `threading`). Идея синхронизации базируется на блокировке/разблокировке общих для потоков ресурсов и обмене сообщениями между потоками.

Задания для самостоятельной работы

> Форму будете создавать под моим личным контролем. Форме сегодня придается большое содержание.
>
> *Из к/ф «Чародеи»*

1. Напишите программу, в которой описывается функция с произвольным количеством аргументов. В качестве результата функция возвращает сумму значений целочисленных аргументов. При вычислении результата использовать обработку исключений.

2. Напишите программу, в которой пользователю предлагается ввести два целочисленных значения. Эти значения определяют границы диапазона, в котором отображаются целые числа. Использовать обработку исключений.

3. Напишите программу для решения квадратного уравнения вида (A^2-1) $x = B$. В процессе поиска решения использовать обработку исключительных ситуаций.

4. Напишите программу, в которой решается уравнение вида $A(A - 1)\, x = \sin(A)$, причем при значении $A = 0$ должно вычисляться решение $x = -1$.

5. Напишите программу, в которой, по аналогии с примером из листинга 10.8, с использованием пользовательского класса исключения и рекурсивного вызова функции создается объект исключения со списком, содержащим (в обратном порядке) буквы алфавита.

6. Напишите программу, в которой с использованием потоков, вычисляется сумма квадратов натуральных чисел. Сумма вычисляется определенное время, по аналогии с примером из листинга 10.15.

7. Напишите программу, в которой с использованием двух дочерних потоков заполняется список. Первый поток присваивает буквенные значения элементам с четными индексами. Второй поток присваивает числовые значения элементам с нечетными индексами.

8. Напишите программу, в которой создается объект с двумя списками. Списки заполняются значениями с помощью двух дочерних потоков: один список заполняется символами, а второй список заполняется числами.

9. Напишите программу, в которой создается и построчно заполняется двумерный список (список, элементами которого являются списки). Для заполнения каждой строки (каждого внутреннего списка) используется отдельный дочерний поток.

10. Напишите программу, в которой создается три дочерних потока. В первом потоке вычисляется факториал числа (произведение натуральных чисел). Во втором потоке вычисляется двойной факториал числа (произведение чисел через одно). В третьем потоке вычисляется число из последовательности Фибоначчи (первые два числа равны единице, а каждое следующее равно сумме двух предыдущих).

Глава 11
ПРОГРАММЫ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ

Видел чудеса техники, но такого...

Из к/ф «Иван Васильевич меняет профессию»

В этой главе мы познакомимся с принципами создания *графического интерфейса*. Существуют различные библиотеки, позволяющие создавать приложения с графическим интерфейсом. Мы рассмотрим встроенную библиотеку Tkinter. Для использования ее утилит в программе подключается модуль `tkinter`. Способы использования этой библиотеки и приемы, применяемые при создании графического интерфейса, будем рассматривать на конкретных примерах. Начнем мы с самых простых ситуаций.

Создание простого окна

Это дело очень интересное. И простое!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Рассмотрим достаточно простую программу, в которой создается пустое окно (оно не содержит ничего, кроме заголовка). Программный код примера представлен в листинге 11.1

Листинг 11.1. Создание простого окна

```
# Импорт класса из пакета:
from tkinter import Tk
# Создание объекта окна:
wnd=Tk()
# Заголовок для окна:
```

```
wnd.title("Простое окно")
# Геометрические размеры окна:
wnd.geometry("250x150")
# Отображение окна на экране:
wnd.mainloop()
```

В результате выполнения этого программного кода на экране появляется окно, представленное на рис. 11.1.

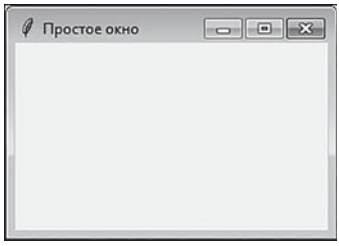


Рис. 11.1. Результат создания простого окна

Ширина окна составляет 250 пикселей, а высота окна равна 150 пикселям. При этом размеры окна можно изменять с помощью курсора мыши.

Проанализируем программный код, с помощью которого создается окно. В первую очередь стоит отметить, что создание окна (как и любого другого графического компонента) означает создание объекта определенного класса. Для окна это класс `Tk`. После того как класс `Tk` импортирован в программу инструкцией `from tkinter import Tk`, командой `wnd=Tk()` создается объект окна, и ссылка на него записывается в переменную `wnd`. Но сам по себе факт создания окна еще не означает, что окно отображается на экране. Для отображения окна на экране из объекта окна вызывается метод `mainloop()` (команда `wnd.mainloop()` в конце программы). Предварительно для окна выполняются минимальные настройки. Так, командой `wnd.title("Простое окно")` задается заголовок для окна. Геометрические размеры окна определяются командой `wnd.geometry("250x150")`.

ПОДРОБНОСТИ

Размеры окна определяются текстовой строкой, в которой указана ширина и высота окна в пикселях. В качестве разделителя в текстовой строке используется буква `"x"`.

При вызове метода `mainloop()` запускается главный цикл обработки событий: программа фактически отображает окно и находится в режиме ожидания действий, выполняемых пользователем. В этом смысле приложения с графическим интерфейсом принципиально отличаются от консольных приложений.

Также стоит отметить, что файлы с реализацией приложений с графическим интерфейсом, сохраняется с расширением `.pyw`.

Окно с меткой и кнопкой

> Мы вам ничего не позволим показывать. Мы вам сами все покажем.
>
> *Из к/ф «Гараж»*

На следующем этапе мы решим более сложную задачу — создадим окно, в котором будут текстовая *метка* и *кнопка*.

ПОДРОБНОСТИ

Хочется верить что любой, кто хоть раз имел дело с программами с графическим интерфейсом, представляет, что такое кнопка. Что касается метки, то это графический компонент, который содержит текст. Это его основное назначение.

Как отмечалось выше, создание любого графического компонента, в том числе кнопки и метки, означает, что на основе определенного класса создается объект. Объект метки создается на основе класса `Label`, а объект кнопки создается на основе класса `Button`. Как эти объекты используются на практике, иллюстрирует программа в листинге 11.2.

Листинг 11.2. Окно с меткой и кнопкой

```
# Импорт классов из пакета:
from tkinter import *
# Создание объекта окна:
wnd=Tk()
# Заголовок для окна:
wnd.title("Окно с кнопкой")
```

```
# Геометрические размеры окна:
wnd.geometry("250x150")
# Окно постоянных размеров:
wnd.resizable(False, False)
# Создание объекта метки:
lbl=Label(wnd, text="Всем привет!", font=("Arial Bold",20))
# Размещение метки в окне:
lbl.place(x=40, y=30)
# Объект копки:
btn=Button(wnd, text="Закрыть", font=("Courier New Bold",13), command=wnd.destroy)
# Размещение объекта кнопки в окне:
btn.place(x=40, y=100, width=170, height=30)
# Отображение окна на экране:
wnd.mainloop()
```

При выполнении этой программы на экране появляется окно, показанное на рис. 11.2.

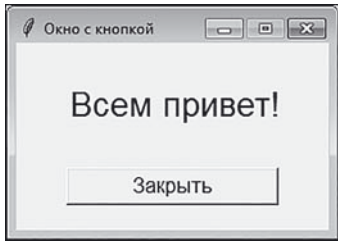


Рис. 11.2. Окно с меткой и кнопкой

В центральной части окна отображается текст, а в нижней части окна расположена кнопка **Закрыть**, нажатие которой приводит к закрытию окна и завершению выполнения программы.

НА ЗАМЕТКУ

Создается окно постоянных размеров — размеры этого окна нельзя изменить с помощью курсора мыши.

Помимо уже знакомых нам команд, программа содержит и новые инструкции. Так, мы хотим на этот раз создать окно постоянных размеров, поэтому после создания объекта окна `wnd`, командой `wnd.resizable(False, False)` для окна устанавливается режим постоянных размеров.

ПОДРОБНОСТИ

Два аргумента `False`, переданные методу `resizable()`, означают запрет изменять размеры окна соответственно по горизонтали и вертикали.

Объект кнопки `lbl` создается командой `lbl=Label(wnd, text="Всем привет!", font=("Arial Bold",20))`. Первый аргумент `wnd` конструктора класса `Label` означает, что метка будет размещаться в окне `wnd`. Аргумент `text` определяет надпись, содержащуюся в текстовой метке. Наконец, аргумент `font` задает шрифт, которым будет отображаться текст в метке. Значением этого аргумента указан кортеж с элементами `"Arial Bold"` и `20`, определяющими соответственно тип и размер шрифта. Размещается метка в окне с помощью команды `lbl.place(x=40, y=30)`. Аргументы метода `place()` определяют координаты метки в области окна. Координаты задаются для левого верхнего угла метки по отношению к левому верхнему углу окна. Координаты задаются в пикселях. Первая горизонтальная координата отсчитывается вдоль горизонтали слева направо, а вторая вертикальная координата отсчитывается сверху вниз.

НА ЗАМЕТКУ

Этот же способ определения координат применяется и для прочих графических компонентов.

ПОДРОБНОСТИ

Существует несколько способов, которыми компонент может добавляться в контейнер (в данном случае контейнером является окно). Если для размещения компонента используется метод `place()`, то координаты компонента указываются явно. Можно также использовать методы `pack()` и `grid()`. С помощью метода `pack()` компонент размещается путем определения положения компонента по отношению к сторонам контейнера. Если используется метод

grid(), то положение компонента определяется указанием ячейки в воображаемой таблице, которая размещена в области окна.

Объект кнопки btn создается командой btn=Button(wnd, text="Закрыть", font=("Courier New Bold",13), command=wnd.destroy). Первый аргумент конструктора класса Button определяет принадлежность кнопки. Аргумент text задает название кнопки. Аргумент font определяет шрифт, применяемый при отображении названия кнопки. Важна роль аргумента command. Он определяет действие, выполняемое при нажатии кнопки. Значение wnd.destroy аргумента означает, что при нажатии кнопки из объекта wnd будет вызываться метод destroy(). При вызове этого метода окно закрывается (и, как следствие, программа прекращает выполнение). Наконец, размещается кнопка в окне командой btn.place(x=40, y=100, width=170, height=30). Аргументы x и y определяют координаты кнопки в области окна, а аргументы width и height задают соответственно ширину и высоту кнопки (в пикселях).

НА ЗАМЕТКУ

Вызов метода destroy() приводит к удалению объекта окна, но это не означает автоматического завершения программы. Если нужно завершить всю программу, используют метод quit() (или иные решения, в зависимости от контекста задачи). Если нужно окно убрать с экрана (не уничтожая объект окна), то используют метод withdraw(). Чтобы снова отобразить скрытое окно, используют метод deiconify().

Использование текстового поля

Фигуры, может, и нет, а характер — налицо.

Из к/ф «Девчата»

Текстовое *поле* — графический компонент, который предназначен для ввода текста. Текстовые поля реализуются в виде объектов класса Entry. Далее рассматривается программа, в которой отображается окно с текстовым полем. В это поле пользователю предлагается ввести свое имя. Окно представлено на рис. 11.3.

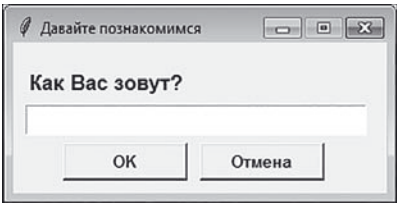


Рис. 11.3. Окно с полем ввода появляется при запуске программы на выполнение

Окно содержит две кнопки: **OK** и **Отмена**. Если пользователь нажимает кнопку **Отмена** или закрывает окно с помощью системной пиктограммы в правом верхнем углу окна, то окно закрывается, и программа на этом прекращает выполнение. Если пользователь вводит в поле текст и нажимает кнопку **OK**, то введенный пользователем текст считывается, запоминается, и затем после первого окна отображается еще одно окно с сообщением, в котором содержится текст, введенный пользователем в поле ввода в первом окне. На рис. 11.4 показано окно с именем, которое ввел пользователь, перед нажатием кнопки **OK**.

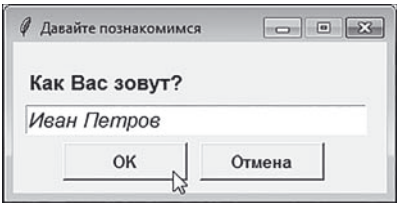


Рис. 11.4. Окно с именем в поле ввода перед нажатием кнопки **OK**

После нажатия кнопки **OK** появится окно, представленное на рис. 11.5.

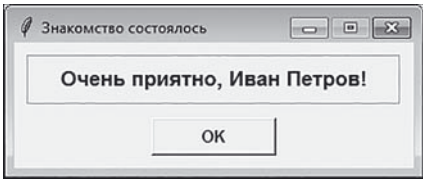


Рис. 11.5. Окно с сообщением появляется после нажатия кнопки **OK** в первом окне

А теперь рассмотрим программный код в листинге 11.3.

Листинг 11.3. Использование поля ввода

```
from tkinter import *
# Функция для обработки нажатия кнопки:
def clicked():
    global t
    # Считывается содержимое текстового поля:
    t=txt.get()
    # Закрывается окно:
    wnd.destroy()
# Создается объект для первого окна:
wnd=Tk()
# Параметры окна:
wnd.title("Давайте познакомимся")
wnd.geometry("300x120")
wnd.resizable(False, False)
# Шрифты:
fnt_1=("Arial",13,"bold")
fnt_2=("Arial",13,"italic")
fnt_3=("Arial",10,"bold")
# Переменная для записи текста из поля ввода:
t=""
# Создание объекта для текстовой метки:
lbl=Label(master=wnd, text="Как Вас зовут?")
# Шрифт для метки:
lbl.configure(font=fnt_1)
# Добавление метки в окно:
lbl.place(x=10, y=20)
# Создание объекта для поля ввода:
txt=Entry(master=wnd, width=30)
# Шрифт для поля ввода:
txt.configure(font=fnt_2)
# Размещение текстового поля в окне:
```

```
txt.place(x=10, y=50)
# Создание объектов для кнопок:
btn_1=Button(master=wnd, text="OK")
btn_2=Button(master=wnd, text="Отмена")
# Параметры кнопок:
btn_1.configure(font=fnt_3)
btn_1.configure(command=clicked)
btn_2.configure(font=fnt_3)
btn_2.configure(command=wnd.destroy)
# Размещение кнопок в окне:
btn_1.place(x=40, y=80, width=100, height=30)
btn_2.place(x=150, y=80, width=100, height=30)
# Отображение первого окна на экране:
wnd.mainloop()
# Если пользователь ввел текст:
if t!="":
    # Создание объекта для второго окна:
    msg=Tk()
    # Параметры второго окна:
    msg.title("Знакомство состоялось")
    msg.geometry("320x100")
    msg.resizable(False, False)
    # Метка с сообщением для второго окна:
    lbl=Label(master=msg, text="Очень приятно, "+t+"!", relief=GROOVE)
    # Шрифт для метки:
    lbl.configure(font=fnt_1)
    # Размещение метки во втором окне:
    lbl.place(x=10, y=10, height=40, width=300)
    # Создание объекта кнопки:
    btn=Button(master=msg, text="OK")
    # Шрифт для кнопки:
    btn.configure(font=fnt_3)
```

```
# Метод для обработки нажатия кнопки:
btn.configure(command=msg.destroy)
# Размещение кнопки во втором окне:
btn.place(x=110, y=60, width=100, height=30)
# Отображение второго окна на экране:
msg.mainloop()
```

Первое окно (объект `wnd`) создается уже знакомым нам способом. В окно добавляется текстовая метка, поле ввода и две кнопки. Для удобства мы создаем три кортежа (`fnt_1`, `fnt_2` и `fnt_3`), определяющие используемые в программе шрифты. Каждый кортеж содержит три элемента: название шрифта, размер шрифта и стиль шрифта. Также мы определяем переменную `t` с пустой текстовой строкой в качестве начального значения. В эту переменную планируется записывать текст, который пользователь введет в поле ввода.

Объект метки создается и размещается в окне практически так же, как мы это делали раньше.

ПОДРОБНОСТИ

Если до этого мы просто указывали первый аргумент, то теперь он передается по ключу `master`. Шрифт для метки мы задаем отдельно. Поэтому из объекта метки вызывается метод `configure()` и значением аргумента `font` указывается переменная, определяющая шрифт, который и будет применяться для метки. Эти же замечания относятся и к созданию прочих компонентов.

Объект для поля ввода создается командой `txt=Entry(master=wnd, width=30)`. Аргумент `width` в данном случае определяет ширину поля в символах (которые в это поле можно ввести). Шрифт для текстового поля задаем командой `txt.configure(font=fnt_2)`. Этот шрифт будет применяться к тексту, вводимому в поле. Для размещения поля в окне использована команда `txt.place(x=10, y=50)`, в которой аргументы метода `place()`, как и ранее, определяют положение (координаты в пикселях) левого верхнего угла поля по отношению к левому верхнему углу окна.

Кроме поля ввода создаются две кнопки. Обработчики событий для кнопок мы регистрируем вызовом из объекта соответствующей кнопки

метода `configure()`, указав значением аргумента `command` имя функции (или метода), вызываемых при нажатии кнопки. Для одной из кнопок таким значением указана ссылка на метод `destroy()` объекта окна `wnd`. В результате вызова этого метода окно убирается с экрана (но программа при этом автоматически работу не прекращает). Для другой кнопки обработчиком события нажатия на кнопке указана функция `clicked()`. Функция описана в программе следующим образом. Командой `t=txt.get()` в глобальную переменную `t` записывается значение, содержащееся в поле ввода. Для получения этого значения из объекта поля `txt` вызывается метод `get()`. После этого командой `wnd.destroy()` окно закрывается.

Для размещения кнопок в окне используется метод `place()`, в качестве аргументов которому передаются координаты кнопки (`x` и `y`) и ее размеры (ширина `width` и высота `height`), указанные в пикселях.

