Hello World!

Программирование для детей и взрослых

Научись разговаривать с компьютером на его языке! Чтобы создать свою игру, начать собственное дело или решить важную проблему при помощи компьютера, для начала нужно научиться программировать.

«Hello World! Программирование для детей и взрослых» – это отлично иллюстрированное введение в программирование.
На примере Python авторы показывают, что учить компьютерные языки легко и интересно.
Эта книга привносит в твою жизнь понятия «алгоритмы», «циклы», «ввод и вывод», «графика» и многое другое.
Написанные отцом и сыном, все три издания бестселлера высоко оценили как дети, так и профессиональные педагоги.

Что внутри:

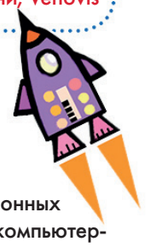
- всё объясняется простым языком – никаких «ботанизмов»;
- картинки и забавные примеры;
- закрепление пройденного: вопросы и упражнения;
- работа с Python 3.

Эта книга прекрасно подойдет новичкам – как детям, так и взрослым!
Если ты умеешь открывать приложения и сохранять файлы, то прекрасно со всем справишься!

«Очень действенный подход, обучающий детей программированию с помощью Python».
Бен Макнамара, DataGeek

«Настоятельно рекомендую... изучайте программирование в простом и интересном формате».
Боб Даст, Рейнольдс Коммьюнити Колледж

«Невероятно легкий способ изучения программирования как для родителей, так и для детей».
Эли Хини, Venovis



Уоррен Сэнд – инженер электронных систем. Python – его любимый компьютерный язык, который он использует на работе, а также обучает ему других. Картер Сэнд начал заниматься программированием в возрасте шести лет. Он помогал отцу в написании первой книги (на тот момент ему было всего девять лет). Сейчас он профессиональный разработчик программного обеспечения. В свободное время он создает игры для ретроконсолей, таких как Game Boy Advance, и любит читать и писать компьютерную фантастику.

Иллюстрации Мартина Муртонена.

ISBN 978-5-97060-881-4

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@alians-kniga.ru

MANNING

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

Hello World! Программирование для детей и взрослых

Программирование

для детей и взрослых

Уоррен и Картер Сэнд

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Hello World!

Computer Programming for Kids and Other Beginners

WARREN SANDE
CARTER SANDE

MANNING
Shelter Island

Hello World!

Компьютерное программирование для детей и начинающих

УОРРЕН СЭНД
КАРТЕР СЭНД

ДМК Пресс
издательство
Москва, 2021

УДК 004.438Python
ББК 32.973.22
С97

Сэнд У., Сэнд К.
С97 Hello World! / пер. с англ. М. А. Райтмана. – М.: ДМК Пресс, 2021. – 486 с.: ил.

ISBN 978-5-97060-881-4

Приступив к изучению этого занимательного руководства, каждый, даже первоклассник, сможет написать на Python – одном из самых простых в освоении языков программирования – свое приветствие миру: Hello World!

Ознакомив читателей с элементарными операциями, авторы постепенно перейдут к более сложным темам: работе с графикой и со звуком, компьютерному моделированию и созданию игр. Материал излагается в доступной, удобной для понимания манере; забавные иллюстрации и подробный разбор примеров кода делают обучение легким и нескучным.

В конце каждой главы приведены упражнения по написанию простых программ.

Книга предназначена для детей, которые могут осваивать материал самостоятельно либо под руководством учителей и родителей, а также для всех, кто хочет освоить программирование с нуля.

УДК 004.438Python
ББК 32.973.22

ISBN 978-1-617297-02-1 (англ.)
ISBN 978-5-97060-881-4 (рус.)

Содержание

Отзывы о первом издании книги

Отличная книга, подойдет как для маленьких, так и для больших деток.

– Гордон Колкхаун,
компьютерный консультант в компании Avalon Consulting Services

Python для подрастающих.

– Доктор Джон Грейсон, автор программ на Python и Tkinter

Книга, с которой интересно учиться!

– Доктор Андре Роберж, президент Университета Святой Анны

Авторы создали прекрасную книгу по программированию для увлекательного обучения.

– Брайан Вайнгартен, архитектор программного обеспечения

Я очень рекомендую эту книгу!

– Хорст Йенс, преподаватель языка Python
и автор книги «*Programming While Playing*»

Язык Python прекрасно подходит для новичков. Приятно видеть ориентированную на ребенка книгу о Python!

– Джеффри Элкнер, педагог

Если вы можете научить своего ребенка чему-то одному, научите его золотому правилу нравственности. Если вы можете научить ребенка двум вещам, научите его золотому правилу и компьютерному программированию. Для обучения программированию этой книги будет более чем достаточно.

– Джош Кронмайер,
старший консультант по программному обеспечению
в компании Thoughtworks

Мне очень понравилось «общение» с Картером в этой книге… А мои ученики будут в восторге от виртуального питомца! Он напоминает мне тамагочи, который был у меня много лет назад.

– Кари Дж. Стеллпфлаг,
преподаватель государственной школы Рочестера, Миннесота

Компьютерное программирование – это мощный инструмент для обучения детей… Дети, которые занимаются программированием, переносят полученные навыки и в другие сферы.

– Николас Негропонте, участник проекта «One Laptop Per Child»

Отзывы о втором издании книги

Самое простое обучение программированию!

– Шон Стебнер, инженер сетевого оборудования в компании Intel Corp

Прочитав эту книгу, вы поймете, что программировать так же легко, как пожарить яичницу.

– Элизабет Гордон, ученица 10-го класса средней школы Игл Харбор

Отличное введение в мир Python для всех. Эта книга очень увлекательна!

– Мейсон Дженкинс, ученик 7-го класса Академии Майрона Б. Томпсона

Книга для людей от 8 до 88 лет. Она не только преподносит программирование на Python в увлекательной форме, но и закладывает хорошую базу, которая может быть использована при работе и с другими языками программирования. Подходит для всех, кто хочет научиться программировать, независимо от возраста.

– Бен Омс, инженер-программист в компании Sogeti

Если вы хотите научиться программированию или научить этому ребенка, эта книга – прекрасный выбор.

– Cuberick.com

Очень хорошее введение в программирование для любого, кто хочет развить этот жизненно важный и очень интересный навык.

– Сью Джи, I-Programmer

Уоррен и Картер начинают с самых азов и проходят с детьми или взрослыми весь путь к созданию забавных 2D-игр и симуляторов. Я бы порекомендовал начинающим программистам изучить в первую очередь Python, а эта книга будет прекрасным помощником. Я рекомендую ее своим студентам с момента выхода первого издания.

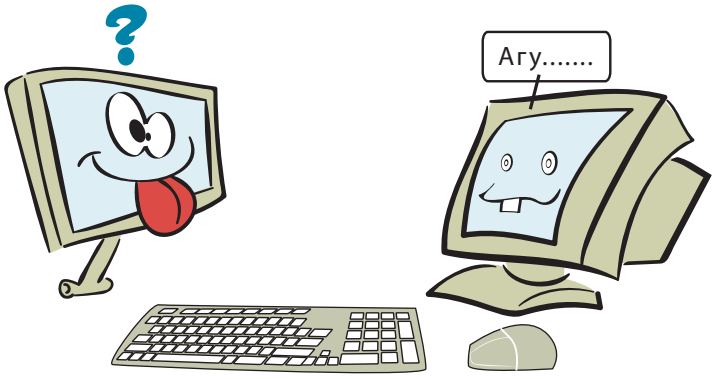
– Дэйв Брикетти,
преподаватель и разработчик программного обеспечения
в компании Dave Briccetti Software LLC

Предисловие

Предисловие – эта та часть в начале книги, которую мы пропускаем, чтобы добраться, наконец, до нужной информации, правильно? Конечно, ты можешь его пропустить, но кто знает, что важного ты не прочитаешь... Оно не очень длинное, поэтому, может быть, все-таки заглянешь?

Что такое программирование?

Если очень кратко, *программирование* означает приказ компьютеру выполнить какое-либо действие. Компьютеры – бестолковые машины. Они не знают, как делать что-либо. Тебе приходится отдавать им команды и учитывать все детали.



Но если ты дашь им верные *инструкции*, они могут сделать много прекрасных и удивительных вещей.

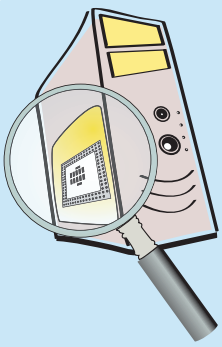
СЛОВАРИК

Инструкция – базовая команда, которую ты даешь компьютеру, обычно для выполнения одного определенного действия.

Компьютерная программа состоит из множества инструкций. Компьютеры выполняют все эти классные вещи сегодня потому, что много умных программистов написали программы, *программное обеспечение*, которое и руководит ими. Программное обеспечение – это программа или набор программ, которые запуска-

ются на твоем или на другом компьютере, к которому ты подключен, например на сервере.

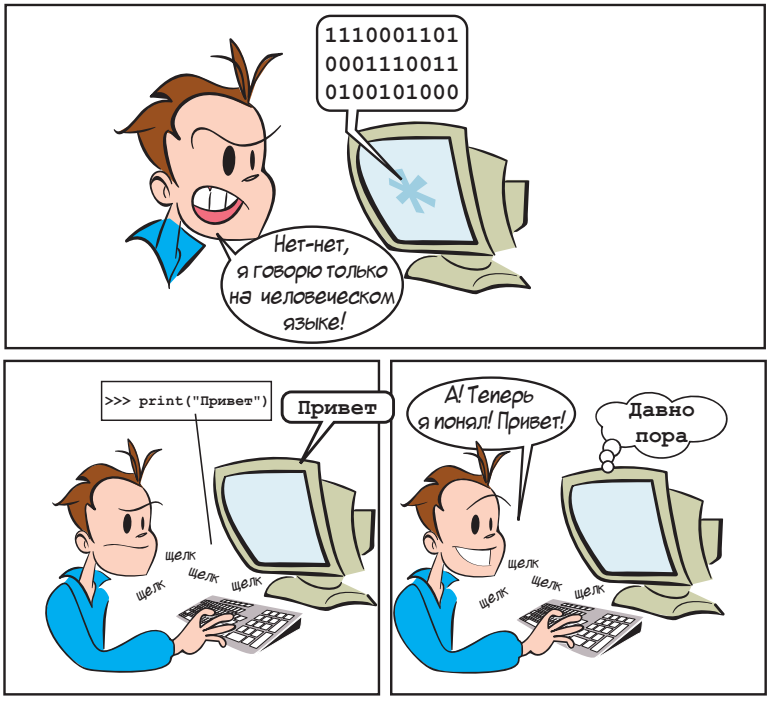
ЧТО ТАМ ПРОИСХОДИТ?



Компьютеры «думают» при помощи многих, многих и МНОГИХ электрических цепей. На самом базовом уровне эти цепи имеют значение ДА или НЕТ. Инженеры и компьютерные специалисты используют значения 0 и 1 вместо ДА и НЕТ. Все эти единицы и нули представляют собой некий код, называемый *двоичным*. Двоичный означает «два состояния». Два состояния – это ДА и НЕТ, или 0 и 1.
Запомни: двоичная цифра = бит.

Python – язык для нас и компьютера

Все компьютеры используют двоичный код. Но большинство людей владеет им не очень хорошо. Нам нужен более простой способ донести до компьютера то, что он должен делать. И люди изобрели языки программирования. Компьютерный язык программирования позволяет нам писать команды тем способом, который мы понимаем, а затем переводит их в двоичный код для использования компьютером.



Существует много разнообразных языков программирования. Эта книга научит тебя пользоваться одним из них – Python (читается как «пайтон») – чтобы ты мог заставить компьютер делать то, что тебе нужно.

Чтобы установить подходящую к этой книге версию Python, воспользуйся ***специальным инсталлятором Hello World***. ***Мы очень рекомендуем использовать именно его***, чтобы ты установил корректную версию программного обеспечения. Ты найдешь его по ссылке **www.dmkpress.com**.

Зачем изучать программирование?

Даже если ты не станешь профессиональным программистом (большинство так и не становится), существует несколько причин изучать программирование:

- самая важная: потому что ты этого хочешь! Программирование – очень интересная и полезная вещь как в качестве хобби, так и в виде профессии;
- если ты интересуешься компьютерами и хочешь знать больше о том, как они работают и как заставить их делать то, что тебе надо, то стоит изучать программирование;
- может быть, ты хочешь создать игру или не можешь найти программу, которая делает то, что тебе нужно. Ты сможешь написать ее самостоятельно;
- компьютеры сейчас везде, ты будешь использовать их на работе, в школе и дома. Изучение программирования поможет тебе лучше понимать принципы работы компьютеров.

Почему именно Python?

При огромном выборе языков программирования (а их очень много!) почему же мы выбрали именно Python для детской книги? На то есть несколько причин:

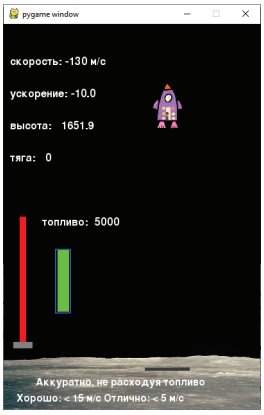
- Python прост для изучения. Программы, написанные на нем, легче всего читать, писать и понимать, чем на каком-либо другом языке;
- Python бесплатен. Ты можешь загрузить его – и много-много веселых и полезных программ на нем – бесплатно. Python – программное обеспечение с открытым исходным кодом. То есть любой пользователь может внести свой вклад в его развитие (создать способы, которые позволят выполнять больше задач с его помощью или выполнять те же самые задачи, но проще). Многие программисты внесли свой вклад, и теперь ты можешь загрузить результаты их работы;
- Python – это не игрушка. Хотя он очень хорош для изучения программирования, им пользуются многие профессионалы во всем мире, включая программистов в таких компаниях, как NASA и Google. Поэтому когда ты изучишь Python, тебе не нужно будет переключаться на «настоящий» язык, чтобы писать «настоящие» программы. Ты уже будешь уметь это делать;
- Python можно запустить в разных операционных системах. Он доступен для Windows, macOS и Linux. В большинстве случаев программа, написанная на Python, будет работать и в операционной системе Windows, и macOS. Ты можешь использовать примеры из этой книги на любом компьютере, на кото-

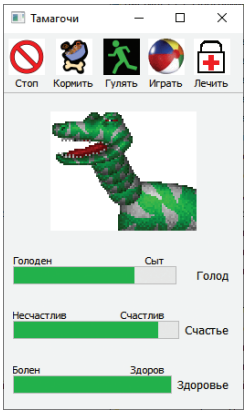
ром установлен Python (и помни, что дистрибутив Python ты всегда можешь скачать бесплатно);

- нам нравится Python. Мы получаем удовольствие от изучения и использования этого языка и думаем, что тебе тоже понравится.

Есть еще кое-что, что мы хотели бы упомянуть...

Один из самых веселых моментов в использовании компьютера (особенно для детей) – игры. Мы научимся делать свои игры, работать с графикой и со звуком. Ниже приведены примеры некоторых программ, которые мы напишем.





Мы думаем (по крайней мере, надеемся), тебе понравится изучать основы языка программирования и писать свои первые программы настолько же сильно, насколько управлять космическими кораблями и лыжниками.

Давай повеселимся!

Благодарности

Благодарности из первого издания

Эта книга никогда не была бы начата и закончена без вдохновения, инициативы и поддержки моей прекрасной жены Патрисии. Когда мы не смогли найти книгу, чтобы удовлетворить интерес нашего сына Картера к программированию, она сказала: «Тебе следует написать ее самому. Это будет отличный проект для вас обоих». Как и в большинстве случаев, она оказалась права. Патрисия умеет раскрывать в людях их лучшие качества. Поэтому мы с Картером начали думать, что будет в этой книге, делать наброски глав и писать примеры программ, искать способы сделать ее забавной и интересной. Раз уж мы взялись за дело, Картер и Патрисия следили за тем, чтобы книга была закончена. Картер отказывался от сказок на ночь, чтобы поработать над ней. А если мы не занимались книгой какое-то время, он напоминал: «Папочка, мы не писали книгу уже несколько дней!» Картер и Патрисия говорили мне, что если ты действительно чего-то хочешь, ты этого добьешься. И все члены нашей семьи, включая нашу дочь Кайру, жертвовали многими часами совместного досуга, пока эта книга не была окончена. Я благодарен им всем за терпение и поддержку, благодаря которым эта книга увидела свет.

Написание рукописи – это одно, а передача книги в руки читателя – совсем другое. Эта книга никогда бы не была опубликована без энтузиазма и постоянной поддержки Майкла Стивенса из издательства Manning Publications. С самого начала он «ухватил» идею и согласился в необходимости такого рода издания. Непоколебимая вера Майкла в этот проект и его терпеливое руководство над неопытным писателем были чрезвычайно ценны. Я также хотел бы сказать спасибо всем сотрудникам издательства, которые помогли выпустить эту книгу, а особенно Мэри Пирджис за терпеливую координацию всех производственных моментов.

Эта книга не была бы таковой без веселых и живых иллюстраций Мартина Мертонена. Его работы говорят сами за себя о таланте и художественном вкусе автора. К сожалению, они не показывают, насколько с ним приятно работать. Это было сущее удовольствие.

Однажды я спросил своего друга и коллегу Шона Кавану: «Как бы ты сделал это на языке Perl?» И он ответил: «Никак. Я бы использовал Python». Поэтому я окунулся в новый язык программирования. Шон ответил на многие мои вопросы, когда я изучал Python и писал первые программы. Он также собрал инсталлятор с примерами для этой книги. Его помощь неоценима.

Я также хотел бы поблагодарить многих людей, которые просматривали книгу во время написания и помогали работать на рукописью: Вибху Чендрешекар, Пэм Колкхоун, Гордон Колкхоун, д-р Тим Купер, Джош Кроунмейер, Саймон Кроунмейер, Кевин Дрисколл, Джеффри Элкнер, Тэд Феликс, Дэвид Гуджер, Лайза Л. Гудеар, д-р Джон Грейсон, Мишель Хаттон, Хорст Дженс, Энди Джадкис, Кейден Кумар, Энтони Линфант, Шэннон Мэдисон, Кеннет МакДональд, Эван Моррис, профессор Александр Репеннинг, Андре Роберж, Кэрри Дж. Стеллпфлаг, Керби Ернер и Брайан Вейнгаартен.

Также огромное спасибо переводчице книги на русский язык – Александре Калмыковой, и отдельная благодарность редактору русского издания, Михаилу Анатольевичу Райтману.

Конечный результат гораздо лучше благодаря им.

– Уоррен Сэнд

Я хотел бы поблагодарить Мартина Мертонена за его отличные карикатуры на меня, мою маму – за разрешение сесть за компьютер в возрасте двух лет и идею написать книгу, и, самое главное, моего отца за все силы, вложенные вместе со мной в эту книгу, и за то, что он научил меня программированию.

– Картер Сэнд

Благодарности из второго издания

Со вторым изданием книги помогали многие из тех, кто внес свой вклад в первое издание. В дополнение к тем людям, которые были перечислены ранее, мы хотели бы поблагодарить тех, кто помогал со вторым изданием: Бена Умса, Брайана Т. Янга, Коди Роузборо, Дэйва Брикетти, Элизабет Гордон, Айрис Фарэуэй, Мейсона Дженкинса, Рика Гордона, Шона Стебнера и Закари Янга. Спасибо также Игнасио Бельтран-Торресу и Даниэлю Солтису, которые тщательно откорректировали окончательный вариант книги незадолго до ее выпуска.

Мы также хотели бы поблагодарить всех сотрудников издательства Manning, которые помогли сделать это второе издание книги даже лучше, чем первое.

Благодарности из третьего издания

В дополнение к тем людям, которые были перечислены ранее, мы хотели бы поблагодарить тех, кто помогал с третьим изданием: Адайла Ретамала, Бена Макнамара, Бисваната Чоудхури, Бьерна Нейхауса, Боба Даста, Эли Хини, Эвиатара Кафкафи, Джеймса Макгинна, Мэрилин Хурет и Мелиссу Айс.

Мы хотели бы поблагодарить нашего друга Шона Кавана за то, что он в очередной раз создал инсталлятор под Windows для третьего издания.

И еще раз спасибо всем в издательстве Manning, кто помог сделать это третье издание книги даже лучше, чем первые два. Надеемся, на этот раз нам наконец удалось все сделать правильно!

Об этой книге

Эта книга обучает основам компьютерного программирования. Она предназначена для детей, но ее может использовать любой желающий.

Тебе не нужны никакие начальные знания о программировании для чтения этой книги, но ты должен уметь пользоваться компьютером. Возможно, ты используешь его для чтения почты, поиска во Всемирной паутине, прослушивания музыки, запуска игр или выполнения домашних заданий. Если ты можешь выполнять базовые операции на компьютере, напримерзапускать программы, открывать и сохранять файлы, у тебя не возникнет проблем с этой книгой.

Что тебе нужно

Эта книга обучает программированию на языке под названием Python. Он бесплатный, его можно скачать из многих источников, в том числе с сайта данной книги. Чтобы научиться программировать с помощью этой книги, тебе потребуется:

- эта книга (конечно же!);
- компьютер под управлением операционной системы Windows, macOS или Linux. Примеры в книге были выполнены на платформе Windows (для получения дополнительных сведений по процессу установки в операционных системах macOS и Linux см. файл *Установка в Linux.pdf* в архиве с примерами для данной книги);
- базовые знания по использованию компьютера (запуск программ, сохранение файлов и т. д.). Если ты испытываешь какие-либо сложности, можно обратиться за помощью к друзьям или родителям;
- разрешение установить Python на компьютер (от родителей, учителя или другого лица, ответственного за него). **Мы настоятельно рекомендуем использовать прилагаемый инсталлятор Hello World**, чтобы установить правильную версию Python, необходимую для этой книги. Ты найдешь его по адресу **www.dmkpress.com**;
- желание изучать новые вещи, даже если сначала у тебя ничего не получается.

Что тебе не нужно

Чтобы изучать программирование с помощью этой книги, тебе *не* нужно:

- покупать какое-либо программное обеспечение. Все, что тебе понадобится, бесплатно и доступно в архиве с примерами для этой книги по адресу **www.dmkpress.com**;
- знание программирования. Это книга для начинающих.

Изучение книги

Если ты собираешься изучать программирование с помощью этого пособия, ниже представлено несколько подсказок, которые тебе помогут:

- Следи за примерами.
- Набирай код программ.
- Отвечай на проверочные вопросы.
- Не волнуйся, успокойся!

Следи за примерами

Примеры в этой книге выглядят так:

```
if timsAnswer == correctAnswer:
    print("Ты все понял!")
    score = score + 10
```

Старайся следить за примерами и набирать код программ самостоятельно. (Мы тебе расскажем подробно, как это делать.) Ты можешь просто сидеть в большом удобном кресле и читать книгу. Возможно, ты даже чему-то научишься. Но ты научишься гораздо большему, если будешь выполнять описываемые действия.

Установка Python

Чтобы получить максимальную пользу от этой книги, тебе нужно установить Python на свой компьютер**. Мы настоятельно рекомендуем использовать инсталлятор Hello World**, чтобы установить правильную версию Python. Установщик Hello World доступен на веб-сайте по адресу **www.dmkpress.com**.

Если ты установишь Python каким-либо другим способом, у тебя может оказаться не та версия Python или необходимые модули. В этом случае ты можешь расстроиться, если твои примеры будут работать не так, как должны.

Набирай код программ

Инсталлятор, который доступен в комплекте с книгой, скопирует все примеры программ на твой жесткий диск (если пожелаешь). Он находится на сайте книги: **www.dmkpress.com**. Ты также можешь использовать прилагаемые файлы приме-

ров, но мы рекомендуем набирать их код самостоятельно. Набирая код программ, ты «прочувствуешь» программирование и Python в частности. (И потренируешь свои навыки работы с клавиатурой.)

Отвечай на проверочные вопросы

В конце каждой главы приведены вопросы для закрепления пройденного. Выполняй их все по возможности. Если ты столкнешься с трудностями, попробуй найти кого-то, кто разбирается в программировании, чтобы помочь тебе. Проработайте их вместе, и ты многому научишься. Не подглядывай в ответы, пока не закончишь или пока по-настоящему не зайдешь в тупик. (Да, некоторые ответы есть в конце книги, но, как мы уже сказали, не подглядывай!)

Слово Картеру

Я хотел убедиться, что эта книга подойдет детям, будет веселой и легкой для понимания. К счастью, у меня был помощник. Картер – мой сын, который любит компьютеры и хочет узнать о них больше. Он помогал мне сделать эту книгу правильной. Места, когда он замечал что-то смешное или необычное, мы оформили так:

Отличия третьего издания

Вот что нового в третьем издании по сравнению со вторым:

- в книге теперь используется Python 3 вместо Python 2 (приложение, описывающее различия между Python 2 и Python 3, ты найдешь в конце книги);
- для программирования графического интерфейса в главе 20 мы переключились с PyQt 4 на PyQt 5. PyQt также используется для программы «Висельник» в главе 22 и программы «Тамагочи» в главе 24;
- мы заменили главу 26, в которой рассказывалось о простейшем игровом искусственном интеллекте, новой главой о сетях.

Родителям и учителям

Python – бесплатное программное обеспечение с открытым кодом, нет никакой опасности в установке и использовании его на своем компьютере. Вы можете скачать программное обеспечение – и все необходимое для книги – бесплатно на сайте **www.dmkpress.com**.

Загруженные файлы легко установить, использовать, они не содержат вирусов и вредоносных программ.

Книги наподобие этой раньше сопровождались дисками с программным обеспечением, но сейчас большинство читателей (и издателей) предпочитают использовать Всемирную паутину.

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте **www.dmkpress.com**, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте (**http://dmkpress.com/authors/publish_book/**) или напишите в издательство по адресу **dmkpress@gmail.com**.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу **dmkpress@gmail.com**. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс. Ссылку на подозрительные материалы можно прислать по адресу электронной почты **dmkpress@gmail.com**.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

ГЛАВА 1

Приступая к работе

Установка Python

Первое, что необходимо сделать, – установить Python на твой компьютер. **Мы настоятельно рекомендуем использовать инсталлятор Hello World**, чтобы установить правильную версию Python; ты найдешь его на сайте **www.dmkpress.com**. Выбери версию инсталлятора, которая соответствует операционной системе твоего компьютера.

В СТАРЫЕ ДОБРЫЕ ВРЕМЕНА

На заре эры персональных компьютеров (ПК) все было просто. На многих из них был установлен язык программирования под названием BASIC. Людям ничего не нужно было устанавливать. Требовалось только включить компьютер, и на экране появлялась надпись `READY`[1], после чего можно было начинать набирать программу на языке BASIC. Звучит неплохо, да?

Но эта надпись `READY` была всем, что ты имел в распоряжении. Никаких программ, окон и меню. Если ты хотел заставить компьютер совершить какое-то действие, тебе нужно было писать программу! Не было текстовых редакторов, мультимедийных проигрывателей, браузеров и других вещей, к которым мы привыкли. Не было даже Всемирной паутины. Не было красивой графики и звуков, кроме системного «бип» при совершении ошибки!

[1] Готов (здесь и далее – прим. ред.).

Существуют версии для операционных систем Windows, macOS и Linux. Все примеры в этой книге написаны для операционной системы Windows, но использование Python в mac OS или Linux аналогично. Просто следуй инструкциям, чтобы установить правильный дистрибутив для твоей системы.

В этой книге мы используем версию Python 3.7.3. Если ты воспользуешься инсталлятором, содержащимся в архиве с примерами для этой книги, ты тоже получишь эту версию. На момент чтения данной книги могут появиться новые версии Python. Все примеры в книге были проверены в версии 3.7.3. Они должны работать и на более поздних версиях, но мы не можем предвидеть будущее, так что все может случиться.

Если Python уже установлен на твоем компьютере и ты не собираешься пользоваться инсталлятором, тебе нужно убедиться, что некоторые дополнительные компоненты, необходимые для изучения примеров из книги, тоже установлены. См. файл *Установка* в Linux.pdf на сайте **www.dmkpress.com**, чтобы узнать, как это сделать. Однако повторюсь: чтобы все операции, описанные в книге, получились и у тебя, лучше используй наш инсталлятор, доступный на сайте **www.dmkpress.com**.

Сравнение Python 3 с Python 2

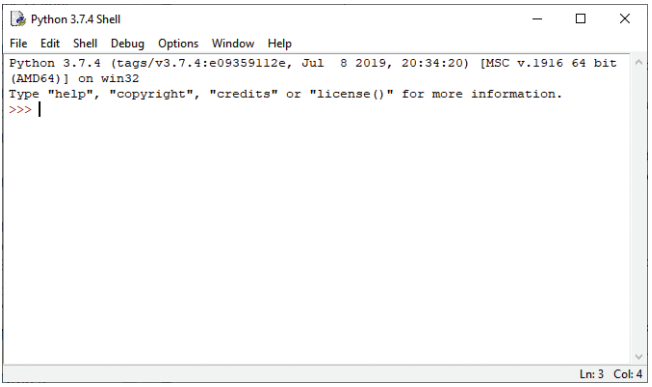
В предыдущих изданиях книги использовался Python 2. С тех пор новая версия программы, Python 3, стала более популярной, поэтому именно ее мы используем в этой книге. Однако оказывается, что Python 3 – на самом деле не столько измененная программа, сколько развилка на дороге. То есть код, написанный на Python 2 (как и пример кода из предыдущих изданий), не всегда будет корректно работать в Python 3, и наоборот.

Для получения более подробной информации о различиях между Python 2 и Python 3 см. приложение Б.

Взаимодействие с Python посредством среды разработки IDLE

Есть несколько способов начать работу с Python. Один из них называется средой разработки IDLE, которой мы сейчас и воспользуемся.

В меню **Пуск** (Start) в папке **Python 3.7** ты увидишь пункт **IDLE (Python 3.7)**. Щелкни мышью по нему – и увидишь открывшееся окно интерпретатора IDLE. Окно будет выглядеть примерно так, как на показано рисунке ниже.



Среда разработки IDLE – это *интерпретатор*[1] Python с графической *оболочкой*. Оболочка – это способ взаимодействия с программой посредством вводимого текста, а эта конкретная оболочка позволяет тебе работать с Python. (Именно поэтому ты видишь слова «Python Shell»[2] в названии окна.) В среде IDLE есть еще кое-что помимо оболочки, о чем ты узнаешь через минуту.

Символы **>>>** на предыдущем рисунке обозначают *строку приглашения*. Строка приглашения отображается, когда программа ждет от тебя ввода какой-либо команды или инструкции. В данном случае она сообщает о том, что Python готов начать работу.

Введи инструкции, пожалуйста

Давай отправим Python первую инструкцию.

Установи указатель мыши после символов строки приглашения **>>>** и введи код:

```
print(«Привет, мир!»)
```

Нажми клавишу **Enter**. (На некоторых клавиатурах она называется **Return**.) Нажимать клавишу **Enter** необходимо после ввода каждой строки. После ее нажатия ты получишь ответ:

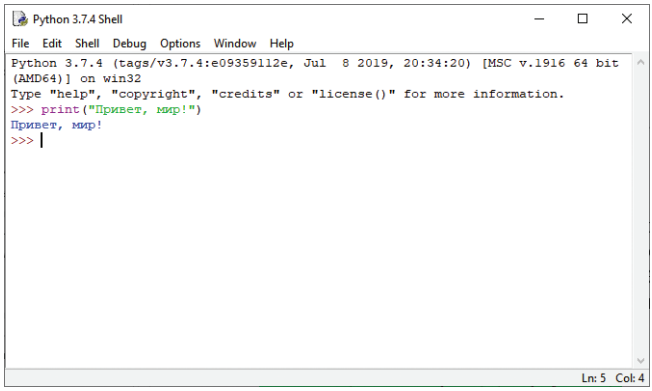
```
Привет, мир!
>>>
```

---

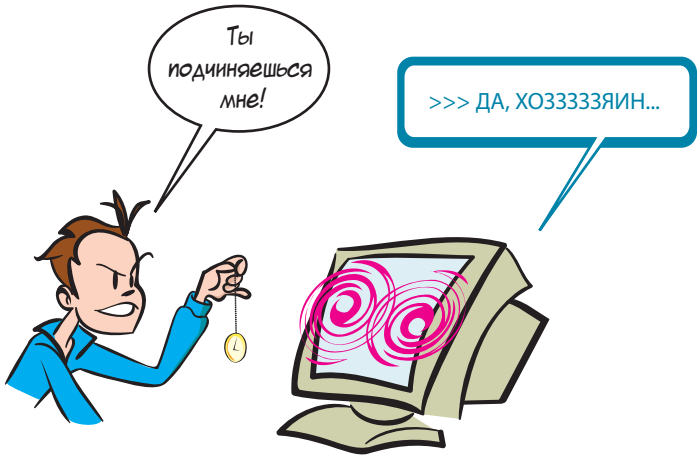
[1] Интерпретатор – программа, позволяющая вводить команды на том или ином языке программирования и тут же выполняющая их.

[2] Оболочка Python (пер. с англ.).

Рисунок ниже показывает, как это выглядит в окне интерпретатора IDLE.

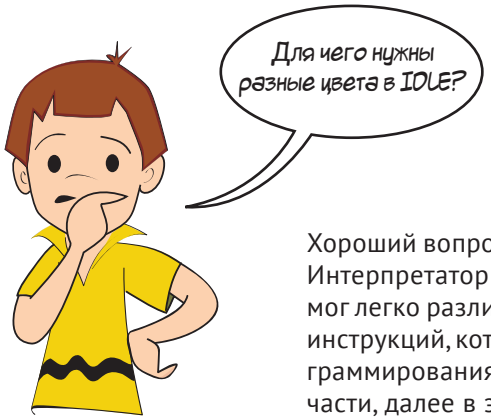


Python сделал именно то, что ты от него потребовал: он написал твое сообщение. (В программировании команда **`print`** часто означает отображение текста на экране, а не его печать на бумаге с помощью принтера[1].) Эта строка и есть *инструкция* для Python. Ты уже начал программировать! Компьютер тебе подчиняется!



Кстати, при изучении программирования есть традиция, что первой командой компьютеру должно быть вывести приветствие «Привет, мир!». Ты следуешь традиции. Добро пожаловать в мир программирования!

[1] Print – печать (пер. с англ.).



Хороший вопрос! IDLE помогает понять некоторые вещи. Интерпретатор отображает их разным цветом, чтобы ты мог легко различать части кода. (*Код* – другой термин для инструкций, которые ты даешь компьютеру на языке программирования.) Мы объясним, что же такое эти разные части, далее в этой книге.

Если команда не сработает

Если ты сделаешь ошибку, то увидишь сообщение наподобие этого:

```
>>> pront("Привет, мир!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pront' is not defined
>>>
```

Это сообщение об ошибке означает, что ты ввел команду, которую Python не понимает. В примере выше команда **print** написана неправильно: **pront**, – и Python не знает, что с ней делать. Если случится подобное, попробуй ввести команду снова и убедись, что она набрана правильно.

Все верно. Так произошло потому, что **print** является ключевым словом Python, а **pront** – нет.

СЛОВАРИК

Встроенные функции и ключевые слова – это специальные слова, которые являются частью языка Python. IDLE выделяет их специальным цветом, чтобы ты знал, что они особенные.

Взаимодействие с Python

То, что ты сейчас делал, – это использовал Python в интерактивном режиме. Ты ввел команду (инструкцию), и Python *выполнил* ее немедленно.

СЛОВАРИК

Выполнение команды, инструкции или программы – просто красивое название для ее «запуска» или «включения».

Давай попробуем еще кое-что в интерактивном режиме. Введи следующий код в командной строке:

```
>>> print(5 + 3)
```

Ты должен получить ответ:

```
8
>>>
```

Python может выполнять операции сложения! Это неудивительно, потому что компьютеры отлично знают математику.

Давай попробуем еще:

```
>>> print(5 * 3)
15
>>>
```

В большинстве компьютерных программ и языков символ * используется для обозначения умножения. Он называется *звездочкой*.

Если на уроках математики ты привык писать «пятью три» как «5×3», тебе нужно привыкать к использованию символа * вместо этого. (Символ в верхней части клавиши с цифрой 8 на большинстве клавиатур.)

Хорошо, а как насчет этого:

```
>>> print(2345 * 6789)
15920205
>>>
```

Или насчет этого:

```
>>> print(12345678987654321234567 89 * 98765432123456789876543 21)
121932632007315960006096522024081660722451126352 69
>>>
```

Все верно. С помощью компьютера можно производить действия с очень и очень большими числами. Вот что еще можно сделать:

```
>>> print(«кот» + «пес»)
котпес
>>>
```

Или попробуй вот это:

```
>>> print(«Привет « * 20)
Привет Привет Привет Привет Привет Привет Привет Привет Привет Привет
Привет Привет Привет Привет Привет Привет Привет Привет Привет Привет
```

Помимо математики, компьютеры хороши еще и в повторении. Только что мы попросили Python двадцать раз повторить слово «Привет».

Мы еще вернемся к интерактивному режиму позже, а прямо сейчас наступило…

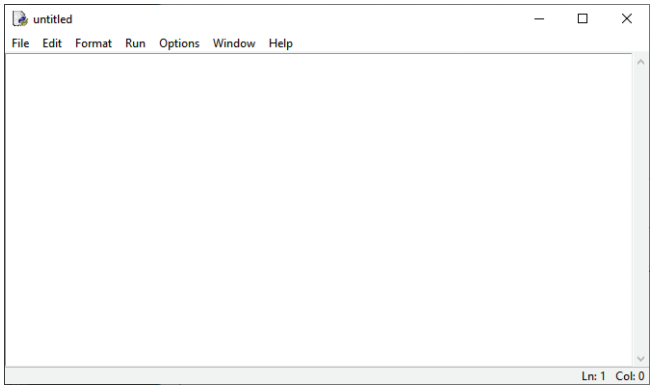
Время программировать

Примеры, которые ты видел до этого, – это простые инструкции (в интерактивном режиме). Они хороши для проверки некоторых возможностей Python, но это еще не программы. Как мы упомянули ранее, программа – это набор инструкций, собранных вместе. Итак, давай напишем свою первую программу на Python.

Для начала нам нужен способ ввода программы. Если ты будешь просто печатать в интерактивном окне, Python ее «не запомнит». Тебе необходимо использовать текстовый редактор (например, «Блокнот» в Windows, TextEdit в macOS или vi в Linux), который позволяет сохранить код программы в файл на жестком диске твоего компьютера. Среда разработки IDLE содержит текстовый редактор, который гораздо удобнее для наших нужд, чем «Блокнот». Чтобы найти его, выбери команду меню **File ⇨ New File** в окне интерпретатора IDLE.

> Когда мы говорим о выборе команд в меню, например **File ⇨ New**, первая часть (**File**) – это название меню в строке. Символ ⇨ указывает на следующую команду (**New**) в меню **File**. Мы будем использовать подобные указания во всей книге.

Ты увидишь окно, как показано на рисунке ниже. В строке с названием будет указано значение **Untitled**[1], поскольку ты еще не присвоил этому файлу имя.



Теперь введи программу из листинга 1.1 в окно редактора.

[1] Без названия (пер. с англ.).

Листинг 1.1. Наша первая настоящая программа

```
print(«Я люблю пиццу!»)
print("пицца " * 20)
print("мням " * 40)
print("Я наелся!")
```

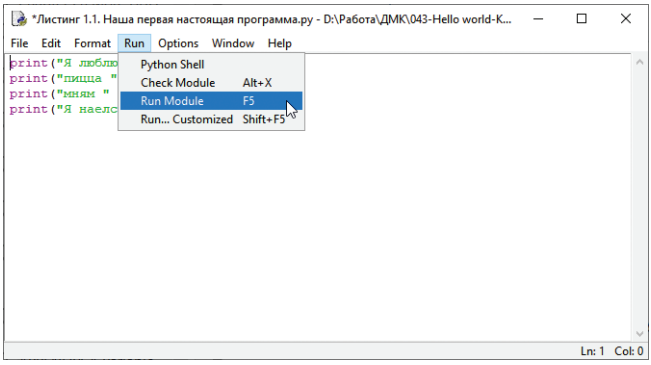
Ты заметил, что в названии написано «Листинг 1.1»? Когда пример кода представляет собой целую программу, мы будем нумеровать его таким образом, чтобы ты мог легко найти его в папке *Примеры*.

Закончив, сохрани программу, выбрав команду меню **File** ⇨ **Save** (Файл ⇨ Сохранить) или **File** ⇨ **Save As** (Файл ⇨ Сохранить как). Присвой файлу имя **pizza.py**. Ты можешь сохранить его в любую папку. Даже создать новую папку для хранения своих программ. Расширение *.py* очень важно, поскольку именно оно сообщает компьютеру, что это программа на языке Python, а не обычный текстовый файл.

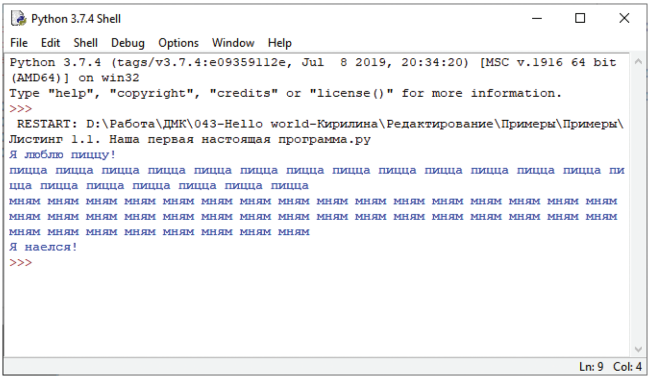
Обрати внимание, что редактор использует разные цвета в программе. Некоторые слова выделены фиолетовым цветом, а некоторые – зеленым. Это происходит потому, что интерпретатор IDLE предположил, что ты будешь набирать программу. Интерпретатор IDLE показывает встроенные функции Python фиолетовым, а значения в кавычках – зеленым цветом. Это упрощает чтение кода программ на языке Python.

Запуск первой программы

Сохранив программу, перейди к меню **Run** (в окне интерпретатора IDLE) и выбери команду **Run Module**, как показано на следующем рисунке. Эта команда запустит твою программу.



Ты увидишь, что окно оболочки Python (первое окно, которое появляется при запуске среды разработки IDLE) снова станет активным, а затем ты увидишь что-то наподобие следующего:



Та часть, где написано слово **RESTART**, говорит о том, что мы запустили программу. (Это помогает ориентироваться, когда ты запускаешь программы одну за другой.)

Потом начинается выполнение программы.

Конечно, она выполнила совсем немного действий. Но ты добился от компьютера выполнения требуемых действий. Наши программы станут более интересными в процессе обучения.

Если что-то пойдет не так

Что случится, если в программе допущена ошибка и она не запустилась? Есть два разных вида ошибок, которые могут случиться. Давай рассмотрим оба вида, чтобы ты знал, что делать в случае каждого из них.

Синтаксические ошибки

Оболочка IDLE выполняет проверку программы перед попыткой запустить ее. Если она находит ошибку, это обычно *синтаксическая ошибка*. Синтаксис – это правописание и грамматические правила языка программирования, то есть синтаксическая ошибка означает, что ты написал нечто неверное в коде. Ниже представлен пример:

```
print(«Я люблю пиццу!»)
print(«пицца « * 20)
print(«мням « * 40)
print(Я наелся!»)
```

Пропущена кавычка

Мы пропустили кавычку между командой **print** и словосочетанием **Я наелся!** Если попробовать запустить эту программу, оболочка отобразит окно с сообщением об ошибке: «SyntaxError» (Синтаксическая ошибка). Тебе потребуется просмотреть свой код и обнаружить ее. Интерпретатор IDLE выделит красным цветом область с ошибкой. Это может быть не совсем точное выделение, но близкое к ошибке.

Ошибки при выполнении

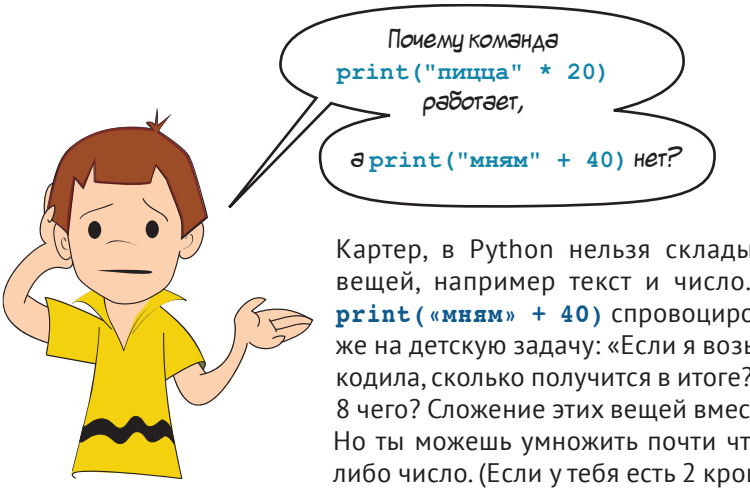
Второй вид ошибок интерпретатор IDLE не может обнаружить до запуска программы. Эти ошибки случаются только при запуске, поэтому их называют *ошибками при выполнении*. Ниже приведен пример подобной ошибки в программе:

```
print(«Я люблю пиццу!»)
print(“пицца “ * 20)
print(“мням “ + 40)
print(“Я наелся!”)
```

Если мы сохраним этот код и попробуем его запустить, программа начнет выполнять инструкции. Первые две строки будут выведены на экран, а затем мы получим сообщение об ошибке:

```
>>>
RESTART: C:/HelloWorld/Примеры/error1.py
Я люблю пиццу!
пицца пицца пицца пицца пицца пицца пицца пицца пицца пицца пицца
пицца пицца пицца пицца пицца пицца пицца пицца пицца
Traceback (most recent call last):          ← Начало сообщения об ошибке
  File “C:/HelloWorld/Примеры/error1.py”, line 3, in <module>
    print(“мням “ + 40)      ← Строка с ошибкой в коде      Где находится ошибка
TypeError: must be str, not int
>>>                          ← Возможная ошибка по версии интерпретатора
```

Строка со словом **`Traceback`** – начало сообщения. Следующая строка содержит информацию о месте ошибки – имя файла и номер строки. Затем выводится сама строка с ошибочным кодом. Это помогает обнаружить ошибку. В последней части сообщения указано мнение интерпретатора Python о том, что именно было неправильно. Вскоре ты научишься понимать подобные сообщения.



Картер, в Python нельзя складывать разные типы вещей, например текст и число. Поэтому команда **`print(«мням» + 40)`** спровоцировала ошибку. Похоже на детскую задачу: «Если я возьму 5 яблок и 3 крокодила, сколько получится в итоге?» Ты получишь 8, но 8 чего? Сложение этих вещей вместе не имеет смысла. Но ты можешь умножить почти что угодно на какое-либо число. (Если у тебя есть 2 крокодила, то, умножив их на 5, ты получишь 10 крокодилов.) Именно поэтому команда **`print(«пицца» * 20)`** работает.

Наша вторая программа

Первая программа выполняла немного действий. Она просто выводила текст на экран. Давай попробуем что-то более интересное.

Следующий код в листинге 1.2 представляет простую игру на угадывание чисел. Открой новый файл в интерпретаторе IDLE, выбрав команду меню **File** ⇨ **New File**, как ты делал в первый раз. Введи код из листинга 1.2 и сохрани его. Ты можешь присвоить файлу любое имя, но его расширение должно быть *.py*. Например, *NumGuess.py*.

Ниже представлено 18 строк инструкций плюс несколько пустых строк для упрощения чтения кода. Ввод кода займет немного времени. Не переживай о том, что мы еще не объяснили тебе значение этого кода. Скоро ты это узнаешь.

Листинг 1.2. Игра по угадыванию чисел

```
import random
secret = random.randint(1, 100)  ← Выбор секретного числа
guess = 0
tries = 0

print(«ЯРРР!  Я страшный пират Робик, и у меня есть секрет!»)
print(«Это число от 1 до 100.  У тебя есть 6 попыток.”)

while guess != secret and tries < 6:
    guess = int(input(“Каков твой ответ?”))  ← Получение ввода от игрока
    if guess < secret:
        print(“Слишком мало, каррамба!”)
    elif guess > secret:
        print(“Слишком много, сухопутная крыса!”)

    tries = tries + 1  ← Использование одной попытки
```

Максимальное число попыток – 6

Вывод сообщения в конце игры

```
if guess == secret:
    print(«Стой! Ты угадал!  Теперь ты знаешь мой секрет!»)
else:
    print(«Больше никаких попыток!  Удачи в следующий раз, приятель!»)
    print(«Секретное число», secret)
```

Когда ты будешь набирать код, обрати внимание на выделение отступами строк после команды **while** и дополнительное выделение строк после **if** и **elif**. Также не пропусти двоеточия в конце некоторых строк. Если ты правильно используешь двоеточие после нужной строки, редактор сам выделит отступом следующую строку.