После того как метка, поле и кнопки добавлены в окно, командой `wnd.mainloop()` оно отображается на экране.

Следующей командой размещен условный оператор. Он будет выполняться только после того, как закроется первое окно (которое отображается предыдущей командой).

ПОДРОБНОСТИ

При вызове метода `mainloop()` из объекта окна фактически запускается бесконечный цикл, в рамках которого интерпретатор ожидает и обрабатывает операции, выполняемые пользователем с окном. Поэтому до выполнения следующей команды дело не доходит. Чтобы начала выполняться следующая команда, необходимо, чтобы окно было закрыто.

В условном операторе проверяется условие `t!=""`. Оно истинно в том случае, если переменная ссылается на непустую текстовую строку. Это происходит в случае, если в первом окне пользователь в поле ввода указал текст и нажал кнопку **OK**. Если так, то командой `msg=Tk()` создается объект для нового кона, выполняется настройка его параметров, а также создаются и размещаются в окне метка и кнопка. Текст для метки вычисляется выражением `"Очень приятно, "+t+"!"`, в котором используется значение переменной `t`. Также мы используем аргумент `relief` со значением `GROOVE`. С помощью этого аргумента задается режим отображения рамки вокруг метки.

ПОДРОБНОСТИ

То, что мы ссылку на объект метки записываем в переменную `lbl`, которую уже использовали ранее, проблемой не является. Первое окно мы больше не используем, а в переменную `lbl` записывается ссылка на новый объект.

Аргумент `relief`, как отмечалось, определяет рамку вокруг метки (точнее, способ выделения метки на фоне контейнера). Возможные значения: `FLAT` (фактически рамка отсутствует), `RAISED` (эффект «приподнятой» метки), `SUNKEN` (эффект «вдавленной» метки), `GROOVE` (вдавленная рамка вокруг метки) и `RIDGE` (приподнятая рамка вокруг метки).

Также стоит еще раз заметить, что аргументы `width` и `height` (равно как аргументы `x` и `y`) в методе `place()` задаются в пикселях.

Кнопка для окна определяется так, что при ее нажатии из объекта окна `msg` вызывается метод `destroy()`, в результате чего окно закрывается и, поскольку на этом код исчерпан, заканчивается выполнение программы. Для отображения окна на экране использована команда `msg.mainloop()`.

НА ЗАМЕТКУ

Если нужно работать с текстами больших объемов, можно использовать текстовую область, которая реализуется как объект класса `Text`.

Раскрывающийся список

Огласите весь список, пожалуйста.

Из к/ф «Операция Ы и другие приключения Шурика»

Еще один полезный графический компонент, который позволяет создавать эффективные приложения с графическим интерфейсом, — *раскрывающийся список*. Для реализации раскрывающегося списка мы воспользуемся классом `Combobox` из подпакета `ttk` пакета `tkinter`.

ПОДРОБНОСТИ

Для использования этого класса в программе, рассматриваемой далее, есть инструкция `from tkinter.ttk import Combobox`.

Что касается самой программы, то она такая: при запуске программы на выполнение появляется окно, показанное на рис. 11.6.

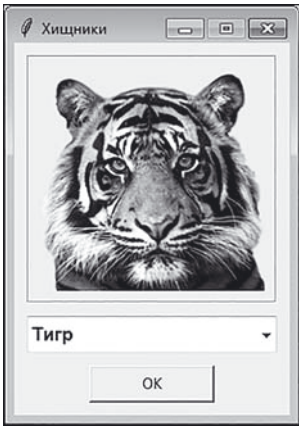


Рис. 11.6. Окно с изображением тигра

В центральной части окна содержится изображение тигра, а внизу под изображением есть раскрывающийся список (сначала в нем выбрано значение **Тигр**) и кнопка **ОК**. При нажатии кнопки окно закрывается. А если нажать пиктограмму со стрелочкой в раскрывающемся списке, то выпадет его содержимое: названия трех животных (**Тигр**, **Лев** и **Медведь**). Окно с раскрытым списком представлено на рис. 11.7.

Рис. 11.7. Выбор нового значения в раскрывающемся списке

После выбора пункта в раскрывающемся списке в центральной области окна автоматически отображается соответствующая картинка.

На рис. 11.8 показано окно с изображением льва (в раскрывающемся списке выбран пункт **Лев**).

Рис. 11.8. Окно с изображением льва

Как будет выглядеть окно, если в списке выбрать пункт **Медведь**, показано на рис. 11.9.

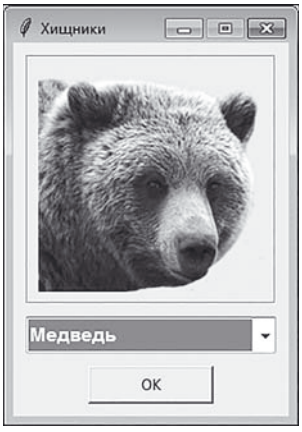


Рис. 11.9. Окно с изображением медведя

Прежде чем приступить к рассмотрению программного кода, сразу отметим несколько моментов, а именно: для размещения изображения в области окна мы используем метку. Хотя она формально называется текстовой, но в действительности может содержать не только текст, но и изображения. Ссылка на объект изображения присваивается

в качестве значения аргументу `image` в конструкторе класса `Label`. Объект изображения будет создаваться на основе класса `PhotoImage`, а в качестве аргументов конструктору передаются ссылка на объект окна (значение аргумента `master`) и текст с полным путем к файлу изображения (значение аргумента `file`).

НА ЗАМЕТКУ

Для корректной работы программы в каталоге `D:\Books\Python\Pictures\` должны находиться файлы `tiger.png`, `lion.png` и `bear.png` с изображениями животных. Каждое изображение имеет размер 180 пикселей в ширину и 180 — в высоту и прозрачный фон (хотя это и не принципиально). Если читатель планирует использовать иные файлы (но с тем же расширением), размещенные в другом месте, то в программу нужно будет внести соответствующие изменения.

Теперь рассмотрим программный код, представленный в листинге 11.4.

Листинг 11.4. Раскрывающийся список

```
# Импорт содержимого пакета:
from tkinter import *
# Импорт класса из подпакета:
from tkinter.ttk import Combobox
# Функция для обработки события, связанного
# с выбором пункта в раскрывающемся списке:
def change(evt):
    # Считывание значения, выбранного в списке:
    t=cb.get()
    # Определение изображения по выбранному значению:
    for k in range(len(names)):
        # Если выбранное значение совпадает
        # с текстом в списке:
        if t==names[k]:
            # Новое изображение в метке:
            lbl.configure(image=imgs[k])
            # Завершение оператора цикла:
```

```
            break
# Переменная с путем к файлам изображений:
path="D:\\Books\\Python\\Pictures\\"
# Названия животных:
names=["Тигр","Лев","Медведь"]
# Названия файлов с изображениями:
files=["tiger.png","lion.png","bear.png"]
# Создание объекта окна:
wnd=Tk()
# Параметры окна:
wnd.title("Хищники")
wnd.geometry("220x300")
wnd.resizable(False, False)
# Список с объектами для изображений:
imgs=[PhotoImage(file=path+f) for f in files]
# Индекс для выбранного в начале пункта:
index=0
# Создание объекта метки:
lbl=Label(wnd, image=imgs[index])
# Параметры метки и ее размещение в окне:
lbl.configure(relief=GROOVE)
lbl.place(x=10, y=10, width=200, height=200)
# Создание объекта для раскрывающегося списка:
cb=Combobox(wnd, state="readonly")
# Содержимое раскрывающегося списка:
cb.configure(values=names)
# Выбранное в начале значение (пункт):
cb.current(index)
# Шрифт для раскрывающегося списка:
cb.configure(font=("Arial",11,"bold"))
# Определение обработчика события, связанного
# с выбором в списке нового значения:
```

```
cb.bind("<<ComboboxSelected>>", change)
# Размещение списка в окне:
cb.place(x=10, y=220, width=200, height=30)
# Создание объекта кнопки:
btn=Button(wnd, text="OK")
# Определение обработчика для события,
# связанного с нажатием кнопки:
btn.configure(command=wnd.destroy)
# Размеры кнопки и ее размещение в окне:
btn.place(x=60, y=260, width=100, height=30)
# Отображение окна на экране:
wnd.mainloop()
```

Кроме инструкций импорта содержимого пакета `tkinter` и импорта класса `Combobox` из подпакета `ttk`, программа содержит описание функции `change()`, несколько глобальных списков и переменных, а также собственно команды, которыми создается окно и все его содержимое. Так, в переменную `path` мы записываем путь к каталогу, в котором хранятся файлы с изображениями. В нашем случае это текст `"D:\\Books\\Python\\Pictures\\"`.

> **НА ЗАМЕТКУ**
>
> Если планируется размещать файлы с изображениями в ином месте, то в качестве значения переменной `path` следует указать текстовое значение с путем к соответствующему каталогу.
>
> Также напомним, что для включения в текст обратной косой черты `\` используется комбинация `\\` из двух обратных косых черт — иначе следующий после обратной косой черты символ будет восприниматься как управляющий.

Список `names` содержит названия животных. Он определяет пункты раскрывающегося списка. Список `files` содержит название файлов с изображениями животных. Между списками `names` и `files` должно быть взаимно однозначное соответствие: индекс элемента с названием животного в списке `names` должен быть таким же, как и индекс элемента с названием файла с изображением животного в списке `files`.

НА ЗАМЕТКУ

Чтобы расширить содержимое раскрывающегося списка, достаточно в список `names` добавить названия животных (новые пункты раскрывающегося списка), а в список `files` добавить названия файлов с изображениями этих животных (и, разумеется, сами файлы должны быть добавлены в каталог с изображениями).

После того как создан объект окна `wnd` и заданы его основные параметры (заголовок, размеры и переход в режим окна постоянных размеров), командой `imgs=[PhotoImage(file=path+f) for f in files]` создается список с объектами изображений, которые будут использоваться для отображения в окне.

ПОДРОБНОСТИ

Для использования изображения в приложении с графическим интерфейсом на основе этого изображения необходимо создать специальный объект. Объект создается на основе класса `PhotoImage`. Значением аргумента `file` указывается путь к файлу, на основе которого создается объект изображения.

Переменная `index` с начальным нулевым значением определяет пункт из раскрывающегося списка и соответствующее ему изображение, которые будут показаны в окне в самом начале.

Метку создаем командой `lbl=Label(wnd, image=imgs[index])`. В данном случае примечательно то, что в качестве значения аргументу `image` передается ссылка на объект изображения из списка `imgs`. Индекс объекта определяется значением переменной `index`.

Командой `lbl.configure(relief=GROOVE)` для метки задается режим отображения вдавленной рамки вокруг метки. Командой `lbl.place(x=10, y=10, width=200, height=200)` задаются ширина и высота метки, а также ее положение в окне.

НА ЗАМЕТКУ

Напомним, что используются изображения размерами 180 пикселей в ширину и 180 пикселей в высоту. Размеры метки немного больше: 200 пикселей в ширину и 200 пикселей в высоту.

Объект для раскрывающегося списка создается командой `cb=Combobox(wnd, state="readonly")`. Значение `"readonly"` для аргумента `state` означает, что поле раскрывающегося списка с выбранным пунктом не будет доступно для редактирования (то есть пункт в раскрывающемся списке можно будет выбрать, но нельзя будет ввести название этого пункта с помощью клавиатуры).

Содержимое раскрывающегося списка определяется командой `cb.configure(values=names)`. Здесь аргументу `values` в качестве значения присваивается ссылка на список `names`, содержащий названия животных. Именно они будут отображаться в раскрывающемся списке. Наконец, командой `cb.current(index)` определяется значение (пункт) в раскрывающемся списке, который будет выбран в самом начале. Этот пункт определяется элементом в списке `names` с таким же индексом, как и у элемента в списке изображений, который указан для отображения в метке. Также мы задаем шрифт для раскрывающегося списка (команда `cb.configure(font=("Arial",11,"bold"))`). Этот шрифт будет применяться для отображения названия выбранного в раскрывающемся списке пункта.

По сравнению с рассмотренными выше графическими компонентами, изменился способ, которым мы задаем для раскрывающегося списка обработчик события, связанного с выбором в раскрывающемся списке нового пункта. На этот раз мы используем метод `bind()`. В качестве аргументов методу передаются название события, для которого регистрируется обработчик, и функция/метод, которые вызываются для обработки этого события. Мы хотим реализовать обработку события `<<ComboboxSelected>>`, которое состоит в том, что в раскрывающемся списке выбран пункт. Для обработки события будет вызываться функция `change()`. Поскольку при обработке события объект этого события передается в функцию, то такая функция должна быть описана с аргументом (аргумент события). Поэтому функцию `change()` мы описываем с аргументом — правда, мы его не используем.

ПОДРОБНОСТИ

Вообще объект события содержит полезную информацию о событии — например координаты курсора мыши в момент наступления события и ряд других характеристик. В принципе эту информацию можно «извлечь» и использовать. Нам она не нужна, но указать аргумент в описании функции мы все равно должны.

Функция `change()`, которая вызывается при выборе нового пункта в раскрывающемся списке, описана следующим образом. Сначала командой `t=cb.get()` считывается и записывается в переменную `t` значение, выбранное в раскрывающемся списке. После этого запускается оператор цикла, в котором выполняется поиск элемента с таким же названием, что записано в переменную `t`. Как только совпадение найдено (истинно условие `t==names[k]` в условном операторе), командой `lbl.configure(image=imgs[k])` для метки задается новое изображение. Это объект из списка `imgs` с таким же индексом `k`, что и индекс элемента из списка `names`, который выбран в раскрывающемся списке. После этого работа оператора цикла завершается с помощью инструкции `break`.

Во всем остальном программный код простой: мы размещаем метку в окне (команда `cb.place(x=10, y=220, width=200, height=30)`), создаем кнопку (команда `btn=Button(wnd, text="OK")`), определяем обработчик для события, связанного с нажатием кнопки (команда `btn.configure(command=wnd.destroy)`), и тоже размещаем ее в окне (команда `btn.place(x=60, y=260, width=100, height=30)`). Наконец, окно отображается на экране (использована команда `wnd.mainloop()`).

Опции, переключатели и другие компоненты

Первый раз таких единоличников вижу.

Из к/ф «Девчата»

В следующем примере мы познакомимся с такими графическими компонентами, как *опции*, *переключатели*, *слайдер* и *статический список*. Кроме этого, для размещения компонентов мы будем использовать *панели*, а также узнаем, как компоненты размещаются в контейнере с помощью метода `pack()`.

Сначала мы рассмотрим результат выполнения программы — после этого нам будет легче проанализировать программный код. Итак, при запуске программы на выполнение на экране появляется окно, представленное на рис. 11.10.

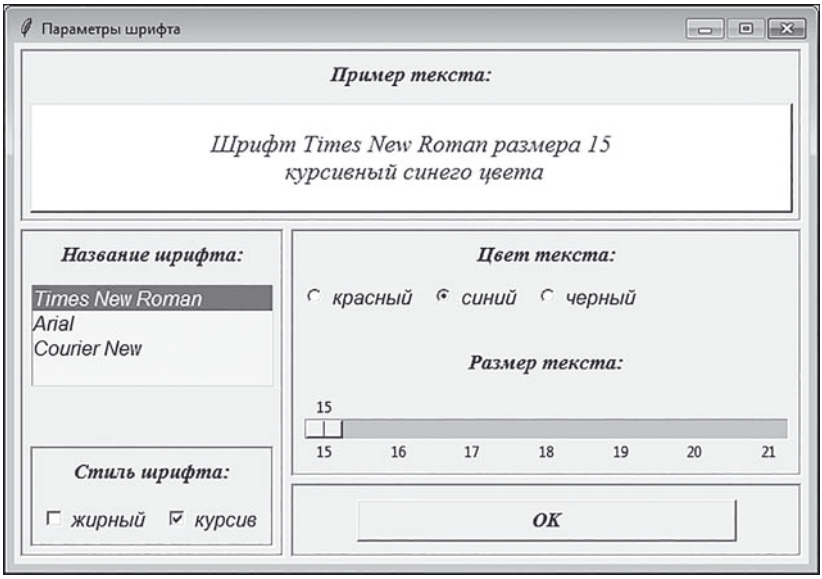


Рис. 11.10. Окно для настроек параметров шрифта

В верхней части окна размещена панель с надписью **Пример текста** и белой областью с текстом. Текст отображается синим цветом курсивным шрифтом **Times New Roman** размера **15**. Собственно сам текст содержит эту информацию. В левой части окна расположена панель с надписью **Название шрифта** и списком для выбора названия шрифта из трех пунктов. В нижней части этой панели, под надписью **Стиль шрифта**, расположены две опции для выбора стиля шрифта (по умолчанию установлен флажок опции **курсив**, и не установлен флажок опции **жирный**).

Справа в области окна под надписью **Цвет текста** расположена панель с группой переключателей (**красный**, **синий** и **черный** — по умолчанию выбран пункт **синий**), предназначенных для выбора цвета шрифта. Также панель под надписью **Размер текста** содержит слайдер для выбора размера шрифта. Далее внизу размещена еще одна панель с кнопкой **ОК**, нажатие которой приводит к закрытию окна.

При изменении настроек в окне автоматически изменения вносятся в шаблонный текст: меняется как содержимое текста, так и шрифт, которым отображается этот текст (в том числе и цвет текста). На рис. 11.11 проиллюстрировано состояние окна, когда выбран шрифт **Arial**, установлены опции **жирный** и **курсив**, а также выбран пункт **красный** в группе переключателей, предназначенных для выбора цвета.