Сохранив файл, запусти программу с помощью команды меню **Run ⇨ Run Module**, как и в случае с первой программой. Попробуй сыграть в игру. Ниже представлен пример запущенной программы:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_1-2.py
ЯРРР!  Я страшный пират Робик, и у меня есть секрет!
Это число от 1 до 100.  У тебя есть 6 попыток.
Каков твой ответ? 40
Слишком много, сухопутная крыса!
Каков твой ответ? 20
Слишком много, сухопутная крыса!
Каков твой ответ? 10
Слишком мало, каррамба!
Каков твой ответ? 11
Слишком мало, каррамба!
Каков твой ответ? 12
Стой! Ты угадал!  Теперь ты знаешь мой секрет!
>>>
```

Мы использовали пять попыток для угадывания числа, которое оказалось 12.

Мы расскажем подробнее об инструкциях **while**, **if**, **else**, **elif** и **input** в следующих главах. Но ты можешь уже сейчас сформировать начальное представление о работе этой программы:

1. Секретное число выбирается программой в случайном порядке.
2. Пользователь вводит свой вариант числа.
3. Программа сравнивает догадку с секретным числом: больше или меньше?
4. Пользователь продолжает угадывать, пока у него не закончатся попытки.
5. Когда догадка совпадет с секретным числом, игрок выигрывает.

Что ты узнал?

Ух! Ты узнал много нового. В этой главе ты:

- установил Python;
- научился запускать среду разработки IDLE;
- узнал об интерактивном режиме;
- предоставил интерпретатору Python некоторые инструкции и выполнил их;
- увидел, как Python выполняет арифметические действия (включая работу с очень большими числами!);
- запустил текстовый редактор IDLE, чтобы набрать код своей первой программы;
- запустил свою первую программу на Python!
- узнал о сообщениях об ошибке;
- запустил свою вторую программу на Python: игру на угадывание чисел.

Проверь свои знания

- Как запустить интерпретатор IDLE?
- Какое действие выполняет команда **`print`**?
- Какой символ служит для представления операции умножения на языке Python?
- Что отображает интерпретатор IDLE при запуске программы?
- Каким другим словом можно назвать запуск программы?

Попробуй самостоятельно

1. В интерактивном режиме интерпретатора Python подсчитай количество минут в неделе.
2. Напиши короткую программу для вывода трех строк: твое имя, дата твоего рождения и твой любимый цвет. Результат должен выглядеть примерно так:

```
Меня зовут Петя Иванов.
Я родился 1 марта 1998 года.
Мой любимый цвет — синий.
```

Сохрани программу и запусти ее. Если программа не выполняет ожидаемые действия или ты получаешь сообщение об ошибке, попробуй исправить ошибки в коде и повторить попытку.

ГЛАВА 2

Память и переменные

Что такое программа? Эй, подожди минуточку, мы думали, что ответили на этот вопрос в главе 1! Мы сказали, что программа – это набор инструкций для компьютера.

Да, это верно. Но почти все программы, которые делают что-то полезное или смешное, имеют и другие качества:

- они получают *ввод* (данные);
- они *обрабатывают* данные;
- они производят *вывод*.

Ввод, обработка, вывод

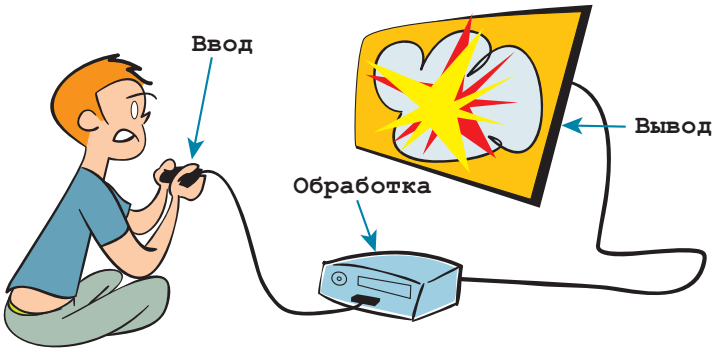
Твоя первая программа не содержала ввода или обработки. Это одна из причин, по которой программа была не очень интересна. Выводом были сообщения программы на экране.

Твоя вторая программа, игра по отгадыванию чисел (листинг 1.2), содержала все три базовых элемента:

- *ввод* ответов игроком;
- *обработку* ответов программой и подсчет их количества;
- *вывод* сообщений.

Вот еще один пример программы со всеми тремя элементами: в видеоигре *ввод* – это сигнал джойстика или контроллера; *обработка* – распознавание твоего

действия (застрелил ли ты пришельца, бросил огненный шар, прошел уровень); *вывод* – надпись или картинка на экране и звук в колонках.



Ввод, обработка, вывод. Запомни.

Хорошо, компьютеру нужен ввод информации. Но что он с ней делает? Чтобы сделать что-то с введенной информацией, компьютер должен ее запомнить или где-то сохранить. Он хранит данные, включая ввод (и саму программу), в своей *памяти*.

ЧТО ТАМ ПРОИСХОДИТ?

Ты, должно быть, слышал о компьютерной *памяти*, но что это означает? Мы сказали, что компьютер – это набор включающихся и выключающихся переключателей. Память похожа на группу таких переключателей, находящихся в том или ином состоянии некоторое время. Когда ты устанавливаешь их в нужное положение, они остаются в нем, пока ты их не изменишь. Они *запоминают* свое положение… Ты можешь *записывать* в память (настраивать переключатели) или *читать* из памяти (просматривать положение переключателей, не изменяя их).

Но как мы назначаем место для хранения в Python? И как мы находим сохраненное?

В Python если ты хочешь, чтобы программа запомнила что-то для дальнейшего применения, тебе нужно всего лишь присвоить этой «вещи» *имя (name)*. Python создаст место для нее в памяти компьютера вне зависимости от того, будет ли это число, текст, изображение или музыкальный файл. Когда ты хочешь снова вернуться к сохраненному объекту, используй то же имя.

Давай снова используем интерпретатор Python в интерактивном режиме и узнаем об именах больше.

Имена

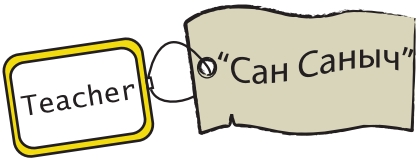
Вернись к окну интерпретатора Python (если ты уже закрыл окно IDLE, открой его снова, как было сказано в главе 1).

В командной строке набери код:

```
>>> Teacher = “Сан Саныч”
>>> print(Teacher)
```

(Напомним, что группа символов **>>>** – это начало командной строки Python. Ты вводишь код после нее и нажимаешь клавишу **Enter**.) В ответ ты увидишь следующее:

```
Сан Саныч
>>>
```



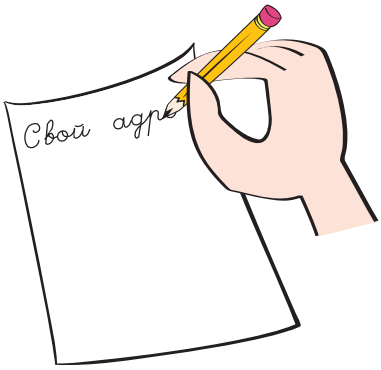
Ты только что создал объект, состоящий из букв «Сан Саныч», и присвоил ему имя **Teacher**.

Знак равенства (**=**) инструктирует Python *присвоить что-то чему-то* или «сделать равным». Ты *присвоил* имя **Teacher** набору букв «Сан Саныч». Имя – это ярлык, метка или наклейка, которую ты прикрепляешь к чему-либо для идентификации.

Где-то в глубинах памяти твоего компьютера существуют буквы «Сан Саныч». Тебе не нужно знать, где именно. Ты сказал Python, что имя этого набора букв – **Teacher**, и именно так ты будешь обращаться к нему теперь.

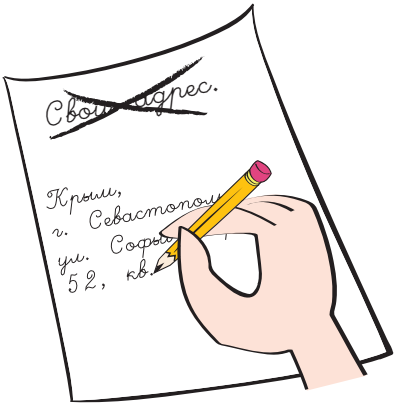
Когда ты ставишь кавычки, Python принимает слова в кавычках буквально. И выводит на экран именно то, что было в кавычках. Когда ты не используешь кавычки, Python должен сам понять, что же это за объект. Это может быть число (например, **5**), выражение (**5 + 3**) или имя (**Teacher**). Поскольку мы создали имя, Teacher, Python выводит то, чему это имя соответствует, а именно – набор букв «Сан Саныч».

Например, если тебе говорят: «Напиши свой адрес», – ты же не будешь писать: «Свой адрес».



(Ну, Картер может так написать, потому что он любит шутить...)

Ты напишешь, например: «Ул. Набережная, 16, кв. 4...»



Если ты пишешь «Свой адрес», это значит, что ты принял побуждение к действию буквально. Python не принимает вещи буквально, если ты не используешь кавычки. Ниже представлен другой пример:

```
>>> print(«53 + 28»)
53 + 28
>>> print(53 + 28)
81
```

Без кавычек Python воспринимает 53+28 как *арифметическое выражение* и *вычисляет* это выражение. В данном случае это было выражение сложения двух чисел, поэтому Python вывел сумму.

СЛОВАРИК

Арифметическое выражение – это сочетание чисел и знаков, значение которых Python может посчитать.
Вычислять означает «определять численное значение».

Python сам определяет количество памяти, необходимое для хранения букв, и часть памяти, в которой они будут храниться. Чтобы вернуть эту информацию (получить ее обратно), тебе нужно просто использовать имя еще раз. Мы использовали *имя* вместе с командой **`print`**, которая выводит на экран числа или текст.

Думай как программист

Когда ты присваиваешь имени значение (например, значение «Сан Саныч» имени **`Teacher`**), оно сохраняется в памяти и называется *переменная*.

В большинстве языков программирования мы говорим, что ты сохраняешь значение в переменной.

Но Python действует немного иначе, чем остальные языки программирования. Вместо присвоения значений переменным он *присваивает имена значениям*.

Некоторые программисты говорят, что в Python нет «переменных», а есть «имена» вместо них. Но ведут они себя почти так же.

Чистый способ хранения

Использование имен в Python похоже на поход в химчистку… Твоя одежда помещается на вешалку, на нее прикрепляют квитанцию с твоим именем и отправляют на длинный кронштейн к остальным вещам. Когда ты возвращаешься забирать свои вещи, тебе не нужно точно знать, где они находятся на кронштейне. Ты просто говоришь свое имя, и тебе возвращают одежду. Фактически вещи могут находиться не там, где ты их оставил. Но сотрудники химчистки найдут их для тебя. Тебе нужно знать только свое имя.

Переменные работают точно так же. Тебе не нужно точно знать, где в памяти хранится твоя информация. Тебе нужно использовать только то же имя, что и при сохранении.

Ты можешь создавать переменные не только для букв. Им можно присваивать численные значения. Вспомним наш старый пример:

```
>>> 5 + 3
8
```

Давай введем в него переменную:

```
>>> First = 5
>>> Second = 3
>>> print(First + Second)
8
```

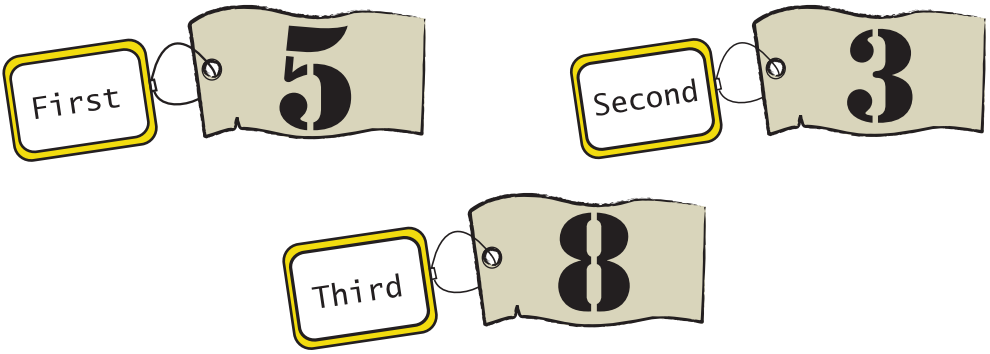
Мы создали два имени, **`First`** и **`Second`**. Числу 5 было присвоено имя **`First`**, а числу 3 – **`Second`**. Затем мы вывели на экран их сумму.

Есть еще один способ это сделать:

```
>>> Third = First + Second
>>> Third
8
```

Обрати внимание: в интерактивном режиме мы можем отобразить значение переменной, просто набрав ее имя, не используя команду **`print`**. (В программе это не будет работать.)

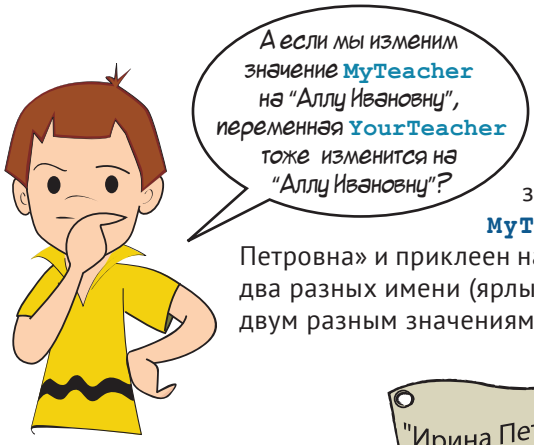
В этом примере вместо выполнения сложения в команде **print** мы взяли переменную по имени **First** и переменную по имени **Second** и сложили их вместе, создавая новую переменную по имени **Third**. **Third** – это сумма переменных **First** и **Second**.



Ты можешь присваивать более одного имени одному значению. Попробуй в интерактивном режиме ввести следующий код:

```
>>> MyTeacher = “Ирина Петровна”
>>> YourTeacher = MyTeacher
>>> MyTeacher
«Ирина Петровна»
>>> YourTeacher
«Ирина Петровна»
```

Это похоже на наклеивание двух ярлыков на одну вещь. На одном из них написано **YourTeacher**, на втором – **MyTeacher**, но они оба прикреплены на объект «Ирина Петровна».



Это очень хороший вопрос, Картер. Ответ: нет. Произойдет вот что: будет создана новая переменная со значением «Алла Ивановна». Ярлык `MyTeacher` будет отклеен от «Ирина Петровна» и приклеен на «Алла Ивановна». У тебя останется два разных имени (ярлыка), но теперь они будут присвоены двум разным значениям, а не одному и тому же.

Что в имени моем?

Ты можешь назвать переменную как угодно (ну, почти как угодно). Имя может быть любой длины и может содержать буквы и цифры, а также знак нижнего подчеркивания (_).

Но есть несколько правил касательно имен переменных. Самое важное – они чувствительны к регистру, то есть к прописным и строчным буквам. Поэтому `teacher` и `Teacher` – два разных имени. Так же, как и `first` и `First`.

Следующее правило говорит о том, что имя переменной должно начинаться с буквы или нижнего подчеркивания. Оно не может начинаться с цифры. Поэтому имя `4fun` недопустимо.

Еще одно правило: имя переменной не может содержать пробелы. Если ты хочешь знать все правила имен переменных в Python, то можешь заглянуть в приложение А в конце книги.

В СТАРЫЕ ДОБРЫЕ ВРЕМЕНА

В некоторых ранних языках программирования имена могли быть длиной только в одну букву. А на некоторых компьютерах были только прописные буквы, что означало, что у тебя было лишь 26 вариантов имен переменных: от A до Z! Если тебе нужно было использовать в программе больше 26 переменных, тебе не повезло!

Числа и строки

Итак, мы создали переменные для букв (текста) и чисел. Но в нашем примере на сложение как Python узнал, что мы имели в виду числа 5 и 3, а не символы «3» и «5»? Ну, как и в последнем предложении, кавычки решают все.

Символ или набор символов (буквы, числа или знаки пунктуации) называются *строкой*. Способ объяснить Python, что ты создаешь строку, – поставить кавычки вокруг символов. Ему не важно, какие кавычки ты используешь.

Оба примера будут работать:

```
>>> teacher = “Сан Саныч”  ← Двойные кавычки
>>> teacher = ‘Сан Саныч’  ← Одинарные кавычки
```

Но нужно использовать одинаковые кавычки в начале и в конце строки.

Если мы введем число без кавычек, Python будет знать, что мы имеем в виду именно численное значение, а не символ. Почувствуй разницу:

```
>>> first = 5
>>> second = 3
>>> first + second
8
>>> first = ‘5’
>>> second = ‘3’
>>> first + second
‘53’
```

Без кавычек 5 и 3 ведут себя как числа, и мы получаем их сумму. С кавычками «3» и «5» принимаются за строки, и мы получаем два символа, выведенных рядом, то есть «53». Ты также можешь складывать строки букв друг с другом, как мы делали это в главе 1:

```
>>> print(«кот» + «пес»)
котпес
```

Обрати внимание, что когда ты добавляешь строки друг к другу таким образом, между ними нет пробела. Они складываются друг с другом.

ДЛИННОЕ УМНОЕ СЛОВО!

Конкатенация

Не совсем верно использовать слово «складывать», когда мы говорим о строках. Когда ты соединяешь строки или символы, чтобы создать более длинную строку, для этого есть свое название. Вместо «сложения» это называется *конкатенацией*. Мы говорим, что ты *конкатенируешь* две строки.

Длинные строки

Если ты хочешь создать строку, которая занимает более одной строки, тебе нужно использовать особый вид строк, называемый *строка в тройных кавычках*. Ниже представлено, как это выглядит:

```
long_string = """Ничего на свете лучше нету,
Чем бродить друзьям по белу свету.
Нам дворцов заманчивые своды
Не заменят никогда свободы!"""
```

Такой вид строки начинается и заканчивается тройными кавычками. Кавычки могут быть как одинарные, так и двойные, то есть то же самое может выглядеть так:

```
long_string = '''Ничего на свете лучше нету,
Чем бродить друзьям по белу свету.
Нам дворцов заманчивые своды
Не заменят никогда свободы!'''
```

Строки с тройными кавычками могут быть полезны, когда у тебя есть несколько строк текста, который нужно вывести на экран вместе, и ты не хочешь использовать разные строки для каждой строки текста.

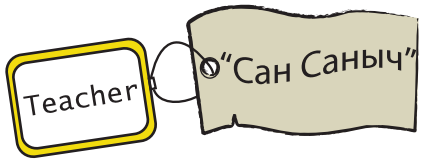
Насколько они «переменные»?

Переменные называются так по какой-то причине. Это потому, что они... ну... переменные! Это значит, что ты можешь *менять* значение, им присвоенное.

В Python это выполняется посредством создания нового объекта, отличного от старого, и присвоения старого ярлыка (имени) новому объекту. Мы делали это с переменной **`MyTeacher`**. Мы сняли ярлык **`MyTeacher`** со значения «Ирина Петровна» и присвоили его новому объекту, «Алла Ивановна». Мы назначили новое значение переменной **`MyTeacher`**.

Давай попробуем еще раз. Помнишь переменную **`Teacher`**, которую мы создали раньше? Что ж, если ты не закрыл IDLE, ее еще можно использовать. Введи следующий код:

```
>>> Teacher
'Сан Саныч'
```



Ага, переменная работает. Теперь мы можем изменить ее значение на какое-нибудь другое:

```
>>> Teacher = 'Петр Петрович'
>>> Teacher
'Петр Петрович'
```

Итак, мы создали новый объект, «Петр Петрович», и назвали его **`Teacher`**. Наш ярлык перешел со старого объекта на новый. Но что же случилось со старым, «Сан Санычем»?

Напомним, что объекты могут иметь более одного имени (более одного ярлыка). Если «Сан Саныч» имеет другой ярлык, то остается в памяти компьютера. Но если у него больше нет ярлыков, Python решает, что он больше никому не нужен, и удаляет объект из памяти.

Таким образом, память не заполняется ненужными вещами. Python производит эту чистку автоматически, и тебе не нужно об этом беспокоиться.

Важно знать, что фактически мы не меняли «Сан Саныча» на «Петра Петровича». Мы переместили ярлык (снова присвоили имя) с одного объекта на другой. Некоторые вещи в Python (например, числа и строки) не могут быть изменены. Ты можешь заново назначить им имена (как мы только что сделали), но ты не можешь изменить сами объекты.

Есть еще некоторые неизменные вещи в Python. Ты узнаешь о них в главе 12, когда будем говорить о *списках*.

Новый я

Ты можешь сделать переменную равной себе самой:

```
>>> Score = 7
>>> Score = Score
```

Готов поспорить, ты сейчас думаешь: «Да это же бесполезно!» И ты прав. Это похоже на изречение «Я – это я». Но с небольшим изменением ты можешь стать *новым собой*! Попробуй:

```
>>> Score = Score + 1
>>> print(Score)
8
```

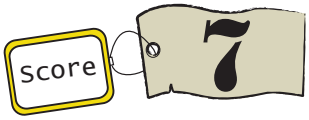
Меняет значение переменной с 7 на 8

Что здесь произошло? В первой строке ярлыку **`Score`** было присвоено значение 7. Мы создали новый объект – **`Score + 1`**, или 7 + 1. Этот новый объект – 8. Затем мы сняли ярлык **`Score`** со старого объекта (7) и присвоили его новому объекту (8). То есть переменная **`Score`** изменила свое значение с 7 на 8.

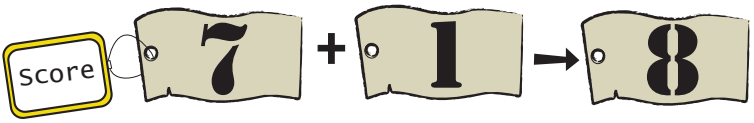
Когда мы делаем переменную равной чему-либо, переменная всегда находится слева от знака равенства. Трюк в том, что переменная может находиться также и справа. Это бывает очень полезно, и ты еще увидишь подобное во многих программах.

Чаще всего это используется при *возрастании* значения переменной (ее увеличении на определенное число), что мы только что выполнили, или, наоборот, для *убывания* значения переменной (ее уменьшения на определенное количество).

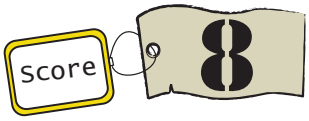
1 Начни со **Score** = 7.



2 Создай новый объект, добавив к нему 1 (который будет 8).



3 Присвой имя **Score** новому объекту.



Переменная **Score** изменит свое значение с 7 на 8.

Ниже приведено несколько важных моментов о переменных, которые нужно запомнить.

- Переменную можно снова назначить (прикрепить ярлык к новому объекту) в любое время с помощью программы. Это очень важно помнить, поскольку самые распространенные «баги» в программировании связаны с изменением не той переменной или с изменением переменной не в то время.
  Один из способов избежать этого – использовать легко запоминаемые имена переменных. Сравни:

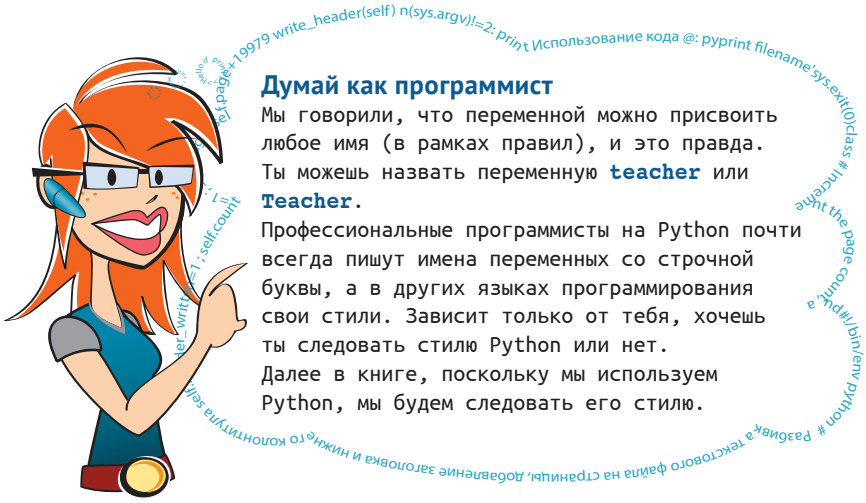
```
t = 'Сан Саныч'
```

  или

```
x1796vc47blahblah = 'Сан Саныч'
```

  Какое имя сложнее запомнить? Но с использованием обоих имен легко допустить ошибку. Старайся присваивать имена, которые говорят тебе что-то о значении переменной.
- Имена переменных чувствительны к регистру. Это значит, что прописные и строчные буквы отличаются. То есть **teacher** и **Teacher** – два разных имени.

И напомним, что если ты хочешь знать все правила присвоения имен переменным в Python, загляни в приложение А.



Думай как программист

Мы говорили, что переменной можно присвоить любое имя (в рамках правил), и это правда. Ты можешь назвать переменную **`teacher`** или **`Teacher`**.

Профессиональные программисты на Python почти всегда пишут имена переменных со строчной буквы, а в других языках программирования свои стили. Зависит только от тебя, хочешь ты следовать стилю Python или нет.

Далее в книге, поскольку мы используем Python, мы будем следовать его стилю.

Что ты узнал?

В этой главе ты научился:

- «запоминать» и хранить объекты в памяти компьютера с помощью переменных;
- называть переменные именами;
- различать типы значений переменных, например строки и числа.

Проверь свои знания

1. Как объяснить Python, что переменная – строка (символы), а не число?
2. Создав переменную, можешь ли ты изменить ее значение?
3. Имя переменной **`TEACHER`** равнозначно имени **`TEACHEr`**?
4. **`«Бах»`** и **`'Бах'`** – это одно и то же для Python?
5. **`'4'`** и **`4`** – это одно и то же для Python?
6. Какое из следующих имен нельзя присвоить переменной? Почему?
   a) **`Teacher2`**
   b) **`2Teacher`**
   c) **`teacher_25`**
   d) **`TeaCher`**
7. **`«10»`** – это число или строка?

Попробуй самостоятельно

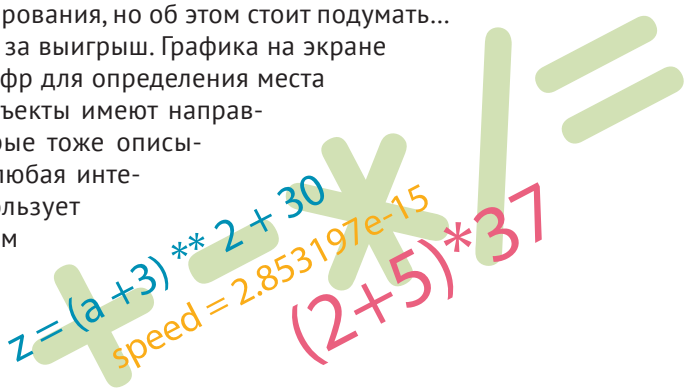
1. Создай переменную и присвой ей число (любое). Отобрази значение своей переменной с помощью команды `print`.
2. Модифицируй свою переменную, заменив старое значение новым или добавив что-то к старому значению. Выведи новое значение с помощью команды `print`.
3. Создай еще одну переменную и присвой ей некую строку (текст). Выведи ее значение с помощью команды `print`.
4. Как в предыдущей главе, в интерактивном режиме Python посчитай количество минут в неделе. Но в этот раз используй переменные. Создай переменные `days_per_week` (дней в неделе), `hours_per_day` (часов в сутках) и `minutes_per_hour` (минут в часе) и перемножь их.
5. Люди всегда говорят, что им не хватает времени. Сколько минут было бы в неделе, если бы в дне было 26 часов? (Подсказка: измени значение переменной `hours_per_day`.)

ГЛАВА 3

Основы математики

Когда мы впервые использовали интерпретатор Python в интерактивном режиме, ты увидел, как он может производить простые арифметические действия. Сейчас ты узнаешь, что же еще он может делать с числами и арифметикой. Ты можешь не понимать этого, но математика везде! Особенно в программировании математика постоянно используется. Это не значит, что тебе потребуется стать математиком для изучения программирования, но об этом стоит подумать...

В каждой игре есть очки за выигрыш. Графика на экране создается с помощью цифр для определения места и цвета. Движущиеся объекты имеют направление и скорость, которые тоже описываются числами. Почти любая интересная программа использует числа и математику тем или иным способом. Итак, давай изучим базовые математические и численные понятия в Python.



Кстати, многое из того, что ты узнаешь, применимо и к другим языкам программирования, и к другим программам, например к электронным таблицам. Не только Python использует математику подобным образом.

Четыре базовые операции

Мы уже наблюдали вычисления в Python в главе 1: сложение с помощью знака **+** и умножение с помощью знака *****.

В языке Python используется дефис (**-**) (который также называют знаком минуса) для вычитания:

```
>>> print(8 – 5)
3
```

Поскольку на клавиатуре нет символа деления (÷), все программы используют для этой цели наклонную черту (**/**).

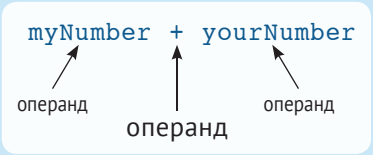
```
>>> print(7/2)
3.5
```

Операторы

Символы **+**, **-**, ***** и **/** называются *операторами*. Потому что они «оперируют», или работают с числами. Знак **=** также является оператором и называется *оператором присвоения*, поскольку с его помощью можно присвоить значение переменной.

ДЛИННОЕ УМНОЕ СЛОВО!

Оператор – это нечто, что имеет влияние или «оперирует» объектами вокруг себя. Эффект может выражаться в присвоении значения, проверке или изменении одного из этих объектов.



Символы **+**, **-**, ***** и **/**, используемые для выполнения арифметических действий, – это *операторы*.
Объекты, над которыми производится действие, – *операнды*.

Порядок операций

Какое из этих выражений верное:

2 + 3 * 4 = 20

или

2 + 3 * 4 = 14?

Это зависит от порядка выполнения действий. Если сначала выполнить сложение, получается:

2 + 3 = 5, затем 5 * 4 = 20.

Если сначала выполнить умножение:

3 * 4 = 12, затем 2 + 12 = 14.

Правильный порядок – второй, и правильный ответ – 14. В математике существует так называемый *порядок операций*, который и указывает, какие операторы выполняются первыми, даже если написаны они в конце выражения.

В нашем примере, хотя символ + написан перед знаком *, умножение выполняется первым. Python следует математическим правилам и также выполняет умножение перед сложением. Ты можешь попробовать в интерактивном режиме:

```
>>> print(2 + 3 * 4)
14
```

Порядок, который использует Python, такой же, которому ты следуешь (или *будешь* следовать) на уроках математики. Сначала идут степени, затем умножение и деление, потом сложение и вычитание.

Если ты хочешь изменить порядок операций и *выполнить* что-то первым, просто заключи это действие в *скобки* (круглые), например:

```
>>> print((2 + 3) * 4)
20
```

В этот раз Python выполнил сложение 2 и 3 первым (из-за скобок), получил 5 и перемножил 5 на 4, чтобы получить ответ 20.

Опять же, то же самое происходит и в математике. Python (и другие языки программирования) следует математическим правилам и порядку операций.

Ну, хотя это и кажется странным, Python действительно старается быть умным. Чтобы понять это, тебе нужно иметь представление о целых числах и десятичных дробях. Если ты не знаешь разницы, загляни в словарик далее.

СЛОВАРИК

Целые числа – это те, которые можно легко посчитать, например 1, 2, 3, а также 0 и отрицательные числа, например –1, –2, –3.
Десятичные дроби (также называемые *действительными числами*) – это числа с десятичной точкой и цифрами после нее, например 1.25, 0.375, –101.2.
В программировании десятичные дроби называют также *числами с плавающей точкой*[1]. Потому что десятичная точка «плавает». У тебя может быть число 0.00123456 или 12345.6.

С помощью оператора `/` Python выполняет десятичное деление, или деление с плавающей точкой. То, что Картер узнал на уроке математики, было целочисленным делением, при котором мы получаем частное и остаток. (Остаток – это та часть, которая остается, если числа не делятся нацело.) У Python тоже есть операторы для этого!

Целочисленное деление: частное и остаток

Если ты хочешь выполнить целочисленное деление в Python, то можешь использовать оператор `//` для получения частного:

```
>>> print(7 // 2)
3
```

Что касается остатка, у Python есть специальный оператор для его вычисления. Он называется оператором деления по модулю и обозначается символом процента (`%`). Вот как это работает:

```
>>> print(7 % 2)
1
```

Таким образом, если ты используешь символы `//` и `%` одновременно, то можешь узнать и частное, и остаток:

```
>>> print(7 // 2)
3
>>> print(7 % 2)
1
```

[1] В России принят термин «число с плавающей запятой», тем не менее в языках программирования используются точки в качестве разделителей дробных чисел. – *Прим. перев.*

Таким образом, если 7 разделить на 2, получится 3 с остатком 1. Если же ты выполнишь деление с плавающей точкой, то получишь ответ с десятичной точкой:

```
>>> print(7 / 2)
3.5
```

Сравнение Python 3 с Python 2

В Python 2 оператор `/` ведет себя немного иначе. Если оба операнда являются целыми числами, это дает тебе коэффициент целочисленного деления.

Напомним, что все программы в этом издании предназначены для работы с Python 3, а не с Python 2.

С помощью четырех основных операций и целочисленного деления ты изучил математические операторы, которые тебе понадобятся в 99 % твоих программ. Есть еще кое-что, что мы хотим тебе показать.

Потенцирование – возведение в степень

Если ты хочешь умножить число 3 на само себя 5 раз, то можешь ввести:

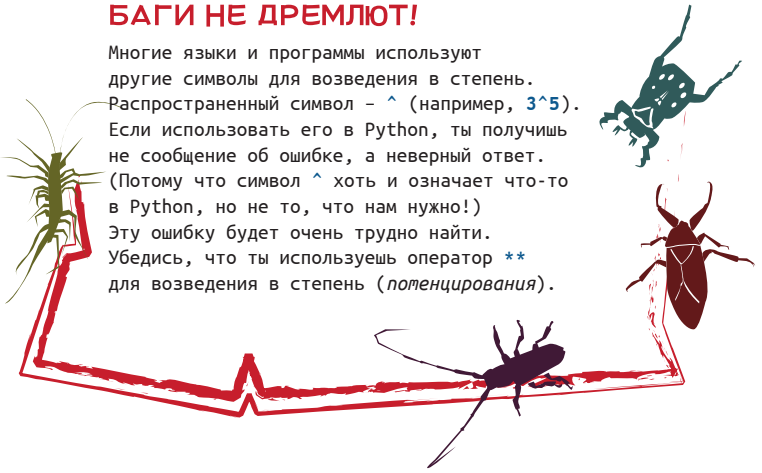
```
>>> print(3 * 3 * 3 * 3 * 3)
243
```

Но это то же самое, что и 3^5, или «три в пятой степени». В языке Python для этого используется двойная звездочка:

```
>>> print(3 ** 5)
243
```

БАГИ НЕ ДРЕМЛЮТ!

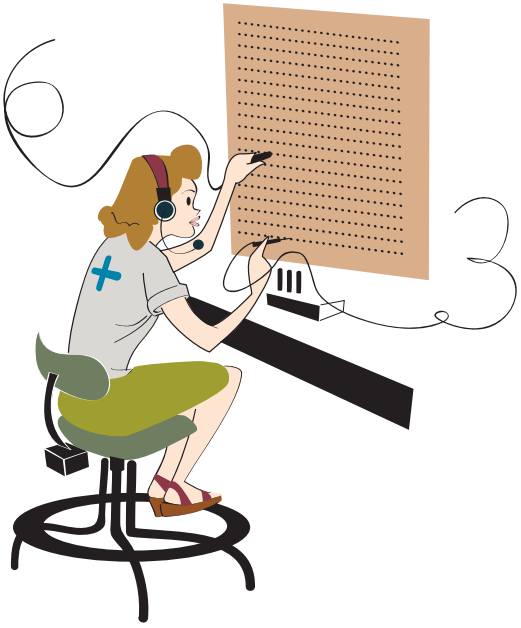
Многие языки и программы используют другие символы для возведения в степень. Распространенный символ – `^` (например, `3^5`). Если использовать его в Python, ты получишь не сообщение об ошибке, а неверный ответ. (Потому что символ `^` хоть и означает что-то в Python, но не то, что нам нужно!) Эту ошибку будет очень трудно найти. Убедись, что ты используешь оператор `**` для возведения в степень (*потенцирования*).



Одна из причин использования степеней вместо многократного умножения – упрощенный ввод. Но важнее то, что с помощью оператора `**` можно использовать степени, которые не являются целыми числами, например:

```
>>> print(3 ** 5.5)
420.888346239
```

Посредством операций умножения выполнить такое вычисление весьма сложно.



Раз уж ты упомянул, они похожи... Арифметический оператор соединяет вместе числа так же, как раньше телефонный оператор соединял телефоны.

Есть еще два оператора, о которых мы хотим тебе рассказать. Да, мы знаем, что говорили *всего* о двух, но эти очень простые!

Приращение и уменьшение

Вспомним пример из предыдущей главы: **`score = score + 1`**. Мы говорили, что это называется *приращением*. Похожее выражение **`score = score – 1`** называется *уменьшением*.

Эти действия так часто выполняются в программировании, что имеют свои собственные операторы: **`+=`** (приращение) и **`-=`** (уменьшение).

Они используются так:

```
>>> number = 7
>>> number += 1 ◄── Значение переменной увеличено на 1
>>> print(number)
8
```

и

```
>>> number = 7
>>> number -= 1 ◄── Значение переменной уменьшено на 1
>>> print(number)
6
```

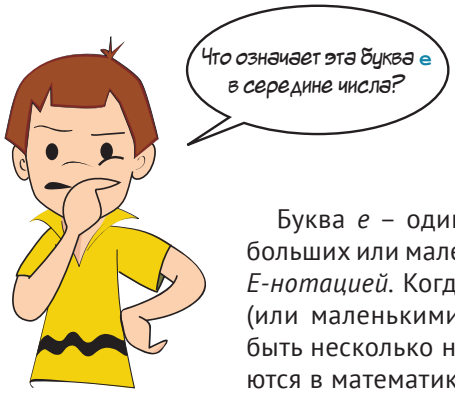
Первый оператор добавляет единицу к числу (меняет с 7 на 8). А второй отнимает единицу (меняет с 7 на 6).

Действительно большой и действительно маленький

Помнишь, в главе 1 мы умножали два больших числа друг на друга? И получали еще большее число в ответ. Иногда Python показывает большие числа немного по-другому. Попробуй:

```
>>> print(9938712345656.34 * 4823459023067.456)
4.793897174132799e+25
```

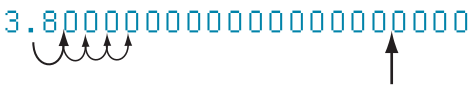
(Числа не имеют особого значения, подойдут любые большие числа с десятичной точкой.)



Буква *e* – один из способов отображения действительно больших или маленьких чисел на компьютере. Она называется *E-нотацией*. Когда мы работаем с действительно большими (или маленькими) числами, отображение всех цифр может быть несколько неудобным. Такие виды чисел часто встречаются в математике и точных науках.

Например, если бы астрономическая программа показывала количество километров от Земли до альфы Центавры, то это было бы 38000000000000000, или 38 000 000 000 000 000 (38 квинтиллионов километров!). В любом случае, ты уже устал считать нули.

Другой способ отобразить это число – использовать *научную нотацию*, или *систему представления чисел*, которая использует степень 10 вместе с десятичными числами. В научной нотации расстояние до альфы Центавра можно записать так: 3.8×10^{16}. (Видишь, что 16 находится над строкой и немного меньше по размеру?) Это выражение читается как «три и восемь умножить на десять в шестнадцатой степени», или «три и восемь умножить на десять в степени шестнадцать». Это значит, что ты берешь число 3.8 и перемещаешь десятичную точку на шестнадцать позиций вправо, добавляя необходимые нули.



Перемести десятичную точку на 16 позиций вправо

$$38000000000000000.0 = 3.8 \times 10^{16}$$

Научная нотация хороша, если ты можешь написать 16 в виде степени, то есть над строкой и меньше по размеру. Если ты работаешь с карандашом и бумагой или с программой, которая поддерживает верхние индексы, то можешь использовать эту систему представления чисел.

СЛОВАРИК

Верхний индекс – это символ или символы, которые располагаются над остальным текстом, например 10^{13}. Здесь 13 является верхним индексом. Обычно верхний индекс меньше по размеру, чем основной текст.
Нижний индекс похож на верхний, но располагается немного ниже основного текста, например $\log_2$. Здесь 2 – нижний индекс.

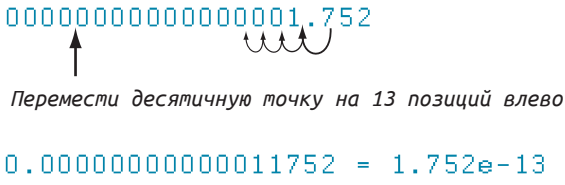
Но ты не всегда можешь использовать верхний индекс, поэтому другим способом будет E-нотация (другой способ записи научной нотации).

E-нотация

В E-нотации наше число будет выглядеть как 3.8E16 или 3.8e16. Это читается как «три и восемь в степени шестнадцать» или «три и восемь е шестнадцать». Подразумевается, что степень принадлежит числу 10. Это то же самое, что и 3.8×10^{16}.

Большинство программ и компьютерных языков, включая Python, позволяют использовать как строчную, так и прописную букву E.

Для очень малых чисел, например 0.0000000000001752, используется отрицательная степень. В научной нотации это будет выглядеть как 1.752×10^{-13}, а в E-нотации – 1.752e-13. Отрицательная степень означает смещение десятичной точки влево, а не вправо.



Ты можешь использовать E-нотацию для ввода больших или малых чисел (или любых чисел) в Python. Позже ты узнаешь, как на языке Python выводить числа с помощью E-нотации.

Попробуй ввести числа:

```
>>> a = 2.5e6
>>> b = 1.2e7
>>> print(a + b)
14500000.0
```

Хотя ты ввел числа в E-нотации, ответ был выведен в виде обычного числа с десятичной точкой. Это произошло потому, что Python не отображает числа в E-нотации, если только ты не задашь ему такую команду или числа не являются достаточно большими или маленькими (много нулей).

Попробуй:

```
>>> c = 2.6e75
>>> d = 1.2e74
>>> print(c + d)
2.72e+75
```

В этот раз Python показал ответ в E-нотации автоматически, потому что не имело смысла отображать число с 73 нулями!

Если ты хочешь, чтобы числа наподобие 14 500 000 отображались тоже в E-нотации, тебе нужно дать Python особую команду. Ты узнаешь о ней в главе 21.

Если ты не совсем понимаешь, как работает E-нотация, не волнуйся. Она не используется для программ в книге. Мы просто хотели показать тебе ее на случай, если она тебе понадобится.

По крайней мере, теперь, если будешь использовать Python для математических действий и получишь число наподобие 5.673745e16 в ответ, то будешь знать, что оно очень большое, и это не ошибка.

Степени и E-нотации

Не перепутай возведение числа в степень (потенцирование) и E-нотацию.

- 3**5 означает 3^5, или «три в пятой степени», или 3*3*3*3*3, что равняется 243.
- 3e5 означает $3*10^5$, или «трижды десять в пятой степени», или 3*10*10*10*10*10, что равняется 300 000.
- Возведение в степень означает умножение числа на само себя степень раз. E-нотация означает умножение на 10 в степени.

Некоторые читают выражения 3e5 и 3**5 как «три в пятой степени», но это два разных выражения. Не имеет большой разницы, как ты их произносишь, пока ты четко понимаешь, что означает каждое из них.

Что ты узнал?

В этой главе ты узнал:

- как выполнять базовые математические операции на языке Python;
- о целых числах и десятичных дробях;
- о потенцировании (возведении в степень);
- как получить модуль (остаток);
- все о E-нотации.

Проверь свои знания

1. Какой символ используется в языке Python для умножения?
2. Какой ответ выведет интерпретатор Python, вычислив выражение 9/5?
3. Как получить остаток от выражения 9/5?
4. Как получить десятичную дробь в ответ на выражение 9/5?
5. Каков другой способ получить ответ на выражение 6*6*6*6 в Python?
6. Как записать число 17 000 000 в E-нотации?
7. Как выглядит число 4.56e-5 в обычной системе обозначения чисел (не E-нотации)?

Попробуй самостоятельно

1. Реши следующие задачи с помощью интерактивного интерпретатора или написав небольшую программу:
   a) Три человека поужинали в ресторане и хотят разделить счет. Общая сумма 3527 рублей, и они хотят оставить 15 % на чаевые. Сколько должен заплатить каждый?
   b) Посчитай площадь и периметр прямоугольной комнаты размером 12,5 м на 16,7 м.
2. Напиши программу для конвертации температуры из градусов Фаренгейта в градусы Цельсия. Формула такова: C = 5/9*(F – 32). (Подсказка: берегись багов с делением целых чисел!)
3. Ты знаешь, как посчитать время, необходимое для того, чтобы добраться куда-либо на машине? Формула: «время пути равно расстоянию, разделенному на скорость». Напиши программу для подсчета времени путешествия на расстояние 200 км при скорости 80 км/ч.

ГЛАВА 4

Типы данных

Мы уже знаем, что переменной можно присвоить, по крайней мере, три типа значений (для хранения в памяти компьютера): целые числа, числа с десятичной точкой и строки. В Python есть и другие типы данных, о которых ты узнаешь позже, а сейчас нам достаточно и этих трех. В этой главе ты научишься определять тип объекта. Ты также научишься отличать один тип от другого.

Изменение типа

Довольно часто требуется конвертировать данные из одного типа в другой. Например, когда мы хотим вывести число, оно должно быть переведено в текст, чтобы текст появился на экране. Команда `print` в Python делает это за нас, но иногда нужно изменить тип без вывода или конвертировать строки в числа (чего команда `print` не может). Это называется *изменением типа*. И как это работает?

Фактически Python не конвертирует объект из одного типа в другой. Он создает новый объект того типа, который нам нужен, из оригинального объекта. Ниже представлены некоторые функции, которые меняют тип:

- `float()` создает новое число с плавающей точкой из строки или целого числа;
- `int()` создает новое целое число, или строки, или числа с плавающей точкой;
- `str()` создает новую строку из числа любого типа.

Скобки в конце имен функций **`float()`**, **`int()`** и **`str()`** нужны потому, что они не являются *ключевыми словами* Python (как **`print`**) – это встроенные *функции* Python. На самом деле ты уже знаком с одной встроенной функцией – **`print()`**!

Ты узнаешь больше о функциях немного позже. Сейчас тебе нужно только знать, что значение, которое ты хочешь изменить, нужно писать *внутри* скобок. Лучше всего это будет понятно на примерах. Давай запустим интерпретатор IDLE.

Изменение `int` на `float`

Давай начнем с целого числа и создадим новое число с плавающей точкой (десятичную дробь) из него с помощью функции **`float()`**.

```
>>> a = 24
>>> b = float(a)
>>> a
24
>>> b
24.0
```

Обрати внимание, что значение **`b`** приобрело десятичную точку и ноль в конце. Это говорит о том, что число стало десятичной дробью и не является более целым. Переменная **`a`** не изменилась, потому что функция **`float()`** не меняет оригинальное значение, а создает новое.

Напомним, что в интерактивном режиме ты можешь просто ввести имя переменной (без команды **`print`**), и Python отобразит ее значение. (Ты видел это в главе 2.) Прием сработает только в интерактивном режиме, не в программе.

Изменение `float` на `int`

А теперь давай попробуем обратное действие: начнем с десятичной дроби и создадим целое число с помощью функции **`int()`**:

```
>>> c = 38.0
>>> d = int(c)
>>> c
38.0
>>> d
38
```

Мы создали новое целое число **`d`**, которое является *недробной* частью числа **`c`**.


```
Python 3.7.4 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 0.1 + 0.2
0.30000000000000004
>>>
                                                                    Ln: 5  Col: 4
```

Ой! Как это случилось? Картер, я думаю, твой компьютер сходит с ума! Шутка. На самом деле этому есть объяснение, которое ты можешь прочесть ниже.

ЧТО ТАМ ПРОИСХОДИТ?

Помнишь, мы говорили, что компьютеры используют двоичный код? Итак, все числа, которые хранит Python, хранятся в виде двоичного кода. Для суммы чисел 0.1 и 0.2 Python создал число с плавающей точкой с достаточным количеством двоичных цифр (битов), чтобы выдать нам 15 десятичных позиций. Но это двоичное число *не совсем* равно 0.3, оно очень, очень близко к нему. (В этом случае погрешность составляет 0.000000000000004.)

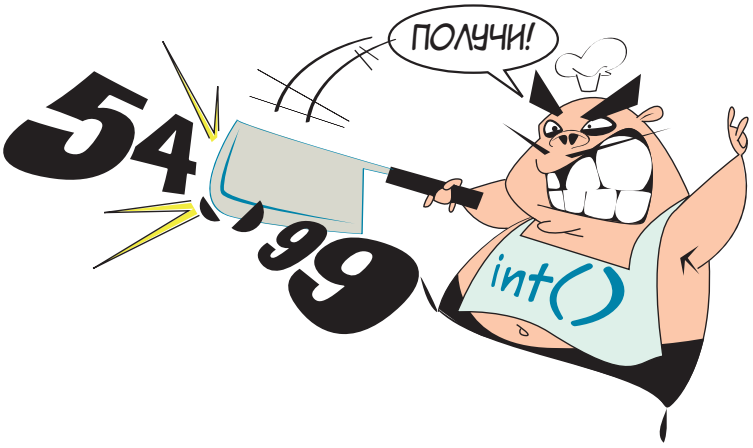
Эта разница называется *ошибкой округления*. Ошибки округления случаются с десятичными дробями во всех языках программирования. Количество точных цифр может варьироваться в зависимости от компьютера или языка, но они все используют один базовый метод хранения чисел с десятичной точкой.

Обычно эта ошибка настолько незначительна, что о ней не стоит беспокоиться.

Давай попробуем еще раз:

```
>>> e = 54.99
>>> f = int(e)
>>> e
54.99
>>> f
54
```

Хотя число 54.99 очень близко к 55, ты все равно получаешь 54 в виде целого числа. Функция **`int()`** всегда округляет. Она показывает не *ближайшее* число, а *следующее меньшее* целое число. Она просто «отрубает» десятичную часть.



Если ты хочешь получить ближайшее целое число, для этого есть способ. Ты узнаешь о нем в главе 21.

Изменение строки в число с десятичной точкой

Мы также можем создать число из строки:

```
>>> a = '76.3'
>>> b = float(a)
>>> a
'76.3'
>>> b
76.3
```

Обрати внимание, что когда мы отобразили значение переменной **`a`**, результат был в кавычках. Это способ языка Python показать нам, что переменная является строкой. Когда мы отобразили значение переменной **`b`**, то увидели число с плавающей точкой и десятичными позициями (как и Картер ранее).

Как получить больше информации: `type()`

В последнем примере мы положились на кавычки, чтобы узнать, является ли значение переменной числом или строкой. Но есть более прямой путь это выяснить.

В Python есть функция **`type()`**, которая точно говорит нам вид переменной.

Давай попробуем:

```
>>> a = '76.3'
>>> b = float(a)
>>> a
'76.3'
>>> b
76.3
```

Функция **`type()`** рассказала нам, что переменная **`a`** имеет вид строки (**`str`**), а переменная **`b`** – числа с плавающей точкой (**`float`**). Больше никаких угадываний!

Ошибки при изменении типа

Конечно, если с функциями **`int()`** или **`float()`** использовать не число, то они не сработают. Проверь:

```
>>> float('филипп')
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    float('филипп')
ValueError: could not convert string to float: 'филипп'
```

Мы получили сообщение об ошибке. Неверное, текстовое значение (**`invalid literal`**) означает, что Python не знает, как создать число из слова **`филипп`**. А ты знаешь?

Что ты узнал?

В этой главе ты узнал:

- об изменении типов (или *создании* типов из других типов): **`str()`**, **`int()`** и **`float()`**;
- как отображать значение переменной без команды **`print`**;
- как проверить тип переменной с помощью функции **`type()`**;
- об ошибках округления и причинах их возникновения.

Проверь свои знания

1. Когда ты используешь функцию `int()`, чтобы конвертировать десятичную дробь в целое число, ты получаешь округленный результат или нет?
2. Если ты введешь `thing1` в командную строку и получишь ответ `'4'`, то каким будет ответ функции `type(thing1)`?
3. (Дополнительный вопрос) Без использования каких-либо функций, кроме `int()`, как получить округленное число, а не округленное с уменьшением? (Например, 13.2 будет округлено с уменьшением до 13, а 13.7 будет округлено до 14.)

Попробуй самостоятельно

1. Используй функцию `float()`, чтобы создать число из строки наподобие `'12.34'`. Убедись, что результат будет числом!
2. Попробуй с помощью функции `int()` создать целое число из десятичной дроби наподобие `56.78`. В какую сторону был округлен ответ?
3. Попробуй с помощью функции `int()` создать целое число из строки. Убедись, что результат является целым числом!

ГЛАВА 5

Ввод

До сих пор, если тебе нужно было произвести действие над какими-либо двумя числами, тебе нужно было помещать эти числа прямо в код. Например, когда ты писал код программы для конвертации температуры в главе 3, то, вероятнее всего, вводил значение температуры для конвертации прямо в код программы. Если тебе нужно было конвертировать другое значение, ты менял код.

А что, если нам нужно, чтобы значение температуры для конвертации вводил пользователь при работающей программе? Мы говорили, что у программы есть три компонента: ввод, обработка и вывод. В нашей первой программе был только вывод. В программе по конвертации температур присутствовала обработка данных (температуры) и вывод, но не ввод. Пришло время добавить третий ингредиент в наши программы: *ввод*. Ввод означает получение чего-то, какой-либо информации программой во время ее работы. Таким образом, мы можем писать программы, взаимодействующие с пользователем, что намного интереснее.

В Python есть встроенная функция, которая называется `input()` и используется для получения информации от пользователя. В этой главе мы научимся ее применять в программах.

Функция `input()`

Функция **`input()`** получает от пользователя строку. Обычно она получает ее с клавиатуры, то есть пользователь вводит информацию.

Эта функция – одна из встроенных функций Python, как и **`str()`**, **`int()`**, **`float()`** и **`type()`**. (Мы о них говорили в главе 4.) Далее ты узнаешь много нового о функциях. Но сейчас тебе нужно лишь не забывать о скобках, когда ты пользуешься **`input()`**.

Ниже показано, как использовать функцию:

```
someName = input()
```

Это позволит пользователю ввести строку под именем **`someName`**.

Теперь давай введем функцию в программу. Создай новый файл в интерпретаторе IDLE и введи код из листинга 5.1.

Листинг 5.1. Получение строки с помощью функции **`input()`**

```
print("Введи свое имя: ")
somebody = input()
print(«Привет», somebody, «как у тебя дела?»)
```

Сохрани код в файл и запусти программу, чтобы посмотреть, как она работает. Ты увидишь нечто подобное:

```
Введи свое имя:
Саша
Привет Саша как у тебя дела?
```

Мы ввели имя, и программа присвоила ему имя **`somebody`**.

Использование функции `input()` в одной строке

Обычно, когда ты хочешь получить ввод информации от пользователя, тебе нужно уточнить, что именно требуется, коротким сообщением наподобие этого:

```
print(«Введи свое имя: «)
```

Тогда ты можешь получить ответ с помощью функции **`input()`**:

```
somebody = input()
```

Если запустить этот код в программе и ввести свое имя, то результат выглядит так:

```
Введи свое имя:
Саша
```

Если ты хочешь, чтобы пользователь вводил ответ в той же строке, где и появилось сообщение, просто добавь **`end=''`** в конце своего запроса:

```
print(«Введи свое имя: «, end='')
someName = input()
```

Если запустить программу с таким кодом, она будет выглядеть так:

```
Введи свое имя: Саша
```

Обычно функция **`print()`** выполняет то же, что и нажатие клавиши **Enter** в конце строки. Добавив **`end=''`**, мы сообщаем функции **`print ()`**, чтобы она ничего не делала в конце строки. Таким образом, следующий отображаемый элемент будет находиться в той же строке.

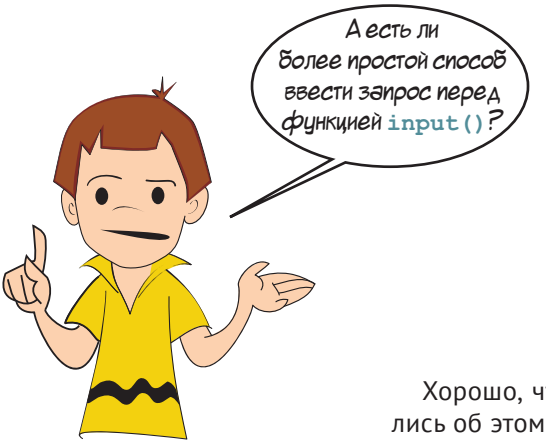
Попробуй ввести следующий код в редакторе IDLE и запустить его.

Листинг 5.2. Как действует `end=''`?

```
print("Меня ", end='')
print("зовут ", end='')
print(«Миша.», end='')
```

В результате ты должен получить:

```
Меня зовут Миша.
```



Хорошо, что ты спросил! Мы как раз собирались об этом рассказать.

Краткий вариант запроса `input()`

Есть более короткий путь для вывода запроса. Функция **`input()`** может сама вывести на экран сообщение, так что тебе не придется использовать команду **`print()`**:

```
someName = input(«Введи свое имя: «)
```

Это похоже на то, что в функцию **`input()`** уже встроена команда **`print()`**. Теперь мы будем использовать этот способ.

Ввод чисел

Ты уже видел, как функция **`input()`** работает со строками. Но что, если ты захочешь получить число? В конце концов, мы начали говорить о вводе для того, чтобы позволить пользователю задавать значение в программе по конвертации температур.

Ты уже знаешь ответ, если читал главу 4. Мы можем использовать функции **`int()`** или **`float()`**, чтобы создать число из строки, которую нам предоставит функция **`input()`**. Это будет выглядеть так:

```
temp_string = input()
fahrenheit = float(temp_string)
```

Мы получили информацию от пользователя в виде строки с помощью **`input()`**. Затем мы сделали из этой информации число с помощью функции **`float()`**. И присвоили температуре в виде десятичной дроби имя **`fahrenheit`**.

Но мы можем выполнить все это одной строкой:

```
fahrenheit = float(input())
```

Этот способ дает точно такой же результат. Он получает строку от пользователя, создает из нее число. То же самое при более коротком коде.

Теперь давай используем это в нашей программе по конвертации температуры. Попробуй набрать код из листинга 5.3 и посмотри, что у тебя выйдет.

Листинг 5.3. Конвертация температур с помощью функции **input()**

```
print(«Эта программа конвертирует температуру по Фаренгейту
      в градусы Цельсия.»)
fahrenheit = float(input(«Введи температуру по Фаренгейту: «))
celsius = (fahrenheit – 32) * 5.0 / 9
print(“равно “, end=’’)
print(celsius, end=’’)
print(“ градусов Цельсия.”)
```

Используем **float(input())** *для получения значения температуры по Фаренгейту от пользователя*

Ты можешь совместить три последние строки программы в одну:

```
print(«равно «, celsius, « градусов Цельсия.»)
```

Это сокращение для трех запросов **print()**.

Использование функции int() вместе с input()

Если число, вводимое пользователем, всегда будет целым (без десятичных дробей), ты можешь конвертировать его с помощью функции **int()**:

```
response = input(«Сколько учеников в твоем классе: «)
numberOfStudents = int(response)
```

Ввод из Всемирной паутины

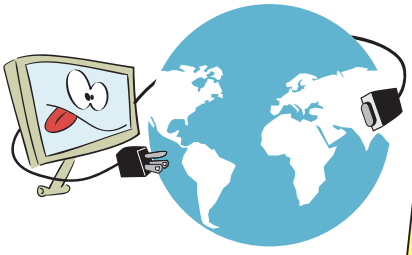
Обычно ты получаешь данные в программе от пользователя. Но есть и другие источники. Ты можешь получить данные из файла на твоем компьютере (мы научимся делать это в главе 22) или из Всемирной паутины.

Если твой компьютер подключен к интернету, ты можешь исполнить программу из листинга 5.4. Она открывает файл с сайта и показывает сообщение из этого файла.

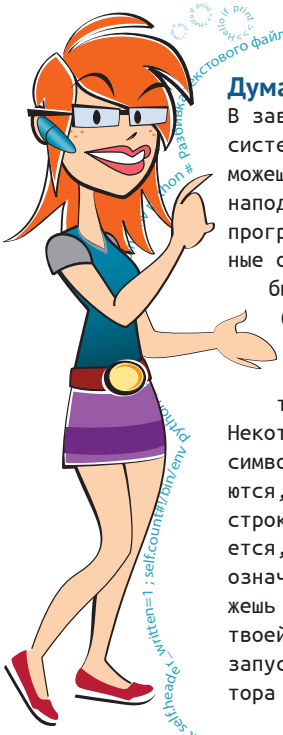
Листинг 5.4. Получение информации из файла во Всемирной паутине

```
import urllib.request
file = urllib.request.urlopen(‘http://spliff-guru.ru/message.txt’)
message = file.read().decode(‘utf-8’)
print(message)
```

И все. Всего лишь с помощью четырех строк кода твой компьютер получил из Всемирной паутины файл и вывел его содержимое на экран. Если ты попробуешь запустить эту программу (при активном подключении к интернету), то увидишь сообщение.



Если ты пробуешь запустить эту программу на школьном или рабочем компьютере, то она может не сработать. Потому что некоторые школы или офисы используют прокси-сервер для подключения к интернету. Прокси-сервер – это другой компьютер, который работает посредником или шлюзом между интернетом и твоей школой или офисом. В зависимости от настроек прокси-сервера эта программа может не подключиться к интернету через него. Если же ты запустишь программу на твоем домашнем компьютере, то она должна работать.



Думай как программист

В зависимости от используемой операционной системы (Windows, Linux или macOS) ты можешь увидеть маленькие квадратики или что-то наподобие `\r` в конце каждой строки при запуске программы из листинга 5.4. Причина в том, что разные операционные системы используют разные способы обозначения конца строки в тексте. Windows (и MS-DOS) использует два символа: CR (Carriage Return, возврат каретки) и LF (Line Feed, перевод строки). Linux использует только LF. В macOS применяется лишь CR. Некоторые программы могут справиться с любым символом, а некоторые, например IDLE, путаются, когда не видят именно те символы окончания строки, которых они ожидали. Когда такое случается, они отображают маленькие квадратики, что означает: «Я не понимаю этот символ». Ты можешь увидеть эти квадратики в зависимости от твоей операционной системы и от того, как ты запускаешь программу (с помощью интерпретатора IDLE или нет).

Что ты узнал?

В этой главе ты узнал:

- как вводить текст с помощью функции **`input()`**;
- как добавлять сообщение с запросом в **`input()`**;
- как вводить числа с помощью функций **`int()`** и **`float()`** с **`input()`**;
- как выводить несколько значений в одной строке с помощью запятой.

Проверь свои знания

1. При таком коде:

   ```
   answer = input()
   ```

   если пользователь введет значение 12, каким типом данных будет значение переменной **`answer`**? Это строка или число?
2. Как с помощью функции **`input()`** вывести на экран запрос для нее?
3. Как получить целое число с помощью функции **`input()`**? Как получить десятичную дробь с ее помощью?

Попробуй самостоятельно

1. В интерактивном режиме создай две переменные, одну для своего имени и одну для фамилии. Затем с помощью одной команды **`print()`** выведи на экран свои имя и фамилию вместе.
2. Напиши программу, которая спрашивает твое имя, затем фамилию, а потом выводит сообщение с твоими именем и фамилией.
3. Напиши программу, которая спрашивает размеры (в метрах) прямоугольной комнаты, затем подсчитывает и выводит на экран количество ковролина, необходимого, чтобы покрыть пол во всей комнате.
4. Напиши программу, которая выполняет то же, что и программа в пункте 3, но также запрашивает цену ковролина. Затем попробуй получить следующие результаты:
   - общее количество ковролина в квадратных метрах;
   - общую стоимость ковролина.
5. Напиши программу, которая помогает пользователю посчитать сдачу. Она должна спрашивать:
   - «Сколько у тебя 10-рублевых монет?»
   - «Сколько у тебя 5-рублевых монет?»
   - «Сколько у тебя 2-рублевых монет?»
   - «Сколько у тебя 1-рублевых монет?»

   Затем должен быть получен ответ, сколько у тебя всего мелочи.

ГЛАВА 6

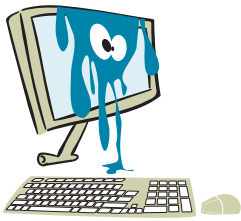
Графические пользовательские интерфейсы

До сих пор весь наш ввод и вывод информации имел вид простого текста в окне интерпретатора IDLE. Но современные компьютеры и программы используют графику, много графики. Было бы хорошо, если бы мы могли ввести ее в свои программы. В этой главе мы начнем разрабатывать простые графические интерфейсы (GUI). Это значит, что наши программы начнут выглядеть более похожими на то, к чему ты привыкли: окна, кнопки и т. д.

Что такое GUI?

GUI – это сокращение, означающее *графический пользовательский интерфейс* (graphical user interface). В GUI вместо ввода текста с клавиатуры и получения текста в ответ пользователь видит графические объекты – окна, кнопки, текстовые блоки и т. д. – и может использовать мышь, чтобы щелкать по объектам, или клавиатуру для ввода команд. Программы, которые мы писали до сих пор, – это *консольные*, или *текстовые*, *программы*. Графический интерфейс – это просто другой способ взаимодействия с программой. Программы с GUI также содержат три базовых элемента: ввод, обработку и вывод. Просто их ввод и вывод выглядят немного привлекательнее.

Кстати, сокращение GUI обычно произносится как «гуи», а не по буквам. GUI на компьютере полезен, а вот клея лучше избегать, иначе будет трудно печатать: клавиши прилипают (игра слов: англ. *gooey* – липкий, клейкий).



Наш первый GUI

Мы уже использовали GUI, даже несколько. Браузер для выхода в интернет – это GUI. IDLE – это GUI. А теперь мы создадим свой собственный графический интерфейс. Для этого нам потребуется помощь под названием EasyGUI.

EasyGui – это модуль Python, который позволяет очень легко создавать простые графические интерфейсы. Мы еще не говорили о модулях (мы будем говорить о них в главе 15), но модуль – это способ добавления чего-либо, еще не встроенного в Python.

Если ты установил Python с помощью инсталлятора этой книги, у тебя уже есть EasyGUI. Если нет, ты можешь загрузить его по адресу **easygui.sourceforge.net**.

Приступим

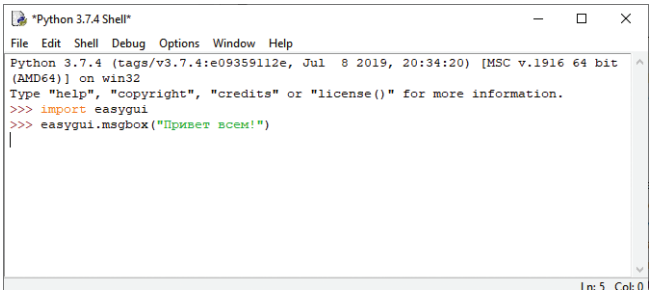
Запусти оболочку IDLE и введи следующее в интерактивном интерпретаторе:

```
>>> import easygui
```

Эта команда говорит Python, что ты собираешься использовать модуль EasyGUI. Если ты не получил сообщение об ошибке, значит, модуль найден. Если ты получил сообщение об ошибке или EasyGUI не работает, перейди на сайт книги (**www.dmkpress.com**), чтобы получить дополнительную информацию.

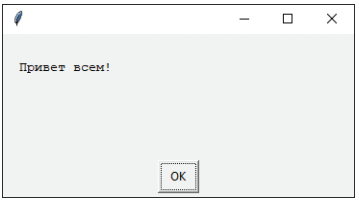
А теперь давай сделаем простое окно с кнопкой **OK**:

```
>>> easygui.msgbox(«Привет всем!»)
```



Функция **`msgbox()`** в EasyGUI используется для создания окна сообщений. В большинстве случаев имена функций EasyGUI представляют собой сокращенные версии английских слов.

Когда ты используешь функцию **`msgbox()`**, то получаешь подобный результат:



Если ты щелкнешь по кнопке **OK**, окно сообщения закроется.

Ввод с помощью GUI

Только что ты видел вывод с помощью GUI – окно с сообщением. А как насчет ввода? Ты также можешь выполнить ввод информации с помощью EasyGUI.

Когда ты выполнял предыдущий пример в интерактивном режиме, ты нажал кнопку **OK**? Если да, ты должен был увидеть что-то подобное в окне интерпретатора:

```
>>> import easygui
>>> easygui.msgbox(«Привет всем!»)
'OK'
```

Строка с буквами **`'OK'`** как раз и сообщает, что пользователь нажал кнопку **OK**. EasyGUI выдает тебе ответную информацию о том, что сделал пользователь в графическом интерфейсе: какую кнопку нажал, что ввел и т. д. Ты можешь присвоить этому ответу имя (присвоить его переменной). Попробуй:

```
>>> user_response = easygui.msgbox(«Привет всем!»)
```

Щелкни по кнопке **OK** в окне сообщения, чтобы закрыть его. Затем набери:

```
>>> print(user_response)
OK
```

Теперь ответ пользователя, **`OK`**, имеет имя переменной **`user_response`**. Давай взглянем на некоторые другие способы получить ввод с EasyGUI.

Окно с сообщением, которое ты только что видел, – один из примеров так называемых *диалоговых окон*. Диалоговые окна – элементы графических интерфейсов, которые используются для подачи информации пользователю или получения информации от него. Вводом может быть кнопка (например, **OK**), имя файла или текст (строка).

`Msgbox` в EasyGUI – просто диалоговое окно с сообщением и одной кнопкой, **OK**. Но мы можем создавать разные диалоговые окна с несколькими кнопками и другими объектами.

Выбери на свой вкус

Мы используем пример выбора любимого мороженого, чтобы изучить разные способы получения ввода (вкус мороженого) от пользователя с помощью EasyGUI.

Диалоговое окно с несколькими кнопками

Давай сделаем диалоговое окно (похожее на окно с сообщением) с более, чем одной кнопкой. Это можно выполнить с помощью функции **`buttonbox`**. Давай напишем еще одну программу, но не в интерактивном режиме.

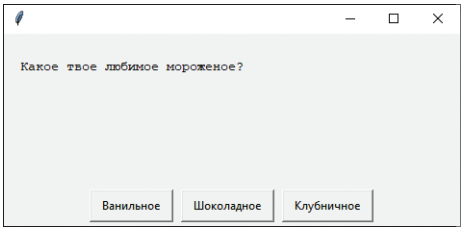
Создай новый файл в IDLE. Введи код программы из листинга 6.1.

Листинг 6.1. Получение ввода с помощью кнопок

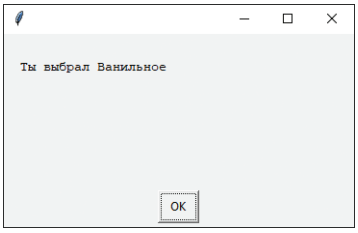
```
import easygui
flavor = easygui.buttonbox(«Какое твое любимое мороженое?»,
                 choices = [‘Ванильное’, ‘Шоколадное’, ‘Клубничное’] )
easygui.msgbox(«Ты выбрал « + flavor)
```

Список вариантов выбора

Сохрани файл (я назвал свой *ice_cream1.py*) и запусти его. Ты увидишь следующее:



А затем, в зависимости от выбранного вкуса, ты увидишь нечто подобное:



Как это работает? Надпись с выбранной пользователем кнопки и была *вводом*. Мы присвоили вводу имя переменной – в данном случае **`flavor`**. Это похоже на использование функции **`input()`**, но пользователь не вводит информацию с клавиатуры, а просто щелкает по кнопкам. В этом вся соль GUI.

Окно выбора

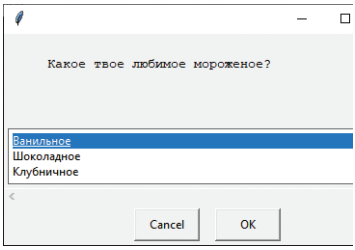
Давай попробуем реализовать другой способ выбора вкуса мороженого. В EasyGUI есть *окно выбора* (**choicebox**), которое представляет собой список вариантов выбора. Пользователь выбирает один из них и щелкает по кнопке **OK**.

Чтобы реализовать этот вариант, нужно внести одно маленькое изменение в нашу программу: сменить значение **buttonbox** на **choicebox**. Новую версию кода ты увидишь в листинге 6.2.

Листинг 6.2. Получение ввода с помощью окна выбора

```
import easygui
flavor = easygui.choicebox(«Какое твое любимое мороженое?»,
                 choices = [‘Ванильное’, ‘Шоколадное’, ‘Клубничное’] )
easygui.msgbox(«Ты выбрал « + flavor)
```

Сохрани программу из листинга 6.2 и запусти ее. Ты увидишь нечто подобное:



После того как ты выберешь вкус и щелкнешь по кнопке **OK**, ты увидишь такое же сообщение, как и раньше. Обрати внимание, что помимо выбора вкуса мышью ты можешь использовать клавиши ↑ и ↓ на клавиатуре.

Если ты щелкнешь по кнопке **Cancel** (Отмена), программа завершится, и ты увидишь сообщение об ошибке. Оно появляется потому, что в последней строке программа ожидает увидеть текст (например, «Ванильное»), а ты нажал кнопку **Cancel**, и программа его не получила.

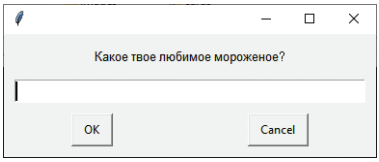
Текстовый ввод

Примеры в этой главе позволяют делать выбор из списка вариантов, которые ты как программист предоставил. А что, если тебе нужно что-то, более похожее на функцию **input()**, когда пользователь вводит текст? Таким образом, он может ввести любой вкус мороженого, который ему нравится. В EasyGUI есть функция **enterbox**, которая позволяет это реализовать. Попробуй программу из листинга 6.3.

Листинг 6.3. Получение ввода с помощью окна ввода

```
import easygui
flavor = easygui.enterbox(«Какое твое любимое мороженое?»)
easygui.msgbox(«Ты выбрал « + flavor)
```

При выполнении программы ты увидишь что-то подобное:



А затем, когда ты введешь свой любимый вкус и щелкнешь по кнопке **OK**, программа выведет такое же сообщение, как и раньше.

Эта функция работает как **`input()`**. Она получает от пользователя текст (строку).

Ввод по умолчанию

Иногда, когда пользователь вводит информацию, есть точный ответ, которого ожидает программа, наиболее вероятный или часто встречающийся. Это называется *ответом по умолчанию*. Ты можешь сэкономить время для ввода информации пользователем, автоматически введя самый ожидаемый от него ответ. При этом он будет вводить ответ сам, только если он отличается от ответа по умолчанию.

Чтобы добавить ответ по умолчанию в окно ввода, измени код программы согласно листингу 6.4.

Листинг 6.4. Как создавать переменные по умолчанию

```
import easygui
flavor = easygui.enterbox(«Какое твое любимое мороженое?»,
                          default = 'ванильное')
easygui.msgbox(«Ты выбрал « + flavor)
```

Здесь указано значение по умолчанию

Теперь при запуске программы слово «Ванильное» уже будет набрано в текстовом поле. Ты можешь удалить его и ввести другой ответ, но если твое любимое мороженое – ванильное, тебе не нужно ничего набирать, достаточно сразу нажать кнопку **OK**.

Как насчет чисел?

Если ты хочешь ввести число в EasyGUI, ты всегда можешь использовать окно ввода для получения строки, а затем создать из нее число с помощью функции **`int()`** или **`float()`** (как мы делали в главе 4).

В EasyGUI также есть функция **`integerbox`**, которую можно использовать для ввода целых чисел. Ты можешь установить верхний и нижний пределы числа, которое можно ввести.

Эта функция не позволяет вводить десятичные дроби. Чтобы ввести дробь, тебе нужно использовать окно ввода (**`enterbox`**), получить строку и использовать функцию **`float()`**, чтобы конвертировать ее в десятичную дробь.

И снова игра по угадыванию чисел

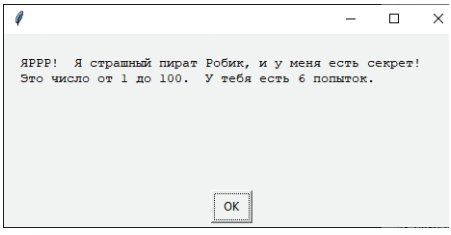
В главе 1 мы написали простую программу по угадыванию чисел. Теперь давай сделаем то же самое, но с помощью EasyGUI. Код приведен в листинге 6.5.

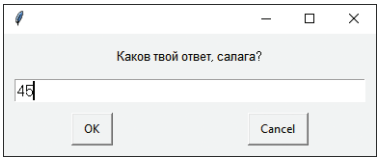
Листинг 6.5. Игра по угадыванию чисел с использованием EasyGUI

```
import random, easygui
secret = random.randint(1, 100)  ← Выбираем секретное число
guess = 0
tries = 0                                                    Получаем ввод от игрока
easygui.msgbox("""Яррр!  Я страшный пират Робик, и у меня есть секрет!
Это число от 1 до 100.  Я даю тебе 6 попыток.""")
while guess != secret and tries < 6:
    guess = easygui.integerbox(«Каков твой ответ, салага?», upperbound=100)
    if not guess: break
    if guess < secret:
        easygui.msgbox(str(guess) + « слишком мало, корабельная крыса!»)
    elif guess > secret:
        easygui.msgbox(str(guess) + « слишком много, каналья!»)
    tries = tries + 1                                  Максимальное число попыток – 6
if guess == secret:          Расходование одной попытки    Вывод сообщения в конце игры
    easygui.msgbox(«Отставить! Ты выиграл!  Узнал мой секрет, да!»)
else:
    easygui.msgbox(«Больше никаких попыток!  Правильный ответ: « + str(secret))
```

Ты можешь спросить: «Зачем нужно писать **`upperbound=100`** в **`integerbox()`**?» А затем, что **`integerbox`** EasyGUI автоматически установил верхний предел до 99. Но ты можешь изменить его до 100.

Мы до сих пор не знаем, как работают все части этой программы, так что просто набери ее и попробуй. При запуске ты увидишь следующее:





Ты познакомишься с ключевыми словами **`if`**, **`else`** и **`elif`** в главе 7, а с **`while`** – в главе 8. О **`random`** ты узнаешь больше в главе 15 и будешь использовать это ключевое слово в главе 23.

Другие части GUI

В EasyGUI доступны и другие фрагменты графических интерфейсов, включая окно выбора с возможностью выбора нескольких вариантов ответа и некоторых особых диалоговых окон для получения имен файлов и т. д. Но тех, которые мы рассмотрели, пока достаточно.

EasyGUi упрощает создание некоторых простых графических интерфейсов и скрывает большинство сложностей GUI, поэтому тебе не нужно о них переживать. Позже ты узнаешь о другом способе создания графических интерфейсов, который дает больше гибкости и контроля над ситуацией.

Если хочешь узнать больше об EasyGUI, посети страницу **easygui.sourceforge.net**.

Думай как программист (на Python)

Если ты хочешь узнать больше о Python, например об EasyGUI (и других модулях), в нем есть встроенная система справки.

В интерактивном режиме можно ввести:

```
>>> help()
```

чтобы попасть в систему справки. Командная строка изменит свой внешний вид на:

```
help>
```

Перейдя в режим справки, просто введи название элемента, о котором хочешь узнать, например:

```
help> time.sleep
```

или

```
help> easygui.msgbox
```

– и ты получишь информацию.

Чтобы выйти из режима справки и вернуться к обычной интерактивной командной строке, просто введи команду **quit**:

```
help> quit
>>>
```

Некоторые материалы в справке тяжело читать и понимать, и ты не всегда найдешь то, что тебе нужно. Но если ты ищешь дополнительную информацию о Python и его элементах, стоит попробовать.

Что ты узнал?

В этой главе ты узнал:

- как создавать простые интерфейсы GUI с помощью EasyGUI;
- как выводить сообщения с помощью окна сообщений **`msgbox`**;
- как получать ввод с помощью кнопок, окон выбора и текстовых окон ввода: **`buttonbox`**, **`choicebox`**, **`enterbox`**, **`integerbox`**;
- как устанавливать значение по умолчанию для текстового окна ввода;
- как использовать встроенную систему справки Python.

Проверь свои знания

1. Как вызвать окно с сообщением с помощью EasyGUI?
2. Как получить строку (текст) в качестве ввода с помощью EasyGUI?
3. Как получить целое число в качестве ввода с помощью EasyGUI?
4. Как получить десятичную дробь в качестве ввода с помощью EasyGUI?
5. Что такое значение по умолчанию? Приведи пример, где его можно использовать.

Попробуй самостоятельно

1. Попробуй изменить программу по конвертации температур из главы 5 таким образом, чтобы она использовала интерфейс GUI для ввода и вывода информации вместо **`input()`** и **`print()`**.
2. Напиши код программы, которая спрашивает твое имя, номер дома, улицу, город и область, индекс (все с помощью диалоговых окон EasyGUI). Затем программа должна выводить полностью твой почтовый адрес, который выглядит примерно так:

```
Петя Иванов
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026
```

ГЛАВА 7

Вычисления

В нескольких первых главах мы познакомились с несколькими основными компонентами программ. Теперь мы можем написать программу с *вводом, обработкой и выводом*. Мы даже можем сделать ее красивее с помощью интерфейса GUI. Мы можем назначить вводу переменную, чтобы использовать ее позднее, и можем обработать ее с помощью математических действий. Теперь ты начнешь узнавать способы управления программой.

Если бы программа выполняла одно и то же постоянно, это было бы скучно и не очень полезно. Программы должны уметь принимать *решения* в зависимости от условий. В этой главе мы добавим в наш репертуар *обработки* некоторые техники принятия решений.

Тестирование

Программы должны уметь совершать разные действия, основываясь на информации ввода. Ниже представлено несколько примеров:

- *Если* Толя дал правильный ответ, добавить ему 1 очко.
- *Если* Женя ударил пришельца, озвучить хлопок.
- *Если* файл отсутствует, вывести сообщение об ошибке.

Чтобы принимать решения, программы проверяют (*тестируют*), выполнено ли необходимое *условие*. В первом примере условием был «правильный ответ».

В Python есть всего несколько способов проверить что-то, и есть только два возможных ответа на каждый тест: правда или ложь (*true* или *false*).

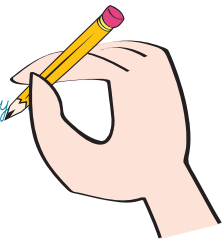
Ниже представлены вопросы, которые может задать Python, чтобы проверить условие:

- Одинаковы ли два объекта?
- Меньше ли один объект, чем другой?
- Больше ли один объект, чем другой?

Подожди, но условие «правильный ответ» – это не тот тест, который мы можем провести, по крайней мере только косвенно. Это значит, что мы должны описать тест таким образом, чтобы Python его понял.

Когда мы хотим узнать, дал ли Толя правильный ответ, мы, очевидно, знаем этот ответ и ответ Толи. Мы можем написать примерно так:

Если ответ Толи идентичен правильному ответу



Если Толя дал правильный ответ, тогда две переменные будут одинаковы, и *условие* будет *истинным* (**`true`**). Если он дал неверный ответ, две переменные будут неодинаковыми, и *условие* будет *ложным* (**`false`**).

СЛОВАРИК

Проведение тестов и принятие решений на основе результатов называется *ветвлением*. Программа решает, каким путем идти или какой ветке следовать, основываясь на результате теста.

Python использует ключевое слово **`if`** для проверки условий, например:

```
if timsAnswer == correctAnswer:
    print(“Ты ответил правильно!”)
    score = score + 1
print “Спасибо за игру.”
```

Эти строки создают «блок» кода, потому что они выделены отступами по сравнению со строками сверху и снизу

СЛОВАРИК

Блок кода – это одна или более строк кода, которые объединены вместе. Они все относятся к определенной части программы (например, утверждение **if**). В Python блоки кода формируются с помощью отступов перед строками кода в блоке.

Двоеточие в конце строки с условием **if** говорит Python о том, что далее идет *блок* инструкций. Блок включает в себя каждую строку, которая выделена отступом по отношению к строке с **if**, вплоть до следующей невыделенной строки.

СЛОВАРИК

Выделение отступами означает, что строка кода немного сдвинута вправо. Вместо того чтобы начинаться с левого края, перед ней присутствует несколько пробелов, поэтому она начинается на несколько символов дальше от левого края.

Если условие выполнено, все строки кода в следующем *блоке* будут исполнены. В предыдущем примере вторая и третья строки образуют *блок* утверждений для условия **if** в первой строке.

Теперь стоит поговорить о *выделении отступами* и *блоках* в коде.

Выделение отступами

В некоторых языках программирования выделение отступами – просто вопрос стиля: ты можешь выделять код, как тебе нравится (или не выделять вообще). Но в Python выделение отступами необходимо при написании кода. Именно отступы сообщают Python, где начинаются и где заканчиваются блоки кода.

Некоторые утверждения в Python, например условие **if**, должны сопровождаться блоком кода, описывающим последующие действия. В данном случае блок говорит Python, что делать, если условие выполнено.

Не имеет значения, насколько велик отступ при выделении, но весь блок должен быть выделен одинаково. По *соглашению* блок в Python выделяется четырьмя пробелами. Желательно следовать стилю выбранного языка.

СЛОВАРИК

Соглашение означает, что много людей просто делают это именно так.

У меня двоится в глазах?

Там действительно два знака равенства в условии **`if`** (**`if timsAnswer == correctAnswer`**)? Да! И вот почему.

Обычно говорят: «Пять плюс четыре равно девять», а еще говорят: «Пять плюс четыре равно девяти?» Первое – утверждение, второе – вопрос.

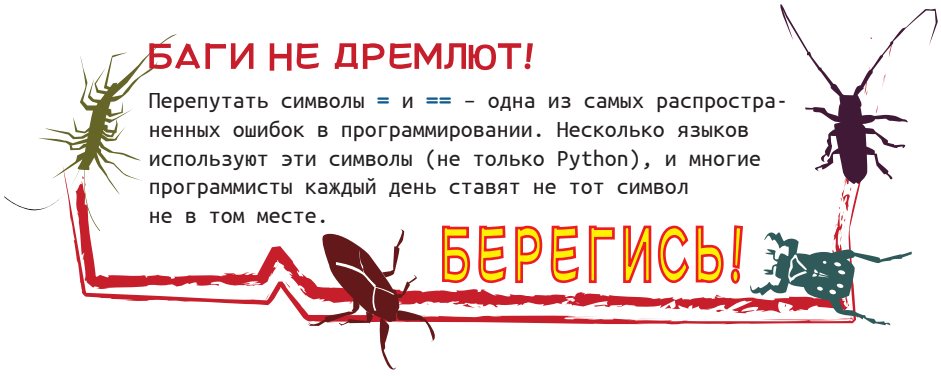
В языке Python тоже есть *утверждения* и *вопросы*. *Утверждение* может присваивать значение переменной. *Вопрос* может проверять, равна ли переменная определенному значению. Первое означает, что ты что-то *устанавливаешь* (присваиваешь или делаешь равным). Второе – что ты *проверяешь* или *тестируешь* что-то (равно оно или нет). Поэтому в Python используются два разных символа.

Ты уже встречал знак равенства (**`=`**) при присвоении значений переменным. Ниже приведены примеры:

```
correctAnswer = 5 + 3
temperature = 35
name = «Миша»
```

Для проверки, равны ли два объекта друг другу, в Python используется знак двойного равенства (**`==`**), например так:

```
if myAnswer == correctAnswer:
if temperature == 40:
if name == «Федя»:
```



Тестирование или проверка также называется *сравнением*. Двойной знак равенства называется *оператором сравнения*. Помнишь, мы говорили об *операторах* в главе 3? Оператор – это особый символ, который оперирует значениями вокруг себя. В данном случае операция – это проверка значений на идентичность.

Другие виды тестов

К счастью, другие операторы сравнения гораздо легче запомнить: меньше, чем (**<**); больше, чем (**>**); не равно (**!=**). (Также для значения «не равно» можно использовать обозначение **<>**, но большинство используют **!=**.) Еще можно использовать комбинацию символа **>** или **<** со знаком **=**, чтобы выразить значение «больше или равно» (**>=**) или «меньше или равно» (**<=**). Ты мог видеть подобное на уроках математики.

Также ты можешь использовать последовательно для оператора «больше, чем» или «меньше, чем», чтобы провести промежуточный тест, например:

```
if 8 < age < 12:
```

Этот тест проверяет, является ли значение переменной **age** числом между, но не включая, 8 и 12. Это условие выполнено, если значение переменной **age** равняется 9, 10 или 11 (или 8.1, или 11.6 и т. д.). Если мы хотим включить значения 8 и 12, мы делаем это так:

```
if 8 <= age <= 12:
```

СЛОВАРИК

Операторы сравнения также называются *операторами отношения* (потому что они тестируют *отношения* между двумя сторонами: равенство или неравенство, больше или меньше). Сравнение еще называется *проверкой условия*, или *логической проверкой*. В программировании слово *логический* относится к чему-либо, где ответ является либо правдой, либо ложью.

Листинг 7.1 показывает пример программы, использующей сравнение. Создай новый файл в редакторе IDLE, введи код программы и сохрани файл под именем *compare.py*. Затем выполни ее. Попробуй запустить ее несколько раз с разными числами. Попробуй указать числа, при которых первое будет бóльшим, потом меньшим и оба числа будут равными.

Листинг 7.1. Использование операторов сравнения

```
num1 = float(input(«Введи первое число: «))
num2 = float(input(«Введи второе число: «))
if num1 < num2:
    print(num1, “меньше, чем”, num2)
```

```
if num1 > num2:
    print(num1, “больше, чем”, num2)
if num1 == num2:
    print(num1, “равно”, num2)
if num1 != num2:
    print(num1, “не равно”, num2)
```

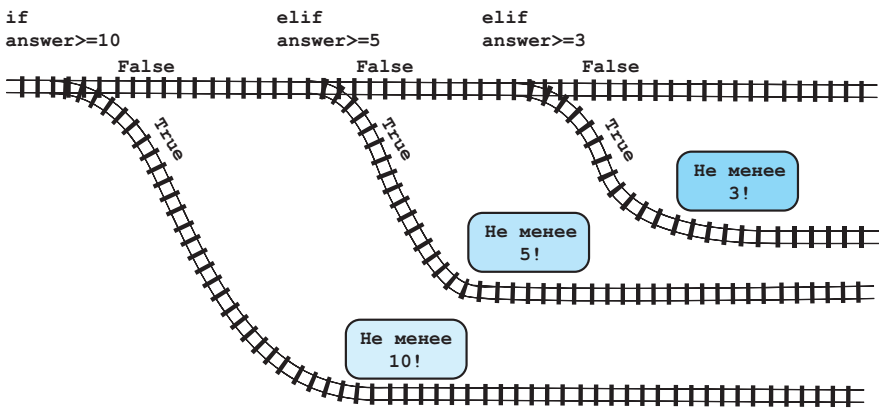
Помни о знаке двойного равенства

Что случается, если результат проверки оказался ложным?

Ты узнал, как на языке Python выполнять определенные действия, если результат проверки оказался правдой. Но что он будет делать, если проверка оказалось ложной? В Python есть три варианта.

- *Провести еще одну проверку.* Если результат первой проверки оказался ложным, ты можешь заставить Python проверить еще что-нибудь с помощью ключевого слова **elif** (сокращение от англ. *else if* – или если), например:

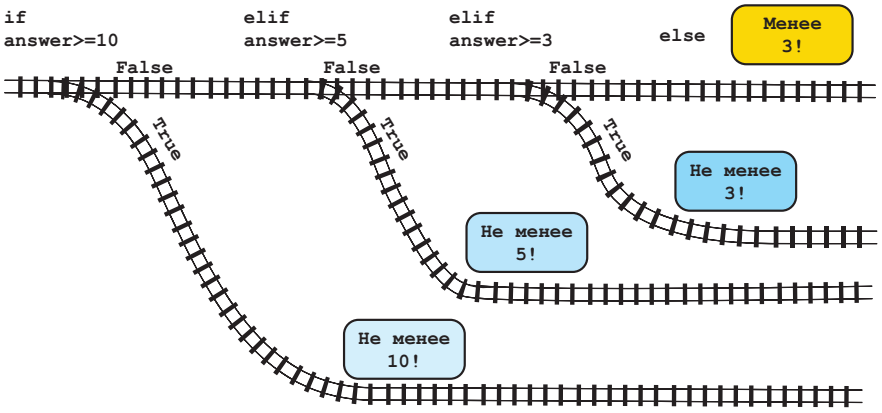
```
if answer >= 10:
print(“Получил не менее 10!”)
elif answer >= 5:
print(«Получил не менее 5!»)
elif answer >= 3:
print(«Получил не менее 3!»)
```



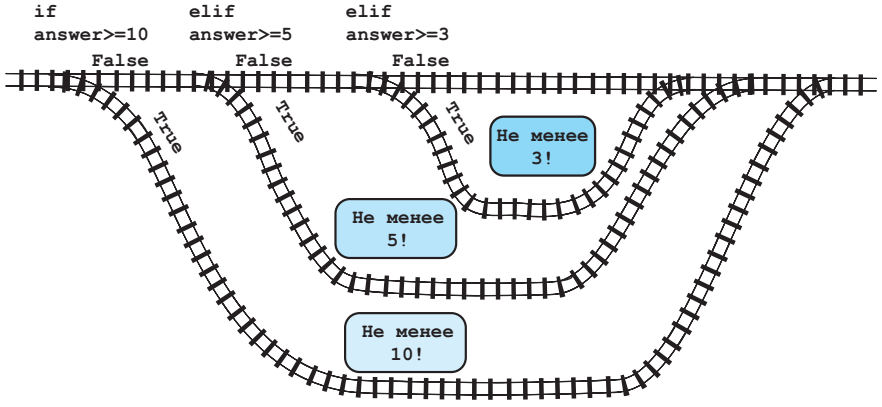
После утверждения **if** можно использовать сколько угодно утверждений **elif**.

- *Выполнить другое действие*, если все остальные проверки дали ложный результат. Для этого используется ключевое слово **else**. Оно всегда указывается в конце, после всех утверждений **if** и **elif**:

```
if answer >= 10:
    print("Получил не менее 10!")
elif answer >= 5:
    print(«Получил не менее 5!»)
elif answer >= 3:
    print(«Получил не менее 3!»)
else:
    print(«Получил менее 3.»)
```



- *Двигаться дальше.* Если ты не ввел никакого другого утверждения после блока **if**, программа продолжит выполнять действия из следующей строки кода (если она есть) или завершится (если кода больше нет).



Попробуй написать программу с кодом, указанным выше, добавив в начале строку для ввода числа:

```
answer = float(input(«Введи число от 1 до 15: «))
```

Не забудь сохранить файл (в этот раз ты выбираешь имя) и запусти программу. Испытай ее несколько раз с разными числами.

Проверка более чем одного условия

А что, если нам нужно проверить выполнение более чем одного условия? Допустим, ты сделал игру для детей от 8 лет и старше и хочешь убедиться, что игрок учится хотя бы в третьем классе. Здесь встречаются два условия. Ниже представлен один из способов проверить выполнение обоих:

```
age = float(input(«Введи свой возраст: «))
grade = int(input(«В каком классе ты учишься: «))
if age >= 8:
    if grade >= 3:
        print(«Ты можешь играть в эту игру.»)
    else:
        print(«Извини, ты не можешь играть в эту игру.»)
else:
    print(«Извини, ты не можешь играть в эту игру.»)
```

Обрати внимание, что строка с первой командой **print** выделена восемью пробелами, а не четырьмя. Это сделано потому, что каждое утверждение **if** требует своего собственного блока, то есть у каждого из них свое собственное выделение.

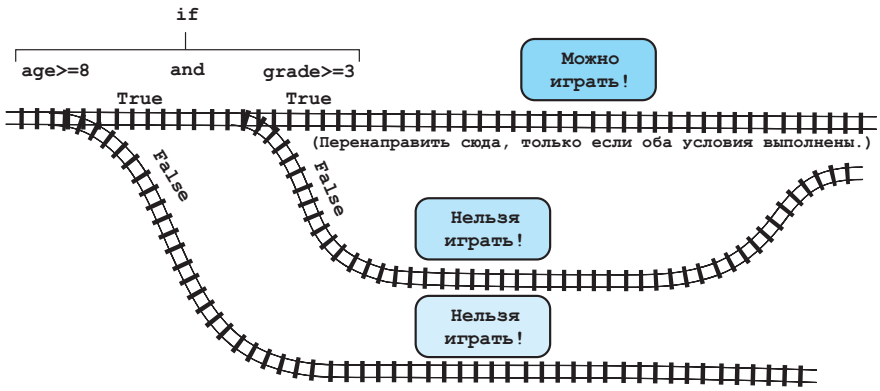
Использование ключевого слова and

Последний пример будет отлично работать. Но есть более простой способ сделать то же самое. Ты можешь совместить условия следующим образом:

```
age = float(input(«Введи свой возраст: «))
grade = int(input(«В каком классе ты учишься: «))
if age >= 8 and grade >= 3:
    print(«Ты можешь играть в эту игру.»)
else:
    print(«Извини, ты не можешь играть в эту игру.»)
```

Совмещение условий с помощью ключевого слова **and**

Мы совместили два условия с помощью ключевого слова **and**. Оно значит, что оба условия должны быть верными для выполнения следующего блока кода.



С помощью ключевого слова **and** можно совмещать более двух условий:

```
age = float(input(«Введи свой возраст: «))
grade = int(input(«В каком классе ты учишься: «))
color = input(«Введи свой любимый цвет: «)
if age >= 8 and grade >= 3 and color == “зеленый”:
    print(«Ты можешь играть в эту игру.»)
else:
    print(«Извини, ты не можешь играть в эту игру.»)
```

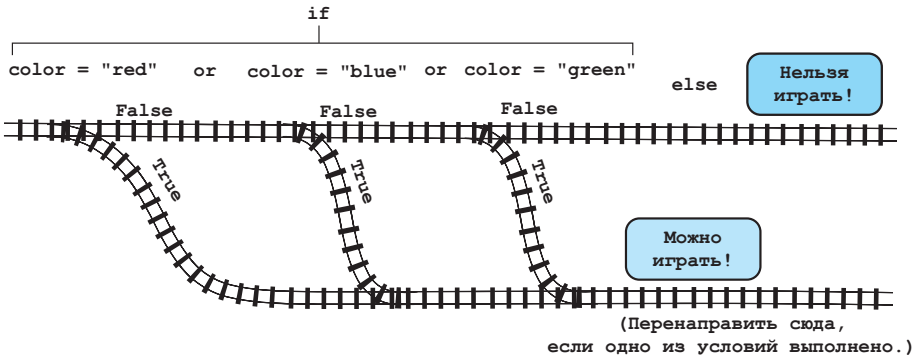
Если условий больше, чем два, *все* они должны быть верными для того, чтобы утверждение **if** было правдой.

Но есть и другие способы совмещения двух условий.

Использование ключевого слова or

Ключевое слово **or** также используется для совмещения условий. Если ты используешь **or**, блок выполняется, если *любое* из условий верно.

```
color = input(«Введи свой любимый цвет: «)
if color == “красный” or color == “синий” or color == “зеленый”:
    print(«Ты можешь играть в эту игру.»)
else:
    print(«Извини, ты не можешь играть в эту игру.»)
```



Использование ключевого слова not

Ты также можешь исказить сравнение так, чтобы оно приняло противоположное значение, с помощью ключевого слова **not**.

```
age = float(input(«Введи свой возраст: «))
if not (age < 8):
    print(«Ты можешь играть в эту игру.»)
else:
    print(«Извини, ты не можешь играть в эту игру.»)
```

Строка

```
if not (age < 8):
```

означает то же самое, что и строка

```
if age >= 8:
```

В обоих случаях блок выполняется, если возраст (значение переменной **age**) равен 8 лет и больше, и не выполняется, если возраст меньше 8 лет.

В главе 4 мы познакомились с *арифметическими операторами* **+**, **-**, ***** и **/**. В этой главе ты узнал об *операторах сравнения* **<**, **>**, **==** и т. д. Ключевые слова **and**, **or** и **not** также являются операторами. Они называются *логическими операторами*. Они используются для модификации сравнений, совмещая два и более из них (**and**, **or**) или изменяя их значение на противоположное (**not**).

Таблица 7.1 Список арифметических операторов и операторов сравнения

Оператор	Название	Действие
Арифметические операторы		
`=`	Присвоение	Присваивает значение имени (переменной)
`+`	Сложение	Складывает два числа вместе. Может использоваться для объединения строк
`-`	Вычитание	Вычитает одно число из другого
`+=`	Приращение	Прибавляет к числу единицу
`-=`	Уменьшение	Вычитает единицу из числа
`*`	Умножение	Умножает два числа друг на друга
`/`	Деление	Делит одно число на другое. Если оба числа целые, в результате будет показана только целая часть ответа без остатка
`//`	Целочисленное деление	Делит одно число на другое. Ответом будет частное без остатка
`%`	Деление по модулю	Получает остаток (или модуль) от деления целых чисел друг на друга
`**`	Возведение в степень	Возводит число в степень. И число, и степень могут быть как целыми числами, так и десятичными дробями
Операторы сравнения		
`==`	Равенство	Проверяет, равны ли две строки
`<`	Меньше чем	Проверяет, меньше ли первое число, чем второе число
`>`	Больше чем	Проверяет, больше ли первое число, чем второе число
`<=`	Меньше чем или равно	Проверяет, меньше ли первое число, чем второе, или равно ему
`>=`	Больше чем или равно	Проверяет, больше ли первое число, чем второе, или равно ему
`!=`	Не равно	Проверяет, что две строки не равны

Ты можешь поместить на эту страницу закладку, чтобы быстро возвращаться к таблице при необходимости.

Что ты узнал?

В этой главе ты узнал:

- о проверке сравнением и операторах отношения;
- о выделении отступами и блоках кода;
- о сочетании проверок с помощью ключевых слов **`and`** и **`or`**;
- об изменении проверки с помощью ключевого слова **`not`**.

Проверь свои знания

1 Каким будет вывод при запуске этой программы:

```
my_number = 7
if my_number < 20:
    print('Меньше 20')
else:
    print('20 или больше')
```

2 В программе из первого вопроса каким будет вывод, если изменить значение переменной **my_number** на **25**?

3 Какой тип утверждения **if** ты применишь, чтобы проверить, что число было больше 30, но меньше или равно 40?

4 Какой тип утверждения **if** ты применишь, чтобы проверить, что пользователь ввел прописную или строчную букву «Ы»?

Попробуй самостоятельно

1 В магазине проходит распродажа. Предоставляется скидка 10 % на покупки от 1000 рублей и меньше и 20 % на покупки суммой более 1000 рублей. Напиши программу, которая запрашивает сумму покупки и выводит скидку на нее (10 % или 20 %) и конечную сумму.

2 Футбольная команда ищет девочек от 10 до 12 лет. Напиши программу, которая спрашивает возраст пользователя и пол (М или Ж). Выведи сообщение, в котором говорится, подходит ли пользователь для игры в этой команде.

3 Дополнительно: напиши код программы так, чтобы она не спрашивала возраст, если пользователь не женского пола.

4 Ты долго ехал на машине и приехал на заправку. До следующей заправки 200 км. Напиши программу для выяснения, нужно ли тебе покупать бензин здесь или можно подождать до следующей заправки. Программа должна задавать три вопроса:

- Каков объем бензобака машины в литрах?
- Насколько заполнен бензобак (в процентах)?
- Сколько км проезжает машина на 1 л бензина?

Вывод должен выглядеть примерно так:

```
Объем бензобака машины в литрах?: 60
Насколько заполнен бензобак (в процентах)?: 40
Сколько км проезжает машина на 1 л бензина?: 10
Машина проедет еще 240 км
Следующая АЗС через 200 км
Ты можешь подождать до следующей АЗС
```

или

```
Объем бензобака машины в литрах?: 60
Насколько заполнен бензобак (в процентах)?: 30
Сколько км проезжает машина на 1 л бензина?: 8
Машина проедет еще 144 км
Следующая заправка через 200 км
Заправься сейчас!
```

Дополнительно: включи в программу погрешность в 5 литров на случай, если датчик уровня топлива передает неточное значение.

5 Напиши программу, в которой пользователь должен ввести пароль для использования программы. Ты будешь знать пароль, но твои друзья должны будут или узнать его от тебя, угадать или изучить Python, чтобы отыскать его в коде!
Программа может быть любой, включая те, что мы уже написали, или просто выводить сообщение «Ты победил!» при вводе правильного пароля.

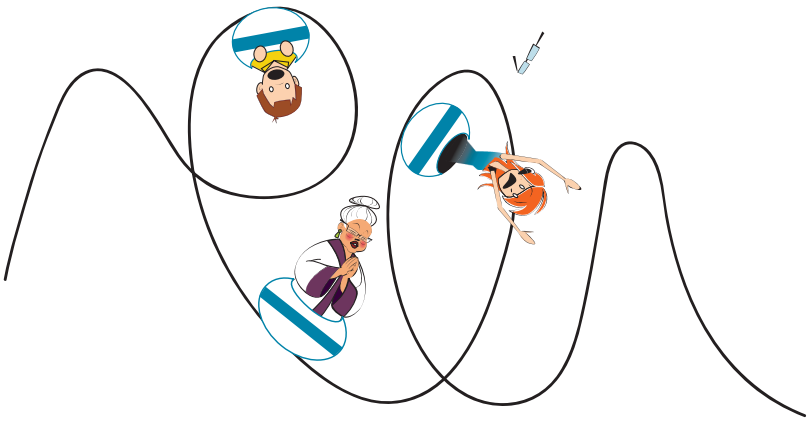
ГЛАВА 8

Циклы

Для большинства людей повторение одного и того же действия снова и снова – очень скучное занятие, поэтому почему не предоставить компьютеру этим заниматься? Компьютерам никогда не становится скучно, поэтому они отлично справляются с повторяющимися заданиями. В этой главе ты узнаешь, как заставить компьютер повторять действия.

Программы часто повторяют одно и то же снова и снова. Это называется *зацикливанием*. Есть два основных вида циклов:

- те, которые повторяются определенное количество раз, – они называются *циклами со счетчиком*;
- те, которые повторяются до выполнения определенного условия, – они называются *условными циклами*, потому что условие для их выполнения должно быть верным.



Циклы со счетчиком

Первый вид циклов называется циклами со счетчиком. Ты также можешь встретить название *цикл for*, потому что во многих языках программирования, включая Python, используется ключевое слово **for** для создания такого цикла в программе.

Давай попробуем написать программу с использованием цикла со счетчиком. Создай новый файл в текстовом редакторе IDLE с помощью команды меню **File ⇨ New** (как мы делали в нашей первой программе). Затем набери код программы из листинга 8.1.

Листинг 8.1. Простой цикл `for`

```
for looper in [1, 2, 3, 4, 5]:
    print(«привет»)
```

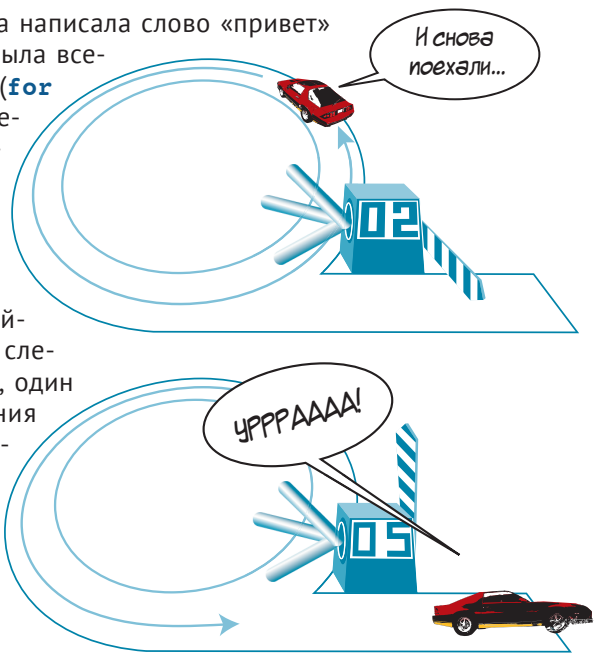
Сохрани программу под именем *Loop1.py* и выполни ее. (Можно использовать команду меню **Run ⇨ Run Module** или клавишу **F5**.)

Ты увидишь что-то подобное:

```
>>>
RESTART: C:/HelloWorld/Примеры/Loop1.py
привет
привет
привет
привет
привет
```

Эй, здесь что, эхо? Программа написала слово «привет» пять раз, хотя команда **`print`** была всего одна. Как так? Первая строка (**`for looper in [1,2,3,4,5]:`**), переведенная на простой человеческий язык, означает:

1. Значение переменной (**`looper`**) начнется с 1 (то есть **`looper = 1`**).
2. Цикл будет выполнять действие, которое описано в следующем *блоке* инструкций, один раз для каждого значения в списке. (Список – это числа в квадратных скобках.)
3. Каждый раз в течение выполнения цикла переменная **`looper`** присваивается следующему значению в списке.



Вторая строка (**`print («привет»)`**) – блок кода, который интерпретатор Python будет выполнять каждый раз во время цикла. Циклу **`for`** необходим блок кода, который будет говорить программе, что ей делать в каждом цикле. Этот *блок* (выделенная отступом часть кода) называется *телом цикла*. (Напомним, что мы говорили о выделении и блоках в предыдущей главе.)

СЛОВАРИК

Каждое повторение цикла называется *итерацией*.

Давай попробуем еще что-нибудь. Вместо вывода одного и того же слова отобразим на экране разные числа. Это делает программа из листинга 8.2.

Листинг 8.2. Выполнение разных действий во время цикла

```
for looper in [1, 2, 3, 4, 5]:
print(looper)
```

Сохрани программу под именем *Loop2.py* и выполни ее. Результат будет выглядеть так:

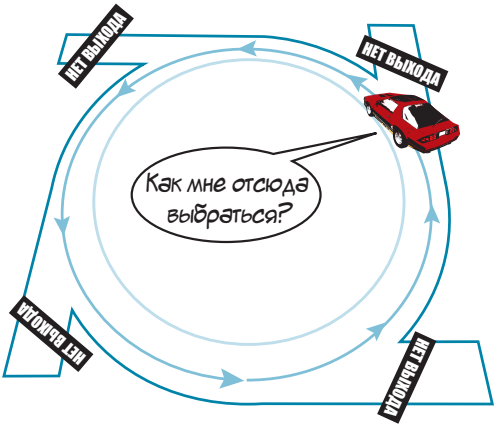
```
>>>
RESTART: C:/HelloWorld/Примеры/Loop2.py
1
2
3
4
5
```

В этот раз вместо пяти слов «привет» программа вывела на экран значение переменной **`looper`**. Каждый раз во время цикла переменная принимала следующее значение из списка.

Неуправляемые циклы

То же самое случилось и со мной, Картер! Неуправляемые циклы (также называемые бесконечными) случаются рано или поздно в практике каждого программиста. Чтобы остановить программу на языке Python в любой момент (даже при неуправляемом цикле), нажми сочетание клавиш **Ctrl**+**C**. То есть нажми и удерживай клавишу **Ctrl**, а затем нажми клавишу **C**. Позже ты к этому привыкнешь! Игры и графические программы

постоянно зациклены. Им нужно все время получать информацию от движений мышью, с клавиатуры или игрового контроллера, обрабатывать ввод и обновлять содержимое экрана. Когда мы начнем писать подобные программы, то будем использовать циклы постоянно. Велика вероятность, что одна из твоих программ застрянет на каком-нибудь цикле, и тебе нужно знать, как ее освободить!



Для чего нужны квадратные скобки?

Ты мог заметить, что наш список значений цикла заключен в квадратные скобки. Квадратные скобки и запятые между числами – это способ создать *список* в Python. Мы скоро познакомимся ближе со списками (в главе 12). А сейчас запомни, что список – это «хранилище» для нескольких объектов вместе. В этом случае объекты – это числа, значения, которые принимает переменная **`looper`** во время итераций цикла.

Использование цикла со счетчиком

А теперь давай сделаем что-нибудь более полезное с помощью цикла. Например, выведем на экран таблицу умножения. Для этого нужно внести небольшие изменения в нашу программу. Новая версия в листинге 8.3.

Листинг 8.3. Вывод таблицы умножения для числа 8

```
for looper in [1, 2, 3, 4, 5]:
    print(looper, "умножить на 8 =", looper * 8)
```

Сохрани файл под именем *Loop3.py* и запусти программу. Ты увидишь следующее:

```
>>>
RESTART: C:/HelloWorld/Примеры/Loop3.py
1 умножить на 8 = 8
2 умножить на 8 = 16
3 умножить на 8 = 24
4 умножить на 8 = 32
5 умножить на 8 = 40
```

Теперь ты видишь весь потенциал цикла. Без циклов нам бы пришлось писать программу наподобие этой, чтобы получить тот же результат:

```
print(«1 умножить на 8 =», 1 * 8)
print(«2 умножить на 8 =», 2 * 8)
print(«3 умножить на 8 =», 3 * 8)
print(«4 умножить на 8 =», 4 * 8)
print(«5 умножить на 8 =», 5 * 8)
```

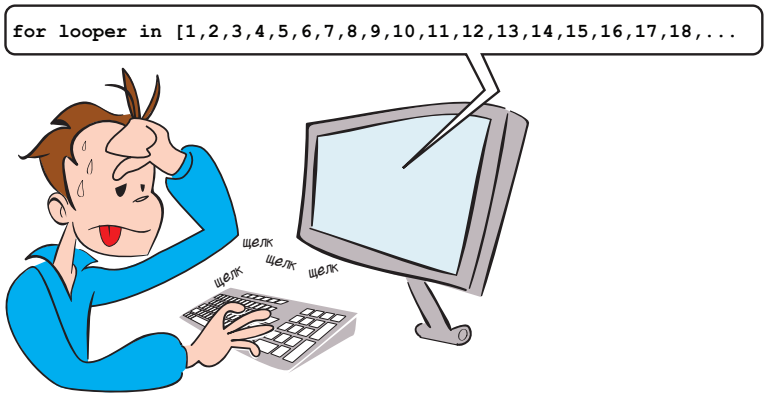
Для более длинной таблицы умножения (например, до 20) эта программа была бы гораздо длиннее, а наша программа с циклом осталась бы практически неизменной (только добавились бы числа в списке). Циклы упрощают жизнь!

Сокращение – `range()`

В предыдущем примере мы использовали цикл 5 раз:

```
for looper in [1, 2, 3, 4, 5]:
```

А если нам нужно использовать цикл 100 или 1000 раз? Это будет долгая песня!



К счастью, есть короткий путь. Функция **`range()`** позволяет вводить начальное и конечное значения и создает промежуточные значения за тебя. Она создает список, содержащий ряд чисел.

В листинге 8.4 используется функция **`range()`** для создания таблицы умножения.

Листинг 8.4. Цикл с использованием функции **`range()`**

```
for looper in range(1, 5):
    print(looper, “умножить на 8 =”, looper * 8)
```

Сохрани файл под именем *Loop4.py* и выполни его. (Ты можешь выбрать команду меню **Run** ⇨ **Run Module** или нажать клавишу **F5**.) Ты увидишь следующее:

```
>>>
RESTART: C:/HelloWorld/Примеры/Loop4.py
1 умножить на 8 = 8
2 умножить на 8 = 16
3 умножить на 8 = 24
4 умножить на 8 = 32
```

Почти то же самое, что и в предыдущей программе... кроме последнего недостающего цикла! Почему? Ответ состоит в том, что функция **`range(1,5)`** дает нам список **`[1,2,3,4]`**. А где же 5? Ну, именно так работает функция **`range()`**. Она дает список чисел, начиная с первого и заканчивая предыдущим числом. Ты должен принимать это во внимание и использовать функцию так, чтобы получить нужное количество циклов.

В листинге 8.5 показано, как нужно изменить программу, чтобы получить таблицу умножения числа 8 от 1 до 10.

Листинг 8.5. Вывод таблицы умножения числа 8 от 1 до 10

```
for looper in range(1, 11):
    print(looper, "умножить на 8 =", looper * 8)
```

И вот что мы получаем:

```
>>>
RESTART: C:/HelloWorld/Примеры/eight_times_table.py
1 умножить на 8 = 8
2 умножить на 8 = 16
3 умножить на 8 = 24
4 умножить на 8 = 32
5 умножить на 8 = 40
6 умножить на 8 = 48
7 умножить на 8 = 56
8 умножить на 8 = 64
9 умножить на 8 = 72
10 умножить на 8 = 80
```

В программе из листинга 8.5 функция **`range(1,11)`** выдает список чисел от 1 до 10, и цикл выполнил одну *итерацию* для каждого числа в списке. Каждый раз во время цикла переменная **`looper`** принимала следующее значение из списка.

Кстати, мы назвали нашу переменную цикла именем **`looper`**, но ты можешь использовать любое имя.

Вопрос стиля – имена переменных цикла

Переменная цикла ничем не отличается от других переменных. В ней нет ничего особенного – это просто имя для значения. Не имеет значения, что мы используем переменную в качестве счетчика циклов.

Ранее мы говорили, что тебе стоит использовать такие имена переменных, которые описывали бы действия этих переменных. Поэтому мы выбрали имя **`looper`** для предыдущего примера (от англ. *loop* – цикл). Но переменные цикла – это как раз те переменные, для которых иногда можно сделать исключение. Потому что есть обычай (напомним, что это значит «распространенная практика») в программировании использовать буквы **`i`**, **`j`**, **`k`** и т. д. в качестве переменных цикла.

Поскольку многие используют буквы **`i`**, **`j`** и **`k`** для обозначения переменных цикла, программисты привыкли видеть их в программах. Можно абсолютно свободно использовать другие имена. Но не стоит использовать **`i`**, **`j`**, **`k`** для других переменных, кроме переменных цикла.

В СТАРЫЕ ДОБРЫЕ ВРЕМЕНА

Почему именно буквы *i*, *j* и *k* стали применяться для именования переменных цикла?

Потому что первые программисты использовали программы для решения математических задач, а в математике буквы *a, b, c* и *x, y, z* уже зарезервированы для других случаев. Также в одном популярном языке программирования переменные *i, j* и *k* всегда были целыми числами – их нельзя было изменить. Поскольку счетчики циклов – всегда целые числа, программисты обычно выбирали буквы *i, j, k* для обозначения, и это стало общей практикой.

Если мы используем эти обозначения, наша программа будет выглядеть так:

```
for i in range(1, 11):
    print(i, "умножить на 8 =", i * 8)
```

И она будет работать точно так же. Попробуй!

Какие имена ты используешь для своих переменных цикла – это вопрос *стиля*. Стиль – это внутренний вид твоей программы, а не то, как она работает. Но если ты будешь использовать тот же стиль, что и другие программисты, твои программы будет легче читать, понимать и налаживать. Ты также привыкнешь к этому стилю и будешь свободно читать код программ других разработчиков.

Сокращение `range()`

Тебе не нужно всегда указывать два числа с функцией **`range()`**, как мы делали в листинге 8.5. Ты можешь указать только одно число:

```
for i in range(5):
```

Это то же самое, что и

```
for i in range(0, 5):
```

что дает нам список чисел `[0,1,2,3,4]`.

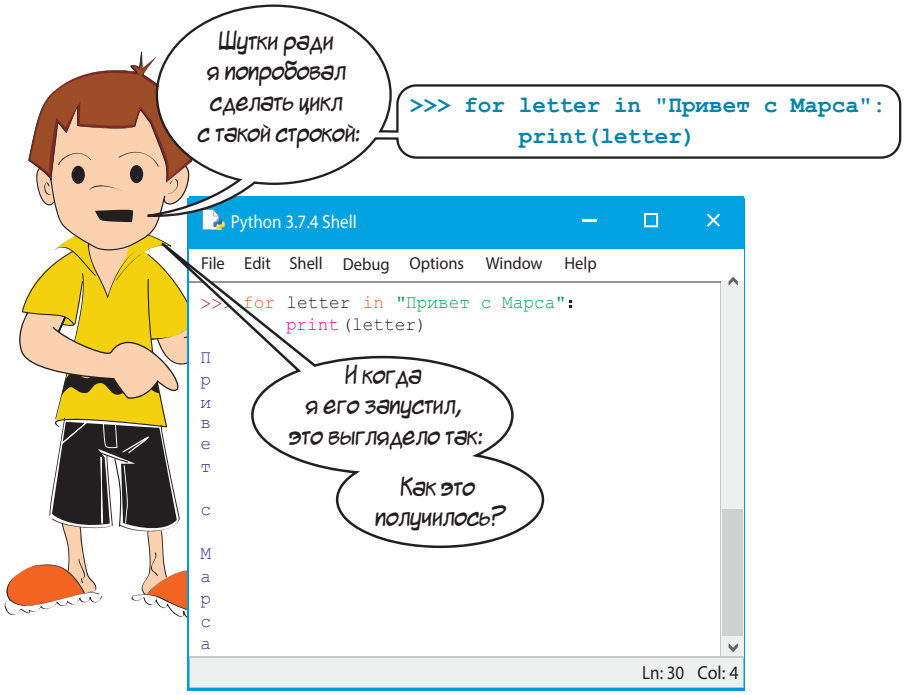
Фактически многие программисты начинают цикл с 0, а не с 1. Если ты используешь функцию `range(5)`, ты получишь пять итераций в цикле, что легко запомнить. Тебе только нужно знать, что в первом цикле `i` будет равно 0, а не 1, а в последнем – 4, а не 5.

В СТАРЫЕ ДОБРЫЕ ВРЕМЕНА

Итак, почему многие программисты начинают циклы с 0, а не с 1?

Ну, в старые добрые времена некоторые начинали с 1, а некоторые – с 0. Они долго спорили о том, какой способ лучше. В конце концов, победили те программисты, которые были за 0.

И вот что мы имеем. Большинство разработчиков начинают теперь цикл с 0, но ты можешь использовать, что хочешь. Просто помни о верхнем пределе и назначай его так, чтобы получить нужное количество итераций.



Ну, Картер, ты узнал что-то новое о строках. Строка похожа на *список символов*. Мы выучили, что циклы со счетчиком используют *списки* для *итераций*. Это значит, что можно создать цикл из строки. Каждый символ в строке – одна итерация цикла. Поэтому если мы выведем на экран переменную цикла, которую Картер назвал **`letter`** в этом примере, мы выведем буквы в строке, по одной на каждую итерацию. Поскольку каждая команда **`print`** начинается с новой строки, каждая буква будет в отдельной строке.

Эксперименты – хороший способ обучения!

Счетчик по шагам

До сих пор в циклах со счетчиками мы использовали 1 в качестве интервала. А если мы хотим использовать 2? Или 5, или 10? А как насчет обратного отсчета?

Функция **`range()`** может содержать дополнительный *аргумент*, который позволяет менять размер шагов со значения по умолчанию (1) на другой размер.

СЛОВАРИК

Аргументы – это значения, которые ты вставляешь в скобки при использовании функции наподобие **`range()`**. Мы говорим, что ты передаешь аргумент функции. Термин *аргумент* используется, как и термин «передавать параметр». Ты узнаешь больше о функциях, аргументах и параметрах в главе 13.

Мы собираемся попробовать использовать циклы в интерактивном режиме. Когда ты наберешь первую строку с двоеточием в конце, редактор IDLE автоматически выделит отступом следующую строку, потому что он «знает», что циклу **`for`** необходим следующий блок кода. Когда ты закончишь ввод блока кода, нажми клавишу **Enter** дважды. Попробуй:

```
>>> for i in range(1, 10, 2):
        print(i)
1
3
5
7
9
```

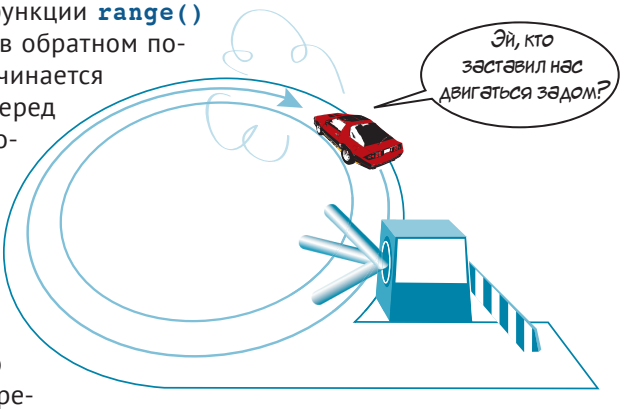
Мы добавили третий параметр, **`2`**, в функцию **`range()`**. Теперь цикл считает с шагом 2. Давай попробуем еще:

```
>>> for i in range(5, 26, 5):
        print(i)
5
10
15
20
25
```

Теперь наш шаг составляет 5. А как насчет обратного счета?