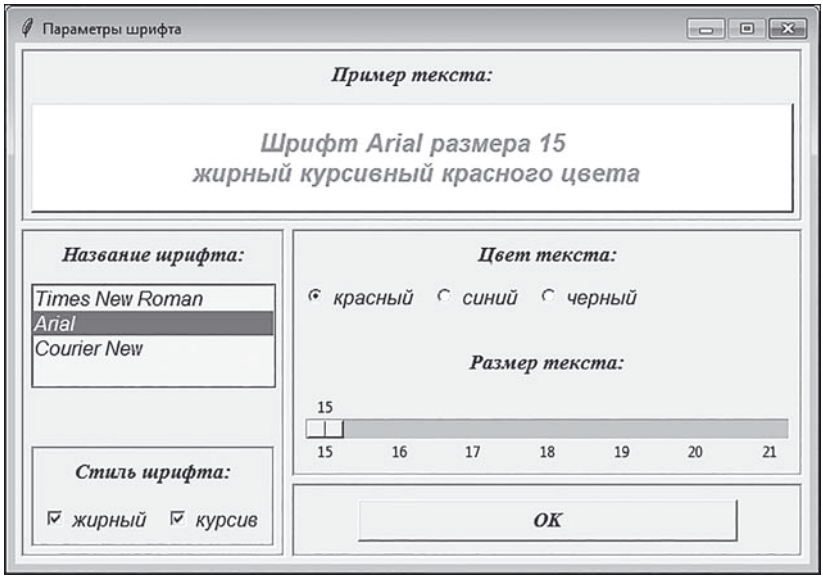


Рис. 11.11. Результат изменения параметров шрифта

Изменение положения слайдера приводит к изменению размера текста. На рис. 11.12 показан ситуация, когда используется шрифт **Courier New**, флажки опций выбора стиля отменены, выбран черный цвет для отображения текста, а слайдер установлен на значении **20**.

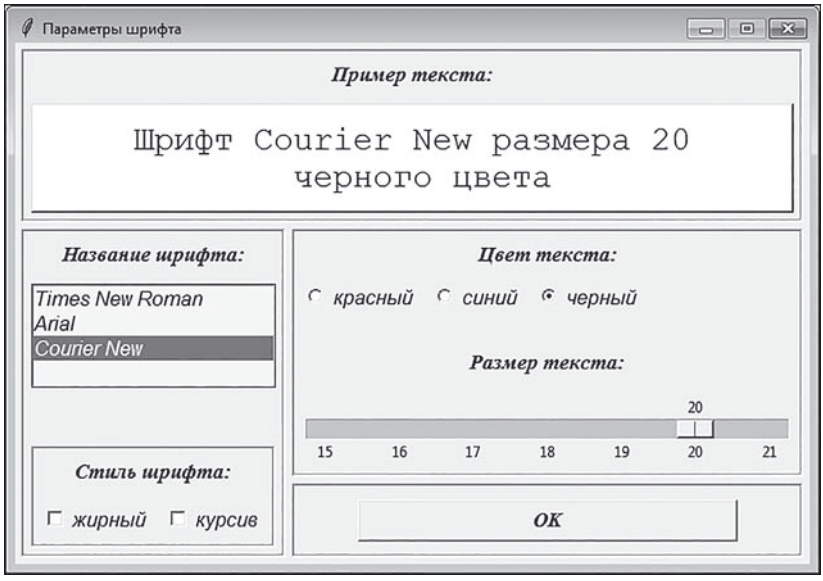


Рис. 11.12. Результат изменения размера, цвета и стиля шрифта

Перед рассмотрением кода сделаем несколько общих замечаний. В первую очередь это относится к классам, на основе которых создаются объекты графических компонентов. Итак, объекты для переключателей создаются на основе класса `Radiobutton`. Класс `Scale` используется для создания объекта слайдера. Объекты для опций создаются на основе класса `Checkbutton`. Класс `Listbox` служит основой для создания объекта статического списка. Кроме этих компонентов, как отмечалось выше, мы используем такой графический компонент, как панель. Объект панели создается на основе класса `Frame`.

ПОДРОБНОСТИ

Главное назначение панели — быть контейнером для размещения других компонентов. Панель представляет собой прямоугольную область, в которую добавляются компоненты, а затем эту панель (вместе с компонентами, которые она содержит) можно добавить в другую панель или окно.

Как иллюстрацию в рассматриваемом далее примере для компоновки (размещения) компонентов на панелях используется метод `pack()` (сами панели, за исключением одной, размещаются в окне с помощью метода `place()`).

ПОДРОБНОСТИ

Если в случае использования метода `place()` положение компонента в контейнере указывается явно, то в случае использования метода `pack()` определяется общее положение компонента в контейнере (например, слева, справа, вверху или внизу в области контейнера).

Также в некоторых случаях мы используем для изменения содержимого элементов управления и считывания состояния элементов управления специальные «переменные» (а если точнее, то объекты) класса `StringVar`. Интересующий нас программный код представлен в листинге 11.5.

Листинг 11.5. Определение параметров шрифта

```
from tkinter import *

# Функция для определения характеристик шрифта:

def getFont():
```

```
    # Пустой список:
    res=[]
    # Название шрифта:
    name=lst.get(lst.curselection())
    # Размер шрифта:
    size=scl.get()
    # Добавление элементов в список:
    res.append(name)
    res.append(size)
    # Если установлена опция применения
    # жирного стиля:
    if bold.get()!="":
        res.append(bold.get())
    # Если установлена опция применения
    # курсивного стиля:
    if italic.get()!="":
        res.append(italic.get())
    # Результат функции:
    return res
# Функция для применения параметров шрифта:
def setAll(*args):
    # Вычисление шрифта:
    fnt=getFont()
    # Применение шрифта к тексту в окне:
    lbl.configure(font=fnt)
    # Определение цвета для текста в окне:
    lbl.configure(fg=color.get())
    # Определение шаблонного текста
    # для отображения в окне:
    txt="\nШрифт "
    # Название шрифта:
    txt+=fnt[0]
```

```
    # Размер шрифта:
    txt+=" размера "+str(fnt[1])+"\n"
    # Если применяется жирный стиль:
    if "bold" in fnt:
        txt+=" жирный"
    # Если применяется курсивный стиль:
    if "italic" in fnt:
        txt+=" курсивный"
    # Цвет шрифта:
    if color.get()=="red":
        txt+=" красного"
    elif color.get()=="blue":
        txt+=" синего"
    else:
        txt+=" черного"
    txt+=" цвета\n"
    # Отображение текста в окне:
    text.set(txt)
# Списки с характеристиками шрифтов:
fnt_1=["Arial",12,"italic"]
fnt_2=["Times New Roman",13,"bold","italic"]
# Список с названиями шрифтов для статического списка:
fonts=["Times New Roman","Arial","Courier New"]
# Минимальный размер шрифта:
min_size=15
# Максимальный размер шрифта:
max_size=21
# Ширина и высота окна:
W=640
H=420
# Высота панели с шаблонным текстом:
Hf=140
```

```
# Ширина и высота панели со статическим списком:
Wl=W/3
Hl=H-Hf-15
# Высота панели с кнопкой:
Hb=60
# Ширина и высота панели со слайдером и переключателями:
Ws=W-Wl-15
Hs=Hl-Hb-5
# Создание окна:
wnd=Tk()
wnd.title("Параметры шрифта")
wnd.geometry(str(W)+"x"+str(H))
wnd.resizable(False, False)
# Создание панелей:
frm_scale=Frame(wnd, bd=3, relief=GROOVE)
frm_text=Frame(wnd, bd=3, relief=GROOVE)
frm_btn=Frame(wnd, bd=3, relief=GROOVE)
frm_list=Frame(wnd, bd=3, relief=GROOVE)
frm_check=Frame(frm_list, bd=3, relief=GROOVE)
# Переменные для определения текстового
# содержимого в элементах управления:
text=StringVar()
color=StringVar()
bold=StringVar()
italic=StringVar()
# Создание текстовых меток:
lbl_text=Label(frm_text, text="Пример текста:", font=fnt_2)
lbl_color=Label(frm_scale, text="Цвет текста:", font=fnt_2)
lbl_size=Label(frm_scale, text="Размер текста:", font=fnt_2)
lbl_font=Label(frm_list, text="Название шрифта:", font=fnt_2)
lbl_style=Label(frm_check, text="Стиль шрифта:", font=fnt_2)
# Метка для отображения шаблонного текста:
```

```
lbl=Label(frm_text, textvariable=text)
# Фон и рамка для метки:
lbl.configure(bg="white", relief=RAISED)
# Переключатели:
rb_1=Radiobutton(frm_scale, text="красный", variable=color)
rb_1.configure(value="red", font=fnt_1)
rb_2=Radiobutton(frm_scale, text="синий", variable=color)
rb_2.configure(value="blue", font=fnt_1)
rb_3=Radiobutton(frm_scale, text="черный", variable=color)
rb_3.configure(value="black", font=fnt_1)
# Устанавливается переключатель:
color.set("blue")
# Создание слайдера:
scl=Scale(frm_scale, orient=HORIZONTAL)
# Диапазон изменения значений:
scl.configure(from_=min_size, to=max_size)
# Интервал для отображения подписей
# и шаг дискретности для положения ползунка:
scl.configure(tickinterval=1, resolution=1)
# Обработчик для события, связанного
# с изменением положения слайдера:
scl.config(command=setAll)
# Создание опций и настройка их параметров:
chb_1=Checkbutton(frm_check, text="жирный", variable=bold)
chb_1.configure(onvalue="bold", offvalue="", font=fnt_1)
chb_2=Checkbutton(frm_check, text="курсив", variable=italic)
chb_2.configure(onvalue="italic", offvalue="", font=fnt_1)
# Начальное состояние опций:
bold.set("")
italic.set("italic")
# Создание статического списка:
lst=Listbox(frm_list, selectmode=SINGLE, font=fnt_1)
```

```
# Цвет фона и цвет для выделения пункта:
lst.configure(bg="gray96", selectbackground="gray")
# Способ выделения пункта и высота списка:
lst.configure(activestyle="none", height=len(fonts)+1)
# Заполнение статического списка пунктами:
for n in fonts:
    lst.insert(END, n)
# По умолчанию выбран первый пункт:
lst.select_set(0)
# Обработчик для статического списка:
lst.bind("<<ListboxSelect>>", setAll)
# Создание кнопки:
btn=Button(frm_btn, text="OK", font=fnt_2)
# Обработчик для кнопки:
btn.configure(command=wnd.destroy)
# Размещение меток и слайдера на панелях:
lbl_text.pack(side="top", fill="x", padx=5, pady=5)
lbl.pack(side="top", fill="both", padx=5, pady=5)
lbl_color.pack(side="top", fill="x", padx=5, pady=5)
scl.pack(side="bottom", fill="x", padx=5, pady=5)
lbl_size.pack(side="bottom", fill="x", padx=5, pady=[25,5])
lbl_font.pack(side="top", fill="x", padx=5, pady=5)
lbl_style.pack(side="top", fill="x", padx=5, pady=5)
# Размещение переключателей:
rb_1.pack(side="left", fill="x", padx=5, pady=5)
rb_2.pack(side="left", fill="x", padx=5, pady=5)
rb_3.pack(side="left", fill="x", padx=5, pady=5)
# Размещение опций:
chb_1.pack(side="left", fill="x", padx=5, pady=5)
chb_2.pack(side="left", fill="x", padx=5, pady=5)
# Размещение статического списка:
lst.pack(side="top", fill="x", padx=5, pady=5)
```

```
# Размещение кнопки:
btn.pack(side="bottom", fill="x", padx=50, pady=10)
# Размещение панелей:
frm_text.place(x=5, y=5, width=W-10, height=Hf)
frm_check.pack(side="bottom", fill="both", padx=5, pady=5)
frm_list.place(x=5, y=Hf+10, height=Hl, width=Wl)
frm_scale.place(x=Wl+10, y=Hf+10, width=Ws, height=Hs)
frm_btn.place(x=Wl+10, y=Hf+Hs+15, width=Ws, height=Hb)
# Применение параметров шрифта к шаблонному тексту:
setAll()
# Режим отслеживания значения переменных:
color.trace("w", setAll)
bold.trace("w", setAll)
italic.trace("w", setAll)
# Отображение окна:
wnd.mainloop()
```

Код объемный, но на самом деле достаточно простой. По большому счету, речь идет о выполнении операций трех типов:

- нам нужно создать (и настроить параметры) объекты для соответствующих графических компонентов;
- следует определить для элементов управления (то есть тех компонентов, которые изменяют свое состояние в процессе работы приложения) способ обработки событий;
- нужно разместить графические компоненты на панелях, а панели разместить в окне.

В программе объявляются два списка `fnt_1` и `fnt_2`, которые определяют шрифты, используемые для отображения текста в графических компонентах. Список `fonts` содержит названия шрифтов, которые будут отображаться в статическом списке в окне. Значения переменных `min_size` и `max_size` определяют соответственно минимальный и максимальный размер для шрифта, применяемого к шаблонному тексту. Также мы используем группу переменных, определяющих размеры

(ширину и высоту) некоторых графических компонентов, в том числе ширину и высоту главного окна (переменные W и H).

ПОДРОБНОСТИ

При создании объекта окна его размеры определяются командой `wnd.geometry(str(W)+"x"+str(H))`. Примечательно здесь то, что текст, определяющий ширину и высоту окна, мы сформировали объединением текстовых значений, в том числе используя значения переменных W и H.

Главное окно приложения создается как объект класса Tk, и ссылка на этот объект записывается в переменную wnd. Мы создаем пять объектов класса Frame (frm_scale, frm_text, frm_btn, frm_list и frm_check), с помощью которых реализуются панели. Все панели, за исключением frm_check (панель с опциями), добавляются в главное окно. Панель frm_check добавляется на панель frm_list (панель со статическим списком). Аргумент bd конструктора класса Frame определяет толщину рамки вокруг панели, а аргумент relief определяет ее стиль (значение GROOVE означает эффект «вдавленной» рамки).

Текстовые метки lbl_text, lbl_color, lbl_size, lbl_font и lbl_style создаются на основе класса Label и содержат неизменяемый текст. Это фактически подписи для элементов управления. Панель, на которую добавляется метка, указана первым аргументом конструктора класса Label. Несколько особо создается метка lbl, предназначенная для отображения шаблонного текста. Для этой метки задается белый цвет фона (инструкция bg="white"), а также определена рамка с эффектом «выдавливания» (инструкция relief=RAISED). Но есть еще одна важная инструкция textvariable=text, указанная в конструкторе класса Label. Эта инструкция связывает содержимое метки с содержимым «переменной» text, причем сама «переменная» создается командой text=StringVar() (конечно, на самом деле речь идет об объекте). Специфика ситуации в том, что если изменится значение «переменной» text, то автоматически изменится содержимое метки, а изменение содержимого метки приведет к автоматическому изменению «переменной» text. То есть в данном случае мы используем такой своеобразный механизм обработки событий: чтобы изменить содержимое метки lbl, достаточно изменить значение «переменной» text. Для присваивания «переменной» нового значения используется метод set(), а для считывания значения «переменной» используется метод get().

ПОДРОБНОСТИ

Еще раз подчеркнем, что, хотя мы говорим о «переменной» `text`, на самом деле речь, конечно, идет об объекте. Кроме текстовых «переменных», которые создаются на основе класса `StringVar`, существуют, например, классы `BooleanVar`, `DoubleVar` и `IntVar`, позволяющие создавать «переменные», аналогичные `text` и предназначенные для работы с данными других типов.

Аналогичная схема использована для работы с переключателями и опциями, для чего командами `color=StringVar()`, `bold=StringVar()` и `italic=StringVar()` создаются соответствующие «переменные».

НА ЗАМЕТКУ

Для некоторых компонентов настройка параметров выполняется вызовом метода `configure()` из объекта компонента, причем метод может вызываться несколько раз. Сделано это исключительно из эстетических соображений. На самом деле можно все аргументы, определяющие настройки компонента, задать в одной команде с вызовом метода `configure()` или задать эти параметры в команде создания компонента.

Переключатели `rb_1`, `rb_2` и `rb_3` реализуются как объекты класса `Radiobutton`. Первым аргументом конструктору передается ссылка на панель `frm_scale`, на которую будет добавляться переключатель. Аргумент конструктора `text` определяет надпись для соответствующего переключателя. Для всех трех переключателей для аргумента `variable` указана «переменная» `color`. Это означает, что, запросив значение `color` с помощью метода `get()`, мы можем получить значение переключателя, который установлен на данный момент. «Значения» переключателей задаются с помощью аргумента `value`. Мы в качестве значений используем англоязычные названия для цвета (значения `"red"`, `"blue"` и `"black"`). Аргумент `font` определяет шрифт, которым отображается подпись для переключателя.

В результате выполнения команды `color.set("blue")` устанавливается переключатель со значением `"blue"` (соответствует синему цвету) — поэтому при первом отображении окна в нем установлен переключатель выбора синего цвета.

Нечто похожее происходит при создании опций `chb_1` и `chb_2`. Обе опции добавляются на панель `frm_check`. Для опции `chb_1` значение

аргумента text равно "жирный", а для опции chb_2 значение аргумента text установлено "курсив". Это подписи, которые будут отображаться возле опций. Шрифт, которым эти надписи будут отображаться, определяется значением аргумента font.

Опция, как известно, может находиться в двух состояниях: у нее может быть установлен или отменен флажок. С каждым из этих состояний ассоциируется определенное значение. Значение, отождествляемое с состоянием с установленным флажком, определяется аргументом onvalue. Если флажок не установлен, то такое состояние отождествляется со значением, которое определяется аргументом offvalue. Узнать или задать эти значения можно с помощью «переменной», которая указана в качестве значения аргумента variable. Для обеих опций в качестве значения аргумента offvalue указан пустой текст. Значение аргумента onvalue для опции chb_1 равно "bold", а для опции chb_2 оно равно "italic". Опция chb_1 связана с «переменной» bold, а опция chb_1 связана с «переменной» italic. Поэтому при выполнении команд bold.set("") и italic.set("italic") для первой опции флажок отменяется, а для второй — устанавливается.