```
>>> for i in range(10, 1, -1):
        print(i)
10
9
8
7
6
5
4
3
2
```

Когда третий параметр в функции **`range()`** отрицательный, цикл считает в обратном порядке. Напомним, что цикл начинается с первого числа и считает вперед (или назад), но не включает второе число, поэтому в нашем последнем примере мы досчитали до 2, а не до 1.



Мы можем использовать это для создания программы отсчета времени. Нам нужно добавить только несколько строк. Открой новый файл в редакторе IDLE и набери код программы из листинга 8.6. Затем попробуй ее выполнить.

Листинг 8.6. Готов стартовать?

```
import time
for i in range(10, 0, -1): ◄—— Обратный счет
    print(i)
    time.sleep(1) ◄—————————— Ждем 1 секунду
print("СТАРТ!")
```

Не переживай об элементах программы, которые тебе еще незнакомы, например ключевых словах **`import`**, **`time`** и **`sleep`**. Ты узнаешь о них в следующих главах. Просто выполни программу из листинга 8.6, и ты увидишь, как она работает. Важной частью здесь является функция **`range(10,0,-1)`**, которая создает цикл с обратным отсчетом от 10 до 1.

Счетчик без чисел

Во всех предыдущих примерах переменной цикла было число. В терминах программирования мы говорим, что *цикл повторялся в пределах* списка чисел. Но спи-

сок необязательно должен быть списком чисел. Как ты уже видел в эксперименте Картера, он также может быть списком символов (строкой). Он также может быть списком строк или чем угодно.

Лучше всего это видно на примере. Попробуй программу из листинга 8.7.

Листинг 8.7. Кто самый крутой?

```
for cool_guy in [«Шварценеггер», «Сталлоне», «Чан», «Райтман»]:
    print(cool_guy, «- самый крутой чувак на свете!»)
```

Теперь мы повторяем не числа, а список строк. И вместо **i** в качестве переменной цикла мы использовали значение **cool_guy**. Переменная цикла **cool_guy** принимает разные значения в соответствии со списком. Это все еще в некотором смысле *цикл со счетчиком*, потому что даже хотя в списке нет чисел, Python *считает*, сколько элементов находится в списке, чтобы знать, сколько раз повторять цикл. (В этот раз мы не будем показывать, как выглядит вывод программы, ты увидишь его сам в программе.)

А что, если мы не знаем заранее, сколько итераций нам понадобится? Что, если у нас нет списка значений, которые мы можем использовать? Не задавайся этими вопросами, потому что ответы уже на подходе!

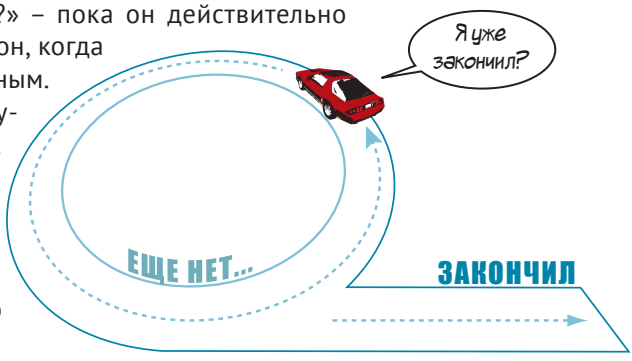
Раз мы коснулись этой темы...

Ты только что узнал о первом виде циклов, *циклах со счетчиком* или *циклов for*. Второй тип называется *условными циклами*, или *циклами while*.

Цикл **for** хорош, если ты знаешь заранее, сколько раз тебе нужно повторить цикл. Но иногда тебе нужно повторять его до тех пор, пока не произойдет нечто. И ты не знаешь, сколько итераций будет выполнено, пока это нечто не случится. Циклы while позволяют это реализовать.

В предыдущей главе ты узнал об *условиях, проверке* и утверждении **if**. Вместо подсчета количества итераций цикла циклы **while** используют *проверку*, для того чтобы определить, когда остановить цикл. Этот вид также называется *условными циклами*. Условный цикл повторяется, пока не будет выполнено *условие*.

Обычно условный цикл постоянно задается вопросом: «Я уже закончил?.. Я уже закончил?.. Я уже закончил?» – пока он действительно не закончится. А закончится он, когда условие больше не будет верным.



Этот вид цикла использует ключевое слово **while**. В листинге 8.8 приведен пример. Набери программу, попробуй ее запустить – и увидишь, как она работает. (Напомним, что ее нужно сохранить и выполнить.)

Листинг 8.8. Условный цикл, или цикл **while**

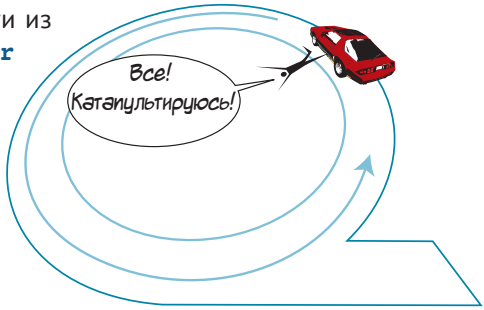
```
print(«Введи 3, чтобы продолжить, или другое значение, чтобы закончить.»)
someInput = input()
while someInput == '3': ←—— Продолжать цикл, пока someInput = «3»
    print(«Спасибо за 3. Ты очень добр.»)
    print(«Введи 3, чтобы продолжить, или другое значение,
          чтобы закончить.»)
    someInput = input()
print(«Это не 3, я заканчиваю.»)
```

Тело цикла

Эта программа запрашивает ввод у пользователя. Пока (**while**) ввод равен 3, условие является верным (**true**) и цикл выполняется. Поэтому этот вид цикла называют циклами **while**. Когда ввод *не* равен 3, условие становится ложным (**false**), цикл останавливается.

Выход из цикла – break и continue

Бывают моменты, когда тебе нужно выйти из цикла в середине, до того, как цикл **for** закончит считать или цикл **while** найдет условие. Есть два способа это сделать: ты можешь перепрыгнуть к следующей итерации с помощью команды **continue** или остановить цикл полностью с помощью команды **break**. Давай посмотрим внимательнее.



Прыжок вперед – continue

Если ты хочешь остановить выполнение текущей итерации и перейти сразу к следующему, тебе нужно утверждение **continue**. Лучший способ – пример. Внимание на листинг 8.9.

Листинг 8.9. Использование утверждения **continue** в цикле

```
for i in range(1, 6):
    print()
    print('i =', i, ' ', end='')
    print(Привет, как ', end='')
    if i == 3:
        continue
    print('у тебя дела?', end='')
print()
```

При запуске программа будет выглядеть так:

```
>>>
RESTART: C:/HelloWorld/Примеры/hello_how_continue.py
i = 1 Привет, как у тебя дела?
i = 2 Привет, как у тебя дела?
i = 3 Привет, как
i = 4 Привет, как у тебя дела?
i = 5 Привет, как у тебя дела?
```

Обрати внимание, что в третьей итерации цикла (когда **`i == 3`**) тело цикла не закончено – программа сразу перешла к следующей итерации (**`i == 4`**). Это вступило в силу утверждение **`continue`**. Точно так же оно работает и с циклами **`while`**.

Завершение цикла – `break`

Что, если мы хотим полностью выйти из цикла – не заканчивать подсчет, не ждать выполнения условия? Нам поможет утверждение **`break`**.

Давай изменим строку 6 в листинге 8.9, заменив утверждение **`continue`** на **`break`**, и запустим программу снова, чтобы увидеть, что у нас вышло.

```
>>>
RESTART: C:/HelloWorld/Примеры/hello_how_break.py
i = 1 Привет, как у тебя дела?
i = 2 Привет, как у тебя дела?
i = 3 Привет, как
```

В этот раз цикл не просто пропустил окончание третьей итерации; он полностью остановился. Так работает **`break`**. Точно так же оно работает и с циклами **`while`**.

Должен тебе сказать, что некоторые люди считают использование утверждений **`break`** и **`continue`** плохой идеей. Лично мы не думаем, что эти утверждения *плохи*, но мы редко их используем. И посчитали нужным рассказать тебе о них на случай, если они когда-нибудь тебе понадобятся.

Что ты узнал?

В этой главе ты узнал:

- о циклах **`for`** (циклах со счетчиками);
- о функции **`range()`** – сокращении для циклов со счетчиками;
- о разных вариантах шагов для функции **`range()`**;
- о циклах **`while`** (условных циклах);
- о переходе к следующему повтору с помощью утверждения **`continue`**;
- о выходе из цикла с помощью утверждения **`break`**.

Проверь свои знания

1 Сколько раз будет повторяться следующий цикл:

```
for i in range(1, 6):
    print('Привет, Михаил')
```

2 Сколько раз будет повторяться следующий цикл? И каким будет значение **i** в каждой итерации?

```
for i in range(1, 6, 2):
    print('Привет, Михаил')
```

3 Какой список чисел выдаст функция **range(1,8)**?

4 Какой список чисел выдаст функция **range(8)**?

5 Какой список чисел выдаст функция **range(2,9,2)**?

6 Какой список чисел выдаст функция **range(10,0.-2)**?

7 Какое ключевое слово используется для прекращения текущей итерации цикла и перехода к следующей?

8 Когда заканчивается цикл **while**?

Попробуй самостоятельно

1 Напиши программу для вывода таблицы умножения. В начале она должна спрашивать пользователя, какую таблицу выводить. Вывод должен выглядеть примерно так:

```
Какая таблица умножения тебе нужна?
5
Вот твоя таблица:
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

2 Ты, вероятно, использовал цикл **for** для программы из пункта 1. Так делает большинство людей. Но, просто для практики, попробуй сделать то же самое с циклом **while**. Или, если ты использовал его, с циклом **for**.

3 Добавь еще что-нибудь в программу с таблицей умножения. После вопроса о том, какую таблицу выводить, спроси пользователя, насколько длинной должна быть таблица. Вывод должен выглядеть примерно так:

```
Какая таблица умножения тебе нужна?
7
До какого множителя считать?
12
Вот твоя таблица:
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
7 x 11 = 77
7 x 12 = 84
```

Ты можешь выполнить эту задачу с помощью циклов **`for`** или **`while`**. Или сделать оба варианта.

ГЛАВА 9

Комментарии

До сих пор все, что мы набирали в виде кода программ (и в интерактивном интерпретаторе), – это инструкции компьютеру. Но было бы неплохо включать в код программ комментарии для самого себя, описывающие, что делает программа и как она работает. Это поможет тебе (или кому-то еще) читать код программы и понимать, что же ты написал.

В программах такие заметки называются *комментариями*.

Добавление комментариев

Комментарии предназначены только для чтения человеком, а не для выполнения компьютером. Комментарии являются частью *документации* программы, и компьютер игнорирует их во время ее выполнения.

В Python есть несколько способов добавления комментариев в программу.

СЛОВАРИК

Документация – это информация, которая описывает программу и принципы ее работы. Комментарии – одна из частей документации, но есть и другие, помимо самого кода, которые описывают:

- почему была написана программа (ее цели);
- кто ее написал;
- для кого она предназначена (аудитория);
- как она организована.

И так далее. Большие сложные программы обычно имеют более развернутую документацию.

Справка Python, которую мы упомянули во врезке «Думай как программист (на Python)» в главе 6, – это в некотором роде документация. Она создана для помощи пользователям – тебе – в понимании принципов работы Python.

Однострочные комментарии

Ты можешь сделать любую строку комментарием, начав ее с символа #. (Он называется *октоторпом*, или *«решеткой»*.)

```
# Это комментарий в программе
print('Это не комментарий')
```

Если ты выполнишь эти две строки, то получишь следующий вывод:

```
Это не комментарий
```

Во время выполнения кода программы первая строка игнорируется программой. Комментарий, который начинается символом #, предназначен только для тебя или других людей, читающих код.

Комментарии в конце строки

Ты также можешь поместить комментарии в конце строки кода, например так:

```
area = length * width      # Вычисление площади прямоугольника
```

Комментарий начинается знаком #. Символы до него – обычная строка кода. После него – комментарий.

Многострочные комментарии

Иногда понадобится использовать не одну строку для комментария. Можно использовать несколько строк с символом # в начале каждой из них:

```
# ***************
# Эта программа показывает, как использовать комментарии в Python
# Линия звездочек используется для визуального отделения
# комментариев от остального кода.
# ***************
```

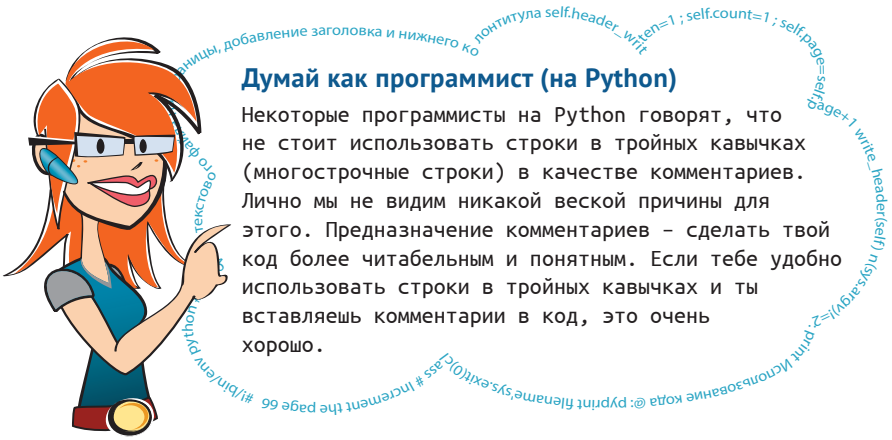
Многострочные комментарии хороши для визуального разделения секций кода при чтении. В них можно пояснить, что делает каждая секция. Многострочный комментарий в начале программы может содержать имя разработчика, название программы, дату написания или обновления и любую другую полезную информацию.

Строки в тройных кавычках

Есть другой способ сделать нечто, что работает как многострочный комментарий, в Python. Ты можешь создать стоку в тройных кавычках без имени. Напомним, что в главе 2 мы говорили о строке в тройных кавычках, которая может распространяться на несколько строк. Это можно использовать так:

```
«»» Это комментарий на нескольких строках,
сделанный с помощью строки в тройных кавычках.
Это не комментарий на самом деле, но ведет себя так же.
«»»
```

Поскольку строка не имеет имени и программа ничего с ней не делает, строка не имеет никакого влияния на выполнение программы. То есть она ведет себя как комментарий, на самом деле им не являясь.



Думай как программист (на Python)

Некоторые программисты на Python говорят, что не стоит использовать строки в тройных кавычках (многострочные строки) в качестве комментариев. Лично мы не видим никакой веской причины для этого. Предназначение комментариев – сделать твой код более читабельным и понятным. Если тебе удобно использовать строки в тройных кавычках и ты вставляешь комментарии в код, это очень хорошо.

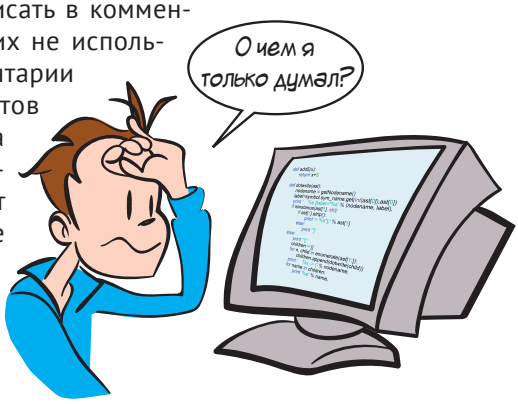
Если ты наберешь несколько комментариев в редакторе IDLE, то увидишь, что они выделяются отдельным цветом. Это тоже облегчает чтение кода.

Большинство редакторов позволяют менять цвет комментариев (и других частей кода). В интерпретаторе IDLE цветом комментариев по умолчанию установлен красный. Поскольку строки в тройных кавычках не являются настоящими комментариями в Python, они выделены другим цветом. В интерпретаторе IDLE они зеленые, потому что для строк в нем установлен зеленый цвет.

Стиль комментирования

Итак, теперь ты знаешь, как добавлять комментарии. Но что стоит в них писать?

Поскольку они не влияют на выполнение программы, мы говорим, что это вопрос «стиля». То есть ты можешь писать в комментариях все, что угодно (или вообще их не использовать). Но это не значит, что комментарии не важны. Большинство программистов приходят к этому своим путем, когда возвращаются к программе, написанной несколько недель, месяцев или лет назад (или даже вчера), и не могут ее понять! Обычно это случается из-за того, что в программу не поместили достаточное количество комментариев для объяснения принципов ее работы. Они могут быть очевидными при написании, но стать полнейшей загадкой по прошествии времени.



Не существует твердых правил по написанию комментариев, но мы рекомендуем добавлять столько пометок, сколько тебе нужно. На данный момент чем больше, тем лучше. Лучше грешить слишком обильными комментариями, чем слишком скудными. При росте твоего опыта программирования ты найдешь свою золотую середину.

Комментарии в этой книге

Ты не увидишь много комментариев в листингах и коде программ этой книги. Потому что здесь мы используем «выноски» – пометки по мере написания кода. Но если ты взглянешь на листинги программ в папке *Примеры*, то увидишь комментарии во всех программах.

Комментирование кода

Также с помощью комментариев можно временно исключать части программы из ее выполнения. Все, что является комментарием, будет проигнорировано.

```
# print(«Привет,»)
print(«мир»)
>>>
RESTART: C:/HelloWorld/Примеры/commenting_out.py
мир
```

Поскольку команда `print(«Привет,»)` была закомментирована, эта строка не была выполнена программой, и слово «Привет» не выведено на экран.

Это полезно при отладке программы, и тебе нужно, чтобы одни части выполнялись, а другие нет. Просто добавь символ `#` в начале любой строки или окружи фрагмент кода тройными кавычками, чтобы компьютер ее проигнорировал.

Большинство редакторов кода (включая IDLE) позволяют быстро закомментировать целые блоки кода. В редакторе IDLE ищи соответствующую команду в меню **Format**.

Что ты узнал?

В этой главе ты узнал:

- о том, что комментарии предназначены для тебя (и других пользователей), а не для компьютеров;
- комментарии можно использовать для блокировки частей кода, чтобы препятствовать их выполнению;
- о том, что можно использовать строки в тройных кавычках в качестве комментариев, которые занимают несколько строк.

Проверь свои знания

Поскольку комментарии – довольно простая штука, мы устроим перерыв и обойдемся без вопросов в этой главе.

Попробуй самостоятельно

Вернись к программе конвертации температур (из раздела «Попробуй самостоятельно» в главе 3) и добавь в нее комментарии. Запусти программу, чтобы проверить, что она работает без изменений.

ГЛАВА 10

Время поиграть

Одной из замечательных традиций обучения программированию является набор кода, который ты не понимаешь. Честно!

Иногда простой ввод кода дает тебе «ощущение» того, как он работает, даже если ты не понимаешь каждую строку или ключевое слово. Мы делали это в главе 1 с игрой по угадыванию чисел. Сейчас мы сделаем то же самое, но программа будет длиннее и интереснее.

Игра «Лыжник»

«Лыжник» – очень простая игра в духе игры SkiFree (о ней можно узнать по адресу **en.wikipedia.org/wiki/SkiFree**).

Ты съезжаешь на лыжах вниз по склону, пытаясь огибать деревья и собирать флажки. Один флажок добавляет тебе 10 очков. Столкновение с деревом лишает тебя 100 очков.

Когда ты запустишь программу, она должна выглядеть так:

В игре «Лыжник» используется Pygame для помощи с графикой. Pygame – это *модуль* Python. (Ты узнаешь подробнее о модулях в главе 15.) Если ты запускал инсталлятор этой книги, Pygame уже установлен на твоем компьютере. Если нет, ты можешь загрузить его по адресу **www.pygame.org**. В главе 16 мы познакомимся с Pygame ближе.

Ниже приведены графические файлы, которые тебе понадобятся:

- skier_down.png
- skier_crash.png
- skier_tree.png
- skier_flag.png
- skier_right1.png
- skier_right2.png
- skier_left1.png
- skier_left2.png

Ты найдешь их в папке *Примеры*. Просто помести их в ту же папку, куда сохранил программу. Это важно. Если эти файлы не будут находиться в одной папке с программой, Python не найдет их, и программа не будет работать.

Код игры Лыжник приведен в листинге 10.1. Листинг довольно длинный, около 115 строк кода (плюс пустые строки для облегчения чтения), но мы предлагаем тебе потратить время на его набор. В листинге есть пометки с объяснениями.

Обрати внимание на слово **`__init__`** в коде. С двух сторон расположено по два знака подчеркивания. Не перепутай: именно по два знака с каждой стороны, а не по одному.

Листинг 10.1. Игра Лыжник

```
import pygame, sys, random
skier_images = ["skier_down.png", "skier_right1.png",
                "skier_right2.png", "skier_left2.png",
                "skier_left1.png"]

class SkierClass(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load("skier_down.png")
        self.rect = self.image.get_rect()
        self.rect.center = [320, 100]
        self.angle = 0

    def turn(self, direction):
        self.angle = self.angle + direction
        if self.angle < -2:  self.angle = -2
        if self.angle >  2:  self.angle = 2
        center = self.rect.center
        self.image = pygame.image.load(skier_images[self.angle])
        self.rect = self.image.get_rect()
        self.rect.center = center
        speed = [self.angle, 6 - abs(self.angle) * 2]
        return speed

    def move(self, speed):
        self.rect.centerx = self.rect.centerx + speed[0]
        if self.rect.centerx < 20:  self.rect.centerx = 20
        if self.rect.centerx > 620: self.rect.centerx = 620
```

Создание лыжника

Поворот лыжника

Движение лыжника вправо и влево

```
class ObstacleClass(pygame.sprite.Sprite):
    def __init__(self, image_file, location, obs_type):
        pygame.sprite.Sprite.__init__(self)
        self.image_file = image_file
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.center = location
        self.obs_type = obs_type
        self.passed = False

    def update(self):
        global speed
        self.rect.centery -= speed[1]
        if self.rect.centery < -32:
            self.kill()

def create_map():
    global obstacles
    locations = []
    for i in range(10):
        row = random.randint(0, 9)
        col = random.randint(0, 9)
        location  = [col * 64 + 20, row * 64 + 20 + 640]
        if not (location in locations):
            locations.append(location)
            obs_type = random.choice(["tree", "flag"])
            if obs_type == "tree": img = "skier_tree.png"
            elif obs_type == "flag":  img = "skier_flag.png"
            obstacle = ObstacleClass(img, location, obs_type)
            obstacles.add(obstacle)

def animate():
    screen.fill([255, 255, 255])
    obstacles.draw(screen)
    screen.blit(skier.image, skier.rect)
    screen.blit(score_text, [10, 10])
    pygame.display.flip()

pygame.init()
screen = pygame.display.set_mode([640,640])
clock = pygame.time.Clock()
skier = SkierClass()
speed = [0, 6]
obstacles = pygame.sprite.Group()
map_position = 0
points = 0
create_map()
font = pygame.font.Font(None, 50)

running = True
while running:
    clock.tick(30)
```

Создание деревьев и флажков

Прокручивание карты вниз

Удаление препятствий, которые прокрутились в верхней части экрана

Создание одного окна с деревьями и флажками в случайном порядке

Рисуем экран заново при движении элементов

Подготовка всех элементов

Начало основного цикла

Обновление графики 30 раз в секунду

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                speed = skier.turn(-1)
            elif event.key == pygame.K_RIGHT:
                speed = skier.turn(1)
    skier.move(speed)

    map_position += speed[1]

    if map_position >= 640:
        create_map()
        map_position = 0

    hit = pygame.sprite.spritecollide(skier, obstacles, False)
    if hit:
        if hit[0].obs_type == "tree" and not hit[0].passed:
            points = points - 100
            skier.image = pygame.image.load("skier_crash.png")
            animate()
            pygame.time.delay(1000)
            skier.image = pygame.image.load("skier_down.png")
            skier.angle = 0
            speed = [0, 6]
            hit[0].passed = True
        elif hit[0].obs_type == "flag" and not hit[0].passed:
            points += 10
            hit[0].kill()

    obstacles.update()
    score_text = font.render("Счет: " + str(points), 1, (0, 0, 0))
    animate()
pygame.quit()
```

Проверка нажатий клавиатуры или закрытия окна

Перемещение лыжника

Прокручивание карты

Создание нового экрана с подробной картой

Проверка на столкновение с деревьями и флажками

Вывод счета

Код из листинга 10.1 находится в папке *Примеры*, поэтому если ты зайдешь в тупик или не захочешь набирать его вручную, можешь использовать этот файл. Но веришь или нет, ты узнаешь больше, если наберешь код самостоятельно, чем если просто откроешь файл и посмотришь его содержимое.

В следующих главах ты узнаешь обо всех ключевых словах и техниках, использованных в игре Лыжник. А в конце книги есть целая глава, подробно объясняющая, как работает игра Лыжник. Сейчас просто введи код и запусти программу.

Попробуй самостоятельно

Все, что тебе нужно сделать в этой главе, – это набрать код игры «Лыжник» (листинг 10.1) и запустить ее. Если ты получишь сообщение об ошибке, попытайся определить, где ты ошибся.

Удачи!

ГЛАВА 11

Вложенные и переменные циклы

Мы уже встречали подобное, в теле цикла (которое представляет собой блок кода) можно совместить объекты со своими собственными блоками. Если еще раз взглянуть на программу по угадыванию числа из главы 1, то можно заметить:

```
while guess != secret and tries < 6:
    guess = int(input(«Каким будет твое слово? «))
    if guess < secret:
        print(«Слишком мало, сухопутная крыса!»)
    elif guess > secret:
        print(«Слишком много, салага!»)
    tries = tries + 1
```

Блок цикла **while**

Блок **if**

Блок **elif**

Светло-серым цветом выделен блок цикла **while**, а более темным – блоки **if** и **elif** в блоке цикла **while**.

Также можно поместить один цикл внутри другого. Эти циклы называются *вложенными*.

Вложенные циклы

Помнишь программу с таблицей умножения из главы 8? В той части, где требуется ввод от пользователя, код мог быть примерно таким:

```
multiplier = 5
for i in range (1, 11):
    print(i, "x", multiplier, "=", i * multiplier)
```

Что, если нам нужно вывести на экран три таблицы умножения за один раз? Именно для таких задач идеально подходит *вложенный цикл*. Вложенный цикл – это один цикл внутри другого. За *каждую* итерацию внешнего цикла внутренний цикл выполняет *все* свои итерации.

Для того чтобы вывести на экран три таблицы умножения, мы просто заключим оригинальный цикл (для одной таблицы умножения) во внешний цикл (который выполняется три раза). Это заставит программу вывести три таблицы вместо одной. Листинг 11.1 содержит код программы.

Листинг 11.1. Вывод трех таблиц умножения

```
for multiplier in range(5, 8):
    for i in range(1, 11):
        print(i, "x", multiplier, "=", i * multiplier)
    print()
```

Этот внутренний цикл выводит одну таблицу

Этот внешний цикл запускает три итерации для чисел 5, 6, 7

Обрати внимание, что нам нужно выделить внутренний цикл и утверждение **`print`** дополнительными четырьмя отступами от цикла **`for`**. Эта программа выводит таблицы умножения для 5,6 и 7 (до значения 10 в каждой таблице):

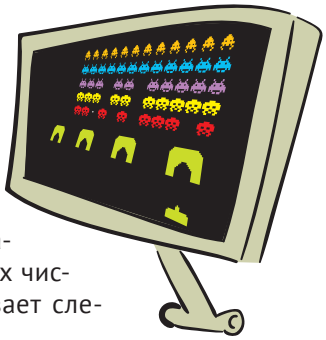
```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_11-1.py
1 x 5 = 5
2 x 5 = 10
3 x 5 = 15
4 x 5 = 20
5 x 5 = 25
6 x 5 = 30
7 x 5 = 35
8 x 5 = 40
9 x 5 = 45
10 x 5 = 50

1 x 6 = 6
2 x 6 = 12
3 x 6 = 18
4 x 6 = 24
5 x 6 = 30
6 x 6 = 36
7 x 6 = 42
8 x 6 = 48
9 x 6 = 54
10 x 6 = 60
```

```
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
6 x 7 = 42
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70
```

Хотя это может показаться делом скучным, но хороший способ наглядно увидеть работу вложенных циклов – вывести несколько звездочек на экран и посчитать их. Мы сделаем это в следующем разделе.

Переменные циклы

Фиксированные числа, например те, что мы использовали с функцией **`range()`**, также называются *константами*. Если использовать константы в функции **`range()`** цикла **`for`**, цикл будет выполнен одно и то же число раз при каждом запуске программы. В этом случае мы говорим, что число циклов *жестко задано*, потому что оно определено в твоем коде и никогда не меняется. Но нам не всегда нужен именно такой результат.



Иногда нам нужно, чтобы количество циклов определялось пользователем или другой частью программы. Для этого нам нужна переменная.

Например, ты делаешь игру – комическую стрелялку. Тебе нужно обновлять экран каждый раз при уничтожении всех пришельцев. Тебе нужен некий счетчик, чтобы отслеживать, сколько пришельцев осталось, был ли обновлен экран, цикл обновления экрана в зависимости от количества оставшихся пришельцев. А их число изменяется каждый раз, когда пользователь убивает следующего.

Поскольку мы еще не научились рисовать пришельцев на экране, ниже приведен пример программы с использованием цикла с переменной:

```
numStars = int(input(«Сколько звезд тебе нужно? “))
for i in range(1, numStars):
    print(‘* ‘, end=’’)

>>>
RESTART: C:/HelloWorld/Примеры/stars1.py
Сколько звезд тебе нужно? 5
* * * *
```

Программа спросила пользователя, сколько звезд необходимо, а затем использовала переменный цикл, чтобы их вывести на экран. Ну, почти! Мы просили пять звездочек, а получили только четыре! Ой, мы забыли, что цикл **for** останавливается на предыдущем числе в функции **range**. Поэтому нам нужно добавить единицу к вводу пользователя.

```
numStars = int(input(«Сколько звезд тебе нужно? “))
for i in range(1, numStars + 1):
    print(‘* ‘, end=’’)
```

Добавляет 1. Если попросим 5 звезд, получим 5 звезд

Еще один способ сделать то же самое – начать цикл с 0, а не с 1. (Мы говорили об этом в главе 8.) Это очень распространенная в программировании практика, и ты узнаешь, почему, в следующей главе. Это будет выглядеть так:

```
numStars = int(input(«Сколько звезд тебе нужно? “))
for i in range(0, numStars):
print(‘* ‘, end=’’)
>>>
RESTART: C:/HelloWorld/Примеры/stars2.py
Сколько звезд тебе нужно? 5
* * * * *
```

Вложенные переменные циклы

А теперь давай попробуем вложенный цикл с переменной. Это просто тот же вложенный цикл, где один или более циклов используют переменную в функции **range()**. В листинге 11.2 приведен пример.

Листинг 11.2. Вложенный цикл с переменной

```
numLines = int(input(‘Сколько строк звезд тебе нужно? ‘))
numStars = int(input(‘Сколько звезд в строке? ‘))
for line in range(0, numLines):
    for star in range(0, numStars):
        print(‘* ‘, end=’’)
    print()
```

Попробуй запустить программу, чтобы увидеть, работает ли она. Ты увидишь нечто подобное:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_11-2.py
Сколько строк звезд тебе нужно? 3
Сколько звезд в строке? 5
*****
*****
*****
```

Первые две строки в программе спрашивают у пользователя, сколько линий звезд ему нужно и по сколько звезд в каждой линии. Она запоминает ответы с помощью переменных **numLines** и **numStars**. Затем идут два цикла:

- внутренний цикл (**for star in range(0,numStars):**) выводит каждую звезду и выполняется один раз для каждой звездочки в строке;
- внешний цикл (**for line in range(0,numLines):**) выполняется один раз для каждой строки звезд.

Вторая команда **print** нужна для того, чтобы начать новую строку со звездами. Если бы ее не было, все звезды были бы выведены в одну линию из-за `end=''` в первой команде **print**.

Мы даже можем использовать вложенные-вложенные циклы (*двойные вложенные циклы*). Они будут выглядеть как в листинге 11.3.

Листинг 11.3. Блоки звезд с помощью двойных вложенных циклов

```
numBlocks = int(input('Сколько блоков звезд тебе нужно? '))
numLines = int(input('Сколько строк в каждом блоке? '))
numStars = int(input('Сколько звезд в строке? '))
for block in range(0, numBlocks):
    for line in range(0, numLines):
        for star in range(0, numStars):
            print('* ', end='')
        print()
    print()
```

Ниже приведен результат:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_11-3.py
Сколько блоков звезд тебе нужно? 3
Сколько строк в каждом блоке? 4
Сколько звезд в строке? 8
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *

* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *

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

Можно сказать, что цикл вложен «трижды».

Еще более глубоко вложенные циклы с переменной

В листинге 11.4 приведена более хитроумная версия программы из листинга 11.3.

Листинг 11.4. Более хитроумная версия блоков звезд

```
numBlocks = int(input('Сколько блоков звезд тебе нужно? '))
for block in range(1, numBlocks + 1):
    for line in range(1, block * 2):
        for star in range(1, (block + line) * 2):
            print('* ', end='')
        print()
    print()
```

Формулы для количества строк и звезд

Результат выглядит так:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_11-4.py
Сколько блоков звезд тебе нужно? 3
* * *

* * * * *
* * * * * * *
* * * * * * * * *

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

В листинге 11.4 переменные внешних циклов позволяют настроить последовательности внутренних циклов. Поэтому вместо одинакового количества строк в каждом блоке и одинакового количества звезд в каждой строке мы имеем разное число звезд и строк в каждом цикле.

Ты можешь вкладывать циклы столько раз, сколько тебе нужно. Может быть трудно уследить за тем, что при этом происходит, поэтому можно иногда выводить значения переменных циклов, как показано в листинге 11.5.

Листинг 11.5. Вывод переменных цикла во вложенных циклах

```
numBlocks = int(input('Сколько блоков звезд тебе нужно? '))
for block in range(1, numBlocks + 1):
    print('блок =', block)
    for line in range(1, block * 2 ):
        for star in range(1, (block + line) * 2):
            print('* ', end='')
        print('  строка =', line, 'звезд =', star)
    print()
```

Вывод переменных

Вывод:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_11-4.py
Сколько блоков звезд тебе нужно? 3
блок = 1
* * *  строка = 1  звезда = 3

блок = 2
* * * * *  строка = 1 звезд = 5
* * * * * * *  строка = 2 звезд = 7
* * * * * * * * *  строка = 3 звезд = 9

блок = 3
* * * * * * *  строка = 1 звезд = 7
* * * * * * * * *  строка = 2 звезд = 9
* * * * * * * * * * *  строка = 3 звезд = 11
* * * * * * * * * * * * *  строка = 4 звезд = 13
* * * * * * * * * * * * * * *  строка = 5 звезд = 15
```

Вывод значений переменных может помочь тебе во многих ситуациях – не только с циклами. Это один из самых распространенных методов отладки программы.

Использование вложенных циклов

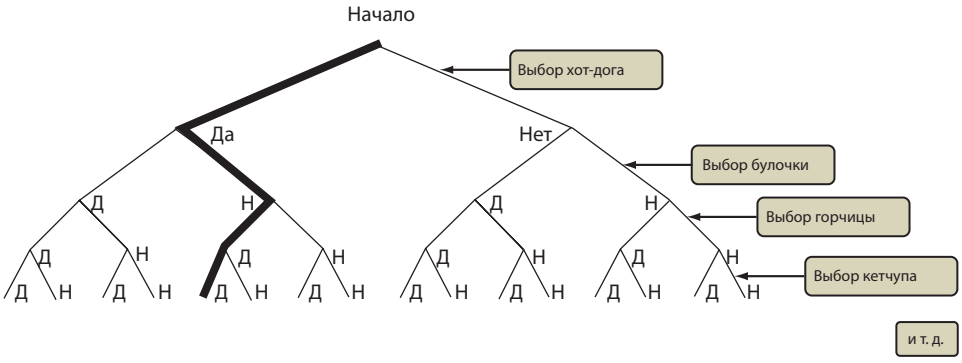
Итак, и что мы можем сделать со всеми этими вложенными циклами? Ну, одна из областей их применения – выяснение всех возможных *пермутаций* и *комбинаций* серии решений.

СЛОВАРИК

Перестановка – это математический термин, который означает уникальный способ комбинации нескольких объектов. *Комбинация* очень похожа на нее по смыслу. Разница между ними в том, что при комбинации порядок не важен, а при перестановке он имеет значение.
Если мы попросим тебя выбрать три числа от 1 до 20, ты можешь выбрать:
  5, 8, 14;
  2, 12,20.
И так далее. Если бы мы задались целью составить список всех перестановок трех чисел от 1 до 20, эти два варианта были бы разными:
  5, 8, 14;
  8, 5, 14,
– поскольку в перестановке важен порядок чисел. Если бы нам нужно было сделать список всех комбинаций, все три варианта были бы одним элементом списка:
  5, 8,14;
  8, 5, 14;
  8, 14, 5.
Потому что в комбинации порядок не важен.

Лучше всего это видно на примере. Давай представим, что ты открыл палатку с хот-догами на весенней ярмарке. И хочешь сделать плакат, иллюстрирующий все возможные для заказа комбинации сосиски, булочки, кетчупа, горчицы и лука по номерам. То есть нам нужно выяснить, каковы эти комбинации.

Первый способ – так называемое *дерево решений*. На рисунке показано дерево решений для вопроса о хот-догах.



Каждое решение имеет два варианта: Да и Нет. Каждая ветка дерева описывает разные комбинации частей хот-дога. Выделенная ветка говорит Да сосиске, Нет булочке, Да горчице и Да кетчупу.

Теперь давай с помощью вложенных циклов создадим список всех комбинаций – всех веток дерева решений. Поскольку точек принятия решений пять и пять

уровней в нашем дереве, то в программе будет пять вложенных циклов. (Рисунок выше показывает только первые четыре уровня дерева решений.)

Набери код из листинга 11.6 в редакторе IDLE и сохрани файл под именем *hotdog1.py*.

Листинг 11.6. Комбинации хот-догов

```
print(«\tСосиска \tБулочка \tКетчуп\tГорчица\tЛук»)
count = 1
for dog in [0, 1]:
    for bun in [0, 1]:
        for ketchup in [0, 1]:
            for mustard in [0, 1]:
                for onion in [0, 1]:
                    print("#", count, "\t", end='')
                    print(dog, "\t", bun, "\t", ketchup,
                          "\t", end='')
                    print(mustard, "\t", onion)
                    count = count + 1
```

Цикл для сосиски

Цикл для булочки

Цикл для кетчупа

Цикл для горчицы

Цикл для лука

Видишь, как циклы размещаются друг в друге? Вот чем вложенные циклы являются на самом деле – циклами внутри других циклов.

- Внешний цикл (сосиска) выполняется дважды.
- Цикл с булочкой выполняется дважды для каждой итерации цикла сосиски. То есть он выполняется 2 × 2 = 4 раза.
- Цикл кетчупа выполняется дважды для каждой итерации цикла сосиски. То есть 2 × 2 × 2 = 8 раз.

И так далее.

Самый глубокий цикл (тот, который отделен самым большим количеством отступов, – цикл лука) выполняется 2 × 2 × 2 × 2 × 2 = 32 раза. Это покрывает все возможные комбинации. То есть у нас есть 32 комбинации.

Если запустить программу из листинга 11.6, ты получишь следующее:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_11-6.py
      Сосиска   Булочка   Кетчуп   Горчица   Лук
№ 1       0         0        0         0        0
№ 2       0         0        0         0        1
№ 3       0         0        0         1        0
№ 4       0         0        0         1        1
№ 5       0         0        1         0        0
№ 6       0         0        1         0        1
№ 7       0         0        1         1        0
№ 8       0         0        1         1        1
№ 9       0         1        0         0        0
№ 10      0         1        0         0        1
№ 11      0         1        0         1        0
```

```
№ 12    0    1    0    1    1
№ 13    0    1    1    0    0
№ 14    0    1    1    0    1
№ 15    0    1    1    1    0
№ 16    0    1    1    1    1
№ 17    1    0    0    0    0
№ 18    1    0    0    0    1
№ 19    1    0    0    1    0
№ 20    1    0    0    1    1
№ 21    1    0    1    0    0
№ 22    1    0    1    0    1
№ 23    1    0    1    1    0
№ 24    1    0    1    1    1
№ 25    1    1    0    0    0
№ 26    1    1    0    0    1
№ 27    1    1    0    1    0
№ 28    1    1    0    1    1
№ 29    1    1    1    0    0
№ 30    1    1    1    0    1
№ 31    1    1    1    1    0
№ 32    1    1    1    1    1
```

Пять вложенных циклов описывают все возможные комбинации сосиски, булочки, кетчупа, горчицы и лука.

В листинге 11.6 мы использовали символ *tab* для выравнивания. Это символы **`\t`** в коде. Мы еще не говорили о форматировании вывода, но если ты хочешь узнать побольше, можешь заглянуть в главу 21.

Мы использовали переменную **`count`**, чтобы пронумеровать каждую комбинацию. Например, хот-дог с булочкой и горчицей будет № 27. Конечно, некоторые из этих 32 комбинаций не имеют смысла. (Хот-дог без булочки, но с горчицей и кетчупом будет странным.) Но знаешь, что говорят? – «Клиент всегда прав!»

Подсчет калорий

Поскольку всех сейчас волнует проблема здорового питания, давай добавим счетчик калорий для каждой комбинации в меню. (Ты можешь не беспокоиться о калориях вообще, но мы уверены, твоим родителям будет интересно!) Это позволит нам использовать некоторые математические способности Python, о которых мы узнали в главе 3.

Мы уже знаем, какие части присутствуют в каждой комбинации. Теперь нам всего лишь нужны значения их калорийности. Потом мы можем добавить эти значения в самый внутренний цикл.

Ниже приведен код, который устанавливает значения калорийности для каждого продукта:

```
dog_cal = 140
bun_cal = 120
mus_cal = 20
ket_cal = 80
onion_cal = 40
```

Теперь нам нужно их добавить. Мы знаем, что для каждого продукта в каждой комбинации есть значение либо 0, либо 1. Поэтому мы можем просто умножить это количество на значение калорийности для каждого продукта:

```
tot_cal = (dog * dog_cal) + (bun * bun_cal) + \
          (mustard * mus_cal) + (ketchup * ket_cal) + \
          (onion * onion_cal)
```

Поскольку умножение все равно выполняется раньше, чем сложение, тебе не нужны скобки. Мы поставили их, чтобы было удобнее просмотреть, что же происходит в коде.

Длинные строки кода

Ты заметил обратный слеш (\) в конце строк предыдущего примера кода? Если у тебя длинное выражение, которое не помещается в одну строку, ты можешь использовать обратный слеш, чтобы сказать Python: «Эта строка еще не закончена. Относись к содержимому следующей строки как к части этой». В коде выше мы использовали два обратных слеша, чтобы разделить длинную строку на три короткие. Этот символ называется символом *перехода строки*, и он присутствует в нескольких языках программирования.

Также можно заключить выражение в дополнительную пару скобок и разбить его на несколько строк без использования обратного слеша:

```
tot_cal = ((dog * dog_cal) + (bun * bun_cal) +
           (mustard * mus_cal) + (ketchup * ket_cal) +
           (onion * onion_cal))
```

Все вместе собрано в новую программу в листинге 11.7.

Листинг 11.7. Программа комбинаций хот-догов со счетчиком калорий

```
dog_cal = 140
bun_cal = 120
ket_cal = 80      # Список калорий для каждой части хот-дога
mus_cal = 20
onion_cal = 40

print("\tСосиска\tБулочка\tКетчуп\tГорчица\tЛук\tКалории") # Вывод заголовков
count = 1
for dog in [0, 1]: # Сосиска является внешним циклом
```

```
    for bun in [0, 1]:
        for ketchup in [0, 1]:
            for mustard in [0, 1]:
                for onion in [0, 1]:
                    total_cal = (bun * bun_cal)+(dog * dog_cal) + \
                        (ketchup * ket_cal)+(mustard * mus_cal) + \
                        (onion * onion_cal)
                    print("#", count, "\t", end='')
                    print(dog, "\t", bun, "\t", ketchup, "\t", end='')
                    print(mustard, "\t", onion, end='')
                    print("\t", total_cal)
                    count = count + 1
```

Подсчет калорий во внутреннем цикле

Вложенные циклы

Попробуй запустить программу в IDLE. Вывод должен выглядеть так:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_11-7.py
      Сосиска   Булочка   Кетчуп   Горчица   Лук   Ккалорий
№ 1      0         0        0         0       0       0
№ 2      0         0        0         0       1       40
№ 3      0         0        0         1       0       20
№ 4      0         0        0         1       1       60
№ 5      0         0        1         0       0       80
№ 6      0         0        1         0       1       120
№ 7      0         0        1         1       0       100
№ 8      0         0        1         1       1       140
№ 9      0         1        0         0       0       120
№ 10     0         1        0         0       1       160
№ 11     0         1        0         1       0       140
№ 12     0         1        0         1       1       180
№ 13     0         1        1         0       0       200
№ 14     0         1        1         0       1       240
№ 15     0         1        1         1       0       220
№ 16     0         1        1         1       1       260
№ 17     1         0        0         0       0       140
№ 18     1         0        0         0       1       180
№ 19     1         0        0         1       0       160
№ 20     1         0        0         1       1       200
№ 21     1         0        1         0       0       220
№ 22     1         0        1         0       1       260
№ 23     1         0        1         1       0       240
№ 24     1         0        1         1       1       280
№ 25     1         1        0         0       0       260
№ 26     1         1        0         0       1       300
№ 27     1         1        0         1       0       280
№ 28     1         1        0         1       1       320
№ 29     1         1        1         0       0       340
№ 30     1         1        1         0       1       380
№ 31     1         1        1         1       0       360
№ 32     1         1        1         1       1       400
```

А теперь представь, как утомительно было бы считать все эти калории вручную, даже с использованием калькулятора. Гораздо интереснее написать программу, которая сделает это за тебя. Циклы и немного математики в Python решают все!

Что ты узнал?

В этой главе ты узнал:

- о вложенных циклах;
- о циклах с переменными;
- о перестановках и комбинациях;
- о деревьях решений.

Проверь свои знания

1. Как в Python создать цикл с переменной?
2. Как в Python создать вложенный цикл?
3. Какое общее количество звездочек будет выведено на экран при выполнении следующего кода:

```
for i in range(5):
    for j in range(3):
        print('* ', end='')
    print()
```

4. Как будет выглядеть вывод кода из вопроса 3?
5. Если в дереве решений есть четыре уровня и два варианта выбора на каждом уровне, сколько возможных вариантов (ветвей) содержится в нем?

Попробуй самостоятельно

1. Помнишь программу обратного отсчета времени из главы 8? Ниже представлен ее код:

```
import time
for i in range(10, 0, -1):
    print(i)
    time.sleep(1)
print("СТАРТ!")
```

Измени программу так, чтобы она использовала цикл с переменной. Программа должна спрашивать пользователя, когда начинать отсчет. Например:

```
Таймер: Сколько секунд? 4
4
3
2
1
СТАРТ!
```

2 Возьми код программы из вопроса 2 и выведи линию звездочек после каждого числа. Например:

```
Таймер: Сколько секунд? 4
4 * * * *
3 * * *
2 * *
1 *
СТАРТ!
```

(Подсказка: тебе, вероятно, нужно использовать вложенный цикл.)

ГЛАВА 12

Собираем все воедино: списки и словари

Ты уже видел, как Python хранит объекты в своей памяти и вызывает их оттуда с помощью имен. До сих пор мы хранили *строки* и *числа* (*целые* и *десятичные дроби*). Иногда полезно хранить несколько объектов вместе в виде «группы» или «коллекции». Тогда ты сможешь производить действия с целой коллекцией и отслеживать группы. Одним из видов таких коллекций является *список, а другим – словарь*. В этой главе мы собираемся познакомиться с ними поближе – что это такое, как их создавать, изменять и использовать.

Списки используются во многих программах. Мы будем широко применять их в примерах программ из следующих глав, когда начнем создавать графику и программировать игры, поскольку многие графические объекты из игр хранятся в списках.

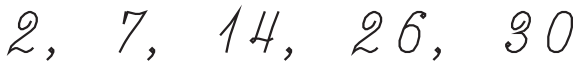
Что такое список?

Если бы мы попросили тебя составить список членов твоей семьи, ты бы написал что-то подобное:

А в Python ты бы написал:

```
family = [«мама», «папа», «старший», «младший»]
```

Если бы мы попросили тебя написать свои удачные числа, ты бы написал их так:



А в Python они выглядели бы так:

```
luckyNumbers = [2, 7, 14, 26, 30]
```

И **`family`**, и **`luckyNumbers`** – примеры списков в Python, а отдельные объекты внутри списков называются *элементами*. Как ты видишь, списки в Python не очень отличаются от обычных списков в реальной жизни. Списки используют квадратные скобки для обозначения начала и конца списка и запятые для разделения элементов внутри него.

Создание списка

`family` и **`luckyNumbers`** – это переменные. Ранее мы упоминали, что можно присвоить любое значение переменной. Мы уже использовали их с числами и строками, но переменные можно присвоить еще и спискам.

Список создается так же, как и любая другая переменная: методом присвоения ей чего-либо, как мы и сделали в случае с **`luckyNumbers`**. Так же можно создать пустой список:

```
newList = []
```

В квадратных скобках нет ни одного элемента, поэтому список пуст. Но что хорошего в пустом списке? Зачем он нам нужен?

Ну, довольно часто мы не знаем заранее, что будет в списке. Мы не знаем, сколько элементов в нем будет или каковы эти элементы. Мы просто знаем, что будем хранить что-то в списке. Если у нас есть пустой список, программа может добавить в него элементы. И как нам это сделать?

Добавление элементов в список

Чтобы добавить элемент в список, используй функцию **`append()`**. Попробуй в интерактивном режиме:

```
>>> friends = []  ← Создание нового пустого списка
>>> friends.append('Сережа')  ← Добавление элемента "Сережа" в список
>>> print(friends)
```

В результате ты получишь:

```
['Сережа']
```

Попробуй добавить еще один элемент:

```
>>> friends.append('Кристина')
>>> print(friends)
['Сережа', 'Кристина']
```

Запомни, что тебе нужно создать список (пустой или нет) прежде, чем ты начнешь добавлять в него элементы. Это похоже на приготовление торта: ты не можешь просто смешать ингредиенты – сначала нужно достать миску для них. Иначе все продукты разольются по столу.

Что за точка?

Зачем мы использовали точку между именами **`friends`** и **`append()`**? Ну, для этого нужно вникнуть в очень большую тему: объекты. Мы познакомимся с объектами в главе 14, а сейчас мы тебе предоставим простое объяснение.

СЛОВАРИК

Когда ты добавляешь что-то в список с помощью функции **`append()`**, объект *добавляется в конец* (англ. *append*) списка.

Многие вещи в Python являются *объектами*. Чтобы что-то сделать с объектом, тебе нужно его имя (имя переменной), затем точка, а потом само действие. Итак, чтобы добавить что-то в список **`friends`**, ты пишешь:

```
friends.append(что-то)
```

Списки могут содержать все, что угодно

Списки могут содержать любые данные, которые Python может хранить. Это могут быть числа, строки, объекты и даже *другие списки*. Элементы списка необязательно должны быть все одного вида. То есть один список может содержать как числа, так и строки, например. Список может выглядеть так:

```
my_list = [5, 10, 23.76, 'Привет', myTeacher, 7, another_list]
```

Давай создадим новый список с чем-нибудь простым, например с буквами алфавита, чтобы нам лучше было видеть, что происходит со списком. Введи в интерактивном режиме:

```
>>> letters = ['а', 'б', 'в', 'г', 'д']
```

Получение элементов из списка

Отдельные элементы из списка можно получить по их *порядковому номеру* или *индексу*. Нумерация в списке начинается с 0, поэтому первым элементом в нашем списке будет **`letters[0]`**.