Объект слайдера scl создается на основе класса Scale. Слайдер добавляется на панель frm_scale. Значение HORIZONTAL для аргумента orient означает, что слайдер ориентирован горизонтально.

ПОДРОБНОСТИ

Кроме значения HORIZONTAL можно использовать значение VERTICAL, означающее, что слайдер ориентирован вертикально. Также при создании объекта слайдера можно задать значение для аргумента label, которое будет определять подпись для слайдера. Для горизонтально расположенного слайдера она располагается в левом верхнем углу, а для вертикально расположенного слайдера — в правом верхнем углу. Мы вместо этого используем отдельную метку.

Диапазон изменения значений на слайдере определяется с помощью аргументов from_ и to. Аргумент tickinterval определяет интервал, с которым отображаются метки (подписей) на шкале слайдера, а аргумент resolution определяет шаг дискретности для изменения положения ползунка. В качестве обработчика для события, связанного с изменением положения ползунка на слайдере, определяется функция setAll() (имя функции присваивается в качестве значения аргументу command).

ПОДРОБНОСТИ

В результате вызова функции `setAll()` считываются настройки всех элементов управления в окне, и в соответствии с этими настройками формируется и отображается нужным шрифтом шаблонный текст. Наша «стратегия» состоит в том, чтобы при изменении состояния любого из элементов управления (статический список, опции, переключатели или слайдер) вызывалась функция `setAll()`. При этом речь идет об обработке разных событий, и в некоторых случаях при вызове функции-обработчика ей передаются аргументы (например, ссылка на объект события). Мы эти аргументы в процессе обработки событий не используем, но они могут быть, и шаблон функции должен соответствовать формату, в котором функция будет вызываться. Поэтому мы применяем искусственный прием — в описании функции указан аргумент со звездочкой `*`. Это означает, что функции при вызове может передаваться произвольное количество аргументов. Более детально код функции `setAll()` анализируется немного позже.

Статический список `lst` создается на основе класса `Listbox`. Статический список добавляется на панель `frm_list`. Значение `SINGLE` аргумента `selectmode` означает, что в списке можно выбрать только один пункт.

ПОДРОБНОСТИ

Возможные значения аргумента `selectmode`: `SINGLE` (возможен выбор одного пункта), `BROWSE` (возможен выбор одного пункта с возможностью перетаскивания выделения мышью), `MULTIPLE` (возможен выбор нескольких пунктов) и `EXTENDED` (возможен выбор нескольких пунктов, причем они могут быть в разных местах списка).

Аргумент `bg` определяет цвет фона для статического списка. Аргумент `selectbackground` задает цвет, которым выделяется выбранный пункт. Значения `"gray96"` и `"gray"` соответствуют разным оттенкам серого цвета.

ПОДРОБНОСТИ

Среди текстовых констант, определяющих разные цвета, есть группа констант, определяющих оттенки серого цвета. Название констант получается объединением текста `"gray"` и текстового представления для числа от `"0"` до `"100"`. Константа `"gray0"` соответствует черному цвету, а константа `"gray100"` соответствует белому цвету. Вместо текста `"gray"` можно также использовать текст `"grey"`.

Аргумент `height` определяет высоту статического списка (в строках). Значение `len(fonts)+1` означает, что статически список содержит на одну строку больше, чем пунктов в этом списке (поскольку количество пунктов в статическом списке определяется количеством элементов в списке `fonts`). Также мы для аргумента `activestyle` указываем значение `"none"`. Это означает, что пункт в статическом списке при выборе будет выделяться только цветом.

ПОДРОБНОСТИ

Возможные значения для аргумента `activestyle`: `"dotbox"` (вокруг выбранного пункта списка отображается пунктирная рамка), `"underline"` (выбранный пункт выделяется подчеркиванием) и `"none"` (дополнительные эффекты не используются).

После создания статического списка его необходимо заполнить (добавить в него пункты). Для этого запускается оператор цикла, в котором перебирается содержимое списка `fonts`. За каждый цикл выполняется команда `lst.insert(END, n)`, которой в конец статического списка добавляется очередной элемент `n` из списка `fonts`.

НА ЗАМЕТКУ

Поскольку первым аргументом метода `insert()` указано значение `END`, но соответствующий элемент (второй аргумент метода) добавляется в конец статического списка. Вообще же первым аргументом можно передавать индекс, определяющий позицию, на которую в статическом списке помещается добавляемый элемент.

Выполнение команды `lst.select_set(0)` означает, что по умолчанию в статическом списке будет выбран элемент с нулевым индексом. Наконец, командой `lst.bind("<<ListboxSelect>>", setAll)` для статического списка задается обработчик события `<<ListboxSelect>>`, связанного с выбором пункта в списке. В этом случае вызывается функция `setAll()`.

Кнопка, предназначенная для закрытия окна, создается уже знакомым нам способом на основе класса `Button`.

После создания объектов для графических компонентов они размещаются на панелях, а затем панели размещаются в окне. Как отмечалось ранее, для размещения компонентов мы используем метод `pack()`.

Аргумент `side` этого метода определяет место, в которое добавляется компонент. Возможные значения этого аргумента: `"top"` (размещение вверху), `"bottom"` (размещение внизу), `"left"` (размещение слева) и `"right"` (размещение справа). Аргумент `fill` определяет режим масштабирования компонента. Значение `"both"` для аргумента означает, что компонент в случае необходимости масштабируется и вдоль горизонтали, и вдоль вертикали. Значение `"x"` означает, что масштабирование выполняется вдоль горизонтали. Значение `"y"` используется, если масштабирование должно выполняться вдоль вертикали. Значение `"none"` соответствует ситуации, когда масштабирование не применяется.

Аргументы `padx` и `pady` определяют внешние отступы (в пикселях) по отношению к смежным компонентам соответственно вдоль горизонтали и вертикали. Если значением аргумента указан список из двух элементов, то они определяют отступы с одной и другой стороны компонента.

НА ЗАМЕТКУ

Команды, использованные для размещения компонентов, все однотипные и комментариев не требуют. Единственное, на что хочется обратить внимание, это то, что важен порядок, в котором компоненты добавляются в контейнер. Например, если мы последовательно добавляем два компонента и в обоих случаях в качестве значения аргумента `side` указано `"left"`, то компоненты будут добавляться в левую часть контейнера один за другим. Если для обоих указано значение `"top"`, то они добавляются один под другим. Если для первого указано значение `"left"`, а для другого `"top"`, то в левую часть контейнера добавляется первый компонент, а второй добавляется в верхнюю часть свободной области контейнера. Если поменять порядок добавления компонентов, то сначала будет добавлен компонент в верхнюю часть контейнера, а затем в левую часть свободной области добавляется следующий компонент. Это особенно важно, если используется масштабирование компонентов.

Для размещения панелей в окне использован уже знакомый нам метод `place()`. Достойно внимания здесь то, что для вычисления положения и размеров (ширины и высоты) компонентов мы используем созданные ранее переменные. Такой подход удобен тем, что если нам впоследствии понадобится изменить размеры того или иного компонента, то, скорее всего, все сведется к изменению значения соответствующей переменной.

ПОДРОБНОСТИ

Напомним, что при работе с методом `place()` положение компонента определяется с помощью аргументов `x` и `y`, а его размеры (в пикселях) задаются с помощью аргументов `width` (ширина) `height` (высота).

После того как все компоненты размещены, вызывается функция `setAll()`, в результате чего в соответствии с настройками управляющих компонентов вычисляются параметры шрифта для шаблонного текста, определяется сам шаблонный текст и этот текст (с использованием соответствующего шрифта) отображается в окне. Также перед отображением окна командой `wnd.mainloop()`, командами `color.trace("w", setAll)`,`bold.trace("w", setAll)` и `italic.trace("w", setAll)` для «переменных» `color`, `bold` и `italic` устанавливается режим отслеживания значения: каждый раз, когда будет изменяться значение соответствующей переменной, будет вызываться функция `setAll()`. Данными командами фактически определяется способ обработки событий, связанных с изменением состояния переключателей и опций.

ПОДРОБНОСТИ

Первым аргументом методу `trace()` можно передавать такие значения: `"w"` (отслеживается изменение значения «переменной"), `"r"` (отслеживается считывание значения «переменной") и `"u"` (отслеживается ситуация, связанная с удалением «переменной").

Необходимость в отслеживании значений «переменных» связана с тем, что эти значения могут происходить вследствие манипуляций пользователя с соответствующими элементами управления. Мы хотим, чтобы в таких случаях вызывалась функция `setAll()`. Имя этой функции передается вторым аргументом методу `trace()`.

Команды с вызовом метода `trace()` размещены в конце программного кода, перед командой отображения окна, чтобы при настройке параметров графических компонентов, влияющих на значения «переменных» `color`, `bold` и `italic`, не вызывалась функция `setAll()`.

Нам осталось проанализировать код двух функций, которые использованы в программе, причем одна из них вызывается в другой.

Функция `getFont()` предназначения для определения шрифта по настройкам компонентов окна. Результатом она возвращает список, определяющий название, размер и стиль шрифта. Результат функции

вычисляется следующим образом. Сначала создается пустой список res. Командой name=lst.get(lst.curselection()) определяется название шрифта. Для этого мы из объекта статического списка lst вызываем метод get(). Метод возвращает в качестве результата название пункта в статическом списке с индексом, указанным аргументом метода. Значением выражения lst.curselection() является индекс выделенного в статическом списке элемента.

Размер шрифта вычисляется командой size=scl.get(). В данном случае мы из объекта слайдера scl вызываем метод get(), который в качестве результата возвращает значение, установленное на слайдере. Командами res.append(name) и res.append(size) название и размер шрифта добавляются в список res. После этого начинается определение стиля шрифта. В условном операторе проверяется условие bold.get()!="". Оно истинно, если у опции применения жирного шрифта установлен флажок. Если так, то командой res.append(bold.get()) в список res добавляется значение "bold".

ПОДРОБНОСТИ

Стоит напомнить, что для опции выбора жирного стиля установлены два значения: пустой текст "", если флажок опции не установлен, и текст "bold", если флажок установлен. Также с опцией связана «переменная» bold, позволяющая определить состояние опции. Чтобы прочитать значение (определяющее состояние опции), используется инструкция bold.get(). Если флажок опции не установлен, то значением выражения bold.get() является пустой текст "". Если флажок опции установлен, то значением выражения bold.get() является текст "bold".

Это же замечание относится и к определению состояния опции, предназначенной для применения курсивного стиля. Следует лишь сделать поправку на то, что с этой опцией связана «переменная» italic, а возможными значениями выражения italic.get() является пустой текст "" (флажок опции не установлен) или текст "italic" (флажок опции установлен).

Если установлена опция применения курсивного стиля, то истинным является условие italic.get()!="" и командой res.append(italic.get()) в список res добавляется элемент "italic". По завершении вычислений ссылка на список res возвращается в качестве результата функции.

Функция `setAll()` предназначена для применения параметров шрифта. Как отмечалось ранее, для того чтобы функцию можно было вызывать с разными аргументами (которые мы не собираемся использовать), она описана как такая, что может иметь произвольное количество аргументов (аргумент `*args` в описании функции). В теле функции командой `fnt=getFont()` на основе настроек графических компонентов вычисляется шрифт, и ссылка на список с параметрами шрифта записывается в переменную `fnt`. Командой `lbl.configure(font=fnt)` к метке отображения шаблонного текста применяется шрифт, вычисленный функцией `getFont()`. Командой `lbl.configure(fg=color.get())` задается цвет для отображения текста. Для этого аргументу `fg` в качестве значения присваивается выражение `color.get()`. Здесь мы учли, что «переменная» `color` связана с группой переключателей, и при вызове метода `get()` мы получаем значение, связанное с переключателем, который в данный момент установлен. Поскольку при создании переключателей с каждым из них мы ассоциировали текстовое название соответствующего цвета, то результатом выражения `color.get()` является название цвета, что нам и нужно.

После этого начинается формирование собственно шаблонного текста. Переменная `txt` получает начальное значение `"\nШрифт "`, и затем это значение «уточняется». Сначала командой `txt+=fnt[0]` к текстовой строке дописывается название шрифта, которое является первым элементом в списке `fnt`. Также в текст добавляется информация о размере шрифта (второй элемент `fnt[1]` в списке `fnt`). С помощью условных операторов мы проверяем, есть ли в списке `fnt` элементы `"bold"` (жирный стиль) и `"italic"` (курсивный стиль), и если элемент в списке представлен, то к текстовой строке дописывается соответствующий текстовый фрагмент. После этого в зависимости от цвета шрифта к текстовой строке `txt` добавляется и такая информация. После того как текст сформирован, командой `text.set(txt)` он применяется для метки `lbl` с шаблонным текстом. Здесь мы учли, что с этой меткой связана «переменная» `text`. Применение нового значения для «переменной» с помощью метода `set()` приводит к автоматическому обновлению содержимого метки.

Использование меню

— По-вашему, это не интересно?
— Интересно. Для любителей древности.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Следующая программа, которую мы рассмотрим, является вариацией предыдущего примера. Как и ранее, мы будем с помощью элементов управления определять параметры шрифта, а в области окна приложения с применением этого шрифта будет отображаться шаблонный текст. Но теперь вместо статического списка, слайдера, опций и переключателей мы будем использовать *панель меню* и *панель инструментов*. Окно, которое будет появляться при запуске программы на выполнение, показано на рис. 11.13.

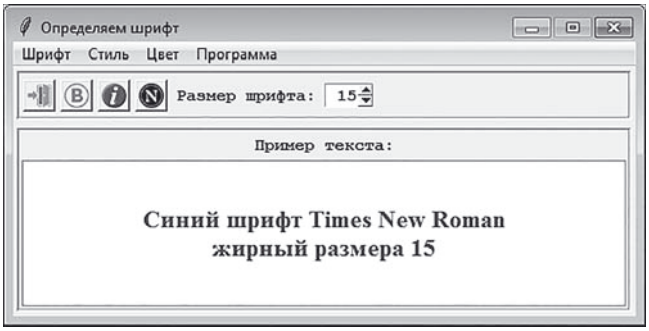


Рис. 11.13. Окно приложения с меню и панелью инструментов

Окно содержит панель меню с пунктами **Шрифт**, **Стиль**, **Цвет** и **Программа**. Под панелью меню расположена панель инструментов с четырьмя кнопками, метка с текстом и *спиннер*, в котором можно выбирать размер шрифта (допустимые значения находятся в диапазоне от **15** до **20**). Под панелью инструментов располагается еще одна вспомогательная панель с подписью и шаблонным текстом на белом фоне. Текст содержит информацию о названии, стиле, размере и цвете шрифта. По умолчанию используется шрифт **Times New Roman**, жирный, размера **15**, синего цвета. Выбрать новый шрифт можно в пункте меню **Шрифт**, как показано на рис. 11.14.

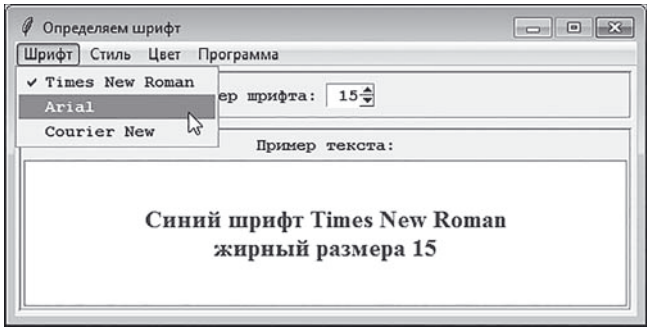


Рис. 11.14. Выбор шрифта в меню **Шрифт**

В этом пункте меню используются команды типа переключателей: возле выбранного пункта отображается флажок, и этот флажок один и только один. При выборе нового шрифта (или изменении любого иного параметра шрифта) он применяется к шаблонному тексту, причем сам текст тоже меняется.

На рис. 11.15 проиллюстрировано содержимое пункта меню **Стиль**, предназначенного для установки стиля шрифта.

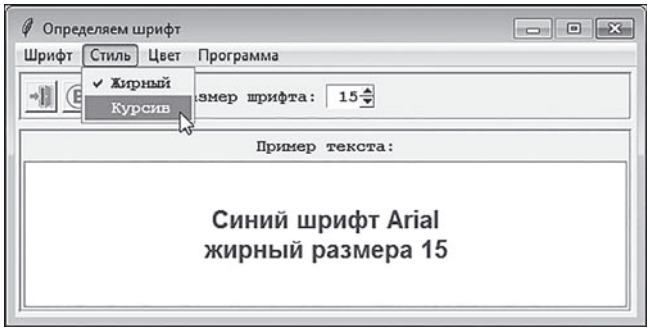


Рис. 11.15. Выбор стиля шрифта в меню **Стиль**

В пункте меню **Стиль** две команды (для установки/отмены жирного и/или курсивного стиля), причем это команды опционного типа: возле каждой из них, независимо от другой, может быть установлен или отменен флажок.

В пункте меню **Цвет**, содержимое которого представлено на рис. 11.16, можно выбрать цвет для отображения текста.

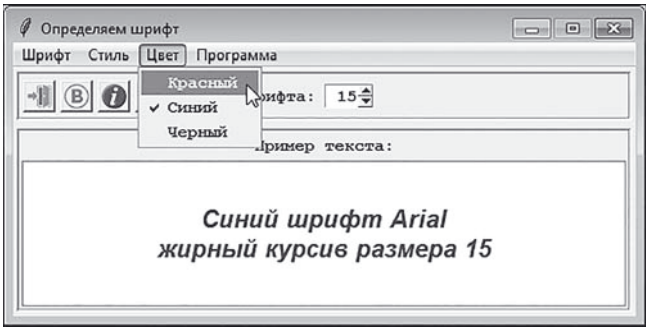


Рис. 11.16. Выбор цвета шрифта в меню **Цвет**

Три команды для выбора цвета также относятся к командам типа переключателей: возле одной из этих команд отображается флажок.

Наконец, в пункте меню **Программа** всего две команды (между которыми расположен разделитель). Содержимое этого пункта меню можно видеть на рис. 11.17.

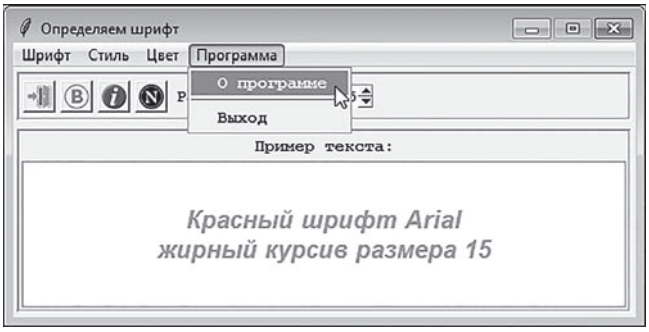


Рис. 11.17. Содержимое меню **Программа**

Там всего две команды. При выборе команды **Выход** приложение завершает работу. Если выбрать команду **О программе**, то на фоне окна приложения будет открыто модальное окно с информацией о программе, как показано на рис. 11.18.

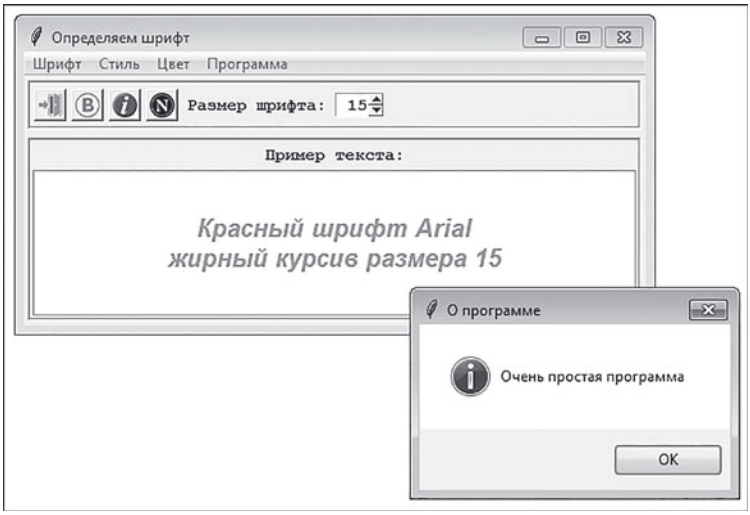


Рис. 11.18. Окно с сообщением на фоне окна приложения

Пока не будет закрыто модальное окно, основное окно будет заблокировано.

Также в приложении используется *контекстное меню*: если в рабочей области приложения нажать правую кнопку мыши, то появится контекстное меню с командами, позволяющими выбрать шрифт, стиль, цвет, или завершить работу с приложением. Открытое контекстное меню представлено на рис. 11.19.

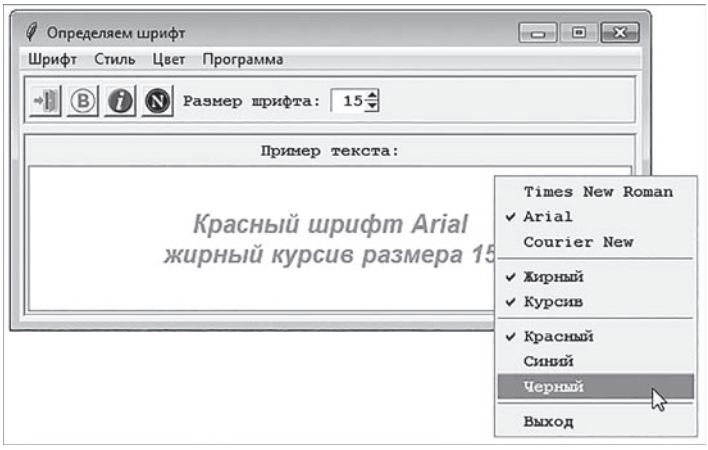


Рис. 11.19. Выбор цвета в контекстном меню

Следует отметить, что контекстное меню полностью синхронизировано с командами в главном меню — например, если в контекстном меню выбрать цвет, то информация об этом выборе отобразится и в главном меню, и наоборот.

Размер шрифта можно изменить с помощью спиннера, как показано на рис. 11.20.

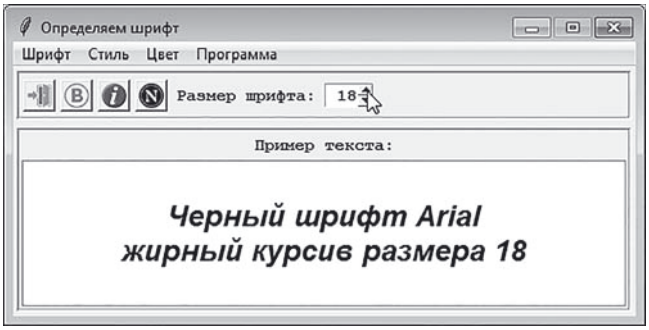


Рис. 11.20. Изменение размера шрифта с помощью спиннера

Панель инструментов содержит, кроме спиннера, еще четыре кнопки. При нажатии первой кнопки слева окно закрывается и приложение завершает работу. Следующие две кнопки (с символами **B** и **i**) предназначены для применения/отмены соответственно жирного и курсивного стилей. Нажатие кнопки приводит к изменению настроек на «противоположные». Например, если был установлен жирный стиль, то нажатие кнопки с символом **B** приведет к отмене жирного стиля. Если жирный стиль не был установлен, то нажатие кнопки приведет к применению жирного стиля. Кнопка с литерой **N** предназначения для отмены жирного и курсивного стиля: при нажатии кнопки стиль будет обычным (не жирным и не курсивным). На рис. 11.20 проиллюстрированы последствия нажатия кнопки отмены жирного и курсивного стилей.

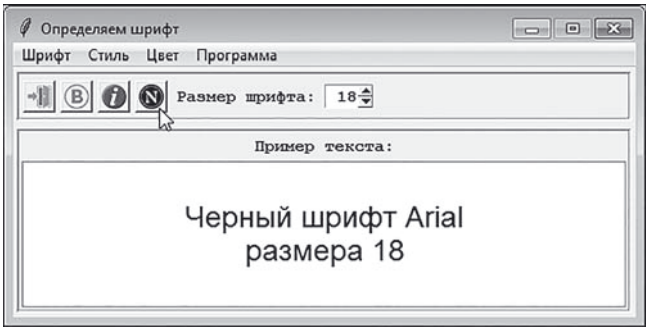


Рис. 11.21. Результат нажатия кнопки отмены жирного и курсивного стилей

Так выглядит и функционирует приложение. Теперь проанализируем соответствующий программный код, представленный в листинге 11.6.

Листинг 11.6. Работа с меню

```
from tkinter import *
from tkinter.messagebox import *
# Класс для создания окна:
class MyApp:
    # Конструктор:
    def __init__(self):
        # Настройка параметров:
        self.setValues()
        # Создание главного окна:
        self.makeMainWindow()
        # Определение "переменных" для обработки событий:
        self.setVars()
        # Создание главного меню:
        self.makeMainMenu()
        # Создание панели инструментов:
        self.makeToolBar()
        # Создание вспомогательной панели:
        self.makeFrame()
        # Создание контекстного меню:
        self.makePopupMenu()
        # Вычисление параметров шаблонного текста:
        self.apply()
        # Режим отслеживания "переменных":
        self.traceVars()
        # Отображение главного окна:
        self.showMainWindow()
    # Метод для настройки параметров:
    def setValues(self):
        # Ширина и высота окна:
```

```
        self.W=500
        self.H=200
        # Высота панели инструментов:
        self.h=40
        # Размеры главного окна:
        self.position=str(self.W)+"x"+str(self.H)
        # Названия шрифтов:
        self.fonts=["Times New Roman","Arial","Courier New"]
        # Словарь с названиями цвета:
        self.colors={"red":"Красный","blue":"Синий", "black":"Черный"}
        # Список с названиями стиля:
        self.style=[["bold","Жирный"],["italic","Курсив"]]
        # Список с названиями файлов с изображениями:
        self.imgFiles=["exit.png","bold.png","italic.png", "normal.png"]
        # Путь к каталогу с файлами:
        self.path="D:\\Books\\Python\\Pictures\\"
        # Основной шрифт:
        self.font=("Courier New",10,"bold")
    # Метод для создания главного окна:
    def makeMainWindow(self):
        self.wnd=Tk()
        self.wnd.title("Определяем шрифт")
        self.wnd.geometry(self.position)
        self.wnd.resizable(False, False)
    # Метод для отображения главного окна:
    def showMainWindow(self):
        self.wnd.mainloop()
    # Метод для создания главного меню:
    def makeMainMenu(self):
        # Создание объекта для панели меню:
        self.menubar=Menu(self.wnd)
        # Создание пунктов главного меню:
```

```
        self.mFont=Menu(self.wnd, font=self.font, tearoff=0)
        self.mStyle=Menu(self.wnd, font=self.font, tearoff=0)
        self.mColor=Menu(self.wnd, font=self.font, tearoff=0)
        self.mAbout=Menu(self.wnd, font=self.font, tearoff=0)
        # Заполнение пунктов меню:
        self.setMenuFont(self.mFont)
        self.setMenuStyle(self.mStyle)
        self.setMenuColor(self.mColor)
        # Заполнение последнего пункта меню:
        self.mAbout.add_command(label="О программе", command=self.showDialog)
        # Добавление разделителя:
        self.mAbout.add_separator()
        self.mAbout.add_command(label="Выход", command=self.clExit)
        # Добавление пунктов меню на панель меню:
        self.menubar.add_cascade(label="Шрифт", menu=self.mFont)
        self.menubar.add_cascade(label="Стиль", menu=self.mStyle)
        self.menubar.add_cascade(label="Цвет", menu=self.mColor)
        self.menubar.add_cascade(label="Программа", menu=self.mAbout)
        # Задаем главное меню для окна:
        self.wnd.config(menu=self.menubar)
    # Метод для создания панели инструментов:
    def makeToolBar(self):
        # Список с названиями методов для обработки
        # события, связанного с нажатием кнопок:
        mt=[self.clExit, self.clBold, self.clItalic, self.clNormal]
        # Панель для размещения кнопок:
        self.toolbar=Frame(self.wnd, bd=3, relief=GROOVE)
        # Список для изображений:
        self.imgs=[]
        # Список для кнопок:
        self.btns=[]
        # Создание изображений и кнопок:
```

```
        for f in self.imgFiles:
            # Создание изображения:
            self.imgs.append(PhotoImage(file=self.path+f))
            # Создание кнопки:
            self.btns.append(Button(self.toolbar, image=self.imgs[-1]))
            # Добавление кнопки на панель:
            self.btns[-1].pack(side="left", padx=2, pady=2)
        # Определение обработчиков для кнопок:
        for k in range(len(mt)):
            self.btns[k].configure(command=mt[k])
        # Создание текстовой метки:
        self.lblSize=Label(self.toolbar, text="Размер шрифта:", font=self.font)
        # Размещение метки на панели:
        self.lblSize.pack(side="left", padx=2, pady=2)
        # Создание спиннера:
        self.spnSize=Spinbox(self.toolbar, from_=15, to=20, font=self.font,
                 width=3, justify="right", textvariable=self.size)
        # Размещение спиннера на панели:
        self.spnSize.pack(side="left", padx=2, pady=2)
        # Размещение панели в окне:
        self.toolbar.place(x=3, y=3, width=self.W-6, height=self.h)
    # Метод для создания вспомогательной панели:
    def makeFrame(self):
        # Создание вспомогательной панели:
        self.frame=Frame(self.wnd, bd=3, relief=GROOVE)
        # Создание метки и добавление ее на панель:
        Label(self.frame, text="Пример текста:", font=self.font).pack(side="top")
        # Создание метки для отображения
        # шаблонного текста:
        self.lblText=Label(self.frame, textvariable=self.text, relief=GROOVE,
                bg="white", height=5)
        # Размещение метки на вспомогательной панели:
```

```
        self.lblText.pack(side="top", fill="both", padx=1, pady=1)
        # Размещение вспомогательной панели в окне:
        self.frame.place(x=3, y=self.h+9, width=self.W-6, height=self.H-self.h-12)
    # Метод для создания контекстного меню:
    def makePopupMenu(self):
        # Создание объекта контекстного меню:
        self.popup=Menu(self.wnd, tearoff=0, font=self.font)
        # Добавление команд в контекстное меню:
        self.setMenuFont(self.popup)
        # Добавление разделителя:
        self.popup.add_separator()
        # Добавление команд в контекстное меню:
        self.setMenuStyle(self.popup)
        # Добавление разделителя:
        self.popup.add_separator()
        # Добавление команд в контекстное меню:
        self.setMenuColor(self.popup)
        # Добавление разделителя:
        self.popup.add_separator()
        # Добавление команды в контекстное меню:
        self.popup.add_command(label="Выход", command=self.clExit)
        # Определение обработчика события
        # для контекстного меню:
        self.wnd.bind("<Button-3>", lambda evt: self.popup.tk_popup(evt.x_root,
                      evt.y_root))
    # Метод для формирования команд меню,
    # связанных с выбором шрифта:
    def setMenuFont(self, menu):
        for f in self.fonts:
            menu.add_radiobutton(label=f, value=f, variable=self.name)
        self.name.set(self.fonts[0])
    # Метод для формирования команд меню,
```

```
    # связанных с выбором стиля:
    def setMenuStyle(self, menu):
        for k in range(len(self.style)):
            menu.add_checkbutton(label=self.style[k][1], onvalue=True,
                    offvalue=False, variable=self.bi[k])
        self.bi[0].set(True)
        self.bi[1].set(False)
    # Метод для формирования команд меню,
    # связанных с выбором цвета:
    def setMenuColor(self, menu):
        for r in self.colors.keys():
            menu.add_radiobutton(label=self.colors[r], value=r,
                    variable=self.color)
        self.color.set("blue")
    # Метод для определения параметров шрифта и
    # вычисления шаблонного текста:
    def apply(self,*args):
        # Цвет для шрифта:
        clr=self.color.get()
        # Название шрифта:
        nm=self.name.get()
        # Размер шрифта:
        sz=self.size.get()
        # Применение цвета к метке:
        self.lblText.configure(fg=clr)
        # Список с параметрами шрифта:
        fnt=[nm, sz]
        # Формирование шаблонного текста и
        # определение параметров шрифта:
        txt=self.colors[clr]+" шрифт "+nm+"\n"
        for k in range(len(self.style)):
            if self.bi[k].get():
```

```
            fnt.append(self.style[k][0])
            txt+=self.style[k][1].lower()+" "
        txt+="размера "+str(sz)
        # Применение шрифта к метке:
        self.lblText.configure(font=fnt)
        # Задаем текст для метки:
        self.text.set(txt)
    # Метод для создания "переменных", используемых
    # для обработки событий:
    def setVars(self):
        self.text=StringVar()
        self.name=StringVar()
        self.bi=[BooleanVar(), BooleanVar()]
        self.size=IntVar()
        self.color=StringVar()
    # Метод для перехода в режим отслеживания
    # значений "переменных":
    def traceVars(self):
        mt=self.apply
        self.name.trace("w", mt)
        self.color.trace("w", mt)
        for k in range(len(self.bi)):
            self.bi[k].trace("w", mt)
        self.size.trace("w", mt)
    # Метод для обработки нажатия кнопки
    # для завершения работы:
    def clExit(self):
        self.wnd.destroy()
    # Метод для обработки нажатия кнопки
    # применения/отмены жирного стиля:
    def clBold(self):
        self.bi[0].set(not self.bi[0].get())
```

```
    # Метод для обработки нажатия кнопки
    # применения/отмены курсивного стиля:
    def clItalic(self):
        self.bi[1].set(not self.bi[1].get())
    # Метод для обработки нажатия кнопки
    # отмены жирного и курсивного стиля:
    def clNormal(self):
        self.bi[0].set(False)
        self.bi[1].set(False)
    # Метод для отображения окна с сообщением:
    def showDialog(self):
        showinfo("О программе","Очень простая программа")
# Отображение окна на экране:
MyApp()
```

В этой программе мы немного изменили общий стратегический подход, а именно, в ней описывается класс `MyApp`, а затем на его основе создается объект (анонимный, поскольку ссылка на этот объект никуда не записывается). Ссылки на объекты графических компонентов (и другие важные характеристики) реализуются как поля объекта класса. Также для удобства восприятия мы разбили выполняемый код на отдельные блоки, реализованные как методы объекта.

При создании объекта класса вызывается, как известно, конструктор. Конструктор класса `MyApp` описан так, что в нем вызывается несколько других методов. Сначала вызывается метод `setValues()`, с помощью которого задаются все основные параметры объекта, используемые в работе приложения. В частности, при вызове метода создаются:

- поля `W` и `H` объекта с данными о ширине и высоте главного окна;
- поле `h` со значением высоты панели инструментов;
- текстовое поле `position` с размерами окна (для передачи в качестве аргумента методу `geometry()`);
- поле-список `fonts` с названиями шрифтов;
- поле-словарь `colors` с названиями цвета (ключом является английское название, а значение — русское название);

- поле-список из двух элементов `style`, и каждый его элемент содержит английское и русское название для стиля;
- поле-список `imgFiles` с названиями файлов с изображениями (используется при создании кнопок на панели инструментов);
- текстовое поле `path`, определяющее путь к каталогу с файлами изображений;
- поле-кортеж `font`, определяющее параметры шрифта, используемого в окне для отображения подписей.