```
>>> print(letters[0])
a
```

И попробуем еще раз:

```
>>> print(letters[3])
г
```

Почему нумерация начинается с 0, а не с 1?

Об этом многие программисты, инженеры и компьютерщики спорят со времен изобретения компьютеров. Мы не собирамся встревать в этот спор, поэтому давай просто примем ответ «потому что» и пойдем дальше...

Ладно, ладно! Загляни в раздел «Что здесь происходит», чтобы найти объяснение, почему нумерация начинается с 0, а не с 1.

ЧТО ТАМ ПРОИСХОДИТ?

Напомним, что компьютеры используют двоичные цифры или биты для хранения информации. В давние времена эти биты стоили дорого. Каждый должен быть собран вручную и доставлен на ослике с битовой плантации… шучу. Но они действительно были дорогими.

Двоичный счет начинается с 0. Поэтому, чтобы использовать биты наиболее эффективно, вещи наподобие ячеек памяти и индексов списков начинаются также с 0.

Ты быстро привыкнешь к нумерации с 0. Она очень распространена в программировании.

ДЛИННОЕ УМНОЕ СЛОВО!

Индекс означает позицию чего-либо. Множественное число от *индекс* будет *индексы*.
Если ты четвертый человек в ряду, то твой индекс будет 4. Но если ты четвертый в списке Python, то твой индекс будет 3, потому что нумерация в списках Python начинается с 0!

«Нарезка» списка

Индексы также можно использовать, чтобы получить более одного элемента из списка за один раз. Это называется *нарезкой* списка.

```
>>> print(letters[1:4])
['б', 'в', 'г']
```

Как и функция **`range()`** в циклах **`for`**, нарезка выводит числа, начиная с первого индекса, но останавливается *перед* вторым индексом. Поэтому мы получили три элемента вместо четырех в предыдущем примере. Один из способов это запомнить: количество получаемых элементов всегда равно разнице между двумя индексами. (4 – 1 = 3, и мы получили три элемента.)

Что еще важно запомнить о нарезке списка: при этом ты получаешь обычно другой (более короткий) список. Этот более короткий список называется *сектором* оригинального списка. Оригинальный список не изменился. Сектор – это частичная *копия* оригинала.

Взгляни на разницу:

```
>>> print(type(letters[1]))
<class 'str'>
>>> print(type(letters[1:2]))
<class 'list'>
```

Вывод типа каждого варианта наверняка покажет тебе, что в одном случае ты получаешь отдельный элемент (в данном случае строку), а во втором – *список*.

Меньший список, который ты получаешь при нарезке, – копия элементов оригинального списка. То есть ты можешь его изменять, не затрагивая оригинальный список.

Сокращения при нарезке

Есть некоторые сокращения, которые можно использовать при нарезке списка. Они не экономят много времени, но программисты – ленивые ребята, поэтому они используют сокращения повсюду. Мы хотим, чтобы ты о них знал, чтобы мог определить их в чужом коде и понимать, что происходит. Это важно, потому что просмотр чужого кода и понимание его – хороший способ выучить новый язык программирования или программирование в целом.

Если фрагмент, который тебе нужен, включает в себя начало списка, простым методом будет использовать двоеточие, а за ним – число элементов, которое тебе требуется:

```
>>> print(letters[:2])
['a', 'б']
```

Обрати внимание, что перед двоеточием нет числа. Это сокращение выводит все элементы с начала списка вплоть до (но не включая) индекса, который ты указал.

Можно использовать нечто подобное, чтобы получить окончание списка:

```
>>> print(letters[2:])
['в', 'г', 'д']
```

С помощью числа и последующего за ним двоеточия можно получить все элементы от индекса, указанного тобой, до конца списка.

Если не указывать никакие числа, а просто использовать двоеточие, ты получишь весь список:

```
>>> print(letters[:])
['а', 'б', 'в', 'г', 'д']
```

Помнишь, мы говорили, что нарезка создает копию списка? Поэтому **`letters[:]`** создает копию всего списка. Это очень удобно, если ты хочешь внести некоторые изменения в список, но сохранить оригинальный неизменным.

Изменение элементов

Можно использовать индекс для изменения одного из элементов списка:

```
>>> print(letters)
['а', 'б', 'в', 'г', 'д']
>>> letters[2] = '3'
>>> print(letters)
['а', 'б', '3', 'г', 'д']
```

Но ты не можешь добавлять новые элементы с помощью индекса. Сейчас у нас в списке пять элементов с индексами от 0 до 4. Поэтому мы *не* можем сделать нечто подобное:

```
letters[5] = 'ф'
```

Это просто не сработает. (Попробуй, если хочешь.) Это похоже на попытку изменить то, чего еще не существует. Чтобы добавить элементы в список, нужно сделать нечто иное, и ты скоро узнаешь, что. Но сначала давай вернем список к первоначальному виду:

```
>>> letters[2] = 'в'
>>> print(letters)
['а', 'б', 'в', 'г', 'д']
```

Другие способы добавления элементов в список

Ты уже видел, как добавлять элементы в список с помощью функции **`append()`**. Но есть и другие способы. Фактически их три: **`append()`**, **`extend()`** и **`insert()`**:

- **`append()`** добавляет один элемент в конец списка;
- **`extend()`** добавляет несколько элементов в конец списка;
- **`insert()`** добавляет один элемент в любое место списка, необязательно в конец. Ты сам указываешь, куда его добавить.

Добавление в конец списка: функция `append()`

Ты уже видел функцию **`append()`** в работе. Она добавляет один элемент в конец списка:

```
>>> letters.append('н')
>>> print(letters)
['а', 'б', 'в', 'г', д', 'н']
```

Давай добавим еще один элемент:

```
>>> letters.append('ж')
>>> print(letters)
['а', 'б', 'в', 'г', 'д', 'н', 'ж']
```

Обрати внимание, что буквы идут не по порядку. Потому что **`append()`** добавляет элемент в конец списка. Если мы хотим упорядочить элементы, то их нужно *отсортировать*. Мы скоро займемся сортировкой.

Расширение списка: функция `extend()`

Функция **`extend()`** добавляет несколько элементов в конец списка:

```
>>> letters.extend(['р', 'к', 'п'])
>>> print(letters)
['а', 'б', 'в', 'г', 'д', 'н', 'ж', 'р', 'к', 'п']
```

Обрати внимание, что в скобках функции заключен список. Список имеет квадратные скобки, то есть для **`extend()`** можно использовать как квадратные, так и круглые скобки.

Все элементы в списке, который ты добавляешь с помощью **`extend()`**, появляются в конце оригинального списка.

Вставка элемента: функция `insert()`

Функция **`insert()`** добавляет один элемент в любое место списка. Ты сам указываешь месторасположение для нового элемента:

```
>>> letters.insert(2, 'з')
>>> print(letters)
['а', 'б', 'з', 'в', 'г', 'д', 'н', 'ж', 'р', 'к', 'п']
```

Сейчас мы добавили букву *з* под индексом 2. Индекс 2 – это третья позиция в списке (потому что индексы начинаются с 0). Буква, которая была на третьем месте, *в*, сдвинулась на одну позицию, на четвертую. Каждый следующий элемент в списке также сдвинулся на одну позицию.

Разница между функциями `append()` и `extend()`

Иногда функции **`append()`** и **`extend()`** кажутся очень похожими, но они производят разные действия. Давай вернемся к оригинальному списку. Во-первых, попробуй использовать функцию **`extend()`** с тремя разными элементами:

```
>>> letters = ['а', 'б', 'в', 'г', 'д']
>>> letters.extend(['е', 'е', 'ж'])
>>> print(letters)
['а', 'б', 'в', 'г', 'д', 'е', 'е', 'ж']
```

А теперь давай то же самое попробуем сделать с помощью функции **`append()`**:

```
>>> letters = ['а', 'б', 'в', 'г', 'д']
>>> letters.extend(['е', 'е', 'ж'])
>>> print(letters)
['а', 'б', 'в', 'г', 'д', ['е', 'е', 'ж']]
```

Что произошло? Ну, мы уже говорили, что функция **`append()`** добавляет *один* элемент к списку. Почему она добавила три? Она не сделала этого. Она добавила один элемент, который оказался *другим списком из трех элементов*. Поэтому мы получили дополнительный набор квадратных скобок внутри первого списка. Напомним, что список может содержать что угодно, включая другие списки. Это мы и получили.

Функция **`insert()`** работает так же, как и **`append()`**, кроме того что ты сам говоришь ей, куда поместить новый элемент. Функция **`append()`** всегда помещает его в конце.

Удаление элементов из списка

Как удалять элементы из списка? Есть три способа: **`remove()`**, **`del`** и **`pop()`**.

Удаление с помощью функции `remove()`

Функция **`remove()`** удаляет элемент, выбранный тобой, из списка и «отбрасывает» его:

```
>>> letters = ['а', 'б', 'в', 'г', 'д']
>>> letters.remove('в')
>>> print(letters)
['а', 'б', 'г', 'д']
```

Тебе не нужно знать, где именно в списке находится элемент. Тебе лишь надо знать, что он там есть. Если ты попробуешь удалить что-то, чего в списке нет, то получишь сообщение об ошибке:

```
>>> letters.remove('ф')
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    letters.remove('f')
ValueError: list.remove(x): x not in list
```

Как же тебе узнать, содержит ли список определенный элемент? Сейчас узнаешь. Во-первых, давай узнаем и об остальных способах удаления элементов.

Удаление с помощью операции `del`

Операция **`del`** позволяет удалять элемент из списка с помощью его индекса, например:

```
>>> letters = ['а', 'б', 'в', 'г', 'д']
>>> del letters[3]
>>> print(letters)
['а', 'б', 'в', 'д']
```

Мы только что удалили четвертый элемент (индекс 3), то есть букву *г*.

Удаление с помощью функции `pop()`

Функция **`pop()`** забирает *последний* элемент списка и отдает его тебе. Таким образом, ты можешь присвоить ему имя, например:

```
>>> letters = ['а', 'б', 'в', 'г', 'д']
>>> lastLetter = letters.pop()
>>> print(letters)
['а', 'б', 'в', 'г']
>>> print(lastLetter)
д
```

Также функцию **pop()** можно использовать с индексом:

```
>>> letters = ['а', 'б', 'в', 'г', 'д']
>>> second = letters.pop(1)
>>> print(second)
б
>>> print(letters)
['а', 'в', 'г', 'д']
```

Сейчас мы удалили вторую букву (индекс 1), то есть *б*. Удаленный элемент был присвоен переменной **second** и удален из списка **letters**.

При отсутствии значения в скобках **pop()** отдает последний элемент и удаляет его из списка. Если в скобки заключить число, **pop(n)** отдает элемент с этим индексом и удаляет его из списка.

Поиск по списку

Если у нас в списке есть несколько элементов, как нам найти нужный? Часто тебе приходится делать со списком две вещи:

- выяснять, есть ли элемент в списке;
- выяснять, где этот элемент в списке находится (его индекс).

Ключевое слово in

Чтобы узнать, есть ли элемент в списке, используется ключевое слово **in**, например:

```
if 'а' in letters:
    print("буква 'а' найдена")
else:
    print(«буква 'а' не найдена»)
```

Та часть, где говорится, что **'а' in letters**, – это *булево*, или *логическое*, выражение. Оно принимает значение **true**, если *а* есть в списке, и **false**, если нет.

СЛОВАРИК

Булево, или логическое, – выражение, которое может принимать одно из двух значений: 0 и 1 или **true** и **false** (правда и ложь). Оно было изобретено математиком Джорджем Булем и используется при сравнении правдивых и ложных условий (представленных 0 и 1) вместе с **and**, **or** и **not** (как ты видел в главе 7).

Ты можешь выполнить следующий код в интерактивном режиме:

```
>>> 'a' in letters
True
>>> 'c' in letters
False
```

Этот пример показывает нам, что в списке с именем **letters** есть элемент *a*, но нет элемента *c*. То есть *a* есть в списке, а *c* в нем нет. Теперь ты можешь совместить ключевое слово **in** и функцию **remove()** и написать код, который не приведет к ошибке, даже если элемента в списке нет:

```
if 'a' in letters:
    letters.remove('a')
```

Этот код лишь удаляет элемент из списка, только если он там есть.

Определение индекса

Чтобы найти местоположение элемента в списке, используй метод **index()**, например:

```
>>> letters = ['а', 'б', 'в', 'г', 'д']
>>> print(letters.index('г'))
3
```

Теперь мы знаем, что буква *г* имеет индекс 3, что значит, что она идет четвертой в списке.

Как и функция **remove()**, **index()** выведет сообщение об ошибке, если значение не будет найдено в списке, поэтому будет неплохо использовать ее вместе с **in**:

```
if 'г' in letters:
    print(letters.index('г'))
```

Зацикливание по списку

Когда мы говорили о циклах, то упоминали, что циклы перебирают *список* значений. Мы также узнали о функции **range()**, использовали ее в качестве сокращения для генерирования списков чисел для циклов. Еще напомним, что функция **range()** выдает список чисел.

Но цикл может перебирать любой список – необязательно содержащий числа. Допустим, мы хотим вывести список букв по одной в каждой строке. Мы можем сделать что-то подобное:

```
>>> letters = ['а', 'б', 'в', 'г', 'д']
>>> for letter in letters:
        print(letter)
а
б
в
г
д
```

В этот раз переменной цикла была **letter**. (Ранее мы использовали такие переменные, как **looper** или **i**, **j** и **k**.) Цикл перебирает все значения списка, и каждую итерацию текущий элемент присваивается переменной цикла **letter** и выводится на экран.

Сортировка списков

Списки – это *упорядоченный* вид коллекции. То есть элементы в списке содержатся в определенном порядке, и у каждого есть свое место, индекс. Когда ты помещаешь элементы в список в определенном порядке, они сохраняют этот порядок, пока ты не изменишь список с помощью функций **insert()**, **append()**, **remove()** или **pop()**. Но этот новый порядок может быть не тем порядком, который тебе нужен. Ты можешь *отсортировать* список перед использованием.

Для этого используется функция **sort()**:

```
>>> letters = ['г', 'а', 'д', 'в', 'б']
>>> print(letters)
['г', 'а', 'д', 'в', 'б']
>>> letters.sort()
>>> print(letters)
['а', 'б', 'в', 'г', 'д']
```

Функция **sort()** автоматически сортирует строки в алфавитном порядке, а числа – в арифметическом, от самого маленького к самому большому.

Важно знать, что **sort()** изменяет список на месте. То есть изменения вносятся в оригинальный список. Она *не* создает новый, отсортированный список. Ты не можешь сделать так:

```
>>> print(letters.sort())
```

Если ты так сделаешь, то получишь ответ «None».

Нужно выполнить два шага:

```
>>> letters.sort()
>>> print(letters)
```

Сортировка в обратном порядке

Есть два способа отсортировать список в обратном порядке. Первый: отсортировать в обычном порядке, а затем *перевернуть* отсортированный список:

```
>>> letters = ['г', 'а', 'д', 'в', 'б']
>>> letters.sort()
>>> print(letters)
['а', 'б', 'в', 'г', 'д']
>>> letters.reverse()
>>> print(letters)
['д', 'г', 'в', 'б', 'а']
```

Мы только что использовали новую функцию списков **`reverse()`**, которая меняет порядок элементов в списке с прямого на обратный.

Второй способ: добавить параметр в **`sort()`**, чтобы сортировка прошла в обратном порядке (от самого большого к меньшему):

```
>>> letters = ['г', 'а', 'д', 'в', 'б']
>>> letters.sort(reverse = True)
>>> print(letters)
['д', 'г', 'в', 'б', 'а']
```

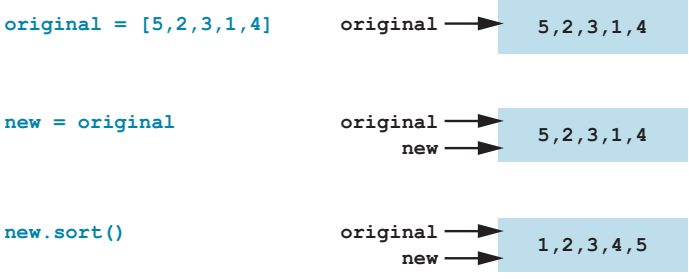
Параметр называется `reverse` и делает именно то, что тебе надо, – сортирует список в обратном порядке.

Напомним, что все эти сортировки и обратные порядки изменяют оригинальный список. То есть начальный порядок элементов в списке теряется. Если ты хочешь его сохранить и сортировать *копию* списка, то можешь сделать копию, о чем мы говорили ранее в этой главе, – другой список, идентичный оригинальному:

```
>>> original_list = ['Толя', 'Женя', 'Маша', 'Федя']
>>> new_list = original_list[:]
>>> new_list.sort()
>>> print(original_list)
['Толя', 'Женя', 'Маша', 'Федя']
>>> print(new_list)
['Федя', 'Женя', 'Маша', 'Толя']
```

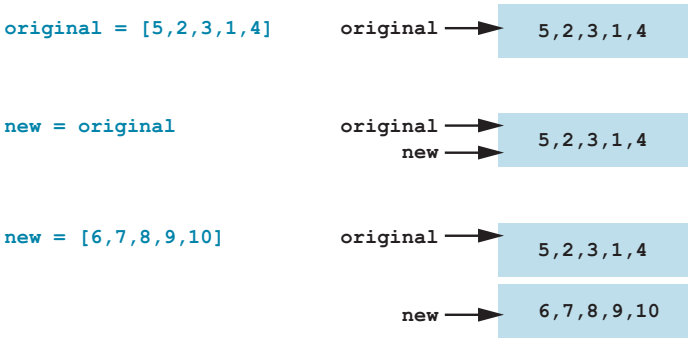
Здóрово, что ты спросил, Картер. Если ты помнишь, раааааааньше, когда мы говорили об именах и переменных (глава 2), мы упоминали, что когда делаешь что-то типа **name1=name2**, то ты просто создаешь новое имя для того же значения. Помнишь картинку:

Поэтому переименование чего-либо просто присваивает новую метку тому же объекту. В примере Картера **new_list** и **original_list** относятся к одному и тому же списку. Ты можешь изменить список (например, отсортировать его) с помощью любого имени. Но список до сих пор *один*. Это выглядит так:



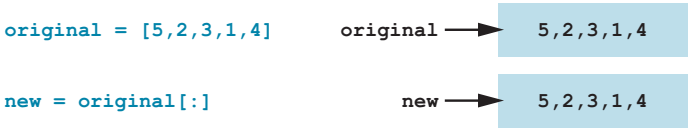
Мы отсортировали **new**, но **original** тоже был отсортирован, потому что new и **original** – два разных имени одного списка. Это *не* два разных списка.

Ты можешь, конечно, присвоить метку **new** абсолютно новому списку:



То же самое мы делали со строками и числами в главе 2.

Это значит, что если ты хочешь сделать *копию* списка, тебе нужны несколько другие действия, чем **new=original**. Легче всего это реализовать с помощью нарезки, как и было показано выше: **new=original[:]**. Это значит «скопируй все из списка от первого до последнего элемента». И ты получаешь следующее:



Теперь у тебя два разных списка. Мы создали копию оригинального списка и назвали его **new**. Теперь при сортировке одного из них второй останется нетронутым.

Еще один способ сортировки – sorted()

Есть еще один способ сортировать копию списка без изменения оригинального. В Python есть функция **sorted()** для этой цели. Она работает так:

```
>>> original = [5, 2, 3, 1, 4]
>>> newer = sorted(original)
>>> print(original)
[5, 2, 3, 1, 4]
>>> print(newer)
[1, 2, 3, 4, 5]
```

Функция **sorted()** выводит *отсортированную копию* оригинального списка.

Изменяемый и неизменяемый

Если ты помнишь главу 2, мы говорили, что ты не можешь изменить саму строку или число, но ты можешь изменить то, какое число или строка были присвоены *имени* (другими словами, перенести метку). Но списки – это из тех вещей в Python, которые *можно* изменить. Как ты только что видел, в списки можно добавлять элементы, удалять их, сортировать и менять порядок на обратный.

Эти два вида переменных называются *неизменными* и *изменяемыми*. *Изменяемый* означает «могущий быть измененным». *Неизменный* значит «не подвергающийся изменениям». В Python числа и строки неизменны, а списки изменяемы.

Кортеж – неизменяемый список

Иногда тебе не нужен изменяемый список. А есть ли в Python неизменяемые списки? Конечно, есть. Есть так называемый *кортеж*, который именно такой, неизменяемый. Создать его можно так:

```
my_tuple = («красный», «зеленый», «синий»)
```

Вместо квадратных скобок используются круглые.

Поскольку кортежи неизменны, ты не можешь сортировать их, добавлять или удалять элементы. Когда кортеж создан, он остается таким всегда.

Списки списков: таблицы данных

Чтобы понять, как данные хранятся в программе, давай представим это визуально.

У переменной есть одно значение.

Список похож на последовательность значений, связанных вместе.

Иногда нужна целая *таблица* со строками и столбцами:

Как нам сохранить таблицу данных? Мы уже знаем, что в списке может находиться несколько элементов. Мы можем поместить оценки каждого ученика в список, например:

```
>>> joeMarks = [55, 63, 77, 81]
>>> tomMarks = [65, 61, 67, 72]
>>> bethMarks = [97, 95, 92, 88]
```

Или можно создать список с оценками для каждого предмета:

```
>>> mathMarks = [55, 65, 97]
>>> scienceMarks = [63, 61, 95]
>>> readingMarks = [77, 67, 92]
>>> spellingMarks = [81, 72, 88]
```

Но еще мы можем собрать все данные вместе в одной *структуре данных*.

СЛОВАРИК

Структура данных – способ сбора, хранения и представления информации в программе. Структуры данных могут включать в себя переменные, списки и другие, еще неизвестные нам объекты. Термин *структура данных* на самом деле относится к способу организации информации в программе.

Чтобы создать структуру данных для нашей таблицы с оценками, можно сделать следующее:

```
>>> classMarks = [joeMarks, tomMarks, bethMarks]
>>> print(classMarks)
[[55, 63, 77, 81], [65, 61, 67, 72], [97, 95, 92, 88]]
```

Это дает нам список элементов, каждый из которых тоже является списком. Мы создали *список списков*. Каждый элемент списка **`classMarks`** сам есть список.

Мы могли создать **`classMarks`** напрямую, не создавая предварительно списки **`ivanMarks`**, **`dimaMarks`**, **`dashaMarks`**:

```
>>> classMarks = [ [55,63,77,81], [65,61,67,72], [97,95,92,88] ]
>>> print(classMarks)
[[55, 63, 77, 81], [65, 61, 67, 72], [97, 95, 92, 88]]
```

Теперь давай выведем нашу структуру данных. В списке `classMarks` три элемента, по одному на каждого ученика. Мы можем создать цикл из них с помощью **`in`**:

```
>>> for studentMarks in classMarks:
        print(studentMarks)

[55, 63, 77, 81]
[65, 61, 67, 72]
[97, 95, 92, 88]
```

Мы сделали цикл по списку **`classMarks`**. Переменная цикла – **`studentMarks`**. Каждый раз во время цикла мы выводим по одному элементу списка. Этот один элемент – оценки для каждого ученика, которые сами по себе являются списком. (Списки для учеников мы создали ранее.)

Обрати внимание, что это выглядит очень похоже на таблицу. То есть мы пришли к структуре данных, хранящей все данные в одном месте.

Получение отдельного значения из таблицы

Как нам получить доступ к значениям в этой таблице (нашем списке списков)? Мы уже знаем, что оценки первого ученика (**`ivanMarks`**) находятся в списке, который является первым в **`classMarks`**.

Давай проверим:

```
>>> print(classMarks[0])
[55, 63, 77, 81]
```

`classMarks[0]` – это список оценок Ивана по четырем предметам. Теперь нам нужно отдельное значение из этого списка. Как нам его получить? Используем второй индекс.

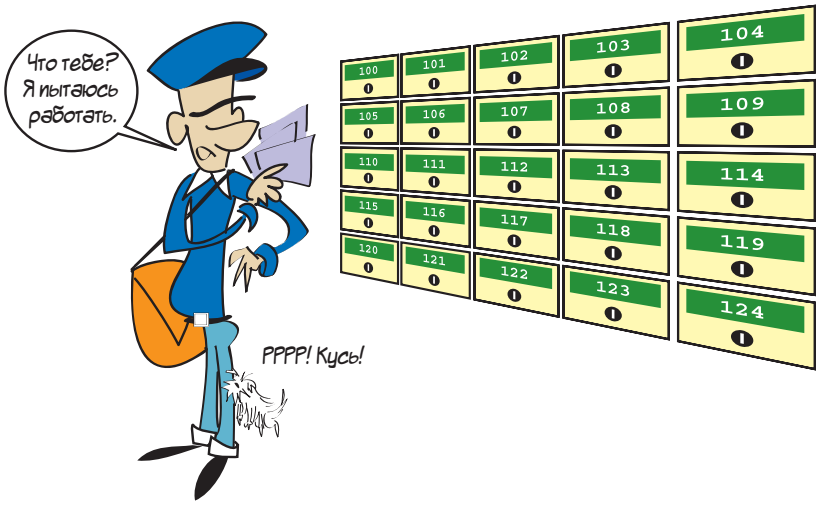
Если нам нужна третья из его оценок (оценка по чтению), индекс которой 2, мы делаем следующее:

```
>>> print(classMarks[0][2])
77
```

Так мы получаем первый элемент в списке **`classMarks`** (индекс 0), который был списком оценок Ивана, и третий элемент в этом списке (индекс 2), который был его оценкой по чтению. Когда ты видишь имя с двумя парами квадратных скобок, например **`classMarks[0] [2]`**, оно обычно относится к списку списков.

classMarks →	Математика	Физика	Чтение	Правописание
Ваня	55	63	77	81
Дима	65	61	67	72
Даша	97	95	92	88

Список **classMarks** на самом деле не знает о предметах или именах Ивана, Димы и Даши. Мы присвоили им такие имена, потому что знаем, что мы сохранили в списке. Но для Python они всего лишь пронумерованные места в списке. Похоже на пронумерованные почтовые ящики на почте. На них нет имен, только номера. Почтальон следит за тем, что находится в каком ящике, а ты знаешь, какой ящик твой.



Более точный способ подписать **classMarks** выглядит так:

classMarks →	[0]	[1]	[2]	[3]
classMarks[0]	55	63	77	81
classMarks[1]	65	61	67	72
classMarks[2]	97	95	92	88

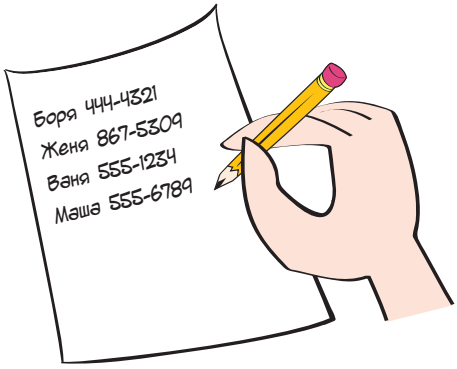
Теперь ты лучше видишь, что оценка 77 хранится в **`classMarks[0] [2]`**.

Если бы мы писали программу с использованием **`classMarks`** для хранения данных, нам бы пришлось следить за тем, какие данные хранятся в какой строке и столбце. Как и почтальон, мы бы следили, какая ячейка предназначена для каждого сегмента данных.

Словари

Ты уже знаешь, что список Python – это способ группировки элементов. Довольно часто в программировании необходимо сгруппировать элементы таким образом, чтобы связать одно значение с другим. Это похоже на телефонную книгу: имена связаны с номерами телефонов. Так же и в словаре: слова связаны с определениями.

Словарь Python – это способ связать два элемента друг с другом. Эти два элемента называются ключом и значением. Каждый элемент или запись в словаре имеет ключ и значение. Чуть позже ты увидишь, что они называются парами ключ-значение. Словарь – это набор пар ключ-значение.



Простой пример – список телефонных номеров. Допустим, ты хочешь сохранить список телефонных номеров своих друзей. Каждому номеру телефона будет соответствовать имя друга. (Надеюсь, у тебя нет друзей с одинаковыми именами.) Имя будет ключом (то, что ты будешь использовать для поиска информации), а номер телефона – значением (то, что ты будешь искать).

Вот один из способов создать словарь Python для хранения имен и телефонных номеров. Для начала давай создадим пустой словарь:

```
>>> phoneNumbers = {}
```

Это очень похоже на создание списков, за исключением того, что при создании словарей ты используешь фигурные скобки, а при создании списков – квадратные.

Затем давай добавим запись:

```
>>> phoneNumbers[«John»] = «555-1234»
```

Если отобразить содержимое словаря, вывод будет следующим:

```
>>> print(phoneNumbers)
{‘Ваня’: ‘555-1234’}
```

Сначала указывается ключ, затем ставится двоеточие, а после пишется значение. Кавычки ставятся потому, что и ключ, и значение в этом случае являются строками (но они не обязательно должны быть). Но есть и другой способ сделать то же самое:

```
>>> phoneNumbers = {«Ваня»: «555-1234»}
```

Давай добавим еще несколько имен. В отличие списков, где есть функция **append()**, в словарях нет функций для добавления новых элементов. Тебе просто нужно ввести новый ключ и значение:

```
>>> phoneNumbers[«Маша»] = «555-6789»
>>> phoneNumbers[«Боря»] = «444-4321»
>>> phoneNumbers[«Женя»] = «867-5309»
```

Вот как выглядит наш словарь целиком:

```
>>> print(phoneNumbers)
{‘Ваня’: ‘555-1234’, ‘Маша’: ‘555-6789’, ‘Боря’: ‘444-4321’, ‘Женя’:
‘867-5309’}
```

Словарь создается для того, чтобы иметь возможность быстро получить доступ к каким-либо элементам. В данном случае мы хотим найти номер телефона по имени. Вот что мы делаем:

```
>>> print(phoneNumbers[«Маша»])
555-6789
```

Обрати внимание, что мы используем квадратные скобки, чтобы указать, какой ключ хотим найти в словаре. Но словарь в целом заключен в фигурные скобки.

Словарь чем-то напоминает список, однако есть несколько основных отличий. И словарь, и список являются коллекциями, то есть представляют собой способ группировки отдельных элементов.

Вот некоторые сходства:

- и списки, и словари могут содержать любой тип данных (даже списки и словари), поэтому у тебя может быть коллекция чисел, строк, объектов и даже других коллекций;
- и списки, и словари позволяют найти элементы в коллекции;
- списки и словари упорядочены. Если ты помещаешь элементы в список или словарь в определенном порядке, они остаются в этом порядке при демонстрации.

А вот в чем главное отличие:

- доступ к элементам списка осуществляется по индексу, а доступ к элементам словаря – по ключу:

```
>>> print(myList[3])
eggs
>>> print(myDictionary[“Ваня”])
555-1234
```

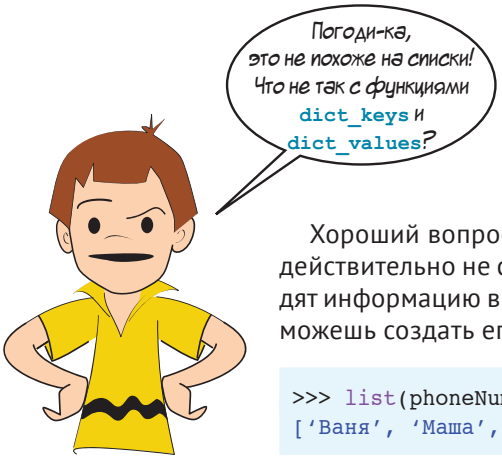
Как мы уже упоминали, многие элементы в Python – это объекты, включая списки и словари. Как в случае со списками, при работе со словарями ты можешь воспользоваться определенными функциями, используя точечную нотацию, которую ты уже видел раньше.

Функция **`keys()`** позволяет отобразить список всех ключей словаря:

```
>>> phoneNumbers.keys()
dict_keys([‘Ваня’, ‘Маша’, ‘Боря’, ‘Женя’])
```

Функция **`values()`** дает возможность отобразить список всех значений словаря:

```
>>> phoneNumbers.values()
dict_values([‘555-1234’, ‘555-6789’, ‘444-4321’, ‘867-5309’])
```



Хороший вопрос, Картер! Функции **`keys()`** и **`values()`** действительно не создают списки – вместо этого они выводят информацию в одну строчку. Если тебе нужен список, ты можешь создать его с помощью функции **`list()`**:

```
>>> list(phoneNumbers.keys())
[‘Ваня’, ‘Маша’, ‘Боря’, ‘Женя’]
```

Функцию **`list()`** также можно применить к другим типам значений, таким как строки и диапазоны:

```
>>> list(«Привет!»)
[‘П’, ‘р’, ‘и’, ‘в’, ‘е’, ‘т’, ‘!’]
>>> list(range(2,5))
[2, 3, 4]
```

Еще несколько слов о словарях

В других компьютерных языках также есть вещи, похожие на словари Python. Их обычно называют *ассоциативными массивами* (потому что они связывают ключи и значения друг с другом), *картами* или *хеш-таблицами*.

Как в списке, элементы в словаре могут быть любого типа: простые (целые числа, числа с плавающей точкой, строки), коллекции (списки или словари) или другие объекты. Это означает, что ты можешь создать словарь, содержащий другие словари, точно так же, как можешь сформировать список списков.

Но на самом деле это не совсем так. Ты можешь использовать все, что хочешь, в качестве значения в словаре, но выбор ключей ограничен. Ранее мы говорили об изменяемых и неизменяемых типах данных. Ключи в словаре могут быть только неизменяемыми (логические (булевы) значения, целые числа, числа с плавающей точкой, строки и кортежи). Также ты не можешь использовать список или словарь в качестве ключа, потому что они относятся к изменяемым данным.

Уже упоминалось, что словари хранят элементы в том порядке, в каком они были введены (как и списки). Но иногда может понадобиться отобразить элементы словаря в другом порядке, например в алфавитном. Это немного сложно, потому что в словарях нет функции **`sort()`**, как в списках. Однако напомним, что ключи словаря представляют собой список, поэтому ты можешь сортировать их и таким образом отсортировать содержимое словаря, например:

```
>>> for key in sorted(phoneNumbers.keys()):
        print(key, phoneNumbers[key])

Боря 444-4321
Ваня 555-1234
Женя 867-5309
Маша 555-6789
```

Это та же самая функция **`sort()`**, которую мы использовали при работе со списками. И в этом нет ничего удивительного, потому что коллекция ключей словаря похожа на список.

А что делать, если тебе нужно сортировать значения элементов, а не ключи? В нашем примере с телефонными номерами это будет означать сортировку по телефонным номерам, от самого меньшего числа до самого большего. По правде говоря, поиск в словаре односторонний. Словарь предназначен для поиска значений по ключам, а не наоборот. Так что будет немного сложнее сортировать элементы по значениям. Однако это возможно – просто нужно приложить чуть больше усилий:

```
>>> for value in sorted(phoneNumbers.values()):
        for key in phoneNumbers.keys():
            if phoneNumbers[key] == value:
                print(key, phoneNumbers[key])

Боря 444-4321
Ваня 555-1234
Маша 555-6789
Женя 867-5309
```

В данном случае, получив отсортированный список значений, мы нашли ключ каждого из них. Затем мы просмотрели все ключи, пока не нашли тот, который был связан с этим значением.

Также при работе со словарями ты можешь использовать следующие функции:

- функцию **`del`** для удаления элемента:

```
>>> del phoneNumbers["John"]
>>> print(phoneNumbers)
{'Маша': '555-6789', 'Боря': '444-4321', 'Женя': '867-5309'}
```

- функцию **`clear()`** для удаления всех элементов (очистки словаря):

```
>>> phoneNumbers.clear()
>>> print(phoneNumbers)
{}
```

- функцию **`in`**, для того чтобы выяснить, есть ли тот или иной ключ в словаре:

```
>>> phoneNumbers = {'Ваня': '444-4321', 'Маша': '555-6789', 'Женя':
                    '867-5309'}
>>> "Ваня" in phoneNumbers
True
>>> "Катя" in phoneNumbers
False
```

Словари часто используются в Python. Конечно, мы рассказали тебе не все о словарях Python, а лишь познакомили тебя с ними. Однако этого достаточно, чтобы ты мог начать использовать их в своем коде и распознавать их в другом коде.

Что ты узнал?

В этой главе ты узнал:

- что такое списки;
- как добавлять элементы в список;
- как удалять элементы из списка;
- как узнать, содержит ли список определенное значение;
- как сортировать список;
- как создавать копии списков;
- о кортежах;
- о списках списков;
- о словарях Python.

Проверь свои знания

1. Каковы два способа добавления элементов в список?
2. Каковы два способа удаления элементов из списка?
3. Каковы два способа получения отсортированной копии списка без изменения оригинального списка?
4. Как узнать, есть ли в списке определенное значение?
5. Как узнать местоположение определенного значения в списке?
6. Что такое кортеж?
7. Как создать список списков?
8. Как получить отдельное значение из списка списков?
9. Что такое словарь?
10. Как добавить элемент в словарь?
11. Как найти элемент в словаре по его ключу?

Попробуй самостоятельно

1. Напиши программу, которая спрашивает у пользователя пять имен. Программа должна хранить имена в списке и выводить их все в конце. Она должна выглядеть примерно так:

```
Введи пять имен:
Тоня
Паша
Коля
Миша
Витя
Имена: Тоня Паша Коля Миша Витя
```

2. Измени программу из вопроса 1 так, чтобы она выводила оригинальный список имен и отсортированный.
3. Измени программу из вопроса 1 так, чтобы она выводила только третье введенное пользователем имя:

```
Третье введенное имя: Коля
```

4. Измени программу из вопроса 1 так, чтобы пользователь мог заменить одно из имен. В ней должен быть возможен выбор имени для замены, а затем должно выводиться новое имя. И наконец, она должна показывать новый список. Например:

```
Введи пять имен:
Тоня
Паша
```

```
Коля
Миша
Витя
Имена: Тоня Паша Коля Миша Витя
Замени одно имя. Какое? (1-5): 4
Новое имя: Петя
Имена: Тоня Паша Коля Петя Витя
```

5 Напиши программу-словарь, которая позволит пользователям вводить слова и их определения, а затем просматривать их позже. Убедись, что программа оповестит пользователя, если запрашиваемого слова еще нет в словаре. Программа должна выглядеть примерно так:

```
Добавить или найти слово (д/н)? д
Введи слово: компьютер
Введи определение: Машина, которая очень быстро выполняет вычисления
Слово добавлено!
Добавить или найти слово (д/н)? н
Введи слово: компьютер
Машина, которая очень быстро выполняет вычисления
Добавить или найти слово (д/н)? н
Введи слово: qwerty
Данного слова нет в словаре
```

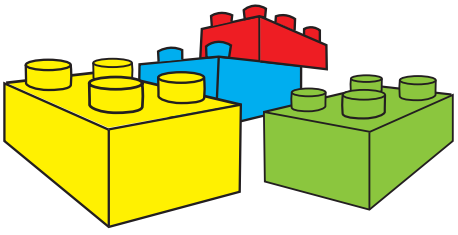
ГЛАВА 13

Функции

Довольно скоро наши программы будут становиться больше и сложнее. Нам нужны способы организации кода в маленькие фрагменты, чтобы их было легче писать и отслеживать.

Есть три основных способа разбиения кода программ на меньшие части. *Функции* похожи на строительные блоки кода, которые можно использовать снова и снова. *Объекты* – автономные части программы. *Модули* – просто отдельные файлы, которые содержат части программ. В этой главе ты узнаешь о функциях, а в следующих двух – об объектах и модулях. И у нас будут все базовые инструменты, необходимые для использования графики, звука и создания игр.

Функции – строительные кирпичики



В самом простом понимании *функция* – это фрагмент кода, который что-то делает. Это небольшой фрагмент, который можно использовать для создания большой программы. Ты можешь совместить фрагмент с остальным кодом, как будто строишь что-то из кубиков «Лего».

Функция создается или *определяется* с помощью ключевого слова **`def`**. Ты можешь *обратиться* к функции с помощью ее имени. Давай начнем с простого примера.

Создание функции

Код в листинге 13.1 определяет функцию и использует ее. Эта функция выводит на экран почтовый адрес.

Листинг 13.1. Создание и использование функции

```
def printMyAddress():
    print(«Петя Иванов»)
    print(«ул. Софьи Перовской, 52, кв. 69»)
    print(«Севастополь, Крым, Россия»)
    print(«299026»)
    print()
printMyAddress()
```

Определение (создание) функции

Обращение (использование) функции

В первой строке мы определяем функцию с помощью ключевого слова **def**. Мы задаем имя функции, которое сопровождается скобками **()**, а затем двоеточием:

```
def printMyAddress():
```

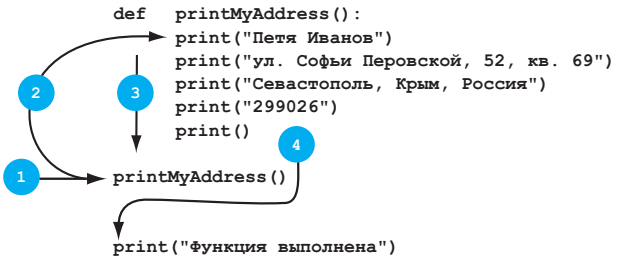
Мы скоро объясним, зачем нужны скобки. Двоеточие сообщает интерпретатору Python о том, что дальше идет блок кода (как и в случае с циклами **for**, **while** и утверждениями **if**).

Затем идет код, который и составляет функцию.

В последней строке листинга 13.1 у нас творится самое главное: мы *обращаемся* к функции с помощью ее имени и скобок. Здесь и начинается выполнение программы. Эта единственная строка заставляет программу выполнить код в функции, которую мы только что определили.

Когда основная программа обращается к функции, то это похоже на то, что функция помогает основной программе выполнить ее работу.

Код внутри блока **def** не является частью основной программы, поэтому при запуске программы она пропускает эту часть и начинает с первой строки, которая не находится в блоке **def**. Рисунок ниже показывает, что происходит, когда ты обращаешься к функции. Мы добавили дополнительную строку в конце программы, которая выводит сообщение после завершения функции.



```
def printMyAddress():
    print(«Петя Иванов»)
    print(«ул. Софьи Перовской, 52, кв. 69»)
    print(«Севастополь, Крым, Россия»)
    print(«299026»)
    print()

printMyAddress()

print(“Функция выполнена”)
```

Ниже приведены шаги с рисунка выше.

1. Начать здесь. Это начало основной программы.
2. Когда мы обращаемся к функции, то перепрыгиваем на первую строку кода в функции.
3. Выполнить каждую строку функции.
4. Когда функция окончена, продолжить основную программу с места остановки.

Обращение к функции

Обращение к функции означает выполнение кода внутри нее. Если мы определим функцию, но никогда к ней не обратимся, то этот код никогда и не будет выполнен.

Мы обращаемся к функции с помощью ее имени и скобок. Иногда в скобках есть символы, а иногда нет.

Попробуй запустить программу из листинга 1.3 – и увидишь, что получится. Должно выйти следующее:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_13-1.py
Петя Иванов
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026
>>>
```

Это тот же самый результат, который ты мог бы получить от простейшей программы:

```
print(«Петя Иванов»)
print(«ул. Софьи Перовской, 52, кв. 69»)
print(«Севастополь, Крым, Россия»)
print(«299026»)
print()
```

Так зачем мы усложняем все созданием функции в листинге 13.1?

Главная причина использования функций – когда ты их определил, то можешь использовать их снова и снова, просто *обратившись* по имени. То есть если бы ты хотел вывести адрес пять раз, мог бы сделать так:

```
printMyAddress()
printMyAddress()
printMyAddress()
printMyAddress()
printMyAddress()
```

И вывод был таков:

```
Петя Иванов
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026

Петя Иванов
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026

Петя Иванов
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026

Петя Иванов
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026

Петя Иванов
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026
```

Ты можешь сказать, что то же самое можно было сделать при помощи цикла, а не функции.

Мы знали, что ты так скажешь... В этом случае ты *мог бы* сделать то же самое с помощью цикла. Но если бы ты хотел вывести адрес пять раз в разных частях программы, а не одновременно, цикл не сработал бы.

Другая причина использовать функцию – ты можешь заставить ее вести себя иначе при каждом выполнении. Сейчас ты узнаешь, как это сделать.

Передача аргументов функции

Настало время узнать, зачем же нужны скобки: *аргументы*!

Нет, Картер, компьютеры очень милы и никогда не спорят. В программировании термин *аргумент* означает фрагмент информации, который ты присваиваешь функции. Мы говорим, что ты *передаешь* аргумент функции.

Представь, что ты хочешь использовать функцию по выводу адреса для любого члена семьи. Адрес был бы тем же самым, но имя нужно изменить. Вместо жестко заданного в функции имени «Петя Иванов» ты можешь сделать его переменной. Переменная *передается* функции, когда ты к ней обращаешься.

Лучше всего это видно на примере. В листинге 13.2 мы изменили функцию по выводу адреса: использовали один аргумент для имени. Аргументы имеют свои имена, как и все остальные переменные. Мы назвали эту переменную **`myName`**.

Когда функция выполняется, переменная **`myName`** заполняется тем аргументом, который мы передали функции при обращении к ней. Мы передаем ей аргумент, помещая его в скобки при вызове функции.

Итак, в листинге 13.2 аргументу **`myName`** присвоено значение «Маша Иванова».

Листинг 13.2. Передача аргумента функции

```
def printMyAddress(myName):
    print(myName)
    print("ул. Софьи Перовской, 52, кв. 69")
    print(«Севастополь, Крым, Россия»)
    print(«299026»)
    print()

printMyAddress(«Маша Иванова»)
```

Передача аргумента **myName** *функции*

Вывод имени

Передача аргумента **«Маша Иванова»** *функции; переменная* **myName** *внутри функции принимает значение* **«Маша Иванова»**

Если мы выполним код из листинга 13.2, мы получим то, что ожидали:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_13-2.py
Маша Иванова
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026
```

Выглядит так же, как и вывод первой программы, когда мы не использовали аргументы. Но теперь мы можем выводить разный адрес каждый раз, например:

```
printMyAddress("Маша Иванова")
printMyAddress("Петя Иванов")
printMyAddress("Толя Иванов")
printMyAddress("Катя Иванова")
```

И теперь вывод будет разным при каждом обращении к функции. Имя меняется, потому что мы *передаем* функции разные имена.

```
>>>
RESTART: C:/HelloWorld/Примеры/many_addresses.py
Маша Иванова
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026

Петя Иванов
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026

Толя Иванов
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026
```

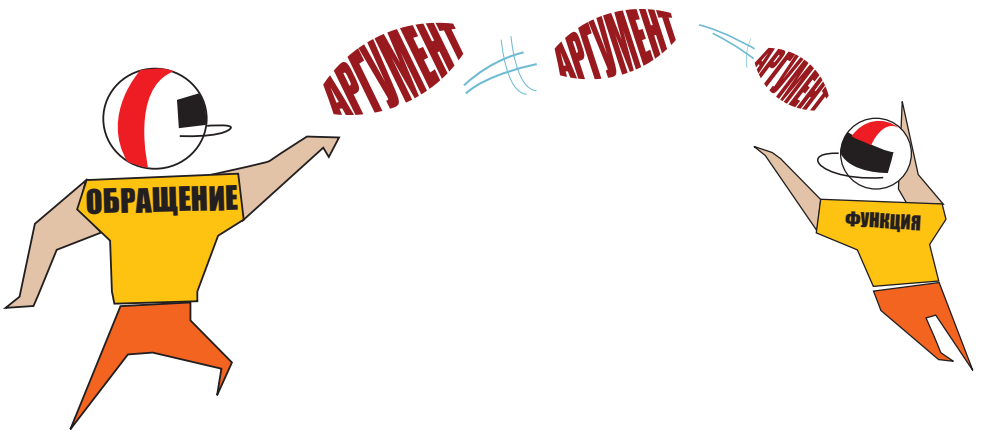
```
Катя Иванова
ул. Софьи Перовской, 52, кв. 69
Севастополь, Крым, Россия
299026
```

Обрати внимание, что какое бы значение не передавалось функции, оно было использовано в функции и выведено в качестве имени в адресе.

Если при выполнении функции меняется больше одного значения, тебе нужно больше одного аргумента. Об этом мы и поговорим далее.

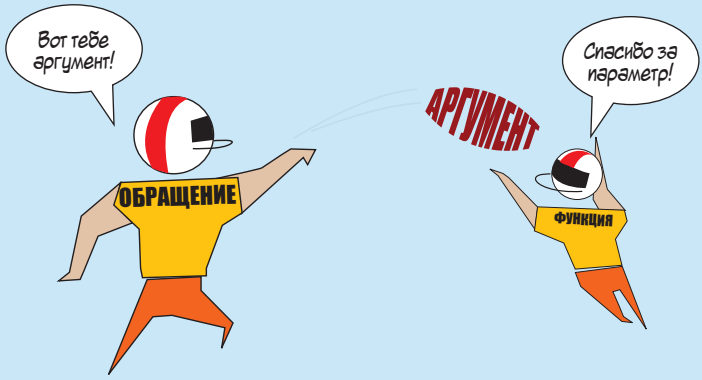
Функции с более чем одним аргументом

В листинге 13.2 наша функция имела только один аргумент. Но функции могут иметь более одного аргумента. Фактически они могут иметь столько аргументов, сколько тебе надо. Давай попробуем пример с двумя аргументами, и я думаю, ты уловишь смысл. Ты сможешь добавлять столько аргументов, сколько тебе нужно в своих программах.



СЛОВАРИК

Есть еще один термин, который можно услышать при разговоре о передаче чего-либо функции: *параметры*. Некоторые люди говорят, что термины *параметр* и *аргумент* взаимозаменяемы. И ты можешь сказать:
«Я передал два параметра этой функции»
или
«Я передал два аргумента этой функции».
Некоторые программисты говорят, что нужно использовать *аргумент* при обозначении передающей части (когда ты обращаешься к функции) и *параметр* при обозначении получающей части (которая находится внутри функции).



Если мы используем *аргумент* или *параметр* при обозначении значений, передаваемых функциям, программисты тебя поймут.

Чтобы отправить письма Картера всем адресатам в доме, нашей функции нужны два аргумента: один для имени получателя и один для номера квартиры. В листинге 13.3 показано, как это сделать.

Листинг 13.3. Функция с двумя аргументами

```
def printMyAddress(someName, apartNum):
    print(someName)
    print(apartNum, « ул. Софьи Перовской, 52»)
    print(«Севастополь, Крым, Россия»)
    print(“299026”)
    print()

printMyAddress(“Петя Иванов”, “69”)
printMyAddress(“Виктор Пчелкин”, “64”)
printMyAddress(“Валерий Филатов”, “22”)
printMyAddress(«Александр Белов», «36»)
```

Использование двух переменных для двух аргументов

Вывод обеих переменных

Обращение к функции и передача ей двух параметров

Когда мы используем несколько аргументов (или параметров), мы разделяем их запятой, как элементы в списке, что и приводит нас к следующей теме...

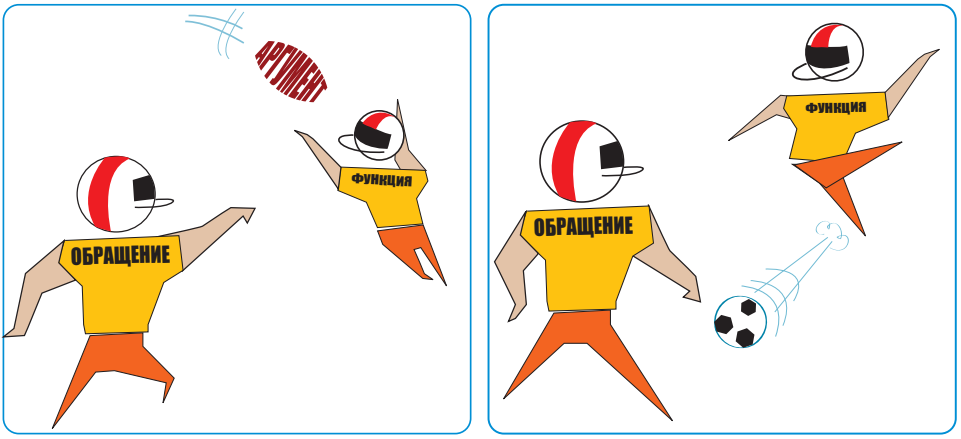
Сколько будет слишком много?

Мы говорили раньше, что ты можешь передать функции столько аргументов, сколько захочешь. Это правда, но если у твоей функции будет хотя бы пять или шесть аргументов, стоит подумать о том, чтобы выполнить задачу другим способом. Первое, что ты можешь сделать, – собрать все аргументы в *список* и затем передать список функции. Так ты передаешь одну переменную (переменную списка), которая всего лишь содержит несколько значений. Это повысит читабельность твоего кода.

Функции, возвращающие значение

До сих пор наши функции выполняли работу за нас. Но еще одним полезным свойством функции является то, что они могут тебе что-то возвращать.

Ты видел, что можно отправлять информацию (аргументы) функциям, но и функции могут отправлять ее обратно обратившимся. Значение, которое возвращается от функции, называется *полученным*, или *возвращенным*, *значением*.



Возвращение значения

Вернуть значение функцией можно с помощью ключевого слова **return** внутри функции. Ниже представлен пример:

```
def calculateTax(price, tax_rate):
    taxTotal = price + (price * tax_rate)
    return taxTotal
```

Этот код отправляет значение **taxTotal** обратно к той части программы, которая вызвала функцию.

Но куда попадает отправленное назад значение? Возвращенные значения отправляются к тому коду, который вызвал функцию. Например:

```
totalPrice = calculateTax(800, 0.2)
```

Функция **calculateTax** вернет значение 960, и оно будет присвоено переменной **totalPrice**.

Ты можешь использовать функцию, чтобы вернуть значения в любое место, где использовано выражение. Ты можешь присвоить возвращенное значение переменной (как мы только что сделали), использовать его в другом выражении или вывести на экран:

```
>>> print(calculateTax(800, 0.2))
60
>>> total = calculateTax(800, 0.2) + calculateTax(700, 0.15)
```

Ты также можешь ничего не делать с возвращенным значением:

```
>>> calculateTax(400, 0.2)
```

В последнем примере функция была выполнена, она посчитала общую сумму с налогом, но мы не использовали полученное значение.

Давай напишем программу с функцией, которая возвращает значение. В листинге 13.4 функция **calculateTax()** будет это делать. Ты задаешь цену без НДС и налоговую ставку, а она возвращает цену с НДС. Мы присвоим это значение переменной. И вместо того, чтобы использовать имя функции, как и раньше, нам нужна переменная, знак равенства и имя функции. Переменная будет присвоена полученному от функции **calculateTax()** значению.

Листинг 13.4. Создание и использование функции, которая возвращает значение

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)
    return total
my_price = float(input("Введи цену: "))
totalPrice = calculateTax(my_price, 0.2)
print("цена = ", my_price, " Цена с НДС = ", totalPrice)
```

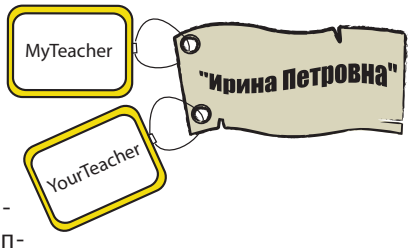
Функция подсчитывает налог и выводит цену с НДС

Отправка результата обратно основной программе

Обращение к функции и сохранение результата в виде **totalPrice**

Попробуй набрать код, сохранить и выполнить программу из листинга 13.4. Обрати внимание, что ставка НДС фиксирована (0.2, то есть 20%-ный налог). Если бы программе необходимо было учитывать разные налоговые ставки, ты мог бы позволить пользователю вводить ставку, как и цену.