Главное окно создается вызовом метода `makeMainWindow()`, при этом ссылка на созданный объект окна записывается в поле `wnd`.

С помощью метода `setVars()` создаются «переменные», которые мы планируем использовать в процессе обработки событий. Каждая такая «переменная» (`text`, `name` и `color` класса `StringVar`, `size` класса `IntVar`, а также поле-список `bi` с двумя «переменными» класса `BooleanVar`) реализована как поле объекта.

Для создания главного меню вызывается метод `makeMainMenu()`. Команды в теле этого метода требуют некоторого внимания. Так, сначала командой `self.menubar=Menu(self.wnd)` создается объект для панели меню. Объект создается на основе класса `Menu`. Аргументом указана ссылка на окно, в которое будет добавляться панель меню. Пункты для главного меню (`mFont`, `mStyle`, `mColor` и `mAbout`) также создаются на основе класса `Menu`. В качестве аргументов конструктору передаются ссылка на объект окна и шрифт, а аргумент `tearoff` со значением `0` означает, что пункт меню нельзя будет открепить главного меню.

> **НА ЗАМЕТКУ**
>
> По умолчанию при раскрытии пункта меню непосредственно под ним отображается пунктирная линия, щелчок на которой приводит к откреплению содержимого пункта и отображению этого содержимого в отдельной панели.

После создания команд меню их необходимо добавить в соответствующий пункт меню. Для решения этой задачи мы используем специально созданные методы `setMenuFont()`, `setMenuStyle()` и `setMenuColor()`, а последний пункт заполняем в явном виде.

НА ЗАМЕТКУ

В качестве аргументов методам `setMenuFont()`, `setMenuStyle()` и `setMenuColor()` передаются, соответственно, ссылки `mFont`, `mStyle` и `mColor` на заполняемые пункты меню. Сами методы и необходимость передачи им упомянутых аргументов обусловлены тем, что мы используем эти же методы при формировании содержимого контекстного меню. Методы описываются далее.

Для добавления команд в пункт меню `mAbout` из объекта пункта меню вызывается метод `add_command()`. Аргумент `label` определяет надпись (название команды), аргумент `command` определяет метод, вызываемый при выборе команды. Для добавления разделителя (линии) между командами используется метод `add_separator()`.

НА ЗАМЕТКУ

При вызове метода `showDialog()` отображается модальное окно справки. При вызове метода `clExit()` закрывается окно приложения. Методы описываются немного позже.

Для добавления пунктов в главное меню используется метод `add_cascade()`, который вызывается из объекта главного меню `menubar`. Аргумент `label` определяет название пункта меню, а аргумент `menu` содержит ссылку на объект добавляемого пункта меню.

После того как главное меню сформировано и заполнено, из объекта окна `wnd` вызывается метод `config()` в качестве аргумента `menu` которому передается ссылка `menubar` на объект главного меню.

Теперь проанализируем методы, использованные для формирования команд для пунктов меню. Второй аргумент метода `setMenuFont()` (как и прочих методов) называется `menu`. Мы исходим из того, что это ссылка на объект пункта меню, который заполняется. В операторе цикла переменная `f` принимает значения из списка `fonts` (названия шрифтов). Из объекта заполняемого пункта `menu` вызывается метод `add_radiobutton()`, в пункт меню добавляется команда типа переключателя. Значение `f` для аргументов `label` и `value` означает, что название шрифта будет отображаться как название команды, и оно же будет значением «переменной» `name`, ссылка на которую указана в качестве значения аргумента `variable`, в том случае, если выбрана соответствующая команда. После завершения оператора цикла командой

`self.name.set(self.fonts[0])` задается значение «переменной» `name` — это название первого шрифта в списке `fonts` (и соответствующая команда будет выделена флажком в пункте меню).

Прочие методы реализованы аналогично. В методе `setMenuStyle()` с помощью индекса `k` перебираются элементы списка `style`. Для добавления команды опционного типа используется метод `add_checkbutton()`. Отображаемое русское название стиля (аргумент `label`) определяется элементом `style[k][1]`. С опционными командами ассоциируются логические «переменные» — элементы поля-списка `bi` (инструкция `variable=self.bi[k]`). При установленном флажке для команды соответствующая логическая «переменная» принимает значение `True` (значение аргумента `onvalue`), а если флажок не установлен, то значение «переменной» равно `False` (значение аргумента `offvalue`). Командами `self.bi[0].set(True)` и `self.bi[1].set(False)`, задаются значения элементов из списка `bi`, что соответствует установленному флажку первой опции (опция выбора жирного стиля) и отсутствию флажка у другой опции (опция выбора курсивного стиля).

В теле метода `setMenuColor()` перебираются (с помощью переменной `r`) ключи словаря `colors`. В пункт меню с помощью метода `add_radiobutton()` добавляются команды типа переключателя. Название команды определяется значением элемента из словаря `colors` с ключом `r`. Ключ `r` является значением «переменной» `color` — все это благодаря инструкциям `value=r` и `variable=self.color`.

НА ЗАМЕТКУ

Получается, что отображается русское название текста, а при выборе команды значением «переменной» `color` будет английское название текста.

С помощью команды `self.color.set("blue")` мы задаем значение `"blue"` для «переменной» `color` (поэтому по умолчанию будет выбран синий цвет для отображения текста).

Панель инструментов создается при вызове метода `makeToolBar()`. В теле метода командой `mt=[self.clExit, self.clBold, self.clItalic, self.clNormal]` создается список с названиями методов, предназначенных для обработки события, связанного с нажатием кнопок на панели инструментов.

НА ЗАМЕТКУ

Методы указаны в том порядке, в котором кнопки будут добавляться на панель. И это тот же порядок, в котором в списке `imgFiles` указаны названия файлов с изображениями, предназначенными для размещения на кнопках.

Панель инструментов создается как обычная панель — объект класса `Frame`. Ссылка на панель записывается в поле `toolbar`. Кроме этого, мы создаем еще два поля-списка: `imgs` для записи в него ссылок на объекты изображений, а также `btns` для запоминания ссылок на объекты кнопок. Изображения и кнопки создаются в операторе цикла, в котором переменная `f` принимает значения из списка `imgFiles` (названия файлов с изображениями). Объекты изображений создаются на основе класса `PhotoImage` и добавляются в список `imgs` с помощью метода `append()`.

Кнопки создаются на основе класса `Button`, причем значением аргумента `image` указана ссылка на последнее (с индексом `-1`) на данный момент изображение в списке `imgs`. Соответствующая картинка будет отображаться на кнопке.

НА ЗАМЕТКУ

Изображения и кнопки создаются и добавляются в списки синхронно. Поэтому добавляемой (на определенном этапе) в список `btns` кнопке соответствует последнее добавленное в список `imgs` изображение.

Кнопки на панель добавляются с помощью метода `pack()`. При этом ссылку на последнюю (только что добавленную в список `btns`) кнопку можно получить через инструкцию `btns[-1]`.

После добавления кнопок на панель они заново перебираются, и для каждой из них задается обработчик (элемент из списка `mt`).

После этого мы создаем и размещаем на панели инструментов текстовую метку (с текстом **Размер шрифта**). После метки на панель добавляется спиннер. Объект спиннера `spnSize` создается на основе класса `Spinbox`. Аргументами конструктору передаются:

- ссылка на панель, в которую будет добавляться спиннер;
- аргументы `from_` и `to`, определяющие минимальное и максимальное значения, которые можно установить для спиннера;

- аргумент `font` определяет шрифт для отображения содержимого спиннера (число в поле спиннера);
- аргумент `width` (со значением `3`) задает ширину поля спиннера (в символах);
- аргумент `justify` со значением `"right"` задает способ выравнивания содержимого спиннера по правому краю;
- аргумент `textvariable` определяет «переменную» (в данном случае это `size`), значение которой соответствует значению, установленному в поле спиннера.

НА ЗАМЕТКУ

Хотя аргумент `textvariable` в названии содержит намек на текст, «переменная» `size` «целочисленная» — это объект класса `IntVar`.

Спиннер размещается на панели инструментов с помощью метода `pack()`, а затем панель инструментов размещается в окне с помощью метода `place()`.

Метод `makeFrame()` предназначен для создания вспомогательной панели с меткой, содержащей шаблонный текст. Ссылка на создаваемую при вызове метода панель записывается в поле `frame`. Метка с надписью **Пример текста** создается на основе класса `Label` и сразу добавляется на панель — метод `pack()` вызывается прямо из анонимного объекта метки.

Ссылка на метку для отображения шаблонного текста записывается в поле `lblText`. Значением аргумента `textvariable` указана ссылка на «переменную» `text`, поэтому изменение значения этой «переменной» приведет к автоматическому изменению содержимого метки. На панели метка размещается с помощью метода `pack()`, а панель в окне размещается с помощью метода `place()`.

Для создания контекстного меню использован метод `makePopupMenu()`. В общем и целом здесь много общего с созданием главного меню, но есть свои особенности. Объект контекстного меню создается на основе класса `Menu`, а ссылка на него записывается в поле `popup`. Заполняется контекстное меню последовательным вызовом методов `setMenuFont()`, `setMenuStyle()` и `setMenuColor()`, но на этот раз (в отличие от главного меню) в качестве аргумента

методам передается ссылка popup на объект контекстного меню. Команда **Выход** добавляется в контекстное меню отдельно с помощью метода add_command().

Также мы с помощью метода bind() добавляем в контекстное меню обработчик. Обработчик реагирует на событие <Button-3> (нажатие правой кнопки мыши). Обработчиком указано лямбда-выражение, в котором для аргумента evt (объект события) выполняется команда self.popup.tk_popup(evt.x_root, evt.y_root). Методом tk_popup() отображается контекстное меню, а аргументы метода evt.x_root и evt.y_root определяют координаты для отображения меню. Это координаты того места, в котором находился курсор мыши в момент наступления события. Мы эти координаты вычисляем на основе объекта события evt.

Метод apply() вызывается для того, чтобы по настройкам уже созданных элементов управления вычислить параметры шрифта и отобразить соответствующий текст в окне.

Командой clr=self.color.get() считывается цвет для шрифта. Название шрифта определяем с помощью команды nm=self.name.get(). Для определения размера шрифта использована команда sz=self.size.get().

Для применения цвета к метке задействована инструкция self.lblText.configure(fg=clr). После этого командой fnt=[nm, sz] создается список fnt с параметрами шрифта (название и размер) и начинается «уточнение» параметров шрифта и формирование шаблонного текста. Цвет (русское название) мы получаем как значение элемента словаря colors с ключом clr. Английское название для стиля шрифта (при заданном значении индекса k) дается выражением style[k][0], а русское название получаем как значение инструкции style[k][1], причем с помощью метода lower() мы преобразуем все символы в строчные (маленькие).

После того как шрифт окончательно определен, командой self.lblText.configure(font=fnt) он применяется к метке. Текст для метки задаем с помощью команды self.text.set(txt).

Метод traceVars() вызывается для того, чтобы перейти в режим отслеживания значений переменных name, color, size и переменных из списка bi.

НА ЗАМЕТКУ

Поскольку для всех «переменных» используется один и тот же метод-обработчик `apply()`, то сначала командой `mt=self.apply` в переменную `mt` записывается ссылка на этот метод, а затем переменная указывается вторым аргументом при вызове метода `trace()`.

Наконец, с помощью метода `showMainWindow()` отображается главное окно программы. Этот метод описан просто: в нем выполняется команда `self.wnd.mainloop()`.

Нам осталось проанализировать методы, которые используются для обработки событий, связанных с нажатием кнопок на панели инструментов, а также вызываемых при выборе некоторых команд меню.

В теле метода `clExit()` выполняется всего одна команда `self.wnd.destroy()`, которой закрывается окно приложения (и затем завершается выполнение программы). Метод используется для обработки нажатия на первой кнопке на панели инструментов и для обработки выбора команды **Выход** в главном и контекстном меню.

Метод `clBold()` используется как обработчик для события, связанного с нажатием кнопки применения/отмены жирного стиля. В теле метода командой `self.bi[0].set(not self.bi[0].get())` логической «переменной» `bi[0]` присваивается значение, противоположное к текущему. Поэтому если опция применения жирного стиля была установлена, то она будет отменена, и наоборот. Аналогично описан метод `clItalic()`, но подобная операция проделывается с «переменной» `bi[1]` и, соответственно, речь идет об опции применения жирного стиля.

Метод `clNormal()` вызывается при обработке нажатия кнопки отмены жирного и курсивного стиля. В теле метода командами `self.bi[0].set(False)` и `self.bi[1].set(False)` значения обеих логических «переменных» устанавливаются равными `False`.

Метод `showDialog()` предназначен для отображения модального окна справки. Мы в данном случае воспользовались стандартным методом `showinfo()` из подпакета `messagebox` из пакета `tkinter` (поэтому в заголовке программы есть инструкция `from tkinter.messagebox import *`). Аргументы метода `showinfo()` определяют заголовок модального окна и текст сообщения.

Работа с графикой

Маленькая, а с фантазией.

Из к/ф «Девчата»

Помимо использования стандартных графических компонентов, в Python существует еще и очень неплохая поддержка для использования базовых *графических примитивов* — проще говоря, при создании графического интерфейса можно еще и рисовать. Для этого используется такой графический компонент, как *область рисования*. Объект для области рисования создается на основе класса `Canvas`. У такого объекта есть целый ряд методов, позволяющих рисовать различные геометрические фигуры и задавать и изменять их настройки.

Работу с областью рисования мы проиллюстрируем на примере. Традиционно сначала рассмотрим функциональность приложения, после чего проанализируем соответствующий программный код.

При запуске приложения на выполнение отображается окно, представленное на рис. 11.22.

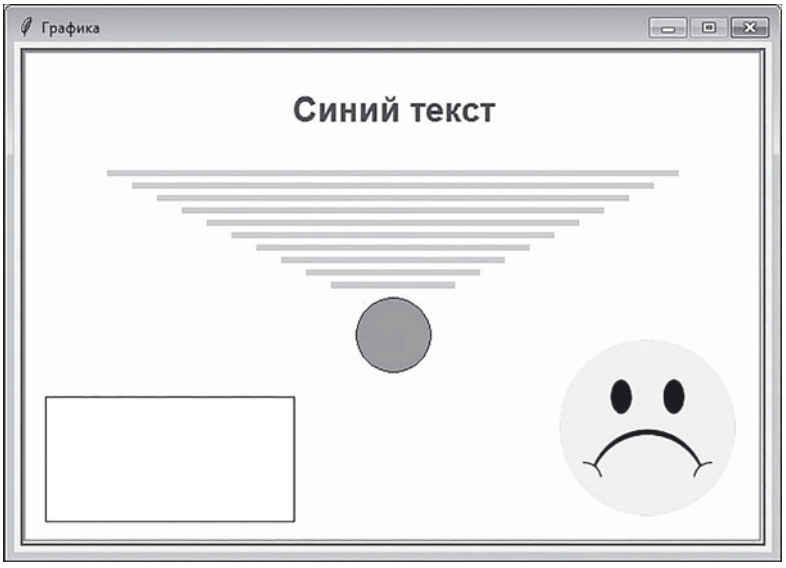


Рис. 11.22. Окно приложения сразу после запуска программы на выполнение

Окно содержит в верхней части надпись **Синий цвет**, которая отображается синим цветом. Под надписью расположена группа горизонтальных линий светло-зеленого цвета. Длина этих линий монотонно уменьшается.

НА ЗАМЕТКУ

Конечно, речь идет об отрезках. Но мы их будем называть линиями.

Под группой линий расположена окружность, закрашенная красным цветом. В левом нижнем углу рабочей области расположен прямоугольник белого цвета. В правом нижнем углу размещено «печальное» изображение. Но если навести курсор мыши на область рисования, «печальное» изображение изменится на «веселое». Более того, через точку, в которой размещен курсор, автоматически проводятся две перпендикулярные красные линии, и при движении курсора эти линии перемещаются вместе с ним. Ситуация проиллюстрирована на рис. 11.23.

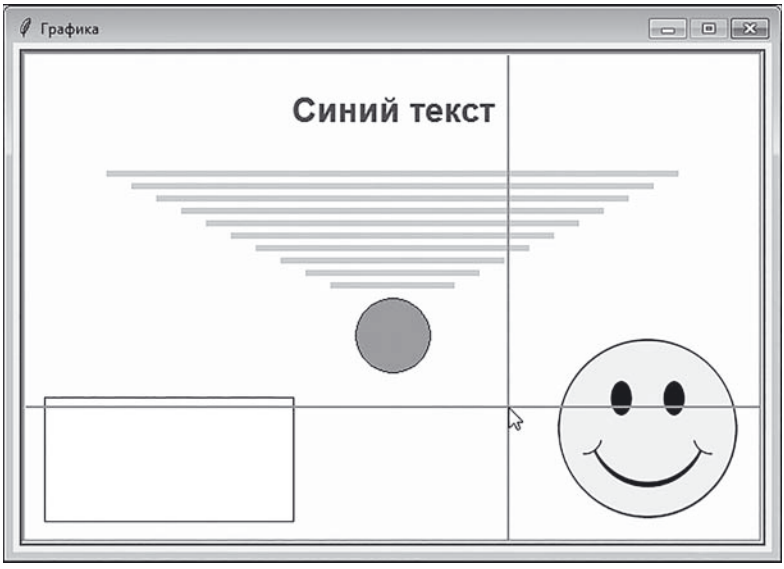


Рис. 11.23. При наведении курсора мыши на область рисования изменяется изображение в окне и отображаются две красные линии

Если курсор мыши навести на область прямоугольника, то он из белого превратится в зеленый, как показано на рис. 11.24.

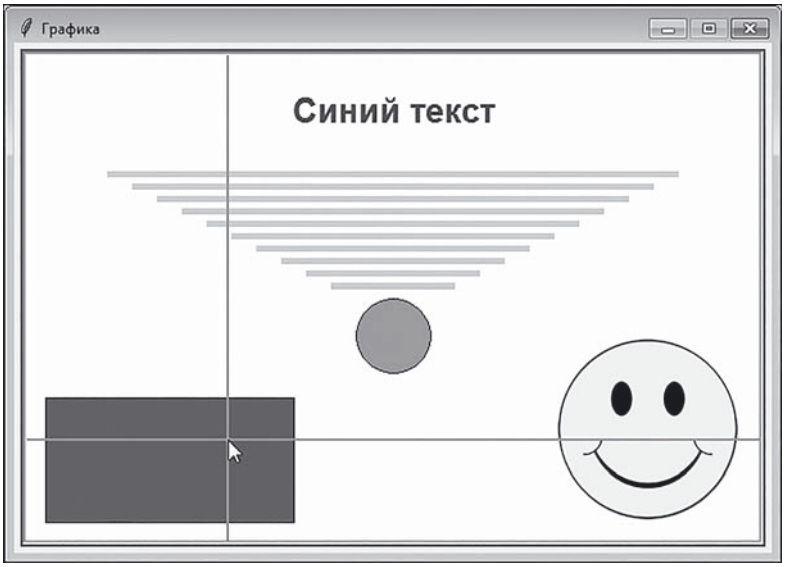


Рис. 11.24. При наведении курсора мыши на прямоугольник он закрашивается зеленым цветом

Если курсор мыши увести из области прямоугольника, прямоугольник снова станет белым. А если курсор окажется над областью окружности, то она станет белой, а прямоугольник — красным. Эта ситуация показана на рис. 11.25.

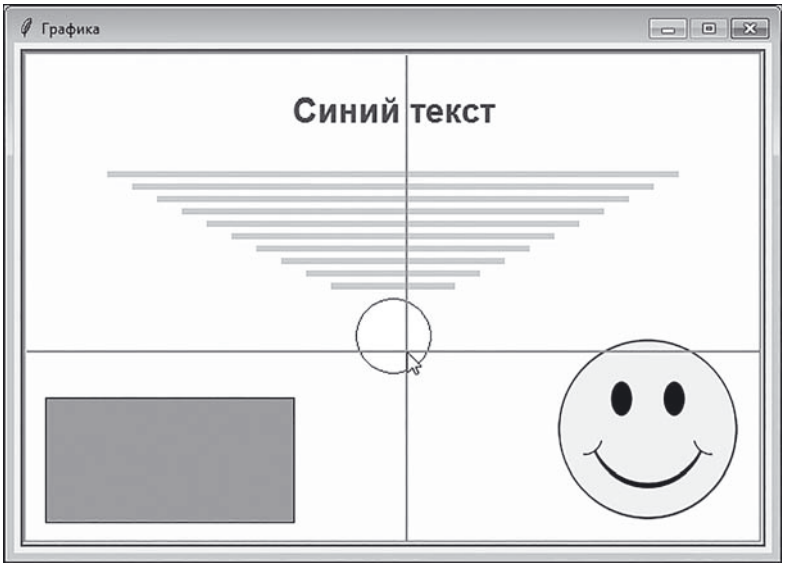


Рис. 11.25. При наведении курсора мыши на окружность она закрашивается белым цветом, а прямоугольник закрашивается красным цветом

По умолчанию фон области рисования светло-желтый. Но если щелкнуть левой кнопкой мыши в области рисования, то фон станет желтым. Чтобы фон снова стал светло-желтым, следует щелкнуть в области рисования правой кнопкой мыши.

Приложение также реагирует на нажатие кнопок со стрелками «вправо», «влево», «вверх» и «вниз». При нажатии кнопок «влево» или «вправо» в соответствующем направлении вдоль горизонтали смещается текст в верхней области окна. На рис. 11.26 показана ситуация, когда цвет фона желтый (был выполнен щелчок левой кнопкой мыши), а текст смещен влево.

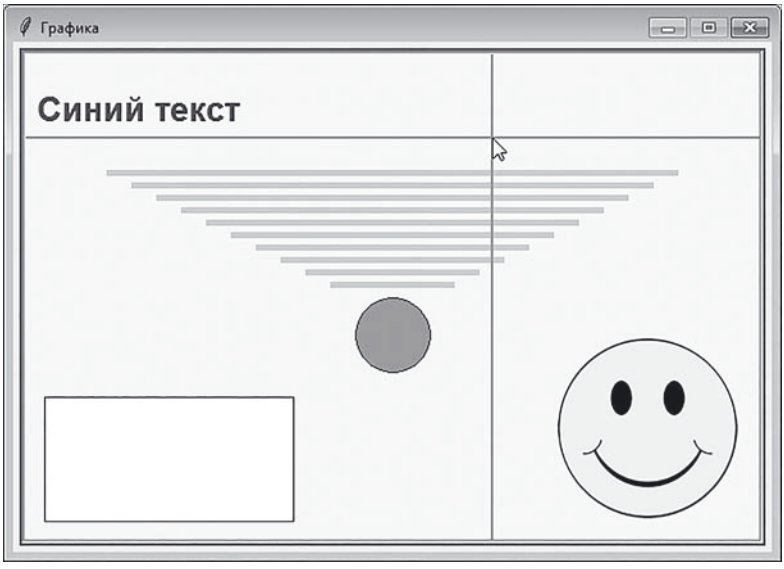


Рис. 11.26. Окно со смещенным влево текстом и желтым цветом фона

Если нажимать кнопку «вниз», то группа линий начинает смещаться вниз, а красная окружность смещается (в три раза быстрее) вверх. На рис. 11.26 показана ситуация, когда текст смещен влево, группа линий смещена вниз, окружность смещена вверх, а в качестве фона используется светло-желтый цвет (была нажата правая кнопка мыши).

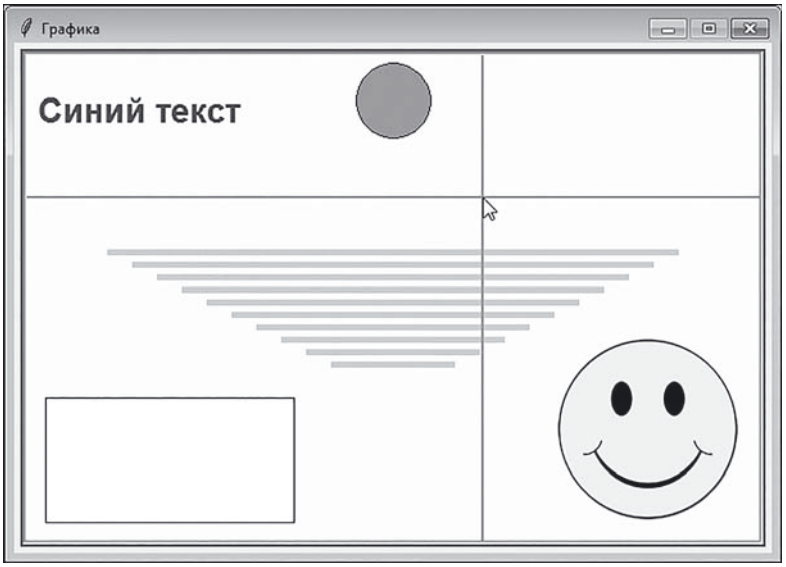


Рис. 11.27. Текст сдвинут влево, группа линий смещена вниз, окружность смещена вверх, курсор находится в области рисования

Если нажимать кнопку «вверх», то группа линий смещается вверх, а окружность смещается вниз. На рис. 11.28 проиллюстрирована ситуация, когда текст смещен вправо, группа линий смещена вверх, окружность смещена вниз, а курсор покинул область рисования.

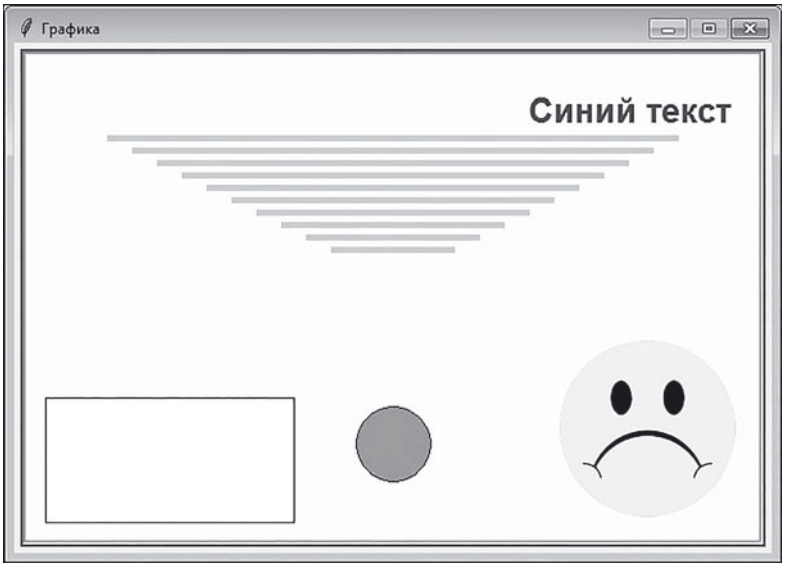


Рис. 11.28. Текст сдвинут вправо, группа линий смещена вверх, окружность смещена вниз, курсор находится вне области рисования

Теперь рассмотрим и проанализируем программный код, представленный в листинге 11.7.

Листинг 11.7. Работа с графикой

```
from tkinter import *
from math import *
# Функция для обработки события, связанного
# с наведением курсора на область рисования:
def msEnter(evt):
    # Смена изображения:
    cnv.itemconfig(Pct, image=img_2)
# Функция для обработки события, связанного
# с тем, что курсор покидает область рисования:
def msLeave(evt):
    # Смена изображения:
    cnv.itemconfig(Pct, image=img_1)
    # Удаление "старых" линий возле курсора:
    cnv.delete("ln")
# Функция для обработки события, связанного
# с перемещением курсора в области рисования:
def msMotion(evt):
    # Координаты курсора в области рисования:
    x=evt.x
    y=evt.y
    # Удаление "старых" линий:
    cnv.delete("ln")
    # Отображение линий:
    cnv.create_line(x,5, x, H-5, fill=clr_C1, width=2, tag="ln")
    cnv.create_line(5, y, W-5, y, fill=clr_C1, width=2, tag="ln")
    # Координаты прямоугольника:
    Rxy=cnv.coords(Rtn)
    # Координаты окружности:
    Cxy=cnv.coords(Crl)
```

```
    # Координаты центра окружности:
    x0=(Cxy[2]+Cxy[0])/2
    y0=(Cxy[3]+Cxy[1])/2
    # Расстояние от курсора до центра окружности:
    r=sqrt((x-x0)**2+(y-y0)**2)
    # Если курсор находится внутри окружности:
    if r<R:
        # Белый цвет заливки для окружности:
        cnv.itemconfig(Crl, fill=clr_C2)
        # Красный цвет заливки для прямоугольника:
        cnv.itemconfig(Rtn, fill=clr_C1)
        # Завершение выполнения функции:
        return
    # Если курсор находится вне окружности:
    else:
        # Красный цвет заливки для окружности:
        cnv.itemconfig(Crl, fill=clr_C1)
    # Если курсор находится внутри прямоугольника:
    if x>Rxy[0] and x<Rxy[2] and y>Rxy[1] and y<Rxy[3]:
        # Зеленый цвет заливки для прямоугольника:
        cnv.itemconfig(Rtn, fill=clr_R2)
    # Если курсор находится вне прямоугольника:
    else:
        # Белый цвет заливки для прямоугольника:
        cnv.itemconfig(Rtn, fill=clr_R1)
# Функция для обработки события, связанного
# с нажатием кнопки "вверх":
def msUp(evt):
    # Группа линий смещается на один пиксель вверх:
    cnv.move("Lns",0,-1)
    # Окружность смещается на три пикселя вниз:
    cnv.move(Crl,0,3)
```

```
# Функция для обработки события, связанного
# с нажатием кнопки "вниз":
def msDown(evt):
    # Группа линий смещается на один пиксель вверх:
    cnv.move("Lns",0,1)
    # Окружность смещается на три пикселя вверх:
    cnv.move(Crl,0,-3)
# Функция для обработки события, связанного
# с нажатием кнопки "влево":
def msLeft(evt):
    # Текст смещается на одну позицию влево:
    cnv.move(Txt,-1,0)
# Функция для обработки события, связанного
# с нажатием кнопки "вправо":
def msRight(evt):
    # Текст смещается на одну позицию вправо:
    cnv.move(Txt,1,0)
# Ширина и высота области рисования:
W=600
H=400
# Ширина и высота прямоугольника:
w=200
h=100
# Количество линий:
N=10
# Декремент для длины линий:
d=20
# Радиус окружности:
R=30
# Шрифт:
fnt=("Arial",20,"bold")
# Текст:
```

```
txt="Синий текст"
# Цвет для линий:
clr="lightgreen"
# Цвет для фона области рисования:
clr_1="lightyellow"
clr_2="yellow"
# Цвет для закраски окружности:
clr_C1="red"
clr_C2="white"
# Цвет для закраски прямоугольника:
clr_R1="white"
clr_R2="green"
# Создание объекта окна:
wnd=Tk()
# Размеры и положение окна:
wnd.geometry(str(W+10)+"x"+str(H+10)+"+400+300")
# Заголовок для окна:
wnd.title("Трафика")
# Окно постоянных размеров:
wnd.resizable(False, False)
# Создание объекта для области рисования:
cnv=Canvas(wnd, bg=clr_1, bd=3, relief=GROOVE)
# Размещение области рисования в окне:
cnv.place(x=5, y=5, width=W, height=H)
# Передача фокуса области рисования:
cnv.focus_set()
# Создание текстового элемента:
Txt=cnv.create_text(W/2,50, text=txt, font=fnt, fill="blue")
# Создание группы горизонтальных линий:
for k in range(N):
    # Координаты линий:
    x1=70+k*d
    y1=H/4+10*k
    x2=W-70-d*k
```

```
    y2=H/4+10*k
    cnv.create_line(x1, y1, x2, y2, fill=clr, width=5, tag="Lns")
# Координаты для создания окружности:
x1=W/2-R
y1=H/2-R+30
x2=W/2+R
y2=H/2+R+30
# Создание окружности:
Crl=cnv.create_oval(x1, y1, x2, y2, fill=clr_C1)
# Координаты для создания прямоугольника:
x1=20
y1=H-h-20
x2=x1+w
y2=y1+h
# Создание прямоугольника:
Rtn=cnv.create_rectangle(x1, y1, x2, y2, fill=clr_R1)
# Создание объектов изображений:
img_1=PhotoImage(file="D:\\Pictures\\sad.png")
img_2=PhotoImage(file="D:\\Pictures\\smile.png")
# Создание объекта изображения:
Pct=cnv.create_image(W-20, H-20, anchor=SE, image=img_1)
# Регистрация обработчиков:
cnv.bind("<Left>", msLeft)
cnv.bind("<Right>", msRight)
cnv.bind("<Up>", msUp)
cnv.bind("<Down>", msDown)
cnv.bind("<Enter>", msEnter)
cnv.bind("<Leave>", msLeave)
cnv.bind("<Motion>", msMotion)
cnv.bind("<Button-1>", lambda evt: cnv.config(bg=clr_2))
cnv.bind("<Button-3>", lambda evt: cnv.config(bg=clr_1))
# Отображение главного окна:
wnd.mainloop()
```

Удобства ради в программе создается несколько переменных (ширина и высота области рисования W и H, ширина и высота прямоугольника w и h, количество линий в группе линий N, декремент для длины линий d, радиус окружности R, шрифт fnt, текст для отображения в области рисования txt, а также переменные clr, clr_1, clr_2, clr_C1, clr_C2, clr_R1 и clr_R2, определяющие цвет).

После создания объекта окна wnd задаются его размеры и положение командой wnd.geometry(str(W+10)+"x"+str(H+10)+"+400+300"). В текстовом аргументе метода geometry() два последних числа (400 и 300), указанные через символ "+", определяют положение окна в области экрана: это координаты левого верхнего угла окна по отношению к левому верхнему углу экрана.

Объект cnv для области рисования создается командой cnv=Canvas(wnd, bg=clr_1, bd=3, relief=GROOVE). Аргументы конструктора определяют принадлежность области рисования окну wnd, цвет фона (аргумент bg), толщину рамки вокруг области (аргумент bd) и тип рамки (аргумент relief). Для размещения области рисования в окне использован метод place(). Командой cnv.focus_set() области рисования передается фокус.

Для создания текстового элемента (это фактически «нарисованная» надпись) Txt использован метод create_text(). Первые два аргумента метода определяют место размещения текстовой надписи. Собственно текст задается аргументом text, шрифт определяется аргументом font, а цвет текста задается с помощью аргумента fill.

Для создания группы горизонтальных линий использован оператор цикла. Каждая отдельная линия создается с помощью метода create_line(). Первые четыре аргумента определяют координаты начальной и конечной точек, которые соединяются линией. С помощью аргумента fill определяется цвет линий, а их толщина определяется значением аргумента width. Также мы используем аргумент tag, который задает дескриптор (или тег), по которому затем можно будет получать доступ к линиям. Поскольку для всех линий указан один и тот же дескриптор "Lns", то с помощью этого дескриптора мы впоследствии сможем управлять группой линий как одним целым.

Для создания окружности Crl использован метод create_oval(). Вообще метод предназначен для рисования овалов. В качестве аргументов методу передаются координаты левого верхнего и правого нижнего

углов воображаемого прямоугольника, в который вписывается овал. Если прямоугольник является квадратом, то получим окружность. Мы, в данном случае, при вызове метода `create_oval()`, кроме координат указываем еще и значение для аргумента `fill=clr_C1`. Это цвет, которым закрашивается окружность.