Область видимости переменной



Ты мог заметить, что у нас есть переменные вне функции, например **`totalPrice`**, и переменные внутри функции, например **`total`**. Это просто два имени для одного и того же. Это именно то, о чем мы говорили в главе 2: **`YourTeacher = MyTeacher`**.

В нашем примере с **`calculateTax`** переменные **`totalPrice`** и **`total`** – два ярлыка, прикрепленных к одной вещи. В функциях имена внутри них создаются только при выполнении функции. Они даже не существуют до выполнения функции или после ее завершения. Python содержит нечто, называемое *распределением памяти*, которое делает это автоматически. Python создает новые имена для использования внутри функции, когда она выполняется, а затем *удаляет их, когда функция закончена*. Последняя часть очень важна: когда функция завершена, все имена внутри нее перестают существовать.

Пока функция выполняется, имена *вне* ее находятся в неком подобии состояния ожидания – они не используются. Только имена внутри функции используются. Часть программы, где переменная используется (или доступна для использования), называется *областью видимости*.

Локальные переменные

В листинге 13.4 переменные **`price`** и **`total`** были использованы только внутри функции. Мы говорим, что **`price`**, **`total`** и **`tax_rate`** находятся *в области видимости* функции **`calculateTax()`**. Еще можно использовать термин *локальный*. Переменные **`price`**, **`total`** и **`tax_rate`** – локальные переменные в функции **`calculateTax()`**.

Одним из способов увидеть, как это работает, будет добавить строку в программу из листинга 13.4, которая пытается вывести значение **`price`** где-нибудь вне функции. Это реализовано в листинге 13.5.

Листинг 13.5. Попытка вывести локальную переменную

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)
    return total
```

Определение функции для подсчета налога и вывода суммы

```
my_price = float(input("Введи цену: "))

totalPrice = calculateTax(my_price, 0.2)
print("цена = ", my_price, " Цена с НДС = ", totalPrice)
print(price)
```

Обращение к функции, сохранение и вывод результата

Попытка вывести **price**

Если ты выполнишь эту программу, то получишь сообщение об ошибке:

```
Traceback (most recent call last):
  File "C:/HelloWorld/Примеры/Листинг_13-5.py", line 9, in <module>
    print (price)
NameError: name 'price' is not defined
```

Эта строка объясняет ошибку

Последняя строка сообщения об ошибке говорит обо всем: когда мы не находимся в функции **calculateTax()**, переменная **price** *не определена*. Она существует, только когда функция выполняется. Когда мы попытались вывести переменную **price** вне функции (когда она не была в процессе выполнения), мы получили сообщение об ошибке.

Глобальные переменные

В противовес *локальной* переменной **price** переменные **my_price** и **totalPrice** в листинге 13.5 определены *вне* функции, в основной части программы. Мы используем термин *глобальный* для переменных с *широкой* областью видимости. В этом случае *широкий* означает основную часть программы, а не то, что находится внутри функции. Если мы расширим программу из листинга 13.5, то сможем использовать переменные **my_price** и **totalPrice** в другом месте программы, и они все равно будут иметь свои значения. Они все равно будут *в области видимости*. Поскольку мы можем использовать их в любой части программы, то называем их *глобальными переменными*.

В листинге 13.5, когда мы были вне функции и попытались вывести переменную, которая находилась внутри функции, мы получили сообщение об ошибке. Переменная не существовала, она была *вне области видимости*. Как ты думаешь, что случится, если сделать обратное: попробовать вывести глобальную переменную внутри функции?

В листинге 13.6 мы пытаемся вывести переменную **my_price** из функции **calculateTax()**. Попробуй – и увидишь, что произойдет.

Листинг 13.6. Использование глобальной переменной внутри функции

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)
    print(my_price)
    return total

my_price = float(input("Введи цену: "))

totalPrice = calculateTax(my_price, 0.2)
print("цена = ", my_price, " Цена с НДС = ", totalPrice)
```

Попытка вывести **my_price**

Сработало? Да! Но почему?

Когда мы начали говорить об области видимости переменной, я сказал, что Python использует управление памятью, чтобы автоматически создавать локальные переменные, когда функция выполняется. Управление памятью делает и другие вещи тоже. В функции, если ты используешь имя переменной, которая была определена в основной программе, Python позволит тебе использовать глобальную переменную, пока ты не попробуешь ее изменить.

То есть ты можешь сделать так:

```
print(my_price)
```

или так:

```
your_price = my_price
```

Потому что при этом переменная **my_price** не изменяется.

Если любая часть функции попробует изменить переменную, Python создаст новую локальную переменную вместо нее. То есть если ты сделаешь так:

```
my_price = my_price + 10
```

то переменная **my_price** будет новой локальной переменной, созданной Python во время выполнения функции.

В примере в листинге 13.6 выведенное значение было *глобальной* переменной **my_price**, потому что функция ее не изменила. Программа из листинга 13.7 показывает тебе, что если ты попробуешь изменить глобальную переменную внутри функции, то получишь новую локальную переменную вместо нее. Попробуй – и увидишь.

Листинг 13.7. Попытка изменить глобальную переменную внутри функции

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)
    my_price = 10000
    print("my_price (внутри функции) = ", my_price)
    return total

my_price = float(input("Введи цену: "))

totalPrice = calculateTax(my_price, 0.2)
print("цена = ", my_price, " Цена с НДС = ", totalPrice)
print("my_price (вне функции) = ", my_price)
```

Изменение переменной **my_price** *внутри функции*

Вывод локальной версии переменной **my_price**

Вывод глобальной версии переменной **my_price**

Переменная **my_price** *здесь – это иная ячейка памяти, чем та же переменная здесь*

Если ты выполнишь программу из листинга 13.7, она будет выглядеть так:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_13-7.py
Введи цену: 800
my_price (внутри функции) =10000 ← Вывод переменной my_price внутри функции
цена = 800 Цена с НДС = 960
my_price (вне функции) = 800 ← Вывод переменной my_price вне функции
```

Как ты видишь, теперь у нас две разные переменные с именем **my_price** с двумя разными значениями. Одна – локальная переменная внутри функции **calculateTax()**, значение которой 10 000. Другая – глобальная переменная, которую мы определили в основной части программы для получения ввода от пользователя, которым было число 800.

Изменение глобальной переменной

В последнем разделе ты видел, что если попробовать изменить значение *глобальной переменной* внутри функции, Python создает новую *локальную переменную* вместо нее. Это имеет своей целью предотвратить случайное изменение глобальных переменных в функциях.

Тем не менее иногда тебе *нужно* изменить глобальную переменную изнутри функции. Как это сделать?

В Python есть ключевое слово, **global**, которое позволяет это сделать. Используется оно так:

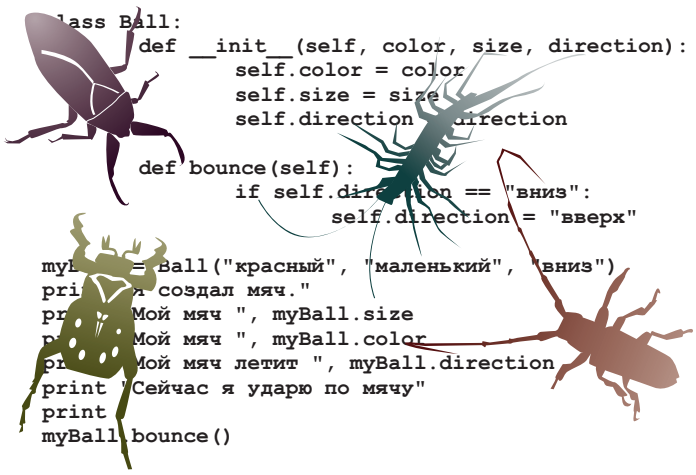
```
def calculateTax(price, tax_rate):
    global my_price ← Говорит Python о том, что ты хочешь использовать глобальную версию переменной my_price
```

Если ты используешь ключевое слово **global**, Python *не* создаст новую локальную переменную с именем **my_price**. Он использует глобальную переменную **my_price**. Если глобальной переменной **my_price** не существует, он ее создаст.

Небольшой совет касательно присвоения имен переменным

Выше ты видел, что можно использовать одни и те же имена для глобальных переменных и локальных. Python автоматически создает новые локальные переменные, когда ему это нужно, или ты можешь предотвратить это с помощью ключевого слова **global**. Тем не менее я крайне рекомендую не использовать имена переменных повторно.

Как ты мог заметить в некоторых примерах, может быть сложно распознать, является переменная глобальной или локальной. Это запутывает код, потому что у тебя разные переменные с одним именем. А где путаница, там и баги.



```
lass Ball:
    def __init__(self, color, size, direction):
        self.color = color
        self.size = size
        self.direction = direction

    def bounce(self):
        if self.direction == "вниз":
            self.direction = "вверх"

myBall = Ball("красный", "маленький", "вниз")
print "Я создал мяч."
print "Мой мяч ", myBall.size
print "Мой мяч ", myBall.color
print "Мой мяч летит ", myBall.direction
print "Сейчас я ударю по мячу"
print
myBall.bounce()
```

Теперь мы рекомендуем использовать разные имена для локальных и глобальных переменных. Таким образом, не будет путаницы и багов.

Что ты узнал?

В этой главе ты узнал:

- что такое функция;
- что такое аргументы (или параметры);
- как передать аргумент функции;
- как передать несколько аргументов функции;
- как заставить функцию вернуть значение;
- что такое область видимости переменной, и что такое локальные и глобальные переменные;
- как использовать глобальные переменные в функции.

Проверь свои знания

1. Какое ключевое слово используется для создания функции?
2. Как обратиться к функции?
3. Как передать информацию (аргументы) функции?
4. Какое максимальное количество аргументов может иметь функция?
5. Как получить информацию от функции обратно?
6. Что происходит с локальными переменными в функции после ее завершения?

Попробуй самостоятельно

1 Напиши функцию для вывода твоего имени прописными буквами, например:

```
PPPPP    OOOO    M       M        A       H    H
P    P  O    O  M  M   M  M      A A      H    H
P    P  O    O  M   M M   M     A   A     H    H
PPPPP   O    O  M    M    M    AAAAAAA    HHHHHH
P       O    O  M         M   A       A   H    H
P        OOOO   M         M  A         A  H    H
```

Напиши программу, которая обращается к функции несколько раз.

2 Создай функцию, которая позволит тебе выводить любое имя, адрес, улицу, город, область, почтовый индекс и страну в мире. (Подсказка: тебе нужно семь аргументов. Ты можешь передать их по одному или в виде списка.)

3 Попробуй использовать пример из листинга 13.7, но сделай **`my_price`** глобальной переменной, чтобы увидеть разницу в итоговом выводе.

4 Напиши функцию для расчета общего количества сдачи – рубли, двухрублевые монеты, пятирублевые монеты и т. д. (как в вопросах из главы 5). Функция должна возвращать общий номинал монет. Затем напиши программу, которая обращается к функции. Вывод должен выглядеть примерно так:

```
10-рублевых монет: 3
5-рублевых монет: 6
2-рублевых монет: 7
1-рублевых монет: 2
Всего у тебя: 76
```

ГЛАВА 14

Объекты

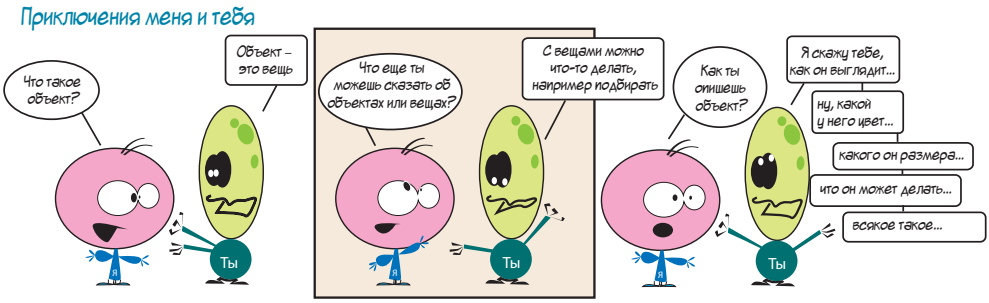
В последних нескольких главах мы рассматривали разные способы организации данных и программ, их сборки. Мы уже знаем, что списки – это способ собирать переменные (данные) вместе, а функции – это способ собирать код в единицу, которую можно использовать снова и снова.

Объекты развивают собирание предметов еще больше. Объекты – это способ *собрать функции и данные вместе*. Это очень полезная вещь в программировании, и используется она во многих, многих программах. Фактически если ты взглянешь на код движка Python, почти все в нем будет объектами. В терминах программирования: Python *объектно-ориентирован*. Это значит, что можно использовать объекты в Python (и довольно легко). Нет *необходимости* создавать свои собственные объекты, но это многое упрощает.

В этой главе ты узнаешь, что же такое объекты, как их создавать и использовать. В дальнейших главах, когда мы займемся графикой, нам предстоит использовать очень много объектов.

Объекты в реальном мире

Что такое объект? Если бы мы говорили не о программировании и задали бы тебе этот вопрос, у нас был бы подобный разговор:



Хорошее начало для определения объектов в Python. Возьмем, к примеру, мяч. Ты можешь выполнять действия с ним: подобрать его, бросить, толкнуть, надуть (некоторые из мячей). Это мы называем *действиями*. А также можешь описать мяч, сообщив его цвет, размер и вес. Это *атрибуты* мяча.

СЛОВАРИК

Ты можешь описать объект, описав его характеристики, или *атрибуты*. Одним из атрибутов мяча является его форма. Большинство мячей круглые. Другие примеры атрибутов: цвет, размер, вес и цена. Синоним атрибутов – *свойства*.

Реальные объекты в реальном мире позволяют:

- с ними что-то *сделать* (действия);
- *описать* их (атрибуты или свойства).

В программировании то же самое.

Объекты в Python

В Python характеристики («то, что ты знаешь» об объекте) также называются *атрибутами*, это легко запомнить. В Python действия («то, что можно сделать» с объектом) называются *методами*.

Если сделать в Python версию или *модель* мяча, мяч будет объектом, и у него будут *атрибуты* и *методы*.

Атрибуты мяча выглядят так:

```
ball.color
ball.size
ball.weight
```

Этими характеристиками можно *описать* мяч.

Методы мяча выглядят так:

```
ball.kick()
ball.throw()
ball.inflate()
```

Это те действия, которые можно *выполнить* с мячом.

Что такое атрибуты?

Атрибуты – это все, что ты знаешь (или можешь узнать) о мяче. Атрибуты мяча – это фрагменты информации: числа, строки и т. д. Звучит знакомо? Да, они переменные. Они просто переменные, включенные в объект.

Ты можешь их вывести:

```
print(ball.size)
```

Присвоить им значения:

```
ball.color = 'зеленый'
```

Присвоить их обычным, не объектным переменным:

```
myColor = ball.color
```

Ты можешь присвоить их атрибутам других объектов:

```
myBall.color = yourBall.color
```

Что такое методы?

Методы – это действия, которые можно *выполнять* с объектом. Это фрагменты кода, к которым можно *обратиться* (*вызвать*), чтобы что-то сделать. Звучит знакомо? Да, *методы* – это *функции*, которые включены в объект.

Ты можешь выполнять с методами те же действия, что и с любой другой функцией, включая *передачу аргументов* и *возвращение значений*.

Объект = атрибуты + методы

Итак, объекты – это способ сборки вместе *атрибутов* и *методов* (то, что ты знаешь, и то, что ты можешь сделать) чего-либо. Атрибуты – это информация, а методы – это действия.

В наших примерах с мячом ты мог заметить точку между именем объекта и именем атрибута или метода. Это обозначение Python для использования атрибутов или методов объекта: **`object.attribute`** или **`object.method()`**. Просто, не правда ли? Это называется *точечной нотацией* и используется во многих языках программирования.

Теперь ты имеешь представление об объектах. Давай начнем их создавать!

Создание объектов

Существует два шага в создании объектов в Python.

Первый шаг: определить, как объект будет выглядеть и вести себя, – его методы и атрибуты. Но создание этого описания не создает сам объект. Это как проект дома. Проект точно говорит, как будет выглядеть дом, но проект – это не дом. Ты не можешь жить в проекте. Ты можешь построить настоящий дом с его помощью. Ты можешь построить много домов по одному проекту.

В Python описание, или план, объекта называется *классом*.

Второй шаг: использовать класс, чтобы создать сам объект. Объект представляет собой экземпляр этого класса.

Давай взглянем на пример создания *класса* и его *экземпляра*. Листинг 14.1 показывает определение класса для простого класса **`Ball`**.

Листинг 14.1. Создание простого класса **`Ball`**

```
class Ball:                          ← Сообщает интерпретатору Python,
                                       где создавать класс
    def bounce(self):                  Это метод
        if self.direction == "вниз":
            self.direction = «вверх»
```

В листинге 14.1 у нас есть определение класса для мяча с одним методом: **`bounce()`**. А как насчет атрибутов? Ну, атрибуты на самом деле не принадлежат классу, они принадлежат каждому экземпляру. Потому что каждый экземпляр может иметь разные атрибуты.

Есть пара способов настройки атрибутов экземпляра. Мы рассмотрим их далее.

Создание экземпляра объекта

Как мы упомянули ранее, определение класса – это не объект. Это только проект. Теперь давай построим этот дом.

Если мы хотим создать *экземпляр* мяча, мы делаем следующее:

```
myBall = Ball()
```

У нашего мяча еще нет атрибутов, поэтому давай присвоим их ему:

```
myBall.direction = “вниз”
myBall.color = “зеленый”
myBall.size = «маленький»
```

Это один из способов определить атрибуты для объекта. Мы увидим второй способ далее.

Теперь давай попробуем какой-нибудь метод.

Ниже показано, как использовать метод **`bounce()`**:

```
myBall.bounce()
```

Давай сложим это все вместе в программе с несколькими утверждениями **`print`**, чтобы увидеть, что будет происходить. Программа представлена в листинге 14.2.

Листинг 14.2. Использование класса `Ball`

```
class Ball:

    def bounce(self):                          # Наш класс, такой же, как и раньше
        if self.direction == “вниз”:
            self.direction = “вверх”

myBall = Ball()                                # Создает экземпляр нашего класса
myBall.direction = “вниз”                      # Настройка атрибутов
myBall.color = “красный”
myBall.size = «маленький»

print(«Я создал мяч.»)                         # Вывод атрибутов объекта
print(“Мой мяч “, myBall.size)
print(“Мой мяч “, myBall.color)
print(“Мой мяч летит “, myBall.direction)
print(«Сейчас я ударю по мячу»)
print()
myBall.bounce()                                # Использование метода
print(“Теперь мой мяч летит “, myBall.direction)
```

Если мы запустим программу из листинга 14.2, то увидим следующее:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_14-2.py
Я создал мяч.                                  # Настроенные нами атрибуты
Мой мяч маленький.
Мой мяч красный.
Мой мяч летит вниз
Сейчас я ударю по мячу                         # Применяем функцию bounce() к мячу

Теперь мой мяч летит вверх                     # Он меняет свое направление
```

Обрати внимание, что после обращения к методу **bounce()** направление мяча изменилось с **вверх** на **вниз**, а это именно то, что должен был сделать код в методе **bounce()**.

Инициализация объекта

Когда мы создали наш мяч, в нем не были заполнены атрибуты **size**, **color** и **direction**. Мы их заполняли *после* того, как создали объект. Но есть способ установки свойств объекта во время его создания. Он называется *инициализацией* объекта.

СЛОВАРИК

Инициализация означает «подготовка чего-либо к запуску». Когда мы *инициализируем* что-то в программном обеспечении, мы делаем это готовым к использованию, приводя его в нужное нам состояние.

Когда ты создаешь определение класса, то можешь определить особый метод, называемый **__init__()**, который будет выполняться каждый раз при создании нового экземпляра класса. Ты можешь передать аргументы методу **__init__()**, чтобы создать экземпляр со свойствами, настроенными так, как тебе нужно.

В листинге 14.3 показано, как это работает.

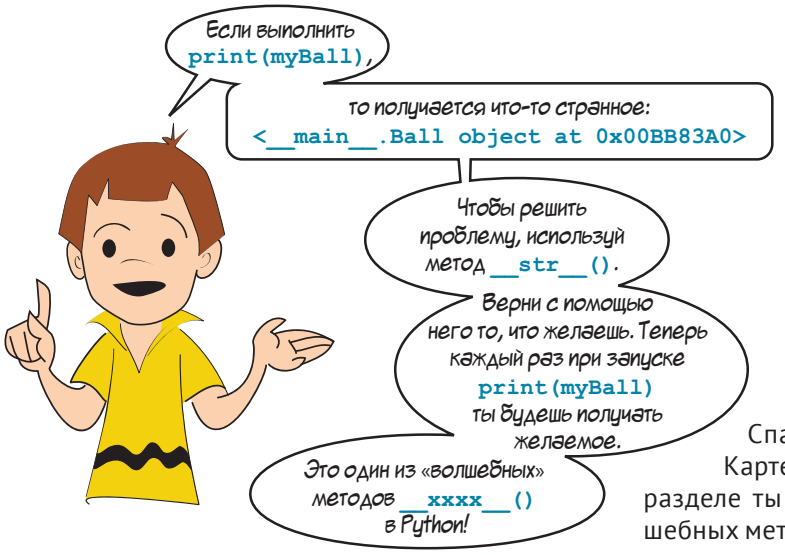
Листинг 14.3. Добавление метода **__init__()**

```
class Ball:
    def __init__(self, color, size, direction):
        self.color = color
        self.size = size
        self.direction = direction

    def bounce(self):
        if self.direction == “вниз”:
            self.direction = “вверх “

myBall = Ball(«красный», «маленький», «вниз»)
print(«Я создал мяч.»)
print(«Мой мяч «, myBall.size)
print(“Мой мяч “, myBall.color)
print(“Мой мяч летит “, myBall.direction)
print(«Сейчас я ударю по мячу»)
print()
myBall.bounce()
print(“Теперь мой мяч летит “, myBall.direction)
```

*Метод **__init__**. С каждой стороны от **init** находится по два подчеркивания, всего четыре подчеркивания с двух сторон*

*Атрибуты, переданные в качестве аргументов методу **__init__***

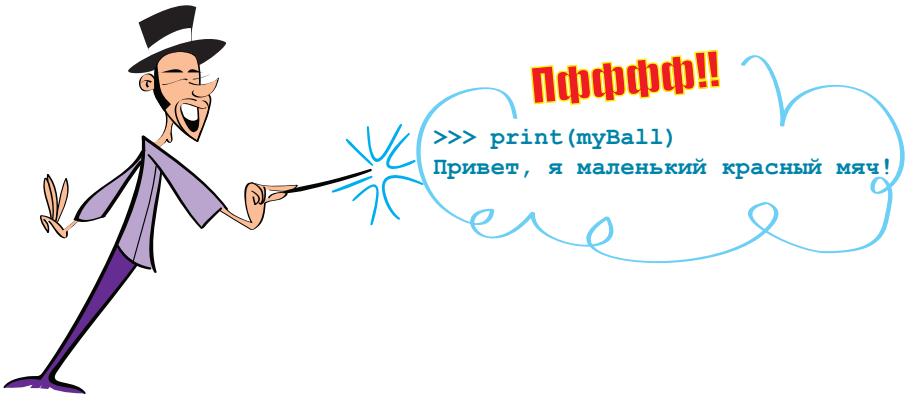
Если ты выполнишь программу из листинга 14.3, то должен получить тот же вывод, что и в предыдущем листинге. Разница лишь в использовании метода **__init()__** для настройки атрибутов.



Спасибо за совет, Картер. В следующем разделе ты узнаешь о волшебных методах подробнее.

«Волшебный» метод: `__str__()`

У объектов в Python есть «волшебные» методы, как их называет Картер. Они, конечно, не волшебные на самом деле! Это просто некоторые методы, которые Python подключает автоматически, когда ты создаешь любой класс. Программисты на Python обычно называют их *особыми методами*.



Ты уже видел метод `__init__()`, который инициализирует объект при его создании. В каждом объекте встроен метод `__init__()`. Если ты не включишь его в определение класса, ему на смену придет встроенный метод и создаст объект.

Другим особым методом является `__str__()`, который указывает Python, что выводить на экран, когда ты задаешь объекту команду `print`. По умолчанию Python сообщает:

- где определен экземпляр (в случае Картера это **__main__**, то есть основная часть программы);
- имя класса (**Ball**);
- место в памяти, где хранится экземпляр (часть с **0x00BB83A0**).

Но если ты хочешь, чтобы команда **print** выводила что-то другое для твоего объекта, то можешь сам назначить метод **__str__()**, который перекроет встроенный. В листинге 14.4 приведен пример.

Листинг 14.4. Использование метода **__str__()** для изменения вывода объекта

```
class Ball:
    def __init__(self, color, size, direction):
        self.color = color
        self.size = size
        self.direction = direction
    def __str__(self):
        msg = "Привет, я " + self.size + " " + self.color + " мяч!"
        return msg
myBall = Ball(«красный», «маленький», «вниз»)
print(myBall)
```

Метод **__str__()**

Если выполнить программу из листинга 14.4, мы получим следующее:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_14-4.py
Привет, я маленький красный мяч!
```

Это выглядит гораздо более дружелюбно, чем **<__main__.Ball object at 0x00BB83A0>**, не правда ли?

Все «волшебные методы» заключены в четыре подчеркивания, по два с каждой стороны.

Что такое `self`?

Ты мог заметить, что термин **self** появляется в нескольких местах в атрибутах класса и определениях метода, например:

```
def bounce(self):
```

Что значит это **self**? Ну, помнишь, мы говорили, что с помощью плана можно построить не один дом? Ты также можешь использовать класс для создания более чем одного экземпляра объекта, например так:

```
cartersBall = Ball(«красный», «маленький», «вниз»)
warrensBall = Ball(«зеленый», «средний», «вверх»)
```

Создание двух экземпляров класса **Ball**

Когда мы обращаемся к методу одного из этих экземпляров:

```
warrensBall.bounce()
```

метод должен знать, какой именно экземпляр к нему обратился. Нужно толкнуть **`cartersBall`** или **`warrensBall`**? Аргумент **`self`** сообщает методу, какой объект к нему обратился. Он называется *указателем экземпляра*.

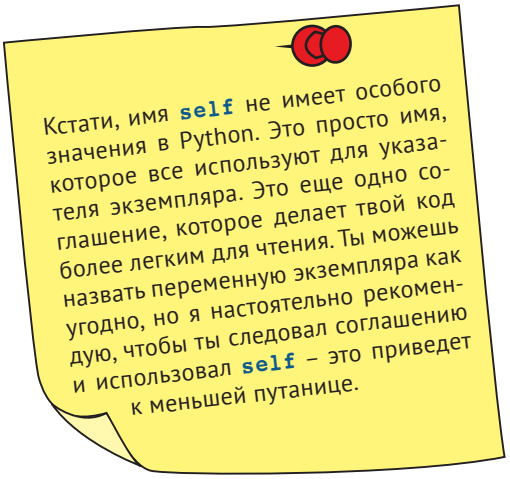
Но постой! Когда мы обратились к методу, не было никакого аргумента в скобках **`warrensBall.bounce()`**, но аргумент **`self`** присутствует в методе. Откуда взялся аргумент **`self`**, если мы ничего не передавали? Это еще один трюк, который Python проводит с объектами. Когда ты обращаешься к методу класса, информация о том, к какому экземпляру ты обратился, – *указатель экземпляра* – автоматически передается методу.

Это все равно, что написать так:

```
Ball.bounce(warrensBall)
```

В этом случае мы сообщили методу **`bounce()`**, какой мяч толкать. Фактически этот код тоже должен работать потому, что это именно то, что Python делает за кадром, когда ты пишешь **`warrensBall.bounce()`**.

В главе 11 мы писали программу о хот-догах. Теперь в качестве примера использования объектов мы создадим класс для хот-дога.



Пример класса – хот-дог

Для этого примера мы примем как данность, что у хот-дога всегда есть булочка. (Иначе будет слишком путано.) Мы присвоим нашему хот-догу некоторые атрибуты и методы.

Атрибуты:

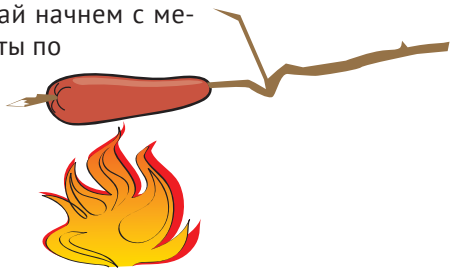
- **`cooked_level`**: число, которое указывает нам, как долго готовилась сосиска. Мы будем использовать числа в диапазоне 0–3 для сырой сосиски, более 3 для полуготовой, более 5 для прожаренной и более 8 для сгоревшей. Сначала наши хот-доги будут сырыми;
- **`cooked_string`**: строка, описывающая, насколько прожарен наш хот-дог;
- **`condiments`**: список того, что есть в хот-доге, например кетчуп, горчица и т. д.

Методы:

- **`cook()`**: готовит хот-дог на протяжении какого-то времени. Делает хот-дог более прожаренным;

- **`add_condiment()`**: добавляет приправы в хот-дог;
- **`__init__()`**: создает экземпляр и устанавливает свойства по умолчанию;
- **`__str__()`**: делает вывод **`print`** красивее.

Сначала нам надо определить класс. Давай начнем с метода **`__init__()`**, который назначит атрибуты по умолчанию для хот-дога:



```
class HotDog:
    def __init__(self):
        self.cooked_level = 0
        self.cooked_string = "Сырой"
        self.condiments = []
```

Начнем с сырого хот-дога без приправ.

Теперь создадим метод для жарки нашего хот-дога:

```
def cook(self, time):
    self.cooked_level = self.cooked_level + time
    if self.cooked_level > 8:
        self.cooked_string = "Сгоревший"
    elif self.cooked_level > 5:
        self.cooked_string = "Прожаренный"
    elif self.cooked_level > 3:
        self.cooked_string = "Полуготовый"
    else:
        self.cooked_string = "Сырой"
```

Увеличение уровня готовности с течением времени

Настройка строк для разных уровней готовности

Перед тем как продолжить, давай проверим эту часть. Сначала нам нужно создать экземпляр хот-дога и проверить атрибуты.

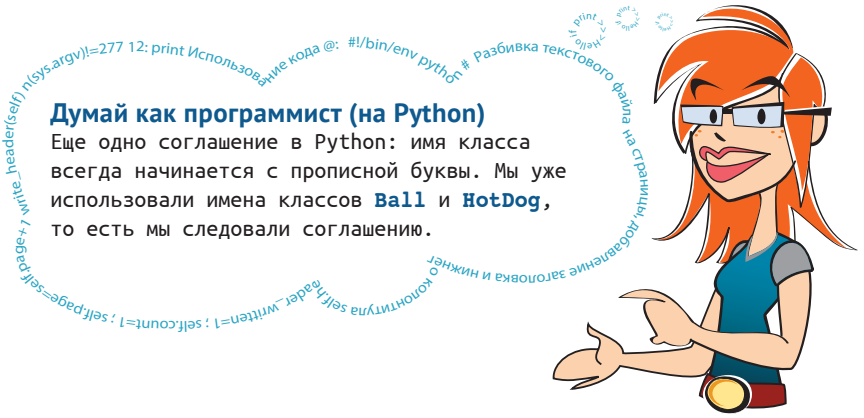
```
myDog = HotDog()
print(myDog.cooked_level)
print(myDog.cooked_string)
print(myDog.condiments)
```

Давай соединим весь код вместе и выполним его. Листинг 14.5 содержит целую программу (на данный момент).

Листинг 14.5. Начало нашей программы обжарки хот-дога

```
class HotDog:
    def __init__(self):
        self.cooked_level = 0
        self.cooked_string = "Сырой"
        self.condiments = []

    def cook(self, time):
        self.cooked_level = self.cooked_level + time
```

```
        if self.cooked_level > 8:
            self.cooked_string = "Сгоревший"
        elif self.cooked_level > 5:
            self.cooked_string = "Прожаренный"
        elif self.cooked_level > 3:
            self.cooked_string = "Полуготовый"
        else:
            self.cooked_string = "Сырой"
myDog = HotDog()
print(myDog.cooked_level)
print(myDog.cooked_string)
print(myDog.condiments)
```



Теперь выполни код из листинга 14.5 и посмотри, что получилось. Должно выглядеть так:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_14-5.py
0   <-------- Cooked_level
Сырой <-------- Cooked_string
[]  <-------- Condiments
```

Мы видим, что атрибуты таковы: **`cooked_level = 0`**, **`cooked_string = "Сырой"`** и **`condiments`** пуст.

Теперь проверим метод **`cook()`**. Добавь эти строки к программе из листинга 14.5:

```
print(«Сейчас я приготовлю хот-дог»)
myDog.cook(4) <-------- Готовка хот-дога в течение 4 минут
print(myDog.cooked_level)
print(myDog.cooked_string) | Проверка новых атрибутов cooked
```

Запусти программу снова. Сейчас вывод должен выглядеть так:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_14-5_modified.py
0
Сырой
[]
Сейчас я приготовлю хот-дог
4
Полуготовый
```

До обжарки

После обжарки

Итак, наш метод **`cook()`** работает. Значение **`cooked_level`** изменилось с **`0`** до **`4`**, и строка тоже изменилась (с **`Сырой`** на **`Полуготовый`**).

Давай добавим приправы. Для этого нам понадобится новый метод. Мы могли бы добавить функцию **`__str__()`**, чтобы объект легче выводился. Измени код программы, чтобы он выглядел, как показано в листинге 14.6.

Листинг 14.6. Класс `HotDog` с методами `cook()`, `add_condiments` и `__str__()`

```
class HotDog:
    def __init__(self):
        self.cooked_level = 0
        self.cooked_string = "Сырой"
        self.condiments = []
    def __str__(self):
        msg = "хот-дог"
        if len(self.condiments) > 0:
            msg = msg + " с "
        for i in self.condiments:
            msg = msg+i+", "
        msg = msg.strip(", ")
        msg = self.cooked_string + " " + msg + "."
        return msg
    def cook(self, time):
        self.cooked_level=self.cooked_level+time
        if self.cooked_level > 8:
            self.cooked_string = "Сгоревший"
        elif self.cooked_level > 5:
            self.cooked_string = "Прожаренный"
        elif self.cooked_level > 3:
            self.cooked_string = "Полуготовый"
        else:
            self.cooked_string = "Сырой"
    def addCondiment(self, condiment):
        self.condiments.append(condiment)

myDog = HotDog()
print(myDog)
print("Готовлю хот-дог 4 минуты...")
myDog.cook(4)
print(myDog)
```

Определение нового метода **`__str__()`**

Определение класса

Определение нового метода **`add_condiments()`**

Создание экземпляра

Проверка всех элементов

```
print("Готовлю хот-дог еще 3 минуты...")
myDog.cook(3)
print(myDog)
print(«А что, если я пожарю его еще 10 минут?»)
myDog.cook(10)
print(myDog)
print(«Сейчас я добавлю приправы»)
myDog.addCondiment(«кетчупом»)
myDog.addCondiment(«горчицей»)
print(myDog)
```

Проверка всех элементов

Этот код длинноват, но я все равно предлагаю тебе набрать его самостоятельно. У тебя уже есть часть его из листинга 14.5. Но если ты устал или у тебя нет времени, то можешь найти его в папке с примерами для данной книги.

Запусти программу и посмотри, что получилось. Должно получиться так:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_14-6.py
Сырой хот-дог.
Готовлю хот-дог 4 минуты...
Полуготовый хот-дог.
Готовлю хот-дог еще 3 минуты...
Прожаренный хот-дог
А что, если я пожарю его еще 10 минут?
Сгоревший хот-дог
Сейчас я добавлю приправы
Сгоревший хот-дог с кетчупом, горчицей.
```

Первая часть программы создает класс. Вторая часть проверяет методы для готовки нашего виртуального хот-дога и добавления приправ. Но, судя по последним строкам, мы его сожгли. Какая трата кетчупа и горчицы!

Сокрытие данных

Ты мог понять, что есть два способа просмотра и изменения данных (атрибутов) внутри объекта. Мы можем получить к ним прямой доступ, например:

```
myDog.cooked_level = 5
```

или можем использовать метод, который изменяет атрибут, например:

```
myDog.cook(5)
```

Если мы начали с сырого хот-дога (`cooked_level = 0`), оба этих способа выполнят одно и то же. Они установят значение `cooked_level` равным `5`. Но зачем тогда усложнять все и создавать для этого метод? Почему бы не сделать все прямо?

Можно назвать две причины:

- если бы мы получали прямой доступ к атрибутам, то приготовление хот-дога потребовало бы по крайней мере двух этапов: изменения `cooked_level` и изменения `cooked_string`. С помощью метода мы обращаемся только к нему, и он выполняет все действия;
- если бы мы получали прямой доступ к атрибутам, мы могли сделать нечто подобное:

```
cooked_level = cooked_level – 2
```

Это сделало бы хот-дог *менее* прожаренным, чем он был раньше. Но ты не можешь *отготовить* хот-дог обратно! Поэтому это не имеет смысла. С помощью метода мы можем быть уверены, что значение `cooked_level` только увеличивается и никогда не уменьшается.

СЛОВАРИК

В терминах программирования ограничение доступа к данным объекта так, что ты можешь изменить их только с помощью методов, называется *сокрытием данных*. Python не позволяет принудительно скрывать данные, но ты можешь написать код так, чтобы следовать этому правилу.

Итак, ты увидел, что у объектов есть атрибуты и методы. А еще узнал, как создавать объекты и как их инициализировать с помощью особого метода `__init__()`. Ты также узнал о еще одном особом методе – `__str__()`, который делает вывод наших объектов более красивым.

Полиморфизм и наследование

Далее мы взглянем на два аспекта объектов, которые, вероятно, являются самыми важными: *полиморфизм* и *наследование*. Это два длинных умных слова, но они позволяют сделать объекты очень полезными. Мы детально объясним, что все это значит.

Полиморфизм: тот же метод, другое поведение

Очень просто, *полиморфизм* означает, что у тебя может быть два (и более) метода с одним и тем же именем для разных классов. Эти методы могут вести себя по-разному, в зависимости от того класса, к которому они применяются.

Например, допустим, что ты делаешь программу по геометрии и тебе нужно посчитать площадь разных фигур, например треугольников и квадратов. Ты можешь создать два класса:

```
class Triangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def getArea(self):
        area = self.width * self.height / 2.0
        return area
class Square:
    def __init__(self, size):
        self.size = size
    def getArea(self):
        area = self.size * self.size
        return area
```

Класс **`Triangle`**

У обоих есть метод **`getArea()`**

Класс **`Square`**

И класс **`Triangle`**, и класс **`Square`** содержат метод **`getArea()`**. Поэтому если бы у нас был экземпляр для каждого класса, например такой:

```
>>> myTriangle = Triangle(4, 5)
>>> mySquare = Square(7)
```

то мы могли бы посчитать площадь каждого с помощью **`getArea()`**:

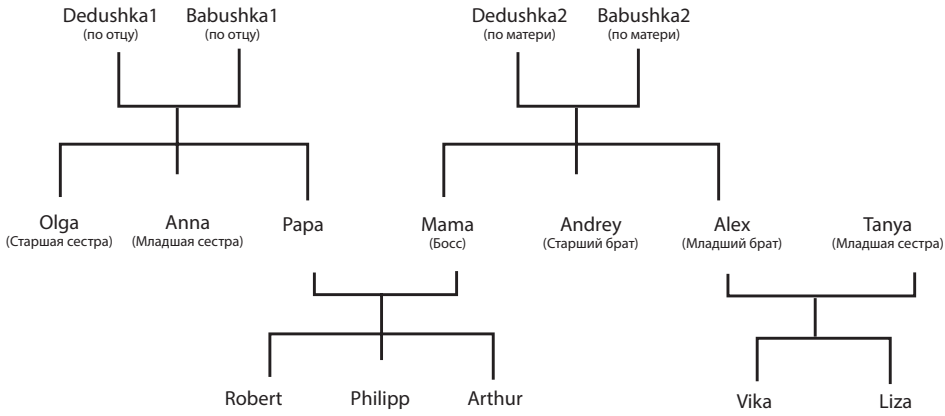
```
>>> myTriangle.getArea()
10.0
>>> mySquare.getArea()
49
```

Мы использовали метод **`getArea()`** для обеих фигур, но метод выполнил разные действия для каждой. Это и есть пример полиморфизма.

Наследование: учимся у родителей

В реальном (непрограммистском) мире люди могут *наследовать* характеристики и вещи своих родителей или родственников. Ты можешь унаследовать рыжие волосы или имущество, например деньги или недвижимость.

В объектно-ориентированном программировании классы могут наследовать атрибуты и методы других классов. Это позволяет тебе создавать целые «семьи» классов с общими атрибутами и методами. Таким образом, тебе не нужно начинать с нуля каждый раз, когда ты хочешь добавить нового родственника в семью.



Класс, который может наследовать атрибуты или методы другого класса, называется *производным классом*, или *подклассом*. Мы объясним на примере.

Представь, что мы делаем игру, где игрок может подбирать разные вещи по пути, например еду, монетки, одежду. Мы можем сделать класс **`GameObject`**. Он будет иметь атрибут **`name`** (например, «монета», «яблоко», «шляпа») и метод **`pickUp()`** (который будет добавлять монету к коллекции предметов игрока). Все игровые объекты будут иметь эти общие методы и атрибуты.

Затем мы создаем *подкласс* монет. Класс **`Coin`** будет *производным* от **`GameObject`**. Он *наследует* атрибуты и методы **`GameObject`**, поэтому класс **`Coin`** будет автоматически иметь атрибут **`name`** и метод **`pickUp()`**. Класс **`Coin`** также будет нуждаться в атрибуте **`value`** (номинал монеты) и методе **`spend()`** (чтобы использовать монету для покупки).

Давай посмотрим, как будет выглядеть код.

```
class GameObject:
    def __init__(self, name):
        self.name = name

    def pickUp(self, player):
        # здесь код для добавления объекта
        # в коллекцию игрока

class Coin(GameObject):
    def __init__(self, value):
        GameObject.__init__(self, "монетка")
        self.value = value
    def spend(self, buyer, seller):
        # здесь код для удаления монеты
        # со счета игрока и
        # ее добавления на счет продавца
```

Определение класса **`GameObject`**

`Coin` – *подкласс* **`GameObject`**

В методе **`__init__`** *наследуется атрибут* **`GameObject`** *и добавляется в него значение*

Новый метод **`spend()`** *для класса* **`Coin`**

Планирование

В последнем примере мы не писали рабочий код в методах, только комментарии, объясняющие, что эти методы будут делать. Это способ планирования заранее того, что ты добавишь позже. Сам код будет зависеть от того, как работает игра. Программисты часто так поступают, чтобы организовать мысли при написании сложного кода. «Пустые» функции или методы называются *заглушками кода*.

Если ты запустишь программу из предыдущего примера, то получишь сообщение об ошибке, потому что определение функции не может быть пустым.

Все верно, Картер, но комментарии не считаются, потому что они только для тебя, а не для компьютера.

Ключевое слово **pass** используется в Python, когда ты хочешь создать заглушку кода. То есть код должен на самом деле выглядеть так:

```
class GameObject:
    def __init__(self, name):
        self.name = name

    def pickUp(self, player):
        pass
        # здесь код для добавления объекта
        # в коллекцию игрока

class Coin(GameObject):
    def __init__(self, value):
        GameObject.__init__(self, "монетка")
        self.value = value
    def spend(self, buyer, seller):
        pass
        # здесь код для удаления монеты
        # со счета игрока и
        # ее добавления на счет продавца
```

Добавление ключевого слова **pass** *в эти два момента*

Мы не будем приводить более подробные примеры использования объектов, полиморфизма и наследования в этой главе. Ты увидишь множество примеров

далее в книге. Ты получишь гораздо более четкое представление о том, как использовать объекты, когда увидишь их в рабочих программах.

Что ты узнал?

В этой главе ты узнал:

- об объектах;
- об атрибутах и методах;
- о классах;
- о создании экземпляров классов;
- об особых методах `__init__()` и `__str__()`;
- о полиморфизме;
- о наследовании;
- о заглушках кода.

Проверь свои знания

1. Какие ключевые слова используются для определения нового типа объекта?
2. Что такое атрибуты?
3. Что такое методы?
4. В чем разница между классом и экземпляром?
5. Какое имя обычно используется для *указателя экземпляра* в методе?
6. Что такое полиморфизм?
7. Что такое наследование?

Попробуй самостоятельно

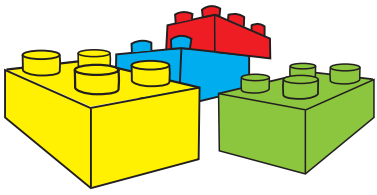
1. Создай определение класса для `BankAccount`. Он должен иметь атрибуты для своего имени (строка), номера счета (строка или целое число), баланса (десятичная дробь). Он должен иметь методы для выведения баланса, создания депозита и снятия средств.
2. Создай класс `InterestAccount`, который приносит проценты. Он должен быть подклассом `BankAccount` (то есть он наследует атрибуты и методы). Он также должен иметь атрибуты для процентной ставки и метод для начисления процентов. Чтобы не усложнять, допустим, что метод `addInterest()` будет использоваться каждый год для подсчета процентов и обновления баланса.

ГЛАВА 15

Модули

Это последняя глава о способах сбора данных и кода вместе. Мы уже знаем о *списках, функциях* и *объектах*. В этой главе ты узнаешь о *модулях*. В следующей – используем модуль Pygame, чтобы приступить к созданию графики.

Что такое модуль?



Модуль – часть или фрагмент чего-либо. Мы говорим, что нечто является модульным, если оно состоит из частей или его можно легко разделить на части. Блоки конструктора «Лего» могут быть идеальным примером чего-то модульного. Ты можешь взять несколько разных элементов конструктора и построить с их помощью много разных моделей.

В Python модули – это меньшие фрагменты кода больших программ. Каждый модуль или фрагмент – это отдельный файл на твоем жестком диске. Ты можешь взять большую программу и разбить ее на не один модуль или файл. Или пойти другим путем: начать с одного маленького модуля и добавлять фрагменты для создания большой программы.

Зачем использовать модули

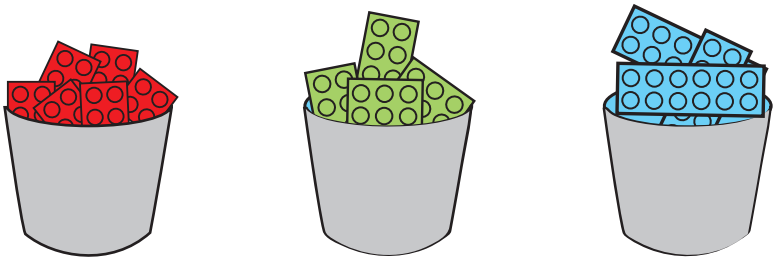
Итак, зачем все усложнять и делить код программы на маленькие фрагменты, если они все нам нужны для того, чтобы программа заработала? Почему бы не оставить все в одном большом файле?

Есть несколько причин:

- уменьшаются размеры файлов, то есть найти что-то в коде проще;
- создав модуль, его можно использовать во многих программах. Это дает тебе возможность не начинать все сначала, если тебе понадобятся те же функции;
- ты можешь не использовать все модули вместе постоянно. Модульность значит, что ты можешь использовать разные комбинации частей для разных задач, как ты можешь собирать разные конструкции из кубиков «Лего».

Ведра блоков

В главе о функциях (глава 13) мы говорили, что они похожи на строительные блоки. Ты можешь представлять себе модули как ведра с блоками. Ты можешь положить так мало или так много блоков в ведро, как тебе надо, и у тебя может быть много разных ведер. Может, у тебя одно ведро для квадратных блоков, одно – для плоских элементов и одно – для блоков неправильной формы. Так обычно программисты используют модули: собирают похожие типы функций вместе в модуле. Или они могут собрать все функции, необходимые для проекта, в модуль, как ты собираешь все блоки, нужные для строительства замка, в ведерко.



Как мы создаем модули?

Давай создадим модуль. Модуль – это файл Python, пример которого приведен в листинге 15.1. Набери код в редакторе IDLE и сохрани файл под именем *my_module.py*.

Листинг 15.1. Создание модуля

```
# это файл my_module.py
# мы используем его в другой программе
def c_to_f(celsius):
    fahrenheit = celsius * 9.0 / 5 + 32
    return fahrenheit
```

И все! Ты только что создал модуль! В твоем модуле одна функция, **`c_to_f()`**, которая конвертирует температуру из градусов Цельсия в градусы Фаренгейта.

Теперь мы задействуем код из файла *my_module.py* в другой программе.

Как мы используем модули?

Чтобы использовать что-то, что находится в модуле, мы должны сначала сообщить Python, какие модули мы собираемся использовать. Ключевое слово Python, которое позволяет тебе включать другие модули в программу, – **import**. Оно используется так:

```
import my_module
```

Давай напишем программу, которая использует наш только что написанный модуль. Мы используем функцию **c_to_f** для конвертации температур.

Ты уже видел, как использовать функцию и передавать ей параметры (или аргументы). Единственная разница состоит в том, что функция будет находиться в файле, отличном от файла нашей программы, поэтому нам нужно использовать ключевое слово **import**. Программа в листинге 15.2 использует написанный нами модуль *my_module.py*.

Листинг 15.2. Использование модуля

```
import my_module  <-------- My_module содержит функцию c_to_f

celsius = float(input(«Введи температуру в градусах Цельсия: «))
fahrenheit = c_to_f(celsius)
print(«Получится», fahrenheit, « градусов Фаренгейта»)
```

Создай новый файл в редакторе IDLE и набери код программы. Сохрани файл под именем *modular.py*, выполни и посмотри, что получится. Тебе нужно сохранить его в ту же папку (или каталог), что и файл *my_module.py*.

Сработало? Ты увидишь что-то подобное:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_15-2.py
Введи температуру в градусах Цельсия: 34
Traceback (most recent call last):
  File “C:/HelloWorld/Примеры/Листинг_15-2.py”, line 4, in <module>
    fahrenheit = c_to_f(celsius)
NameError: name ‘c_to_f’ is not defined
```

Не сработало! Что случилось? Сообщение об ошибке говорит, что функция **c_to_f()** не определена. Но мы знаем, что она определена в **my_module**, и мы *импортировали* этот модуль.

Ответ в том, что нам нужно быть точнее, когда мы говорим Python о функциях, которые определены в других модулях. Первый способ исправить проблему – изменить строку

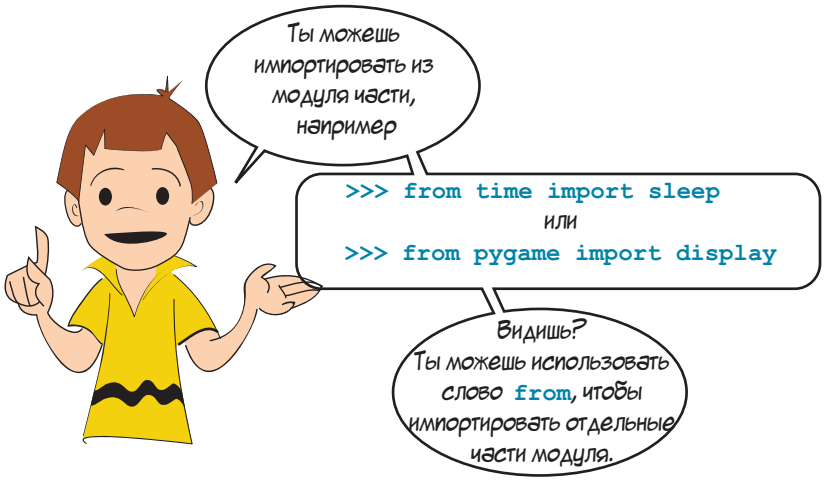
```
fahrenheit = c_to_f(celsius)
```

на

```
fahrenheit = my_module.c_to_f(celsius)
```

Теперь мы точно указали Python, что функция **`c_to_f`** находится в модуле **`my_module`**. Попробуй запустить программу с этими изменениями – и увидишь, что она работает.

Пространство имен

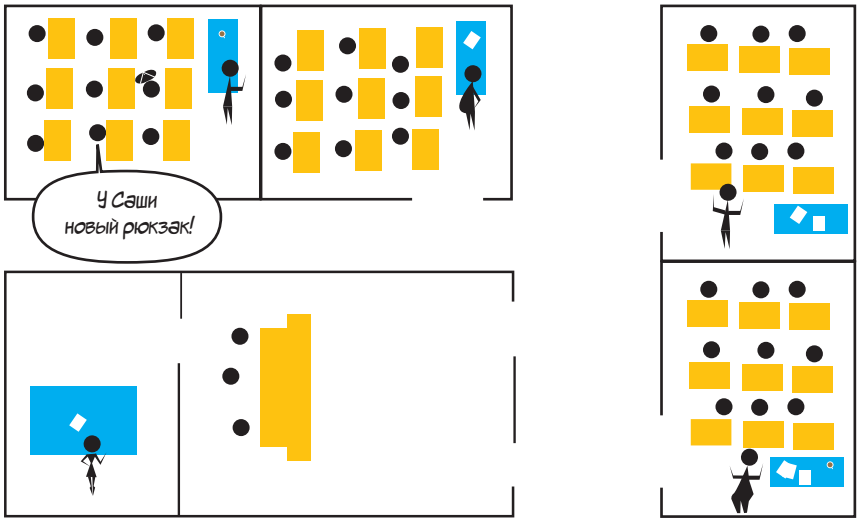


То, что упомянул Картер, относится к так называемому *пространству имен*. Это сложная тема, но тебе нужно ее понимать.

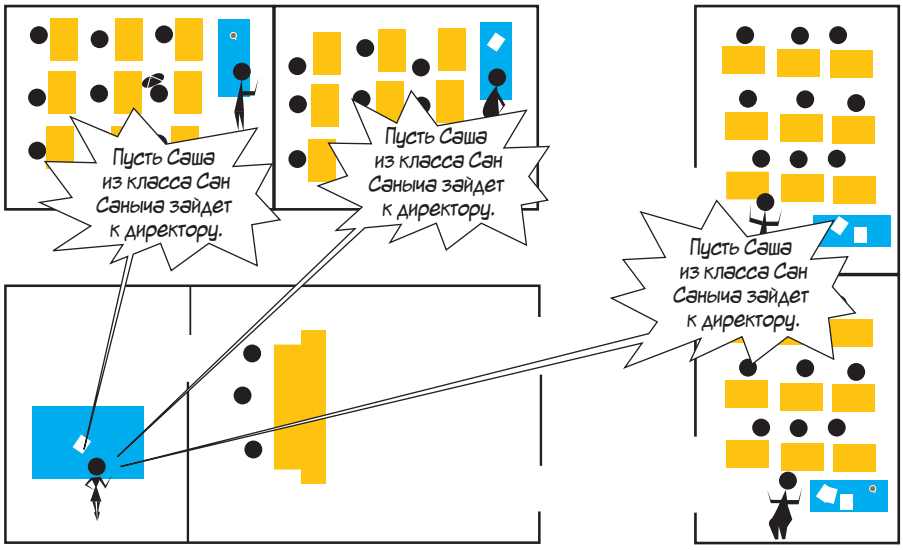
Что такое пространство имен?

Представь, что ты учишься в школе в классе Сан Саныча, и в твоем классе есть мальчик по имени Саша. А теперь представь, что в другом классе твоей школы, классе Ольги Петровны, есть другой мальчик с таким же именем Саша. Если ты в своем собственном классе скажешь: «У Саши новый рюкзак», – все в твоем классе поймут (по крайней мере, допустят), что ты говоришь о Саше из твоего класса. Если ты имел в виду другого Сашу, ты скажешь: «У Саши из класса Ольги Петровны», или «У другого Саши», или что-то подобное.

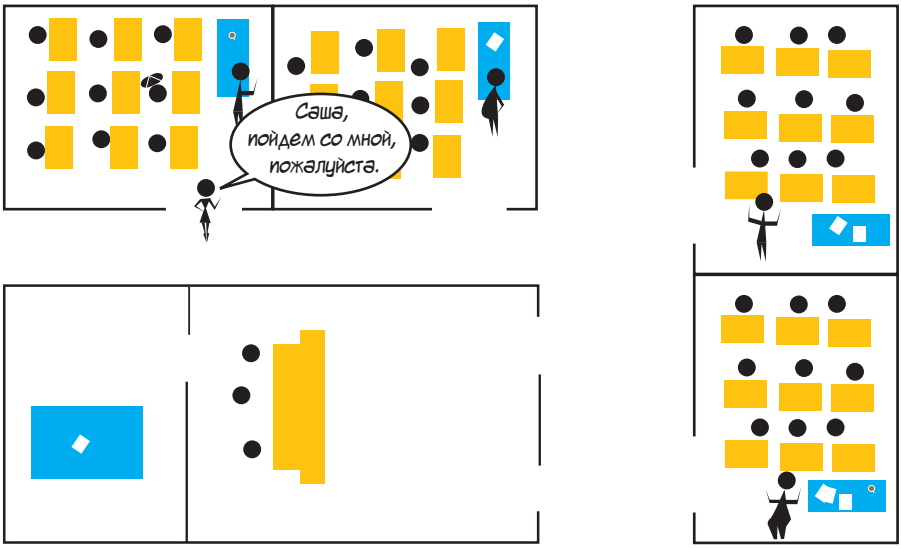
В твоем классе только один Саша, поэтому когда ты говоришь «Саша», твои одноклассники знают, кого ты имеешь в виду. Иначе говоря, в пространстве твоего класса есть только одно имя Саша. Твой класс – твое *пространство имен*, и в этом пространстве есть только один Саша, поэтому нет путаницы.



А если директору нужно вызвать Сашу к себе в кабинет, он не может просто сказать: «Пусть Саша зайдет ко мне». Если он так поступит, у него в кабинете окажутся все Саши. Для директора пространством имен является вся школа. Это значит, что все в школе слышат названное имя, а не только один класс. Поэтому он должен быть более точен в своем обозначении: «Пусть Саша из класса Сан Саныча зайдет ко мне».



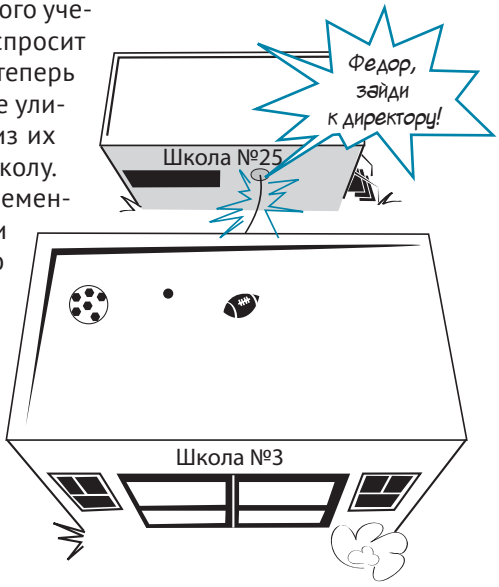
Другой способ для директора найти нужного Сашу – прийти в класс и сказать: «Саша, пойдем со мной». Будет только один Саша в классе, и директор уведет нужного ученика. В этом случае пространством имен снова будет один класс, а не вся школа.



В общих чертах: программисты называют небольшие пространства имен (наподобие твоего класса) *локальными*, а большие (как вся школа) – *глобальными*.

Импорт пространств имен

Допустим, в твоей школе № 25 нет ни одного ученика по имени Федор. Если директор спросит Федора в школе, он никого не найдет. А теперь представим, что в другой школе на той же улице, школе № 3, ведется ремонт, и один из их классов временно переезжает в твою школу. В этом классе есть ученик Федор. Но временный класс не связан с твоей школой. Если директор снова начнет искать Федора, то не найдет его. Но если он свяжет временный класс с системой оповещений всей остальной школы и позовет Федора, то к нему придет Федор из школы № 3.



Подключение временного класса другой школы похоже на импорт модуля в Python. Когда ты импортируешь модуль, то получаешь доступ ко всем именам в нем: всем переменным, всем функциям, всем объектам.

Импортирование модуля означает то же, что и импортирование пространства имен. Когда ты импортируешь модуль, ты импортируешь пространство имен.

Есть два способа это сделать. Можно поступить так:

```
import school_3
```

Если ты так сделаешь, **school_3** все еще будет отдельным пространством имен. У тебя будет доступ к нему, но тебе нужно будет уточнять, какое именно пространство имен тебе нужно, когда ты к нему обращаешься. Поэтому директору придется сделать что-то подобное:

```
call_to_office(school_3.Fedor)
```

Ему все равно придется указать пространство имен (**school_3**) и имя (**Fedor**), если он хочет позвать Федора. Именно это мы сделали ранее в нашей программе по конвертации температур.

Чтобы она заработала, мы написали:

```
fahrenheit = my_module.c_to_f(celsius)
```

Мы указали пространство имен (**my_module**) и имя функции (**c_to_f**).

Импортирование с помощью функции `from`

Другой способ импортировать пространство имен таков:

```
from school_3 import Fedor
```

Если директор поступит так, имя **Fedor** из **school_3** будет включено в его пространство имен, и он сможет позвать Федора так:

```
call_to_office(Fedor)
```

Поскольку **Fedor** теперь в его пространстве имен, ему не нужно идти в пространство **school_3**, чтобы позвать **Fedor**.

В этом примере директор импортировал только одно имя, **Fedor**, из **school_3** в свое локальное пространство имен. Если он хотел бы импортировать всех, он мог бы написать:

```
from school_3 import *
```

Здесь символ * означает *все*. Но ему нужно быть осторожным. Если есть ученики с одинаковыми именами в школе № 25 и школе № 3, будет путаница.

Ох! Сейчас ты можешь еще не совсем понимать суть пространств имен. Не переживай! Все станет ясно на примерах из следующих глав. Когда нам надо будет импортировать модули, мы будем объяснять наши шаги.

Стандартные модули

Теперь, когда мы знаем, как создавать и использовать модули, нужно ли нам всегда писать свои? Нет! В этом одна из основных прелестей Python.

Python содержит огромное количество стандартных модулей, которые позволяют выполнять такие действия, как поиск файлов, сообщать время (или отсчитывать время), генерировать случайные числа и др. Иногда говорят, что Python имеет «батарейки в комплекте», и тогда имеют в виду именно все стандартные модули Python. Также они известны как *стандартная библиотека Python (Python Standard Library)*.

Почему все это должно находиться в отдельных модулях? Ну, не должно, но люди, создавшие Python, решили, что так будет эффективнее. Иначе каждая программа на Python содержала бы все возможные функции. А так ты включаешь только те, которые тебе нужны.

Конечно, для некоторых базовых команд (например, **`print`**, **`for`**, **`if-else`**) в Python не нужен отдельный модуль.

Если в Python нет модуля для реализации твоей идеи (например, графической игры), существуют другие модули, которые можно скачать (обычно бесплатно). Мы включили несколько из них в инсталлятор, прилагаемый к этой книге. Если ты его не установил, то всегда можешь инсталлировать модули отдельно.

Рассмотрим парочку стандартных модулей.

Время

Модуль **`time`** позволяет получать значения системных даты и времени. Также дает возможность добавлять задержки в твои программы. (Иногда компьютер делает все так быстро, что тебе нужно замедлить его.)

Функция **`sleep()`** модуля **`time`** используется, чтобы добавить задержку, то есть заставить программу подождать и ничего не предпринимать какое-то время. Это похоже на «сон» программы, поэтому функция и названа **`sleep()`** (от англ. *sleep* – спать). Ты сам назначаешь, сколько секунд она должна «спать».

Программа в листинге 15.3 демонстрирует, как работает функция **`sleep()`**. Попробуй ее набрать, сохранить и выполнить.

Листинг 15.3. Сонная программа

```
import time
print("Как ", end="")
time.sleep(2)
print("у ", end="")
time.sleep(2)
print("тебя ", end="")
time.sleep(2)
print(«дела?»)
```

Обрати внимание, что когда мы обращались к функции **`sleep()`**, нам пришлось указать значение **`time`** перед ней. Это случилось потому, что хотя мы и импортировали модуль **`time`**, мы не сделали его имя частью пространства имен основной программы. То есть каждый раз, когда мы хотим использовать функцию **`sleep()`**, мы должны обращаться к **`time.sleep()`**.

Если бы мы попробовали следующее:

```
import time
sleep(5)
```

это не сработало бы, потому что **`sleep()`** не входит в наше пространство имен. Мы бы получили сообщение об ошибке:

```
NameError: name 'sleep' is not defined
```

Но если бы мы импортировали ее так:

```
from time import sleep
```

то сказали бы Python: «Найди переменную (или функцию, или объект) по имени **`sleep`** в модуле **`time`** и включи ее в мое пространство имен». Теперь мы можем использовать функцию **`sleep()`** без упоминания **`time`** перед ней:

```
from time import sleep
print('Привет, продолжим через 5 секунд...')
sleep(5)
print('И снова здравствуйте')
```

Если необходимо импортировать имена в локальное пространство (чтобы не приходилось каждый раз уточнять модуль), но ты не знаешь, какие имена в модуле нужны, можно использовать символ `*` импорта всех без исключения имен:

```
from time import *
```

Символ `*` означает *все*, то есть она импортирует все доступные переменные из модуля. С ней нужно быть осторожным. Если мы создадим в своей программе имя, такое же как и в модуле **`time`**, будет конфликт. Импорт с указанием символа `*` – не самый лучший способ. Лучше импортировать только необходимые части.

Помнишь программу отсчета времени, которую мы писали в главе 8 (листинг 8.6)? Теперь ты знаешь, что там делает **`time.sleep(1)`**.

Случайные числа

Модуль **`random`** используется для генерации случайных чисел, что весьма полезно в играх.

Давай попробуем использовать его в интерактивном режиме:

```
>>> import random
>>> print(random.randint(0, 100))
4
>>> print(random.randint(0, 100))
72
```

Каждый раз при использовании функции **`random.randint()`** ты получаешь новое, случайное число. Поскольку мы передали аргументы от 0 до 100, целое число будет из этого диапазона. Мы использовали функцию **`random.randint()`** в программе по угадыванию чисел в главе 1, чтобы загадывать секретное число.

Если тебе нужна случайная десятичная дробь, используй функцию **`random.random()`**. Тебе не нужно ничего писать в скобках, потому что функция **`random.random()`** всегда выдает число в диапазоне между 0 и 1.

```
>>> print(random.random())
0.270985467261
>>> print(random.random())
0.569236541309
```

Если тебе нужно случайное число в диапазоне, например, от 0 до 10, ты можешь просто умножить результат на 10.

```
>>> print(random.random() * 10)
3.61204895736
>>> print(random.random() * 10)
8.10985427783
```

Что ты узнал?

В этой главе ты узнал:

- что такое модуль;
- как создать модуль;
- как использовать модуль в другой программе;
- что такое пространство имен;
- что значит *локальные* и *глобальные* пространства имен и переменные;
- как перенести имена из других модулей в твое пространство имен.

А также ты увидел несколько примеров стандартных модулей Python.

Проверь свои знания

1. Каковы преимущества использования модулей?
2. Как создать модуль?
3. Какое ключевое слово используется, когда ты хочешь использовать модуль?
4. Импорт модуля – то же самое, что и импорт _____.
5. Каковы два способа импортировать модуль **`time`** так, чтобы у тебя был доступ ко всем именам (то есть всем переменным, функциям и объектам) этого модуля?

Попробуй самостоятельно

1. Напиши код модуля, в котором есть функция «напиши свое имя большими буквами» из раздела «Попробуй самостоятельно» главы 13. Затем напиши программу, которая импортирует модуль и обращается к этой функции.
2. Измени код в листинге 15.2 так, чтобы перенести функцию **`c_to_f()`** в пространство имен основной программы. То есть чтобы ты мог написать

   ```
   fahrenheit = c_to_f(celsius)
   ```

   вместо

   ```
   fahrenheit = my_module.c_to_f(celsius)
   ```

3. Напиши код небольшой программы для генерации списка из пяти случайных целых чисел от 1 до 20 и выведи их на экран.
4. Напиши код небольшой программы, которая выводит случайную десятичную каждые 3 секунды в течение 30 секунд.

ГЛАВА 16

Графика

Ты узнал многие базовые приемы программирования: ввод и вывод, переменные, решения, циклы, списки, функции, объекты и модули. Надеемся, тебе было интересно! Теперь пора повеселиться.

В этой главе ты узнаешь, как рисовать на экране линии, фигуры, цвета и даже анимацию. Это поможет нам создать игры и другие программы из следующих глав.

Небольшая помощь – Pygame

Получить графику (и звук) на компьютере с помощью кода – задача непростая. Она включает в себя ресурсы операционной системы, видеокарты и много низкоуровневого кода, о котором мы даже не хотим задумываться. Поэтому мы будем использовать модуль под названием Pygame, чтобы упростить весь процесс.

Модуль Pygame позволяет создавать графику и другие вещи, необходимые для работы игр на разных компьютерах и операционных системах без необходимости изучать все мелкие детали каждой системы. Pygame бесплатен, и его версия есть в файлах этой книги. Он должен быть уже установлен, если ты запускал инстал-

лятор, прилагаемый к книге. В противном случае тебе нужно его установить. Ты можешь скачать модуль на сайте **www.pygame.org**.

Окно Pygame

Первое, что нам нужно сделать, – создать окно, в котором мы будем рисовать свои графические элементы. В листинге 16.1 показана очень простая программа, которая создает окно Pygame.

Листинг 16.1. Создание окна Pygame

```
import pygame
pygame.init()
screen = pygame.display.set_mode([640, 480])
```

Запусти программу. Что ты видишь? Это зависит от того, какая у тебя операционная система: ты можешь увидеть окошко (залитое черным цветом), быстро мелькнувшее на экране, или окошко, которое не получается закрыть. Что это такое?

Модуль Pygame предназначен для создания игр. Игры выполняют действия не сами по себе, они взаимодействуют с игроком. И в модуле Pygame есть так называемый *цикл с ожиданием события*, который постоянно проверяет, не сделал ли пользователь что-то, например не нажал ли клавишу или не передвинул мышь. Программам Pygame часто приходится выполнять цикл с ожиданием события постоянно. В нашей первой программе мы не начинали цикл событий, поэтому программа завершилась очень быстро после запуска.

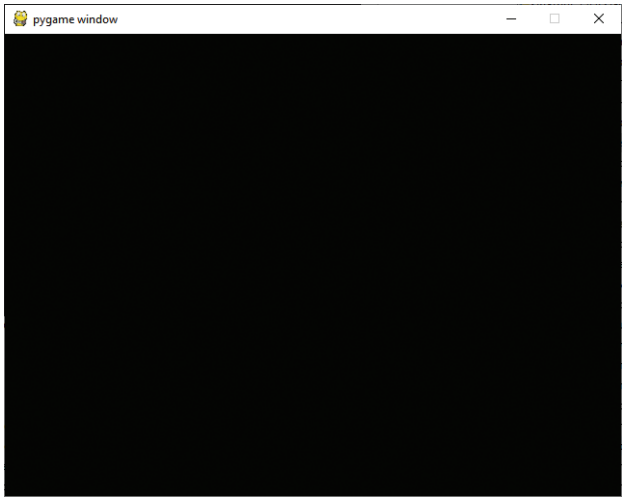
Один из способов инициировать постоянное выполнение цикла событий Pygame – использовать цикл **`while`**. Мы хотим, чтобы цикл продолжал работать до тех пор, пока пользователь запускает нашу игру. Поскольку в программах Pygame обычно нет меню, пользователь закроет программу с помощью крестика в правом верхнем углу окна (в Windows) или кнопки закрытия в левом верхнем углу (для macOS). В системе Linux значок закрытия окна варьируется в зависимости от используемого оконного менеджера и графического интерфейса – но если ты используешь Linux, то, скорее всего, знаешь, как закрыть окно!

Код в листинге 16.2 открывает окно Pygame; оно останется открытым до тех пор, пока пользователь не закроет его:

Листинг 16.2. Удержание окна Pygame открытым

```
import pygame
pygame.init()
screen = pygame.display.set_mode([640, 480])
running  = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Запусти программу из листинга 16.2. Появится окно Pygame, которое остается открытым до тех пор, пока ты сам не закроешь его.



Ты спросишь: «Как именно работает код в этом цикле **`while`**?» Ответим: «Он использует цикл **`event`** в Pygame». Более подробный ответ ты найдешь в главе 18, где мы рассказываем обо всех событиях в Pygame.

Рисование в окне

Теперь у нас есть окно Pygame, которое остается открытым, пока мы его не закроем. Цифры **`[640, 480]`** в третьей строке листинга 16.2 – это размер окна: 640 пикселей в ширину и 480 пикселей в высоту. Давай начнем рисовать графику. Измени свою программу, как показано в листинге 16.3.

Листинг 16.3. Рисуем круг

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255,255,255])
pygame.draw.circle(screen, [255,0,0],[100,100], 30, 0)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Заполняет окно белым цветом

Добавь эти три строки

Обновляет монитор

Рисует круг

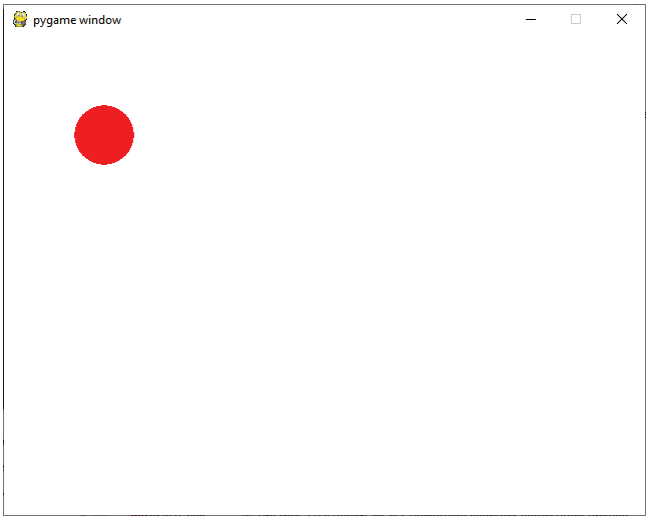
Что за прыжок?

Объект экрана в Pygame (свой мы назвали **`screen`** в строке 3 листинга 16.3) содержит две копии содержимого окна Pygame. Причина этого в том, что когда мы начинаем создавать анимацию, то хотим сделать ее как можно более ровной и быстрой. Поэтому вместо обновления экрана при малейшем изменении в изображении мы можем внести ряд изменений и затем «перепрыгнуть» (англ. *flip*) на новую версию изображения. Это вносит все изменения мгновенно, а не по очереди. Таким образом мы не получаем на экране недорисованные круги.

Представь эти две копии как текущий вид экрана и следующую его версию. Текущий – это то, что мы видим сейчас. «Следующий» экран – то, что мы увидим после «прыжка». Мы вносим все изменения в «следующий» экран и перепрыгиваем на него, чтобы их увидеть.

Как сделать круг

Когда ты запустишь программу из листинга 16.3, то увидишь красный круг возле верхнего левого угла окна.



Функция **`pygame.draw.circle()`** рисует круг. Тебе нужно передать ей пять параметров:

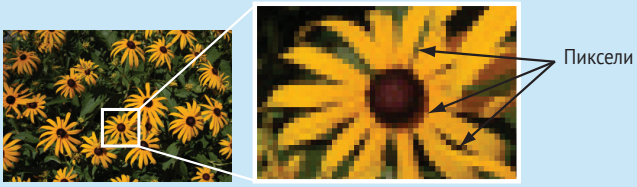
- на какой *поверхности* рисовать круг (в нашем случае он на поверхности, определенной в строке 3, **`screen`**, то есть на поверхности монитора);

- каким *цветом* его рисовать (в нашем случае красным, что представлено в выражении `[255,0,0]`);
- *где* его рисовать (у нас это `[100,100]`, то есть 100 пикселей ниже и 100 пикселей правее верхнего левого угла);
- какого *размера* должен быть круг (у нас это число 30, которое представляет радиус в пикселах – расстояние от центра круга до его внешнего края);
- *ширину* контура (если `width = 0`, круг полностью залит цветом, как в нашем примере).

Теперь взглянем на эти пять моментов более внимательно.

СЛОВАРИК

Слово *пиксель* – это сокращение от словосочетания «picture element». Означает одну точку на твоем экране или изображении. Если ты масштабируешь любое растровое изображение с помощью инструмента увеличения, то видишь отдельные пиксели. Ниже показан обычный размер фотографии и увеличенная версия, где ты можешь рассмотреть пиксели.



Обычный экран компьютера может вместить 1080 строк пикселей по высоте и 1920 столбцов пикселей по ширине. Мы говорим, что у монитора «разрешение 1920×1080». У некоторых мониторов бывает больше пикселей, а у других – меньше.

Поверхности Pygame

Если бы я попросил тебя нарисовать настоящую картину, ты бы сразу же спросил: «На чем?» В Pygame *поверхность* – это то, на чем мы рисуем. *Поверхность экрана* – это то, что мы видим на экране. Это ее мы назвали `screen` в листинге 16.3. Но у программы Pygame может быть много поверхностей, и ты можешь копировать изображение с одной поверхности на другую. Ты можешь выполнять действия с поверхностями, например вращать их, менять размер (делать больше или меньше).

Как мы уже говорили, есть две копии поверхности экрана. На профессиональном жаргоне программного обеспечения мы говорим, что поверхность дисплея имеет *двойную буферизацию*. Именно так мы не видим неоконченные фигуры и рисунки на экране. Мы рисуем свои круги, инопланетян и все остальное в буфере, а затем «прыгаем» сразу к той поверхности экрана, которая показывает готовый рисунок.

Цвета в Pygame

Цветовая схема, используемая в Pygame, широко распространена в других языках программирования и программах. Она называется *RGB*. Большие буквы соответствуют английским названиям основных цветов: красного (Red), зеленого (Green) и синего (Blue).

На уроках физики ты мог узнать, что любой цвет можно создать сочетанием трех *основных* цветов: красного, зеленого и синего. Точно так же это работает и на компьютерах. Каждый цвет – красный, зеленый и синий – получает значение от 0 до 255. Если все значения 0, то никакого цвета нет, полная темнота, то есть черный цвет. Если все значения 255, ты получаешь самую яркую из смеси трех цветов, то есть белый цвет. Если у тебя что-то похожее на [255,0,0], это будет красный цвет без примеси зеленого или синего. Зеленый имеет комбинацию [0,255,0], а синий – [0,0,255]. Если все цвета одинаковые, например [150,150,150], ты получаешь оттенок серого. Чем меньше значение цвета, тем темнее его оттенок, чем больше – тем светлее.

Имена цветов

В Pygame есть список именованных цветов, и ты можешь использовать их, если не хочешь пользоваться обозначениями [R,G,B]. Есть более 600 определенных имен цветов. Мы не будем приводить их все здесь, но если хочешь взглянуть на них, найди в папке с примерами файл *Словарь_цветов.py* и открой его в текстовом редакторе.

Если хочешь использовать имена цветов в программе, нужно добавить следующую строку в ее начале:

```
from pygame.color import THECOLORS
```

Тогда при выборе цвета ты будешь делать это примерно так (на примере нашего круга):

```
pygame.draw.circle(screen, THECOLORS["red"], [100,100], 30, 0)
```

Если хочешь поиграть и поэкспериментировать с созданием разных цветов из красного, зеленого и синего, попробуй запустить программу *Микшер_цвета.py*, которая находится в папке примеров для этой книги. С ее помощью можно попробовать любую комбинацию красного, зеленого и синего и увидеть результат.

ЧТО ТАМ ПРОИСХОДИТ?

Почему 255? Ряд чисел от 0 до 255 дает нам 256 разных значений каждого первичного цвета (красного, зеленого и синего). Но что особенного в этом числе? Почему не 200, 300 или 500?

Двести пятьдесят шесть – это максимальное количество разных значений, которые ты можешь создать с помощью 8 бит. Это все возможные комбинации восьми единиц и нулей. Восемь бит также называются байтом, а байт – это самый маленький объем памяти, который имеет свой собственный адрес. Адрес – это способ компьютера находить отдельные участки памяти.

Как и на твоей улице твой дом или квартира имеют свой адрес, но у твоей комнаты своего адреса нет. Дом – это самая маленькая «единица с адресом» на улице. Байт – самая маленькая «единица с адресом» в памяти компьютера.

Можно было использовать больше, чем 8 бит для каждого цвета, но следующее значение, которое имело бы смысл, – это 16 бит (2 байта), потому что не очень удобно использовать только часть байта. И получается, из-за физических возможностей человеческого глаза 8 бит достаточно, чтобы создать реалистично выглядящие цвета.

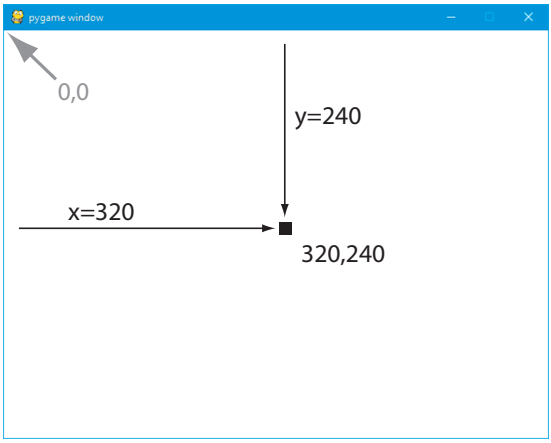
Поскольку есть три значения (красный, зеленый, синий), каждое по 8 бит, вместе это 24 бита, поэтому этот способ представления цветов называется 24-битным. Для каждого пиксела используется 24 бита, 8 бит для каждого первичного цвета.

Расположение – координаты экрана

Если мы хотим нарисовать или поместить что-то на экран, то должны уточнить, где именно. Существует два числа: одно по оси *x* (горизонтальное направление) и второе по оси *y* (вертикальное). В Pygame числа начинаются с [0,0] в верхнем левом углу окна.

Когда ты видишь пару чисел наподобие [320, 240], первое число передает горизонтальное направление или расстояние от левой стороны. Второе число – вертикальное направление или расстояние от верхнего края окна. В математике и программировании буква *x* часто используется для обозначения расстояния по горизонтали, а *y* – по вертикали.

Мы создали окно с высотой 480 пикселей и шириной 640 пикселей. Если бы нам нужно было поместить круг в центр окна, нам надо бы было рисовать его в точке с координатами [320, 240]. То есть 320 пикселей слева и 240 пикселей вниз от левого верхнего угла окна.



Давай попробуем нарисовать круг в центре окна. Набери код программы из листинга 16.4.

Листинг 16.4. Помещение круга в центр окна

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
pygame.draw.circle(screen, [255,0,0],[320,240], 30, 0)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Измени значение [100, 100] на [320, 240]

Местоположение [320, 240] использовано в качестве центра круга. Сравни результат вывода программы из листинга 16.4 с результатами программы из листинга 16.3, чтобы увидеть разницу.

Размер фигур

Когда ты используешь функцию **`draw`** для рисования фигур, тебе нужно указывать размер этих фигур. Для круга размер только один: радиус. Для прямоугольника нужно указать длину и ширину.

В Pygame есть особый вид объекта под названием **`Rect`** (сокращенно от *rectangle* – англ. прямоугольник), который используется для определения прямоугольных областей. Объект **`Rect`** определяется с помощью координат его верхнего левого угла и его высоты и ширины:

```
Rect(left, top, width, height)
```

Так одновременно определяются место и размер. Например:

```
my_rect = Rect(250, 150, 300, 200)
```

Этот код создает прямоугольник, верхний левый угол которого находится на расстоянии 250 пикселей от левого края окна и 150 пикселей от верхнего края. Прямоугольник будет шириной 300 пикселей и высотой 200 пикселей. Давай попробуем и убедимся.

Замени этой строкой строку 5 в листинге 16.4 – и увидишь, что получится:

```
pygame.draw.rect(screen, [255,0,0], [250, 150, 300, 200], 0)
```

Цвет прямоугольника — *Местоположение и размер прямоугольника* — *Ширина контура или заливка*

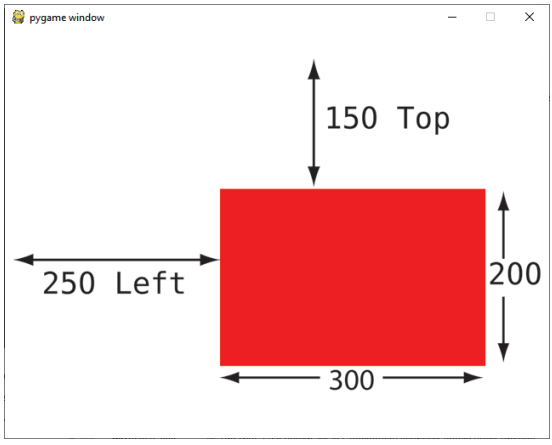
Расположение и размеры прямоугольника могут быть простым списком (или кортежем) чисел или объектом **`Rect`** из Pygame. То есть предыдущую строку можно было заменить двумя такими:

```
my_list = [250, 150, 300, 200]
pygame.draw.rect(screen, [255,0,0], my_list, 0)
```

или

```
my_rect = pygame.Rect(250, 150, 300, 200)
pygame.draw.rect(screen, [255,0,0], my_rect, 0)
```

Вот так должен выглядеть прямоугольник. Мы добавили значения размеров, чтобы показать, какие числа что означают.



Обрати внимание, что мы передаем только четыре аргумента **`pygame.draw.rect`**. Это происходит потому, что объект **`rect`** содержит как расположение, так

и размер в одном аргументе. В **pygame.draw.circle** расположение и размер были двумя разными аргументами, поэтому мы передавали пять аргументов.

Думай как программист (на Pygame)

При создании прямоугольника с помощью объекта **Rect(left, top, width, height)** доступно несколько дополнительных атрибутов, с помощью которых можно передвинуть или выровнять объект **Rect**:

- **четыре края: top, left, bottom, right;**
- **четыре угла: topleft, bottomleft, topright, bottomright;**
- **середина каждой стороны: midtop, midleft, midbottom, midright;**
- **центр: center, centerx, centery;**
- **размеры: size, width, height.**

Это для удобства. Если тебе нужно передвинуть прямоугольник так, чтобы его центр был в определенной точке, тебе не нужно думать, какими будут верхние и левые координаты; ты можешь получить доступ к центральной точке напрямую.

Толщина линии

Последнее, что нужно уточнить при рисовании фигур, – это толщина линии. В наших примерах мы использовали линию толщиной 0, таким образом заполняя всю фигуру. Если бы мы использовали другое значение, ты бы видел контур фигуры.

Попробуй изменить толщину линии на 2:

```
pygame.draw.rect(screen, [255,0,0], [250, 150, 300, 200], 2)
                                                    Измени на 2
```

Попробуй – и увидишь, как будет выглядеть фигура. Попробуй другие значения толщины линии.

Современное искусство?

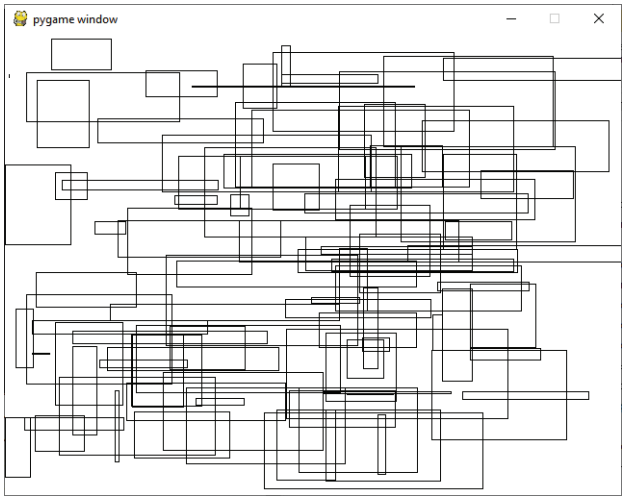
Хочешь создать творение в духе компьютерного современного искусства? Просто ради прикола, выполни код из листинга 16.5. Можно начать с кода в листинге 16.4 и изменить его или написать программу с нуля.

Листинг 16.5. Использование draw.rect для создания предмета искусства

```
import pygame, sys, random
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
```

```
for i in range (100):
    width = random.randint(0, 250)
    height = random.randint(0, 100)
    top = random.randint(0, 400)
    left = random.randint(0, 500)
    pygame.draw.rect(screen, [0,0,0], [left, top, width, height], 1)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Запусти эту программу – и увидишь, что получится. Должно выглядеть примерно так:



Понимаешь, как работает программа? Она рисует сто прямоугольников случайного размера и на случайных позициях. Чтобы сделать ее еще более «художественной», добавь цвет и сделай толщину линии тоже случайной, как показано в листинге 16.6.

Листинг 16.6. Современное искусство в цвете

```
import pygame, sys, random
from pygame.color import THECOLORS
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
for i in range (100):
    width = random.randint(0, 250)
```

```
    height = random.randint(0, 100)
    top = random.randint(0, 400)
    left = random.randint(0, 500)
    color_name = random.choice(list(THECOLORS.keys()))
    color = THECOLORS[color_name]
    line_width = random.randint(1, 3)
    pygame.draw.rect(screen, color, [left, top, width, height], line_width)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Сейчас не беспокойся о том, как работает эта строка

Когда ты запустишь эту программу, ты увидишь нечто, что выглядит каждый раз иначе. Если получишь интересное изображение, дай ему название наподобие «Голос машины» и попробуй продать в местную галерею!

Отдельные пиксели

Иногда нам не нужно рисовать круг или прямоугольник, но нам нужны отдельные точки или пиксели. Может быть, мы пишем математическую программу и хотим нарисовать синусоиду.

Не переживай, если ты не знаешь, что такое синусоида. Для целей этой главы это может быть просто волнистая фигура.

И также не переживай о математических формулах в нескольких последующих примерах. Просто набирай их в листингах по мере появления. Это способ получения волнистой фигуры подходящего размера для заполнения окна Pygame.

Поскольку не существует метода **`pygame.draw.sinewave()`**, нам придется рисовать фигуру самим по отдельным точкам. Первый способ: нарисовать маленькие

кружочки или прямоугольники размером 1–2 пиксела. Листинг 16.7 показывает реализацию с использованием прямоугольников.

Листинг 16.7. Рисование волнистых фигур с помощью маленьких прямоугольников

```
import pygame, sys
import math
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
for x in range(0, 640):
    y = int(math.sin(x/640 * 4 * math.pi) * 200 + 240)
    pygame.draw.rect(screen, [0,0,0],[x, y, 1, 1], 1)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

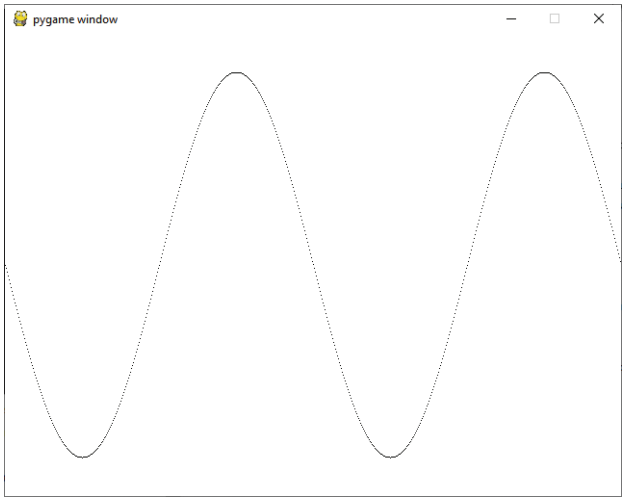
Импорт математических функций, включая **sin()**

Подсчет расположения каждой точки по вертикальной оси y

Циклы слева направо, от x = 0 до 639

Рисуем точку с использованием маленького прямоугольника

И вот как должен выглядеть вывод:



Чтобы нарисовать каждую точку, мы использовали прямоугольник шириной 1 пиксель и высотой 1 пиксель. Обрати внимание, что мы использовали толщину линии 1, а не 0. Если бы мы использовали значение 0, ничего не было бы видно, потому что нет никакой «середины» для заполнения.

Соединение точек

Если присмотреться, ты заметишь, что синусоида не сплошная – есть пробелы между точками в середине. Это происходит потому, что в наклонной части сину-

соиды нам нужно передвинуть ее на 3 пиксела вверх (или вниз) при сдвиге на 1 пиксель вправо. И поскольку мы рисуем отдельные точки, а не линии, нечем заполнить место между пикселями.

Давай попробуем нарисовать ту же фигуру, но с помощью линий, соединяющих каждую точку. В Pygame есть метод для рисования линий, но есть также и метод для рисования нескольких линий между наборами точек (что-то наподобие «соединить точки»). Этот метод называется **`pygame.draw.lines()`**, и ему необходимо передать пять параметров:

- поверхность для рисования – **`surface`**;
- цвет – **`color`**;
- будет ли фигура закрыта (**`closed`**) с помощью линии, соединяющей первую и последнюю точки. Мы не хотим замыкать синусоиду, поэтому этот параметр будет принимать значение **`False`** для нас;
- список (**`list`**) точек для соединения;
- ширина (**`width`**) линии.

Итак, в нашем примере с синусоидой метод **`pygame.draw.lines()`** будет выглядеть так:

```
pygame.draw.lines(screen, [0,0,0], False, plotPoints, 1)
```

В цикле **`for`** вместо рисования каждой точки мы просто создадим список точек для соединения с помощью функции **`draw.lines()`**. Затем мы один раз обратимся к функции **`draw.lines()`**, которая находится вне цикла **`for`**. Весь код программы показан в листинге 16.8.

Листинг 16.8. Правильно соединенная синусоида

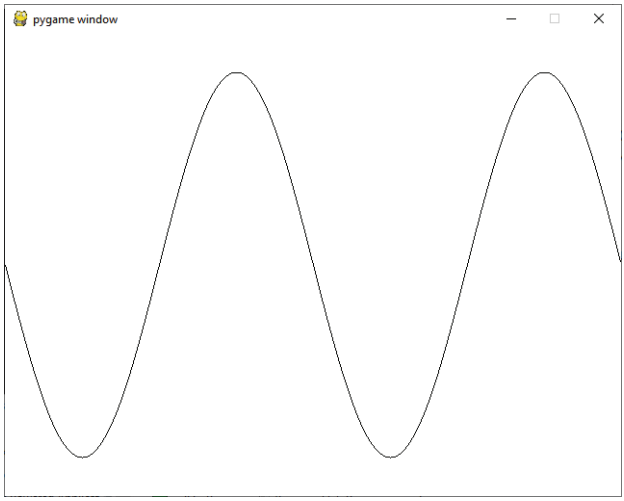
```
import pygame, sys
import math
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
plotPoints = []
for x in range(0, 640):
    y = int(math.sin(x/640 * 4 * math.pi) * 200 + 240)
    plotPoints.append([x, y])
pygame.draw.lines(screen, [0,0,0], False, plotPoints, 1)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Подсчет расположения каждой точки по оси y

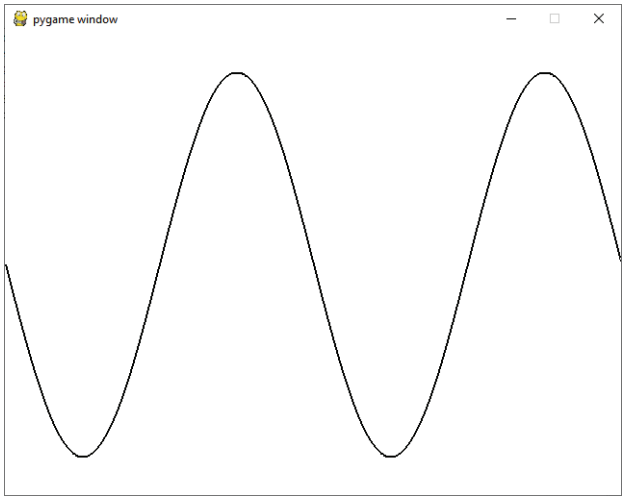
Добавление каждой точки в список

Рисуем всю кривую с помощью функции **`draw.lines()`**

Теперь при запуске вывод программы выглядит так:



Так лучше: нет пробелов между точками. Если увеличить ширину линии до 2, будет еще красивее:



И вновь соединение точек

Помнишь игры на соединение точек из детства? Вот тебе версия от Pygame.

Программа, код которой приведен в листинге 16.9, создает фигуру с помощью функции `draw.lines()` и списка точек. Чтобы увидеть секретную картинку, набери код программы из листинга 16.9. В этот раз никакого мошенничества! Этой программы нет в папке с примерами – тебе нужно набрать ее код вручную, если

хочешь увидеть волшебную картинку. Но ввод всех чисел может быть утомительным, поэтому ты найдешь список точек в текстовом файле *Список точек.txt* в папке с примерами.

Листинг 16.9. Волшебная картинка из точек

```
import pygame, sys
pygame.init()

dots = [[221, 432], [225, 331], [133, 342], [141, 310],
        [51, 230],  [74, 217],  [58, 153],  [114, 164],
        [123, 135], [176, 190], [159, 77],  [193, 93],
        [230, 28],  [267, 93],  [301, 77],  [284, 190],
        [327, 135], [336, 164], [402, 153], [386, 217],
        [409, 230], [319, 310], [327, 342], [233, 331],
        [237, 432]]

screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
pygame.draw.lines(screen, [255,0,0], True, dots, 2) ← В этот раз closed=True
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Рисование по точкам

Давай на минутку вернемся к рисованию по точкам. Кажется немного глупым рисовать маленькие круги и прямоугольники, когда все, что нам нужно, – изменить цвет пиксела. Вместо использования функций рисования можно получить доступ к каждому пикселу на поверхности с помощью метода **`Surface.set_at()`**. Ты указываешь, какой пиксель и каким цветом окрасить:

```
screen.set_at([x, y], [0, 0, 0])
```

Если мы используем эту строку кода в нашем примере с синусоидой (вместо строки 8 в листинге 16.7), результат будет выглядеть так же, как и при использовании прямоугольников шириной 1 пиксель.

Еще можно проверить цвет пиксела с помощью метода **`Surface.get_at()`**. Ты просто передаешь ему координаты пиксела, который нужно проверить: **`pixel_color = screen.get_at([320, 240])`**. В этом примере **`screen`** – имя поверхности.

Изображения

Рисование фигур, линий и отдельных пикселей на экране – один из способов создания графики. Но иногда нам нужно использовать изображения, полученные со стороны, – может быть, фотографию, что-то, скачанное из Всемирной паутины или созданное в графическом редакторе. В Pygame самый простой способ использовать изображения – применить функцию **`image`**.

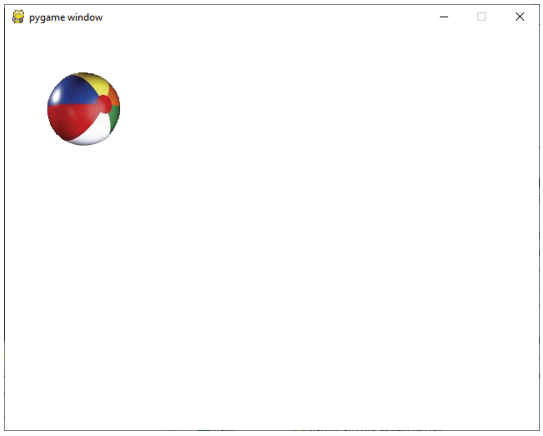
Рассмотрим пример. Мы выведем на экран изображение, которое уже есть на твоем жестком диске, если ты распаковал архив с примерами, прилагаемый к этой книге. В папке *Примеры* ты найдешь файл, который мы будем использовать в этом примере, – *beach_ball.png*. Тебе нужно скопировать файл *beach_ball.png* в ту папку, где ты хранишь свои программы на Python. Таким образом, Python легко найдет его при запуске программы. После этого набери код программы из листинга 16.10 и запусти ее.

Листинг 16.10. Вывод изображения в окне Pygame

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load("beach_ball.png")
screen.blit(my_ball, [50, 50])
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Это единственные новые строки

Когда ты запустишь эту программу, то увидишь изображение мяча возле верхнего левого угла окна Pygame.



В листинге 16.10 единственными новыми строками были 5-я и 6-я. Все остальное ты уже видел в листингах 16.3–16.9. Мы заменили код **`draw`** из предыдущих примеров кодом, который загружает изображение с диска и показывает его.

В строке 5 функция **`pygame.image.load()`** загружает изображение с диска и создает объект под названием **`my_ball`**. Объект **`my_ball`** – это поверхность. (Мы говорили о поверхностях несколькими страницами ранее.) Но мы не видим эту поверхность. Она только в памяти. Единственная поверхность, которую мы видим, – это поверхность *экрана*, названная **`screen`**. (Мы создали ее в строке 3.) Строка 6 копирует поверхность **`my_ball`** на поверхность **`screen`**. Затем функция **`display.flip()`** делает ее видимой, как и ранее.

Хорошо, Картер. Скоро мяч будет двигаться!

Ты мог заметить странную вещь в строке 6 листинга 16.10: **`screen.blit()`**. Что значит **`blit`**? Читай «Словарик» далее, чтобы узнать об этом.

СЛОВАРИК

При программировании графики часто выполняется копирование пикселей из одной позиции в другую (например, из переменной на экран или с одной поверхности на другую). Копирование пикселей имеет специальное название в программировании. Оно называется *блиттингом* (от англ. *to blit* – переносить фрагмент растрового изображения). Мы говорим, что мы *переносим* изображение (или часть изображения, или группу пикселей) из одного места в другое. Это просто красивый способ сказать «копировать», но когда ты видишь «blit», ты знаешь, что это относится к копированию пикселей, а не другого содержимого.

В Pygame мы копируем, или *переносим*, пиксели с одной *поверхности* на другую. Здесь мы скопировали пиксели с поверхности **`my_ball`** на поверхность **`screen`**.

В строке 6 листинга 16.10 мы *перенесли* изображение мяча в местоположение **`[50, 50]`**. То есть 50 пикселей от левого края и 50 пикселей от верхнего края окна. Когда ты работаешь с **`surface`** или **`rect`**, этот код устанавливает положение верхнего левого угла изображения. Итак, левый край мяча отстоит от левого края окна на 50 пикселей, а верхний край мяча отстоит на 50 пикселей от верха окна.

Давай двигаться!

Теперь, когда мы можем поместить графику в окно Pygame, давай начнем ее перемещать. Именно, мы собираемся заняться анимацией! Компьютерная анимация состоит в перемещении изображений (групп пикселей) из одного места в другое. Давай попробуем сдвинуть наш мяч.

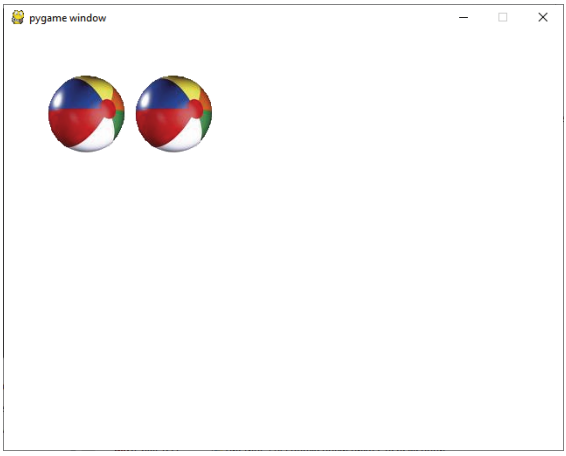
Чтобы его передвинуть, нам нужно изменить его расположение. Для начала подвинем его в сторону. Чтобы убедиться, что мы видим движение, давай передвинем его на 100 пикселей вправо. Горизонтальное направление (лево-право) – первое в паре чисел, которые указывают расположение. Итак, чтобы передвинуть что-то вправо на 100 пикселей, нам нужно увеличить первое число на 100. Мы также введем задержку, чтобы видеть процесс анимации.

Измени код программы из листинга 16.10 согласно листингу 16.11. (Тебе нужно добавить строки 8, 9 и 10 перед циклом **`while`**.)

Листинг 16.11. Попытка сдвинуть мяч

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
screen.blit(my_ball,[50, 50])
pygame.display.flip()
pygame.time.delay(2000)                          # Три новые строки
screen.blit(my_ball,[150, 50])                   # Три новые строки
pygame.display.flip()                            # Три новые строки
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Выполни программу – и увидишь, что получится. Мяч сдвинулся? Что? Ты увидел два мяча?!



Первый находится на оригинальной позиции, а затем появляется второй справа от него несколькими секундами позже. Итак, мы *передвинули* мяч вправо, но мы забыли об одном. Нужно стереть первый мяч!

Анимация

При выполнении анимации компьютерной графики существует два шага в перемещении объектов:

1. рисуем объект на новом месте;
2. стираем объект со старого места.

Ты уже видел первый шаг. Мы нарисовали мяч на новом месте. Теперь нам нужно стереть мяч оттуда, где он находился ранее. Но что значит «стереть»?

Стирание изображений

Когда ты нарисовал что-то карандашом на бумаге или мелом на доске, этот рисунок легко стереть. Ты просто используешь ластик или губку, верно? А если ты написал картину красками? Допустим, ты нарисовал голубое небо и птичку на нем. Как ты сотрешь птичку? Ты же не можешь стереть рисунок красками. Но ты можешь нарисовать еще кусочек голубого неба поверх птички.

В компьютерной графике используется тот же принцип, что и с рисунком красками, а не карандашом или мелом. Чтобы «стереть» что-то, тебе на самом деле нужно это «закрасить». Но чем закрашивать? В случае с небом оно голубое, поэтому ты закрашиваешь птицу голубым цветом. Наш фон белый, поэтому мы должны закрасить мяч белым цветом.

Давай попробуем. Измени код программы из листинга 16.11 согласно листингу 16.12. Нужно добавить только одну строку.

Листинг 16.12. Еще одна попытка передвинуть мяч

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
screen.blit(my_ball,[50, 50])
pygame.display.flip()
pygame.time.delay(2000)
screen.blit(my_ball, [150, 50])
pygame.draw.rect(screen, [255,255,255], [50, 50, 90, 90], 0)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Эта строка стирает «первый» мяч

Мы добавили строку 10, чтобы нарисовать белый прямоугольник поверх первого мяча. Изображение мяча примерно 90 пикселей в ширину и 90 пикселей в высоту, поэтому мы создали прямоугольник такого размера. Если ты запустишь программу из листинга 16.12, должно получиться так, как будто мяч двигается со своего начального места на новое.

А что на заднем плане?

Закрашивание на белом фоне (или голубом) – это просто. А что, если ты нарисовал птицу в облачном небе? Или на фоне дерева? Тогда тебе придется закрашивать птицу облаками или деревьями, чтобы стереть ее. Смысл в том, что тебе нужно следить за тем, что нарисовано на фоне, как бы «под» твоим изображением, потому что при анимации тебе нужно вернуть на место или вновь нарисовать то, что было на фоне.

Это довольно легко в нашем примере с мячом, потому что фон просто белый. Но если фон был бы пляжем, все было бы сложнее. Вместо рисования белого прямоугольника нам пришлось бы рисовать нужный участок фонового изображения. Другой вариант – перерисовать заново всю сцену и затем поместить мяч в новую позицию.

Плавная анимация

До сих пор наш мяч совершал одно движение. Давай посмотрим, получится ли у нас заставить его двигаться более реалистично. При анимации на экране обычно лучше двигать предметы маленькими шагами, чтобы движение было более плавным. Давай попробуем перемещать мяч пошагово.

Мы собираемся уменьшить не только шаги – мы добавим цикл для перемещения мяча (потому что нам нужно много маленьких шагов). Начиная с листинга 16.12 измени код в программе так, чтобы он выглядел в соответствии с листингом 16.13.

Листинг 16.13. Плавное перемещение мяча

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
screen.blit(my_ball,[x, y])
pygame.display.flip()
for looper in range (1, 100):
    pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + 5
    screen.blit(my_ball, [x, y])
    pygame.display.flip()
running = True
while running:
```

Добавь эти строки

Использует x и y (вместо чисел)

Начинает цикл

Изменяет значение **time.delay** *с* **2000** *на* **20**

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Если запустить эту программу, ты увидишь мяч, движущийся со своего первоначального положения в новое в правой части окна.

Мяч продолжает двигаться

В предыдущей программе мяч двигался в правую часть окна и остановился. Теперь попробуем заставить мяч двигаться дальше.

Если бы мы просто увеличивали значение переменной **x**, что бы произошло? Мяч продолжил бы двигаться вправо при увеличении значения **x**. Но наше окно (поверхность экрана) ограничено значением **x = 640**. Поэтому мяч просто исчезнет. Попробуй изменить цикл **for** в строке 10 листинга 16.13 следующим образом:

```
for looper in range(1, 200):
```

Теперь цикл продолжается в два раза дольше, и мяч пропадает за пределами экрана! Если мы хотим и дальше его видеть, у нас есть два варианта:

- мы заставляем мяч *оттолкнуться* от края окна;
- мы заставляем мяч *вернуться* с другой стороны окна.

Давай попробуем оба способа.

Мяч отталкивается

Если мы хотим, чтобы мяч *отпрыгивал* от края окна, нам нужно знать, когда он этого края достигает, и затем изменить его направление на обратное. Если мы хотим, чтобы мяч двигался туда-сюда, нам нужно сделать это для правого и левого краев окна.

Для левого края это легко, потому что мы просто проверяем положение мяча, и оно должно быть равным 0 (или какое-то малое число).

С правой стороны нам нужно проверить, что правая сторона мяча находится у правого края окна. Но положение мяча задано с его левой стороны (верхнего левого угла), а не с правой. Поэтому нам нужно вычесть ширину мяча:

Когда мяч движется по направлению к правой стороне окна, нам нужно оттолкнуть его (изменить его направление) при его положении, равном 550.

Чтобы сделать все проще, внесем изменения в код:

- мяч будет отталкиваться от краев постоянно (или до тех пор, пока мы не закроем окно Pygame). Поскольку у нас уже есть цикл **`while`**, который выполняется, пока открыто окно, мы перенесем наш код, отображающий мяч, в этот цикл (цикл **`while`** – последняя часть программы);
- вместо добавления числа 5 к положению мяча мы создадим новую переменную, **`speed`**, для определения скорости движения мяча для каждой итерации. И также увеличим немного скорость мяча, присвоив переменной значение 10.

Новый код показан в листинге 16.14.

Листинг 16.14. Отталкивание мяча

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
x_speed = 10

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + x_speed
    if x > screen.get_width() - 90  or  x < 0:
    x_speed = - x_speed
    screen.blit(my_ball, [x, y])
    pygame.display.flip()
pygame.quit()
```

Переменная скорости

Помести код для отображения мяча здесь, внутри цикла **`while`**

Когда мяч ударяется о край окна...

...изменить направление движения, обратив скорость

Ключ к отталкиванию мяча от краев окна – в строках 19 и 20. В строке 19 (**`if x > screen.get_width() – 90 or x < 0:`**) мы определяем, находится ли мяч возле края окна, и если да, то изменяем его направление в строке 20 (**`x_speed = – x_speed`**).

Попробуй – и увидишь, как это работает.

Отталкивание в двух измерениях

До сих пор наш мяч перемещался только туда-сюда или движение происходило в одном измерении. Теперь давай заставим его перемещаться еще и вверх-вниз. Для этого нам нужны незначительные изменения, как показано в листинге 16.16.