Аналогично создается прямоугольник `Rtn`, но на этот раз использован метод `create_rectangle()`.

Также в программе есть пример отображения в области рисования изображения из файла. Для этого мы на основе класса `PhotoImage` и файлов с изображениями создаем два объекта `img_1` и `img_2`. Объект изображения `Pct` создается с помощью метода `create_image()`. Первые два аргумента определяют координаты точки в области рисования, в которой размещается изображение. Аргумент `anchor` определяет способ «привязки» изображения к точке, заданной координатами. Значение `SE` означает, что в соответствующей точке будет размещен правый нижний угол области изображения.

НА ЗАМЕТКУ

Значение `SE` получается от англоязычных названий сторон света — в данном случае это South East (юго-восток, что соответствует правому нижнему углу). Значение `NW`, например, означает North West (северо-запад) — левый верхний угол.

Значением аргумента `image` указывается ссылка на отображаемую картинку.

С помощью метода `bind()` для разных событий, связанных с областью рисования `cnv`, регистрируются обработчики: в основном это функции, но есть и два лямбда-выражения. Вся «динамика» связана с этими обработчиками.

НА ЗАМЕТКУ

У всех функций один аргумент (называется `evt`), который отождествляется с объектом обрабатываемого события.

Функция `msEnter()` используется для обработки события `<Enter>`, связанного с наведением курсора на область рисования. В теле функции командой `cnv.itemconfig(Pct, image=img_2)` выполняется смена изображения, которое рисуется в правом нижнем углу области рисования.

ПОДРОБНОСТИ

В программе используется метод `itemconfig()`. Метод вызывается из объекта `cnv`, а первым аргументом методу передается ссылка на объект, для которого выполняется настройка параметров. Далее указываются значения опций, которые применяются для объекта, указанного в качестве первого аргумента.

Функция `msUp()` используется как обработчик события `<Up>`, связанного с нажатием кнопки «вверх». В теле функции командой `cnv.move("Lns",0,-1)` группа линий смещается на один пиксель вверх. Первым аргументом метода `move()` указывается дескриптор `"Lns"`, общий для всех линий в группе, поэтому операция (перемещения) выполняется для каждой линии в группе. Значения `0` и `-1` означают, что изображение смещается по горизонтали на `0` пикселей и по вертикали на `-1` пиксель. Начало координатной системы области рисования находится в левом верхнем углу этой области. Координатные оси направлены слева направо (горизонтальная) и сверху вниз (вертикальная). Поэтому значение `-1` для смещения по вертикали означает, что изображение смещается на `1` пиксель вверх (в противоположном направлении к направлению вертикальной координатной оси). А командой `cnv.move(Crl,0,3)` окружность смещается по горизонтали на `0` пикселей и на `3` пикселя (вниз) вдоль вертикали.

Функция `msDown()` нужна для обработки события `<Down>`, связанного с нажатием кнопки «вниз». В теле функции выполняются команды `cnv.move("Lns",0,1)` и `cnv.move(Crl,0,-3)`, в результате чего группа линий смещается на `1` пиксель вниз, а окружность смещается на `3` пикселя вверх.

Для обработки события `<Left>` (нажатие кнопки «влево») используется функция `msLeft()`. В ней командой `cnv.move(Txt,-1,0)` текст смещается на `1` пиксель влево. В функции `msRight()` (используется для обработки события `<Right>`, связанного с нажатием кнопки «вправо») командой `cnv.move(Txt,1,0)` текст смещается на один пиксель вправо.

Самая «объемная» функция `msMotion()` регистрируется как обработчик события `<Motion>`, связанного с перемещением курсора в области рисования. Сначала командами `x=evt.x` и `y=evt.y` на основе объекта события `evt` вычисляются координаты курсора (на момент возникновения события) в области рисования. Далее командой `cnv.`

delete("ln") удаляются «старые» красные линии, которые были нарисованы при предыдущем вызове функции. А затем командами cnv.create_line(x,5, x, H-5, fill=clr_C1, width=2, tag="ln") и cnv.create_line(5, y, W-5, y, fill=clr_C1, width=2, tag="ln") рисуются две новые линии, которые проходят через точку, в которой находится курсор.

ПОДРОБНОСТИ

При вызове функции рисуются две линии. Функция вызывается каждый раз, когда курсор меняет свою позицию в области рисования. При этом старые линии автоматически не удаляются. Работу по удалению этих уже не нужных линий следует возложить на функцию. Обе линии имеют один и тот же дескриптор "ln". Это позволяет удалить сразу две линии, указав дескриптор в качестве аргумента метода delete().

На следующем этапе мы решаем еще несколько вспомогательных задач. Так, при выполнении команды Rxy=cnv.coords(Rtn) в переменную Rxy записывается ссылка на список, содержащий координаты прямоугольника Rtn. Это координаты левого верхнего и правого нижнего углов прямоугольника. Аналогичные последствия выполнения команды Cxy=cnv.coords(Crl): в переменную Cxy записывается ссылка на список с координатами воображаемого квадрата, в который вписана окружность Crl. Затем с помощью команд x0=(Cxy[2]+Cxy[0])/2 и y0=(Cxy[3]+Cxy[1])/2 мы вычисляем координаты точки, являющиеся центром окружности (эта же точка — центр квадрата, в который вписана окружность). Расстояние от точки, в которой находится курсор, до центра окружности вычисляется командой r=sqrt((x-x0)**2+(y-y0)**2).

ПОДРОБНОСТИ

Для вычисления квадратного корня использована математическая функция sqrt() из модуля math. Для возможности использования функции в заголовке программы размещена инструкция from math import *.

Затем в дело вступает условный оператор, в котором проверяется условие r<R (курсор находится в области окружности). Если так, то командой cnv.itemconfig(Crl, fill=clr_C2) для окружности приме-

няется белый цвет заливки, а командой `cnv.itemconfig(Rtn, fill=clr_C1)` красный цвет заливки применяется для прямоугольника, после чего с помощью инструкции `return` завершается выполнение функции (поскольку нет необходимости проверять, находится ли курсор над областью прямоугольника).

Если же курсор находится вне окружности, то командой `cnv.itemconfig(Crl, fill=clr_C1)` для окружности применяется красный цвет заливки. После этого с неизбежностью выполняется еще один условный оператор. В нем использовано условие `x>Rxy[0] and x<Rxy[2] and y>Rxy[1] and y<Rxy[3]`. Его истинность означает, что координаты курсора попадают внутрь области прямоугольника. В этом случае с помощью команды `cnv.itemconfig(Rtn, fill=clr_R2)` для прямоугольника применяется зеленый цвет заливки. А если курсор не находится в области прямоугольника, то с помощью команды `cnv.itemconfig(Rtn, fill=clr_R1)` для прямоугольника применяется белый цвет заливки.

Для обработки события `<Leave>` (курсор покидает область рисования) мы используем функцию `msLeave()`. В теле функции всего две команды. Командой `cnv.itemconfig(Pct, image=img_1)` выполняется смена изображения в правом нижнем углу области рисования. Командой `cnv.delete("ln")` удаляются линии, объединенные дескриптором `"ln"` (то есть линии, которые «остались» от курсора, который покинул область рисования).

> **НА ЗАМЕТКУ**
>
> Дело в том, что, когда курсор покидает область рисования, на ней может остаться (и скорее всего, останется) как минимум одна из красных линий, которые «сопровождали» курсор. Поэтому эта линия (или линии) удаляются.

События `<Button-1>` возникает, когда пользователь нажимает левую кнопку мыши в области рисования (в данном случае). Обработчиком события указано лямбда-выражение. В нем выполняется команда `cnv.config(bg=clr_2)`, которой задается цвет фона для области рисования. Аналогично обрабатывается событий `<Button-3>` (нажатие правой кнопки мыши в области рисования). Разница лишь в том, какой цвет применяется для фона области рисования.

Резюме

— Интурист хорошо говорит.
— А что он говорит, конкретно что?

Из к/ф «Иван Васильевич меняет профессию»

- Создание программ с графическим интерфейсом подразумевает создание объектов для графических компонентов, их компоновку в контейнере (окне или панели), а также обработку событий.
- Объект главного окна создается на основе класса `Tk`. Также часто используются панели (объекты класса `Frame`), кнопки (объекты класса `Button`), метки (объекты класса `Label`), поля ввода (объекты класса `Entry`), списки (объекты класса `Listbox`), опции (объекты класса `Checkbutton`), переключатели (объекты класса `Radiobutton`) и многие другие. Помимо этого для приложений можно создавать меню (используется класс `Menu`), в том числе и контекстное.
- Для настройки параметров компонентов используются специальные методы, а также задаются значения соответствующих аргументов. Например, для размещения компонентов в контейнерах используются методы `place()`, `pack()` и `grid()`. Для настройки параметров компонентов используется метод `configure()`.
- Для связывания методов и функций, предназначенных для обработки событий, с компонентами используют метод `bind()` или задают значение аргумента `command`. Также можно использовать объекты специальных классов `StringVar`, `BooleanVar`, `IntVar` и `DoubleVar`.
- Для работы с графикой на основе класса `Canvas` создается объект для области рисования. Этот объект содержит методы для создания различных геометрических фигур, таких, например, как линии, прямоугольники, окружности и ряд других. Для таких графических изображений можно выполнять обработку событий.

Задания для самостоятельной работы

Я не бездействовал. Я сразу на каппу нажал. Скрипач свидетель.

Из к/ф «Кин-дза-дза»

1. Напишите программу, в которой отображается окно с кнопкой, изображением и текстовой меткой. При нажатии кнопки окно закрывается.

2. Напишите программу, в которой отображается окно с кнопкой и изображением. При нажатии кнопки окно закрывается. При наведении курсора на область изображения оно меняется на другое. Когда курсор покидает область изображения, оно меняется на исходное.

3. Напишите программу, в которой отображается окно с полем ввода, в которое следует ввести возраст пользователя. Программа должна по возрасту (и текущему году) вычислить год рождения, который отображается в следующем диалоговом окне.

4. Напишите программу, в которой отображается окно с двумя кнопками, полем ввода и текстовой меткой. При вводе текста в текстовое поле он автоматически дублируется в текстовой метке. Нажатие одной из кнопок приводит к очистке поля и метки. Нажатие другой кнопки приводит к закрытию окна.

5. Напишите программу, в которой отображается окно с шаблонным текстом. Окно содержит две кнопки. При нажатии одной кнопки размер шрифта (которым отображается шаблонный текст) увеличивается на единицу. При нажатии другой кнопки размер шрифта уменьшается на единицу.

6. Напишите программу, в которой есть статический список с названиями животных. При выборе пункта в статическом списке в окне отображается изображение выбранного животного.

7. Напишите программу, в которой есть группа переключателей, предназначенная для выбора цвета. В окне отображается область, закрашенная цветом, выбранным в группе переключателей.

8. Напишите программу, в которой с помощью главного меню можно выбирать название животного, и картинка с этим животным отображается в окне.

9. Напишите программу, в которой открывается окно с полем ввода и меткой. В поле вводится выражение (в соответствии с правилами языка Python — например, алгебраическое выражение), а в метке отображается значение этого выражения. Используйте функцию `eval()` и обработку исключительных ситуаций.

10. Напишите программу, в которой отображается окно с областью для рисования. В центре области находится окружность. При нажатии кнопок «вверх», «вниз», «влево» или «вправо» окружность начинает двигаться к одному из углов области рисования.

Заключение
PYTHON И ПРОГРАММИРОВАНИЕ

> Ты не уйдешь от славы, ты еще услышишь фанфары.
>
> *Из к/ф «Гостья из будущего»*

Как отмечалось в самом начале книги, язык Python на сегодня достаточно популярен, и его востребованность постоянно растет. Поэтому изучение Python в определенном смысле является удачной инвестицией в себя — такие инвестиции, как известно, являются наиболее эффективными. Но важно понимать, что прочтением книги эта инвестиция не заканчивается. Причин много. В первую очередь, язык Python очень динамичный, он постоянно развивается. Это естественно, поскольку язык относительно «молодой», а каждый язык во время своего становления проходит очень бурные этапы. С другой стороны, программные технологии тоже не стоят на месте, и эффективный язык должен им соответствовать. То есть данный процесс объективный и закономерный. И с ним нужно идти в ногу. Мы сделали только первые шаги на этом долгом, но очень интересном пути.

В книге рассмотрены все (или почти все) базовые методы, подходы и конструкции, важные для понимания принципов использования языка Python на практике. Но список рассмотренных тем не исчерпывающий — он не может быть исчерпывающим в принципе. Но он показательный. И отталкиваясь от этого списка можно выстраивать свои «взаимоотношения» с языком Python и сопутствующими технологиями.

Отдельно стоит отметить вот какой факт. Все, кто профессионально занимается программированием, как правило, редко ограничиваются лишь одним языком программирования. Конечно, многое зависит от мотивации. Но если человек решил программированием заниматься профессионально, то практически наверняка ему придется иметь дело с разными языками программирования. Даже в этом случае знание языка Python будет полезным, поскольку владение навыками

программирования в таком красивом и гибком языке как значительно расширяет общие горизонты, так и способствует пониманию механизмов, заложенных в любом другом языке.

В завершение хочется выразить скромную надежду, что книга стала полезной для читателя, и поблагодарить за интерес к ней.

Научно-популярное издание

РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Васильев Алексей

ПРОГРАММИРОВАНИЕ НА PYTHON В ПРИМЕРАХ И ЗАДАЧАХ

Главный редактор *Р. Фасхутдинов*
Руководитель направления *В. Обручев*
Ответственный редактор *Е. Горанская*
Литературный редактор *С. Ульянов*
Младший редактор *Ю. Клюшина*
Художественный редактор *В. Брагина*
Компьютерная верстка *Э. Брегис*
Корректоры *Р. Болдинова, А. Баскакова*

Страна происхождения: Российская Федерация
Шығарылған елі: Ресей Федерациясы

ООО «Издательство «Эксмо»
123308, Россия, город Москва, улица Зорге, дом 1, строение 1, этаж 20, каб. 2013.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндіруші: «ЭКСМО» АҚБ Баспасы,
123308, Ресей, қала Мәскеу, Зорге көшесі, 1 үй, 1 ғимарат, 20 қабат, офис 2013 ж.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Интернет-магазин : www.book24.ru

Интернет-магазин : www.book24.kz
Интернет-дукен : www.book24.kz
Импортёр в Республику Казахстан ТОО «РДЦ-Алматы».
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС.
Дистрибьютор и представитель по приему претензий на продукцию,
в Республике Казахстан: ТОО «РДЦ-Алматы»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды
қабылдаушының өкілі «РДЦ-Алматы» ЖШС,
Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»
www.eksmo.ru/certification
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Дата изготовления / Подписано в печать 05.11.2020. Формат 70x100$^{1}/_{16}$.
Печать офсетная. Усл. печ. л. 49,91.
Тираж экз. Заказ

12+

ISBN 978-5-04-103199-2

ЛУЧШИЕ КНИГИ О БИЗНЕСЕ С ЛОГОТИПОМ ВАШЕЙ КОМПАНИИ? ЛЕГКО!

Удивить своих клиентов, бизнес-партнеров, сделать памятный подарок сотрудникам и рассказать о своей компании читателям бизнес-литературы? Приглашаем стать партнерами выпуска актуальных и популярных книг. О вашей компании узнает наиболее активная аудитория.

ПАРТНЕРСКИЕ ОПЦИИ:

- Специальный тираж уже существующих книг с логотипом вашей компании.
- Размещение логотипа на супер-обложке для малых тиражей (от 30 штук).
- Поддержка выхода новинки, которая ранее не была доступна читателям (50 книг в подарок).

ПАРТНЕРСКИЕ ВОЗМОЖНОСТИ:

- Рекламная полоса о вашей компании внутри книги.
- Вступительное слово в книге от первых лиц компании-партнера.
- Обращение первых лиц на суперобложке.
- Отзыв на обороте обложки вложение информационных материалов о вашей компании (закладки, листовки, мини-буклеты).

У вас есть возможность обсудить свои пожелания с менеджерами корпоративных продаж. Как?

Звоните:
+7 495 411 68 59, доб. 2261

Заходите на сайт:
eksmo.ru/b2b

Популярность языка программирования Python постоянно растет, поэтому его изучение является удачной инвестицией в себя. Такие инвестиции, как известно, всегда наиболее эффективны.

В книге рассмотрены базовые методы, подходы и конструкции, важные для понимания принципов использования языка Python на практике. Материал от главы к главе постепенно усложняется. Некоторые важные моменты достаточно часто повторяются (в разном контексте), особенно в начальных главах. Иногда похожие задачи решаются разными методами. Все это сделано намеренно. Цель простая — облегчить процесс усвоения информации и сформировать основы для понимания принципов программирования в Python.

Дополнительные материалы можно скачать по адресу:
http://addons.eksmo.ru/it/Python_Vasiliev_examples.zip

Самое главное:

- Все о языке Python — от базовых знаний до сложных программ
- Подробный разбор каждой главы с примерами и выводами
- Все примеры актуальные и могут применяться в работе
- Доступный язык изложения, понятный новичкам
- Использована методика обучения, многократно проверенная на практике

Об авторе

Алексей Николаевич Васильев — доктор физико-математических наук, профессор кафедры теоретической физики физического факультета Киевского национального университета имени Тараса Шевченко. Автор книг по программированию на языках C++, Java, JavaScript, C#, Python и математическому моделированию.

«Мы собирались писать собственный путеводитель по Python, но, обнаружив замечательную книгу Алексея Васильева, передали по рациям кодовое слово отмены операции. Не тратьте время на дилетантов с YouTube и состояния на дорогие онлайн-курсы. Если нет проблем с самодисциплиной, то эта книга — все, что вам нужно, чтобы начать уверенно писать на Python».

Кирилл Жвалов,
соосновательMoscow Coding School

ISBN 978-5-04-103199-2