Листинг 16.15. Отталкивание мяча в двух измерениях

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
x_speed = 10
y_speed = 10          <-- Добавление кода для скорости по оси y (вертикальная)
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
            pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + x_speed
    y = y + y_speed   <-- Добавление кода для скорости по оси y (вертикальная)
    if x > screen.get_width() - 90 or x < 0:
        x_speed = - x_speed
    if y > screen.get_height() - 90 or y < 0:
        y_speed = -y_speed
    screen.blit(my_ball, [x, y])
    pygame.display.flip()
pygame.quit()
```

Отталкиваем мяч от нижнего или верхнего края окна

Мы добавили строки: 9 (**`y_speed = 10`**), 18 (**`y = y + y_speed`**), 21 (**`if y > screen.get_height() – 90 or y < 0:`**) и 22 (**`y_speed = -y_speed`**) в предыдущую программу. Попробуй ее запустить!

Если ты хочешь замедлить движение мяча, есть два способа:

- ты можешь уменьшить значения переменных скорости (**`x_speed`** и **`y_speed`**). Это уменьшает расстояние, которое проходит мяч за каждый анимационный шаг, поэтому движение будет еще более плавным;
- ты можешь увеличить время задержки. В листинге 16.15 его значение 20. Это значение в миллисекундах, то есть тысячных долях секунды. Каждый раз во время цикла программа ждет 0,02 секунды. Если ты увеличишь это число, движение замедлится. Если уменьшишь, ускорится.

Попробуй поэкспериментировать со скоростью и задержкой, чтобы увидеть разные эффекты.

Мяч оборачивается

Теперь давай посмотрим на второй вариант продолжения движения мяча. Вместо отталкивания от края окна мяч, пропав с правой стороны окна, снова появится с левой.

Чтобы тебе было проще, ограничимся горизонтальным движением. Программа показана в листинге 16.16.

Листинг 16.16. Появление мяча с другой стороны экрана

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
x_speed = 5
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + x_speed
    if x > screen.get_width(): <-- Если мяч находится далеко справа...
        x = 0                  <-- ...рисуем его снова с левой стороны
    screen.blit(my_ball, [x, y])
    pygame.display.flip()
pygame.quit()
```

В строках 17 (**`if x > screen.get_width():`**) и 18 (**`x = 0`**) мы обнаруживаем, когда мяч достигает правого края окна, и возвращаем его или оборачиваем его обратно на левую сторону.

Ты мог заметить, что когда мяч появляется справа, он «выскакивает» при значении **`[0, 50]`**. Было бы более натурально, если бы он «ускользал» с экрана. Измени строку 18 (**`x = 0`**) на **`x = -90`** и посмотри, видна ли разница.

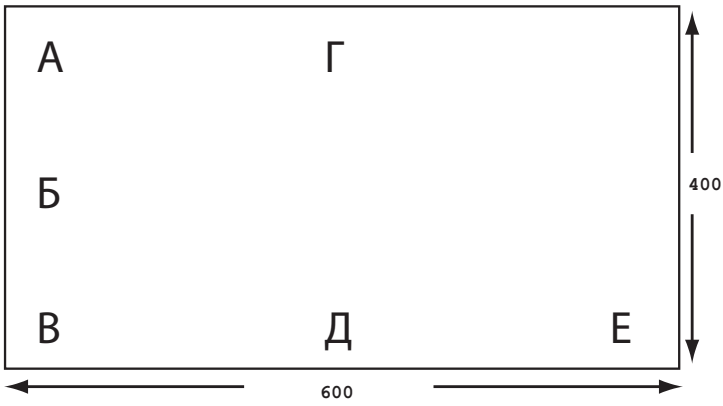
Что ты узнал?

Это была длинная глава! В ней ты узнал:

- как использовать Pygame;
- как создавать графическое окно и рисовать в нем фигуры;
- как настраивать цвета в компьютерных изображениях;
- как копировать изображения в графическое окно;
- как анимировать изображения, включая их «стирание» при перемещении на новое место;
- как заставить мяч «отталкиваться» от края окна;
- как заставить мяч вернуться с другой стороны окна.

Проверь свои знания

1. Какой цвет создает значение RGB [255, 255, 255]?
2. Какой цвет создает значение RGB [0, 255, 0]?
3. Какой метод Pygame можно использовать для рисования прямоугольников?
4. Какой метод Pygame можно использовать для рисования линий, соединяющих вместе некоторое количество точек?
5. Что значит термин «пиксел»?
6. В окне Pygame где находится точка [0, 0]?
7. Если окно Pygame было 600 пикселей в ширину и 400 пикселей в высоту, какая буква на рисунке ниже находится в точке с координатами [50, 200]?



8. Какая буква на рисунке находится в точке с координатами [300, 50]?
9. Какой метод Pygame используется для копирования изображений на поверхность (например, поверхность экрана)?
10. Каковы два основных шага при «перемещении» или анимации изображения?

Попробуй самостоятельно

1. Мы говорили о рисовании кругов и прямоугольников. В Pygame также есть методы для рисования линий, арок, эллипсов и многогранников. Попробуй использовать их для рисования других фигур в программе.
   Ты можешь узнать больше об этих методах в документации к Pygame по адресу **www.pygame.org/docs/ref/draw.html**.
   Ты также можешь использовать систему справки Python (о которой мы говорили в конце главы 6).

```
>>> import pygame
>>> help()
help> pygame.draw
```

Ты увидишь список разных методов рисования и некоторые объяснения к каждому из них.

2 Попробуй изменить код одной из программ в примерах, которые используют мяч, так чтобы она использовала другое изображение. Ты можешь найти примеры изображений в папке *Примеры\images*, загрузить или нарисовать свой рисунок. Ты можешь использовать фрагмент цифровой фотографии.

3 Измени значения **`x_speed`** и **`y_speed`** в листингах 16.15 или 16.16 так, чтобы заставить мяч двигаться быстрее или медленнее, а также в разных направлениях.

4 Измени листинг 16.15 так, чтобы заставить мяч отталкиваться от невидимой стены или пола, который не является краем окна.

5 В листингах 16.5–16.7 (современное искусство, синусоида и волшебная картинка) попробуй перенести строку с **`pygame.display.flip`** в цикл **`while`**. Для этого сдвинь ее четырьмя пробелами. После этой строки и внутри цикла **`while`** добавь задержку с помощью такой строки и посмотри, что получится:

```
pygame.time.delay(30)
```

ГЛАВА 17

Спрайты и обнаружение столкновений

В этой главе мы продолжим создавать анимации с помощью Pygame. Ты узнаешь о *спрайтах*, которые позволяют отслеживать большое количество изображений, перемещающихся на экране. Мы также узнаем, как обнаружить накладывающиеся или сталкивающиеся изображения, например когда мяч ударяется о ракетку или космический корабль об астероид.

Спрайты

В предыдущей главе ты видел, что простая анимация не так уж проста. Если у тебя много изображений и ты их все перемещаешь, может быть довольно трудно уследить, что находится «под» каждым из них, чтобы закрасить эту область при перемещении изображения. В нашем первом примере с мячом фон был просто белым, и это было легко. Но ты можешь себе представить трудности со сложной фоновой картинкой?

К счастью, Pygame может нам помочь. Отдельные изображения или фрагменты изображений, которые свободно перемещаются по экрану, называются *спрай-*

тами, и в Pygame есть особый модуль для управления ими. Он позволяет легко перемещать графические объекты.

В предыдущей главе наш мяч отталкивался от краев окна. А что, если нам нужна целая группа мячей, прыгающих по экрану? Мы могли бы написать программу, управляющую каждым из них отдельно, но вместо этого используем модуль **sprite**, чтобы упростить решение задачи.

СЛОВАРИК

Спрайт – это группа пикселей, которые перемещаются и отображаются как единая единица, тип графического объекта.

« Термин “спрайт” – это пережиток времен старых компьютеров и игровых автоматов. Эти старые машины не могли рисовать и стирать нормальную графику в играх достаточно быстро. У этих машин были специальные приспособления для управления игровыми объектами, которым требовалась быстрая анимация. Эти объекты назывались “спрайтами”, и у них были особые ограничения, но они могли быть нарисованы и стерты очень быстро… В наши дни компьютеры стали довольно быстрыми для управления спрайтоподобными объектами без специальных устройств. Термин “спрайт” до сих пор используется для описания любого анимированного объекта в двумерных играх. »

(Выдержка из книги «Pygame Tutorials – Sprite Module» Пита Шиннерса (Pete Shinners), **www.pygame.org/docs/tut/SpriteIntro.html**)

Что такое спрайт? Представь себе спрайт небольшим фрагментом графики – графическим объектом, который перемещается по экрану и взаимодействует с другими графическими объектами. У большинства спрайтов есть два базовых свойства:

- **image** – графический объект, который отображается в виде спрайта;
- **rect** – прямоугольная область, которая содержит спрайт.

Изображение может быть тем объектом, который ты рисуешь с помощью функций **draw** в Pygame (как в предыдущей главе), или картинкой из файла.

Класс спрайта

Модуль **sprite** в Pygame определяет базовый класс спрайта, который называется, соответственно, **Sprite**. (Напомним, что мы говорили об объектах и классах несколькими главами ранее.) Обычно мы не используем базовый класс напрямую, а создаем вместо него свой собственный подкласс, основанный на **pygame.sprite.Sprite**. Это мы и сделаем в следующем примере и назовем наш класс **Ball**. Код выглядит так:

```
class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
```

Инициализация спрайта

Загрузка файла с изображением в спрайт

Получение прямоугольника, который определяет границы изображения

Настройка первичного расположения мяча

Последняя строка в этом коде заслуживает более пристального внимания. **location** – это координаты [*x*, *y*], которые представлены списком из двух элементов. Поскольку у нас есть список из двух элементов по одну сторону от знака равенства (**x** и **y**), мы можем присвоить два атрибута другой стороне. Здесь мы назначаем атрибуты **left** и **top** прямоугольника спрайта.

Теперь, когда мы определили класс **Ball**, нам нужно создать некоторые из его экземпляров. (Напомним, что определение класса – это всего лишь план; теперь нам нужно построить дома по нему.) Нам все еще нужен тот код, который мы использовали в предыдущей главе для создания окна Pygame. Мы также создадим несколько мячей, организованных в строки и столбцы. Мы сделаем это с помощью вложенного цикла.

```
img_file = «beach_ball.png»
balls = []
for row in range (0, 3):
    for column in range (0, 3):
        location = [column * 180 + 10, row * 180 + 10]
        ball = Ball(img_file, location)
        balls.append(ball)
```

Меняет расположение каждый раз во время цикла

Создает мяч в этой точке

Собирает мячи в список

Нам также нужно *перенести* мячи на поверхность экрана. (Помнишь про *блиттинг*? Мы говорили о нем в предыдущей главе.)

```
for ball in balls:
    screen.blit(ball.image, ball.rect)
pygame.display.flip()
```

Соберем весь код вместе в программе, показанной в листинге 17.1.

Листинг 17.1. Использование спрайтов для создания множественных изображений мяча на экране

```
import sys, pygame

class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
```

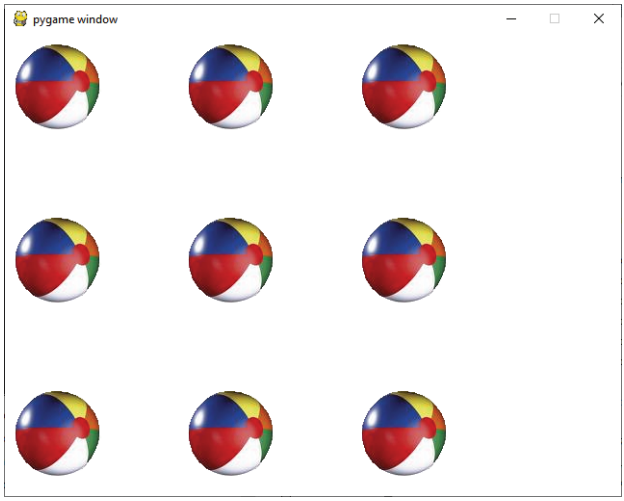
Определение подкласса мяча

```
size = width, height = 640, 480
screen = pygame.display.set_mode(size)
screen.fill([255, 255, 255])
img_file = "beach_ball.png"
balls = []
for row in range (0, 3):
    for column in range (0, 3):
        location = [column * 180 + 10, row * 180 + 10]
        ball = Ball(img_file, location)
        balls.append(ball)

for ball in balls:
    screen.blit(ball.image, ball.rect)
pygame.display.flip()

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Настройка размеров окна

Добавление мяча в список

Если ты запустишь эту программу, то увидишь девять мячей в окне Pygame:



Через минуту мы начнем их перемещать.

Ты заметил небольшое изменение в строках 10 и 11, которые настраивают размер окна Pygame? Мы заменили строку

```
screen = pygame.display.set_mode([640,480])
```

следующим кодом:

```
size = width, height = 640, 480
screen = pygame.display.set_mode(size)
```

Этот код не только настраивает размер окна (как и раньше), но и определяет две переменные, **`width`** и **`height`**, которые мы будем использовать позже. Хорошо то, что мы определили кортеж **`size`** с двумя элементами в нем, а также определили две переменные с целыми числами, **`width`** и **`height`**, в одном выражении. Мы не использовали квадратные скобки в нашем кортеже, и Python воспринял это хорошо.

Мы хотели показать тебе, что одни и те же вещи в Python можно делать по-разному. Один способ необязательно лучше, чем другой (пока они оба работают). Хотя тебе и нужно следовать правилам синтаксиса Python (правилам языка), есть небольшое свободное пространство для выражения. Если ты попросишь десятерых программистов написать одну и ту же программу, то, вероятно, не получишь и двух одинаковых фрагментов кода.

Метод `move()`

Поскольку мы создаем мячи в виде экземпляров класса **`Ball`**, имеет смысл перемещать их с помощью метода класса. Давай создадим новый метод класса, называемый **`move()`**:

```
def move(self):
    self.rect = self.rect.move(self.speed)
    if self.rect.left < 0 or self.rect.right > width:
        self.speed[0] = -self.speed[0]
    if self.rect.top < 0 or self.rect.bottom > height:
        self.speed[1] = -self.speed[1]
```

Проверка на столкновение со сторонами окна и, при положительном результате, изменение скорости по оси x

Проверка на столкновение с верхней и нижней сторонами окна и, при положительном результате, изменение скорости по оси y

Спрайты (вернее, **`rect`** с ними) имеют встроенный метод **`move()`**. Этот метод требует назначения параметра **`speed`**, который указывает, как далеко (то есть как быстро) объект будет перемещаться. Поскольку мы имеем дело с двухмерной графикой, **`speed`** – это список из двух чисел, одно для скорости по оси *x*, второе – по оси *y*. Мы также проверяем мяч на столкновение с краями окна, чтобы «оттолкнуть» его от них.

Давай изменим определение класса **`MyBallClass`**, добавив свойство **`speed`** и метод **`move()`**:

```
class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, location, speed):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed
```

Добавление аргумента **`speed`**

Добавь эту строку, чтобы создать атрибут **`speed`** *для мяча*

```
def move(self):
    self.rect = self.rect.move(self.speed)
    if self.rect.left < 0 or self.rect.right > width:
        self.speed[0] = -self.speed[0]

    if self.rect.top < 0 or self.rect.bottom > height:
        self.speed[1] = -self.speed[1]
```

Добавление метода для перемещения мяча

Обрати внимание на изменение в строке 2 (**`def__init__(self, image_file, location, speed):`**) и добавление строки 7 (**`self.speed = speed`**), а также на новый метод **`move()`** в строках 9–15.

Теперь, когда мы создали каждый экземпляр мяча, нам нужно передать им скорость, а также указать файл с изображением и расположение:

```
speed = [2, 2]
ball = Ball(img_file, location, speed)
```

Вышеуказанный код создает мячи с одной и той же скоростью (и одним направлением движения), но будет веселее, если мячи будут двигаться в случайном порядке. Давай используем функцию **`random.choice()`** для настройки скорости:

```
from random import choice
speed = [choice([-2, 2]), choice([-2, 2])]
```

Так, будет выбрана скорость –2 или 2 для скорости по оси *x* и *y*.

В листинге 17.2 приведен полный код программы.

Листинг 17.2. Программа для перемещения мячей с помощью спрайтов

```
import sys, pygame
from random import choice
class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, location, speed):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > width:
            self.speed[0] = -self.speed[0]

        if self.rect.top < 0 or self.rect.bottom > height:
            self.speed[1] = -self.speed[1]
size = width, height = 640, 480
screen = pygame.display.set_mode(size)
```

Определение класса **`Ball`**

```
screen.fill([255, 255, 255])
img_file = "beach_ball.png"
balls = []          <-- Создание списка для отслеживания мячей
for row in range (0, 3):
    for column in range (0, 3):
        location = [column * 180 + 10, row * 180 + 10]
        speed = [choice([-2, 2]), choice([-2, 2])]
        ball = Ball(img_file, location, speed)
        balls.append(ball)   <-- Добавление каждого мяча в список при его создании

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    pygame.time.delay(20)
    screen.fill([255, 255, 255])
    for ball in balls:
            ball.move()
            screen.blit(ball.image, ball.rect)
    pygame.display.flip()
pygame.quit()
```

Обновление экрана

Эта программа использует список для слежения за мячами. В строке 32 (**`balls.append(ball)`**) каждый мяч добавляется в список при своем создании.

Код в последних пяти строках программы обновляет экран. Здесь мы немного схитрили и вместо «стирания» (закрашивания) каждого мяча просто залили экран белым цветом и нарисовали мячи заново.

Ты можешь поэкспериментировать с кодом, создав больше (или меньше) мячей, изменив их скорость, направление движения, «отталкивание» и т. д. Ты увидишь, что мячи передвигаются и отталкиваются от краев окна, но не друг от друга – пока что!

Бах! Обнаружение столкновений

В большинстве компьютерных игр тебе нужно знать, когда один спрайт сталкивается с другим. Например, шар для боулинга ударяет по кеглям, или ракета попадает в космический корабль.

Ты можешь подумать, что если мы знаем расположение и размер каждого спрайта, то можем написать код для их проверки по отношению к расположению и размерам каждого следующего спрайта, чтобы знать, когда они накладываются друг на друга. Но люди, создавшие Pygame, уже сделали это за нас. В нем есть так называемое встроенное *обнаружение столкновений*.

СЛОВАРИК

Обнаружение столкновений значит просто обнаружение случаев соприкосновения или наложения двух спрайтов. Когда два движущихся объекта подходят впритык друг к другу, это называется столкновением.

В Pygame также есть способ группировки спрайтов вместе. Например, в боулинге все кегли могут быть в одной группе, а шар будет в своей собственной группе.

Группы и обнаружение столкновений идут рука об руку. В примере с боулингом тебе нужно определять попадание шара в любую из кеглей, поэтому ты будешь отслеживать столкновение спрайта мяча с любым спрайтом из группы кеглей. Ты также можешь определить столкновение внутри группы (кегли могут тоже соприкоснуться).

Давай поработаем с примером. Начнем с прыгающих мячей, но для упрощения возьмем четыре мяча вместо девяти. И вместо создания списка мячей, как мы делали в предыдущем примере, используем класс **`Group`**.

Мы также почистим код, введя часть, которая анимирует мячи (последние несколько строк в листинге 17.2), в функцию, именуемую **`animate()`**. Функция **`animate()`** также будет содержать код для обнаружения столкновений. Когда два мяча столкнутся, мы заставим их изменить направление.

Код приведен в листинге 17.3.

Листинг 17.3. Использование группы спрайтов вместо списка

```
import sys, pygame
from random import choice
class Ball(pygame.sprite.Sprite):                                    # Определение класса мяча
    def __init__(self, image_file, location, speed):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > width:
            self.speed[0] = -self.speed[0]
        if self.rect.top < 0 or self.rect.bottom > height:
            self.speed[1] = -self.speed[1]

def animate(group):                                                  # Новая функция animate()
    screen.fill([255,255,255])
    for ball in group:
        group.remove(ball)                                           # Удаление спрайта из группы
        if pygame.sprite.spritecollide(ball, group, False):          # Проверка столкновений между спрайтом и группой
            ball.speed[0] = -ball.speed[0]
            ball.speed[1] = -ball.speed[1]
        group.add(ball)                                              # Добавление мяча обратно в группу
        ball.move()
        screen.blit(ball.image, ball.rect)
    pygame.display.flip()
    pygame.time.delay(20)
size = width, height = 640, 480                                      # Начало основной программы
screen = pygame.display.set_mode(size)
```

```
screen.fill([255, 255, 255])
img_file = "beach_ball.png"
group = pygame.sprite.Group()          ← Создание группы спрайта

for row in range (0, 2):               | Создание только четырех мячей в этот раз
    for column in range (0, 2):
        location = [column * 180 + 10, row * 180 + 10]
        speed = [choice([-2, 2]), choice([-2, 2])]
        ball = Ball(img_file, location, speed)
        group.add(ball)

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    animate(group)
pygame.quit()
```

Самое интересное здесь – это то, как обнаруживаются столкновения. Модуль **sprite** имеет функцию **spritecollide()**, которая следит за столкновениями между отдельным спрайтом и любым другим спрайтом в группе. Если ты отслеживаешь столкновения между спрайтами в *одной и той же* группе, тебе нужно сделать это в три шага:

1. во-первых, ты удаляешь спрайт из группы;
2. далее ты проверяешь наличие столкновений спрайта и остальной группы;
3. наконец, ты добавляешь спрайт обратно в группу.

Это и происходит в цикле **for** в строках 21–29 (в середине функции **animate()**). Если мы сначала не удалим спрайт из группы, функция **spritecollide()** обнаружит столкновение между спрайтом и ним самим, потому что он будет в группе. Это может показаться тебе странным, но ты поймешь, если немного подумаешь над ситуацией.

Запусти программу – и увидишь, как она работает. Заметил странное поведение? Обрати внимание на две вещи:

- когда мячи сталкиваются, они «запинаются» или совершают второе столкновение;
- иногда мяч «приклеивается» к краю окна и замирает на некоторое время.

Это происходит из-за того, как мы написали функцию **animate()**. Обрати внимание, что мы передвигаем один мяч, потом проверяем его на наличие столкновений, затем двигаем второй, проверяем его на столкновения и т. д. Нам следовало бы, наверное, сначала выполнить движение, а затем проверять все столкновения.

Это легче заметить, если увеличить шаг анимации. Увеличь скорость с 2 до 5, а задержку между каждым шагом с 20 до 50.

Итак, нам нужно взять строку 28, **ball.move()**, и поместить ее в собственный цикл:

```
def animate(group):
    screen.fill([255,255,255])
    for ball in group:
        ball.move()
    for ball in group:
        group.remove(ball)

        if pygame.sprite.spritecollide(ball, group, False):
            ball.speed[0] = -ball.speed[0]
            ball.speed[1] = -ball.speed[1]

        group.add(ball)

        screen.blit(ball.image, ball.rect)
pygame.display.flip()
pygame.time.delay(20)
```

Перемещает сначала все мячи

Затем происходит определение столкновений и отталкивание мячей

Попробуй этот вариант и проверь, работает ли он лучше предыдущего.

Ты можешь поэкспериментировать с кодом, изменяя скорость (значение функции **`time.delay()`**), количество мячей, оригинальное расположение мячей, случайный порядок и т. д.), и посмотреть, что будет происходить с мячами.

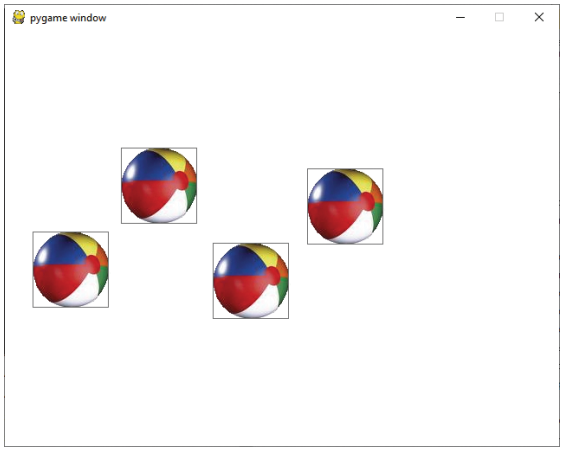
Столкновения на уровне прямоугольников и пикселей

Единственное, что ты заметишь, – это то, что мячи не всегда полностью касаются друг друга при «столкновении». Это происходит потому, что функция **`spritecollide()`** не учитывает круглую форму мячей при определении столкновения. Она учитывает прямоугольники вокруг мячей, **`rect`**.

Для наглядности нарисуй прямоугольник вокруг мяча и используй другое изображение вместо обычного мяча. Мы написали код за тебя, поэтому можешь попробовать его выполнить:

```
img_file = "b_ball_rect.png"
```

Результат должен быть таким:



Если ты хочешь, чтобы мячи отскакивали друг от друга только при соприкосновении их реальных круглых поверхностей (а не краев прямоугольников), тебе нужно отслеживать так называемые «столкновения на уровне пикселей». Функция **`spritecollide()`** не может этого сделать, она обнаруживает лишь столкновение на уровне прямоугольников.

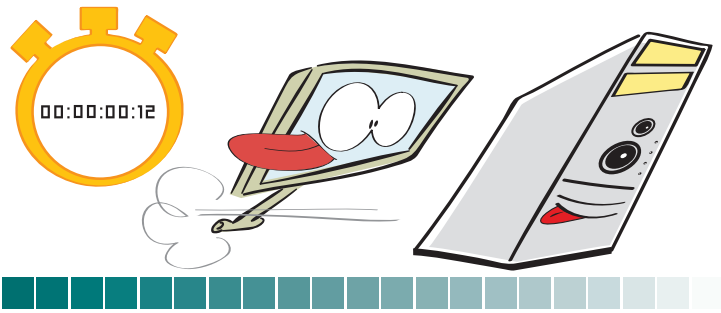
Вот в чем разница. При обнаружении столкновений на уровне прямоугольников два мяча «столкнутся», когда любые части их прямоугольников коснутся друг друга. При обнаружении столкновений на уровне пикселей два мяча столкнутся, только когда друг друга коснутся.

Столкновение на уровне пикселей выглядит более реалистично. (Ты же не чувствуешь никаких невидимых прямоугольников вокруг мячей в реальной жизни, верно?) Но его сложнее реализовать в программе.

Для большинства вещей, которые ты делаешь в Pygame, достаточно обнаружения столкновений на уровне прямоугольников. Для попиксельного обнаружения нужно больше кода, что сделает твои игры более медленными, поэтому используй его, когда оно действительно необходимо. Доступно несколько модулей обнаружения столкновений на уровне пикселей (два из них ты найдешь на веб-сайте Pygame). Также ты можешь поискать соответствующую информацию в интернете.

Отсчет времени

До сих пор мы использовали функцию `time.delay()` для управления скоростью нашей анимации. Но это не лучший способ, потому что при ее использовании ты не знаешь, насколько длинным будет каждый цикл. Код в цикле требует некоторого (неизвестного) количества времени для выполнения, а затем еще идет задержка (известное количество времени). Итак, часть времени известна, а часть нет.



Если мы хотим знать, как часто выполняется наш цикл, нам нужно знать общее время каждого цикла, которое состоит из времени выполнения кода + времени

задержки. Для подсчета времени анимации удобно использовать миллисекунды (мс) или тысячные доли секунды.

В нашем примере допустим, что код выполняется за 15 мс. Это значит, что нужно 15 мс для выполнения кода в цикле **`while`**, не включая время задержки **`time.delay()`**. Мы знаем время задержки, потому что сами установили его на 20 мс с помощью кода **`time.delay(20)`**. Общее время цикла: 20 мс + 15 мс = 35 мс, а в одной секунде 1000 мс. Если каждый цикл занимает 35 мс, мы имеем 1000 мс / 35 мс = 28,57. То есть около 29 циклов в секунду. В компьютерной графике каждый шаг анимации называется *кадром*, и программисты игр говорят о *частоте кадров*, или количестве *кадров в секунду*, когда обсуждают, как быстро обновляется их графика. В нашем примере частота кадров будет около 29 кадров в секунду, или 29 fps (от англ. *frames per second* – кадров в секунду).

Проблема в том, что мы не можем на самом деле контролировать часть уравнения со «временем кода». Если мы добавим или удалим часть кода, время изменится. Даже с тем же самым количеством кода при разном количестве спрайтов (например, игровые объекты появляются и пропадают) изменится время, необходимое для того, чтобы нарисовать их все. Кроме того, скорость работы кода зависит от компьютера. Вместо 15 мс время кода будет равно 10 мс или 20 мс. Было бы хорошо иметь более предсказуемый способ контроля частоты кадров. К счастью, модуль **`time`** из Pygame дает нам такую возможность с помощью класса **`Clock`**.

Управление частотой кадров с помощью функции `pygame.time.Clock()`

Чем добавлять задержку в каждом цикле, можно контролировать частоту выполнения каждого цикла с помощью **`pygame.time.Clock()`**. Это как таймер, который повторяет: «Начинай следующий цикл! Начинай следующий цикл!..»

До того, как ты начнешь использовать функцию времени Pygame, тебе нужно создать экземпляр объекта **`Clock`**. Это делается так же, как и создание экземпляра любого другого класса:

```
clock = pygame.time.Clock()
```

Затем в теле основного цикла ты просто указываешь «часам» функции времени, как часто им «тикать», то есть как часто выполняется цикл:

```
clock.tick(60)
```

Число, которое ты передаешь **`clock.tick()`**, – это не количество миллисекунд. Это количество итераций цикла в секунду. То есть этот цикл должен быть выполнен 60 раз в секунду. Мы говорим «должен быть», потому что цикл может

быть выполнен так быстро, как это может сделать твой компьютер. При 60 циклах (или кадрах) в секунду приходится 1000 / 60 = 16.66 мс (около 17 мс) на один цикл. Если выполнение кода в цикле занимает больше, чем 17 мс, цикл не будет завершен к тому времени, когда `clock` инициирует новый.

Обычно это означает наличие предела количества кадров в секунду, которое может выполнить графическая подсистема компьютера. Этот предел зависит от сложности графики, размера окна и быстродействия компьютера, выполняющего программу. Одну и ту же программу один компьютер может выполнить при 90 fps, а другой, более старый, будет хромать при 10 fps.

При достаточно сложной графике большинство современных компьютеров не будут испытывать сложностей при выполнении программ Pygame со скоростью 20 или 30 fps. Поэтому если ты хочешь, чтобы твои игры выполнялись с одной скоростью на большинстве компьютеров, выбирай частоту кадров 20–30 fps и меньше. Это достаточно высокая скорость для имитации плавного движения. Мы используем функцию `clock.tick(30)` для примеров из этой книги.

Проверка частоты кадров

Если ты хочешь знать, как быстро может быть выполнена твоя программа, можно проверить частоту кадров с помощью функции `clock.get_fps()`. Конечно, если ты установишь частоту кадров 30, она всегда будет 30 (при условии что твой компьютер может выполнить программу так быстро). Чтобы узнать самое короткое время, за которое тот или иной компьютер может выполнить определенную программу, установи значение `clock.tick` очень маленьким (около 200 fps), запусти программу и проверь реальную частоту кадров с помощью функции `clock.get_fps()`. (Пример будет чуть позже.)

Калибровка частоты кадров

Если ты хочешь быть уверенным, что твоя анимация выполняется с одинаковой скоростью на всех компьютерах, для этого есть трюк с использованием функций `clock.tick()` и `clock.get_fps()`. Поскольку ты знаешь желаемую скорость выполнения и действительную скорость, то можешь настроить или *откалибровать* скорость анимации в соответствии с производительностью компьютера.

Например, допустим, указано значение `clock.tick(30)` – значит, ты пытаешься выполнить программу при 30 fps. Если ты применишь `clock.get_fps()` и увидишь, что скорость не превышает 20 fps, то будешь знать, что объекты на экране движутся медленнее, чем требуется. Поскольку ты получаешь меньшее количество кадров в секунду, тебе нужно передвигать твои объекты на большее расстояние в каждом кадре, чтобы их скорость *казалась* требуемой. У тебя, вероятно, есть переменная (или атрибут) `speed` для движущихся объектов, которая указывает им, как далеко передвигаться в каждом кадре. Тебе просто нужно увеличить ее значение для более медленного компьютера.

И на сколько его увеличить? Ты увеличиваешь значение в зависимости от отношения желаемой частоты кадров к действительной. Если текущая скорость объекта равна 10 для 30 fps, а программа на самом деле выполняется при 20 fps, получаем:

```
object_speed = current_speed * (desired fps / actual fps)
object_speed = 10 * (30 / 20)
object_speed = 15
```

Итак, вместо перемещения на 10 пикселей за кадр тебе нужно двигать объект на 15 пикселей за кадр, чтобы скрыть маленькую частоту кадров.

Ниже приведен код программы с мячом, использующий новые для тебя элементы: класс **`Clock`** и функцию **`get_fps()`**.

Листинг 17.4. Использование класса `Clock` и функции `get_fps()`

```
import sys, pygame
from random import choice
class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, location, speed):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed
    def move(self):
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > width:
            self.speed[0] = -self.speed[0]
        if self.rect.top < 0 or self.rect.bottom > height:
            self.speed[1] = -self.speed[1]
def animate(group):
    screen.fill([255,255,255])
    for ball in group:
        ball.move()
    for ball in group:
        group.remove(ball)
        if pygame.sprite.spritecollide(ball, group, False):
            ball.speed[0] = -ball.speed[0]
            ball.speed[1] = -ball.speed[1]
        group.add(ball)
        screen.blit(ball.image, ball.rect)
    pygame.display.flip()
                    ← Функция time.delay() удалена
size = width, height = 640, 480
screen = pygame.display.set_mode(size)
screen.fill([255, 255, 255])
img_file = "beach_ball.png"
clock = pygame.time.Clock() ← Создание экземпляра Clock
group = pygame.sprite.Group()
for row in range (0, 2):
    for column in range (0, 2):
        location = [column * 180 + 10, row * 180 + 10]
        speed = [choice([-4, 4]), choice([-4, 4])]
        ball = Ball(img_file, location, speed)
```

Определение класса мяча

Функция анимации

Инициализация всех элементов и появление каждого мяча

```
        group.add(ball) #add the ball to the group

running = True
while running:          <-- Основной цикл while начинается здесь
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
            frame_rate = clock.get_fps()  <-- Проверка частоты кадров
            print("frame rate =", frame_rate)
    animate(group)
    clock.tick(30)      <-- Clock.tick теперь управляет частотой кадров (ограниченной скоростью компьютера)
pygame.quit()
```

Это основы использования спрайтов и Pygame. В следующей главе мы напишем клевую игру с помощью Pygame и узнаем, что еще можно делать, например о добавлении текста (для ведения счета), звука, ввода с мыши или клавиатуры.

Что ты узнал?

В этой главе ты узнал:

- о спрайтах в Pygame и их использовании в управлении множественными движущимися объектами;
- о группах спрайтов;
- об обнаружении столкновений;
- о `pygame.clock` и частоте кадров.

Проверь свои знания

1. Что такое обнаружение столкновений на уровне прямоугольников?
2. Что такое обнаружение столкновений на уровне пикселей, и чем оно отличается от обнаружения столкновений на уровне прямоугольников?
3. Каковы два способа отслеживания нескольких спрайтов вместе?
4. Каковы два способа управления скоростью анимации в коде?
5. Почему использование `pygame.clock` дает более точный результат, чем `pygame.time.delay()`?
6. Как можно узнать, с какой частотой кадров выполняется твоя программа?

Попробуй самостоятельно

Если ты сам набирал код всех примеров в этой главе, то уже попробовал достаточно. Если нет, вернись обратно и все повтори. Мы обещаем, ты узнаешь нечто новое!

ГЛАВА 18

Новый вид ввода – события

До сих пор у нас были очень простые варианты ввода. Пользователь набирал строки кода на клавиатуре с помощью функции `input()` или мы получали числа и строки с помощью EasyGui (глава 6). Мы также показывали тебе, как закрывать окно Pygame с помощью мыши, но не объяснили, как это работает.

В этой главе ты узнаешь о другом способе ввода – *событиях*. Попутно ты узнаешь, что именно делает код для закрытия окна Pygame. Мы получим ввод с мыши и заставим программы реагировать на нажатие клавиш немедленно, не ожидая нажатия клавиши **Enter**.

События

Если бы мы спросили тебя «Что такое событие?» в реальной жизни, ты ответил бы: «Это то, что происходит». Хорошее определение, и оно также верно для программирования. Многие программы должны реагировать на «то, что происходит», например:

- движение или нажатие кнопки мыши;
- нажатие клавиш;
- истечение определенного промежутка времени.

Большинство программ, которые мы написали до сих пор, следовали по достаточно предсказуемому пути от начала до конца, иногда с некоторыми циклами

или условиями в середине. Но есть целый класс других программ, называемых *управляемыми событиями*, которые работают иначе. Программы, управляемые событиями, обычно ждут и ничего не делают, пока что-то – *событие* – не случится. Когда событие случается, они берутся за дело и делают все необходимое, чтобы *обработать* событие.

Хороший пример подобной программы – операционная система Windows (и любой другой графический пользовательский интерфейс). Если ты включишь компьютер под управлением Windows, он будет спокойно ждать действий после загрузки. Никакие программы не будут запущены, а указатель мыши не будет сам метаться по рабочему столу. Тем не менее, если ты начнешь им двигать или нажимать кнопки, начнут происходить события. Указатель переместится, откроется меню «Пуск» и т. д.

Цикл событий

Чтобы программа, управляемая событиями, «увидела» их, она должна их «искать». Программа должна постоянно сканировать ту часть памяти компьютера, которая раньше сигнализировала о событии. Она делает это снова и снова, пока программа выполняется. В главе 8 мы узнали, как программы повторяют свои действия, – с помощью *циклов*. Особый цикл, который ищет события, называется циклом события.

В программах Pygame, которые мы писали в последних двух главах, в конце всегда присутствовал цикл `while`. Мы говорили, что этот цикл выполняется, пока программа запущена. Этот цикл `while` – цикл события в Pygame. (Это первая часть загадки о коде для закрытия окна Pygame.)

Очередь событий

Все эти события происходят тогда, когда кто-то двигает или щелкает мышью либо нажимает клавишу. И куда эти события направляются? В предыдущей части мы сказали, что цикл событий постоянно сканирует часть памяти. Эта часть памяти, в которой хранятся события, называется *очередью событий*.

СЛОВАРИК

Слово *очередь* в повседневной жизни означает очередь людей. В программировании очередь обычно означает список вещей, которые поступили в определенном порядке или будут использованы в определенном порядке.

Очередь событий – список всех событий, которые случились, в порядке их происхождения.

Обработчики событий

Если ты пишешь программу с графическим интерфейсом или игру, она должна знать, когда пользователь нажимает клавишу или перемещает мышь. Эти нажатия и движения мыши – *события*, и программа должна знать, что с ними делать. Она должна их *обрабатывать*. Часть программы, которая обрабатывает определенный вид событий, называется *обработчиком событий*.

Но не каждое событие будет обработано. Когда ты перемещаешь мышь по столу, происходят сотни событий, потому что цикл событий выполняется очень быстро. Каждую долю секунды, даже если мышь сдвинулась совсем ненамного, генерируется новое событие. Но твоя программа может не обращать внимания на незначительное движение мыши. Она может реагировать, только если пользователь щелкнет по чему-либо. Итак, твоя программа может игнорировать события **`mouseMove`** и обращать внимание на события **`mouseClick`**.

Программы, управляемые событиями, содержат *обработчики* для тех *событий*, которые им нужны. Если в твоей игре используются клавиши со стрелками на клавиатуре для управления движением корабля, ты можешь написать обработчик для события **`keyDown`**. Если ты используешь мышь для управления кораблем, можешь написать обработчик для события **`mouseMove`**.

Сейчас мы взглянем на некоторые особые события, которые можно использовать в программах. Мы опять используем Pygame, поэтому все события, о которых мы будем говорить, будут поступать из очереди событий Pygame. Другие модули Python содержат другой набор событий, который можно использовать. Например, мы познакомимся с другим модулем, PyQt, в главе 20. У него есть свой набор событий, некоторые из которых отличаются от набора Pygame. Тем не менее способ обработки событий обычно одинаков для разных наборов (и даже для разных языков программирования). Он не одинаков для каждой системы событий, но и здесь больше сходств, чем различий.

События клавиатуры

Давай начнем с примера события клавиатуры. Допустим, нам нужно, чтобы что-то произошло при нажатии клавиши на клавиатуре. В Pygame этому событию соответствует событие **`KEYDOWN`**. Чтобы показать, как это работает, давай используем наш прыгающий мяч из листинга 16.15, в котором он двигается в стороны и отталкивается от краев окна. Но перед добавлением событий обновим программу с помощью наших новых знаний:

- используем спрайты;
- используем функцию **`clock.tick()`** вместо **`time.delay()`**.

Во-первых, нам нужен класс для мяча. Этот класс будет иметь метод **`__init__()`** и метод **`move()`**. Мы создадим экземпляр класса, основной цикл **`while`** и используем **`clock.tick(30)`**. Листинг 18.1 содержит код со всеми этими изменениями.

Листинг 18.1. Программа с прыгающим мячом с использованием спрайтов и **`Clock.tick()`**

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
background = pygame.Surface(screen.get_size())
background.fill([255, 255, 255])
clock = pygame.time.Clock()
```

```
class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        if self.rect.left <= screen.get_rect().left or \
                self.rect.right >= screen.get_rect().right:
            self.speed[0] = - self.speed[0]
        newpos = self.rect.move(self.speed)
        self.rect = newpos

my_ball = Ball('beach_ball.png', [10,0], [20, 20])
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:

    clock.tick(30)
    screen.blit(background, (0, 0))
    my_ball.move()
    screen.blit(my_ball.image, my_ball.rect)
    pygame.display.flip()
pygame.quit()
```

Класс **Ball**, *включая метод* **move()**

Создание экземпляра мяча

Скорость, расположение

Часы

Обновление всех элементов

Здесь нужно заметить, что мы сделали нечто иное, чтобы «стереть» мяч при его перемещении. Ты уже ты видел два способа «стирания» спрайтов перед их перерисовкой на новых местах: первый – залить фоновым цветом каждый спрайт на старом месте, второй – полностью залить фон каждого кадра, то есть начинать каждый раз с чистого экрана. В этом случае мы использовали второй способ. Но вместо использования функции **screen.fill()** каждый раз во время цикла мы создали поверхность **background** и залили ее белым цветом. Затем каждый раз во время цикла мы просто обновляли фон поверхности, **screen**. Это дает тот же результат; просто немного иначе.

События клавиш

Теперь добавим обработчик событий, который заставляет мяч двигаться *вверх* при нажатии клавиши ↑ и *вниз* при нажатии клавиши ↓. Pygame состоит из множества модулей. Модуль, который мы используем в этой главе, называется **pygame.event**.

У нас уже есть цикл событий Pygame (цикл **while**). Этот цикл ищет особое событие под названием **QUIT**.

```
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
```

Метод **`pygame.event.get()`** получает список всех событий в очереди событий. Цикл for повторяется для каждого события списка и если находит событие **`QUIT`**, то запускает функцию **`running = False`**, которая закрывает окно Pygame и завершает программу. Теперь ты знаешь все о том, как закрыть программу.

Для этого примера нам также нужен другой тип события. Нам нужно знать, когда нажимается клавиша, поэтому нам нужно событие **`KEYDOWN`**. Необходимо следующее:

```
if event.type == pygame.KEYDOWN
```

Поскольку у нас уже есть утверждение **`if`**, мы можем просто добавить еще одно условие **`elif`**, как мы научились в главе 7:

```
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            # сделать что-то
```

Новая часть, где мы определяем нажатие клавиши

Что именно мы хотим сделать при нажатии клавиши? Мы говорили, что при нажатии клавиши ↑ мы заставим мяч двигаться вверх, а при нажатии клавиши ↓ – двигаться вниз. Итак, мы можем сделать следующее:

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_UP:
                my_ball.rect.top = my_ball.rect.top - 10
            elif event.key == pygame.K_DOWN:
                my_ball.rect.top = my_ball.rect.top + 10
```

Двигает мяч на 10 пикселей вверх

Двигает мяч на 10 пикселей вниз

`K_UP` и **`K_DOWN`** – это имена Pygame для клавиш ↑ и ↓. Внеси эти изменения в листинг 18.1, и код программы должен выглядеть следующим образом:

Листинг 18.2. Отталкивание мяча с помощью клавиш

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
background = pygame.Surface(screen.get_size())
background.fill([255, 255, 255])
clock = pygame.time.Clock()

class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
```

Инициализация всех элементов

*Определение класса **`Ball`**, включая метод **`move()`***

```
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        if self.rect.left <= screen.get_rect().left or \
                self.rect.right >= screen.get_rect().right:
            self.speed[0] = - self.speed[0]
    newpos = self.rect.move(self.speed)
    self.rect = newpos

my_ball = Ball('beach_ball.png', [10,0], [20, 20])
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_UP:
                my_ball.rect.top = my_ball.rect.top - 10
            elif event.key == pygame.K_DOWN:
                my_ball.rect.top = my_ball.rect.top + 10

    clock.tick(30)
    screen.blit(background, (0, 0))
    my_ball.move()
    screen.blit(my_ball.image, my_ball.rect)
    pygame.display.flip()
pygame.quit()
```

Определение класса **Ball**, *включая метод* **move()**

Создание экземпляра мяча

Проверка нажатий клавиш и перемещение мяча

Обновление всех элементов

Запусти программу из листинга 18.2 и попробуй нажать клавишу ↑ и ↓. Работает?

Повтор нажатия клавиши

Ты мог заметить, что если удерживать клавишу ↑ или ↓ нажатой, мяч передвигается только на один шаг. Это происходит потому, что ты не указал в программе, что ей делать, если клавиша *удерживается*. Когда пользователь нажал клавишу, было сгенерировано единственное событие **KEYDOWN**, но в Pygame есть способ генерировать несколько событий **KEYDOWN**, если клавиша удерживается. Он известен как *повтор нажатия клавиш*. Ты указываешь, как долго ждать, перед тем как начать повтор, и как часто его производить. Значения указаны в миллисекундах (тысячных долях секунды). Выглядит это так:

```
delay = 100
interval = 50
pygame.key.set_repeat(delay, interval)
```

Значение **delay** сообщает Pygame, сколько ждать перед началом повтора, а значение **interval** – как быстро клавиша должна повторяться, другими словами – сколько времени проходит между каждым событием **KEYDOWN**.

Имена событий и имена клавиш

Когда мы обнаруживали событие нажатия клавиш ↑ и ↓, мы искали тип события `KEYDOWN` и имена клавиш `K_UP` и `K_DOWN`. А какие еще события нам доступны? Каковы имена других клавиш?

Их довольно много, поэтому я не буду приводить здесь полный список. Они указаны на сайте Pygame. Ты можешь найти список событий в документации Pygame:

www.pygame.org/docs/ref/event.html.

Список названий клавиш:

www.pygame.org/docs/ref/key.html.

Ниже приведены некоторые из популярных событий, которые будем использовать и мы:

- `QUIT`;
- `KEYDOWN`;
- `KEYUP`;
- `MOUSEMOTION`;
- `MOUSEBUTTONUP`;
- `MOUSEBUTTONDOWN`.

В Pygame также есть имена для каждой клавиши, которую можно нажать. Ты уже видел клавишу ↑ – `K_UP` и клавишу ↓ – `K_DOWN`. Ты увидишь некоторые другие имена клавиш, но все они начинаются с `K_`, за которой следует название клавиши, например:

- `K_a`, `K_b` (для букв);
- `K_SPACE`;
- `K_ESCAPE`

и так далее.

События мыши

Ты видел, как получить события с клавиатуры и использовать их для управления чем-либо в программе. Мы заставили мяч двигаться вверх и вниз с помощью соответствующих клавиш. Теперь используем мышь для управления мячом. Ты узнаешь, как обрабатывать события мыши и как использовать информацию о положении указателя мыши.

Три вида широко используемых событий мыши:

- `MOUSEBUTTONUP`;
- `MOUSEBUTTONDOWN`;
- `MOUSEMOTION`.

Самое простое, что мы можем сделать, – заставить мяч следовать за указателем мыши при его движении в окне Pygame. Чтобы двигать мяч, мы используем атрибут `rect.center`. Таким образом, центр мяча будет следовать за указателем мыши.

Мы заменим код, который определял события клавиш в цикле **while**, кодом для определения события мыши.

```
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEMOTION:
            my_ball.rect.center = event.pos
```

Определение перемещения мыши и мяча

Это даже проще, чем в примере с клавиатурой. Внеси это изменение в листинг 18.2 и попробуй запустить программу. Часть **even.pos** – это координаты (*x* и *y*) мыши. Мы лишь переносим центр мяча в эти координаты. Обрати внимание, что мяч следует за мышью до тех пор, пока мышь движется, то есть до тех пор, пока происходят события **MOUSEMOVE**. Изменение **rect.center** мяча изменило также координаты *x* и *y*. Мы уже не просто двигаем мяч вверх и вниз, мы двигаем его и в стороны. Когда события мыши не происходят – потому что мышь не двигается или потому что ее указатель находится за пределами окна Pygame, – мяч продолжает прыгать из стороны в сторону.

Теперь давай попробуем заставить нашу мышь влиять на движение мяча, только когда ее кнопка удерживается нажатием. Перемещение указателя мыши при удерживаемой нажатием кнопке мыши называется *перетаскиванием*. Нет такого события – «MOUSEDRAG», поэтому мы используем те, что есть, чтобы получить нужный эффект.

Как мы можем узнать, перетаскивает ли мышь что-то? Перетаскивание – это движение указателя мыши при нажатой ее кнопке. Мы можем узнать, когда кнопка нажимается, с помощью события **MOUSEBUTTONDOWN** и узнать, когда она отпущена, с помощью события **MOUSEBUTTONUP**. Затем нам просто нужно следить за состоянием кнопки. Мы можем сделать это с помощью переменной, которую назовем **held_down**. Это будет выглядеть так:

```
held_down = False
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            held_down = True
        elif event.type == pygame.MOUSEBUTTONUP:
            held_down = False
        elif event.type == pygame.MOUSEMOTION:
            if held_down:
                my_ball.rect.center = event.pos
```

Определение нажатия кнопки мыши

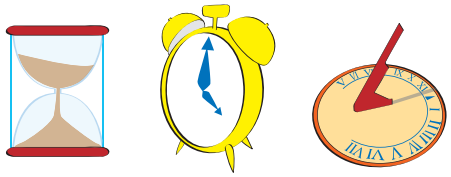
Выполняется при перемещении указателя мыши

Состояние перетаскивания (движение указателя мыши при удерживаемой нажатием кнопке) определяется в последнем блоке **elif** в предыдущем коде. Попробуй внести это изменение в цикл **while** в предыдущую измененную версию листинга 18.2. Запусти ее и посмотри, что получится.

Эй, мы программировали с самой первой главы! Но теперь, когда мы имеем дело с графикой, спрайтами и мышью, все становится интереснее. Мы говорили тебе, что до этого дойдем. Тебе сначала надо было выучить некоторые основы.

События таймера

В этой главе ты видел события клавиатуры и мыши. Другой тип событий, очень полезный, особенно в играх и симуляторах, – события *таймера*. Таймер генерирует событие через равные промежутки времени, как твой будильник. Если ты включишь его, он будет звонить в одно и то же время каждый день.



Таймеры Pygame можно настроить на любой временной интервал. Когда время истекает, происходит событие из цикла событий. Какое событие он генерирует? Этот тип событий называется *пользовательскими событиями*.

В Pygame есть некоторое количество уже определенных типов событий. Эти события пронумерованы, начиная с 0, и имеют имена для скорейшего запоминания. Мы уже встречали некоторые из них, например **MOUSEBUTTONDOWN** и **KEYDOWN**. Также в Pygame есть место для событий, *заданных пользователем*. Это события, которые в Pygame не зарезервированы для чего-то особенного, и ты можешь использовать их для своих нужд. Например, для таймеров.

Для настройки таймера в Pygame используется функция **set_timer()**:

```
pygame.time.set_timer(EVENT_NUMBER, interval)
```

EVENT_NUMBER – это номер события, а **interval** – это как часто (в миллисекундах) таймер срабатывает и создает событие.

Какой же номер события **EVENT_NUMBER** нам использовать? Нам нужен тот номер, который Pygame не использует уже для чего-то другого. Мы можем спросить у Pygame, какие номера уже заняты. Попробуй в интерактивном режиме:

```
>>> import pygame
>>> pygame.USEREVENT
24
```

Этот ответ говорит нам, что Pygame использует номера от 0 до 23 и первый доступный пользователю номер – 24. Итак, нам надо выбрать номер от 24 и выше. Насколько высоко мы можем продвинуться? Давай спросим Pygame еще раз:

```
>>> pygame.NUMEVENTS
32
```

NUMEVENTS сообщает нам о максимальном числе типов событий в Pygame – 32 (от 0 до 31). Поэтому нам надо выбрать номер от 24 до 31. Мы можем настроить таймер так:

```
pygame.time.set_timer(24, 1000)
```

Но если по какой-то причине значение **USEREVENT** изменится, код может не сработать. Лучше сделать так:

```
pygame.time.set_timer(pygame.USEREVENT, 1000)
```

Если бы нам нужно было установить следующее пользовательское событие, мы могли бы использовать **USEREVENT + 1** и т. д. Число **1000** в этом примере означает 1000 миллисекунд, то есть 1 секунду, поэтому этот таймер будет срабатывать каждую секунду. Давай вставим это в нашу программу с мячом.

Как и ранее, мы используем событие для движения мяча вверх и вниз, но поскольку мяч не будет контролироваться пользователем в этот раз, мы заставим его отталкиваться от всех краев окна. Код всей программы на основе измененного листинга 18.2 приведен в листинге 18.3.

Листинг 18.3. Использование события таймера для движения мяча

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
background = pygame.Surface(screen.get_size())
background.fill([255, 255, 255])

clock = pygame.time.Clock()
class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed
    def move(self):
        if self.rect.left <= screen.get_rect().left or \
                self.rect.right >= screen.get_rect().right:
            self.speed[0] = - self.speed[0]
        newpos = self.rect.move(self.speed)
        self.rect = newpos
```

Инициализация всех элементов

Определение класса **Ball**

1 *Эта строка продолжается*

```
my_ball = Ball('beach_ball.png', [10,0], [20, 20])
pygame.time.set_timer(pygame.USEREVENT, 1000)
direction = 1
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.USEREVENT:
            my_ball.rect.centery = my_ball.rect.centery + (30*direction)
            if my_ball.rect.top <= 0 or \
            my_ball.rect.bottom >= screen.get_rect().bottom:
                direction = -direction
    clock.tick(30)
    screen.blit(background, (0, 0))
    my_ball.move()
    screen.blit(my_ball.image, my_ball.rect)
    pygame.display.flip()
pygame.quit()
```

Создание экземпляра **Ball**

Создание таймера 1000 мс = 1 с

Обработчик событий для таймера

1 *Эта строка продолжается*

Обновление всех элементов

Напомним, что символ \ – это символ продолжения строки 1. Ты можешь использовать его для написания кода в две строки, выполняемого как одна строка. (Не допускай пробелов после этого символа, иначе перенос строки не сработает.)

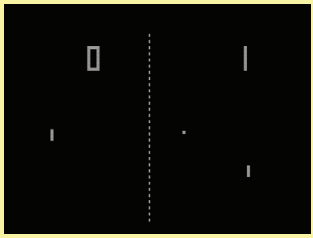
Если ты сохранишь и выполнишь программу из листинга 18.3, то увидишь мяч, двигающийся туда-сюда и вверх-вниз (на 30 пикселей) (каждую секунду). Это движение происходит из события таймера.

Еще одна игра – PyPong

В этом разделе мы совместим наши знания – включая спрайты, обнаружение столкновений и события, – чтобы создать простую игру с мячом и ракеткой, похожую на понг.

В СТАРЫЕ ДОБРЫЕ ВРЕМЕНА

Понг была одной из первых видеоигр, в которые люди играли дома. Оригинальная игра понг содержала только набор операций! Это было еще до появления домашних компьютеров. Она подключалась к телевизору, а управлять «ракетками» нужно было с помощью ручек. Ниже представлено, как выглядела игра на экране:



Небольшой интересный факт:
Бабушка не только была мастером игры понг, но и стала чемпионом мира по пинг-понгу!

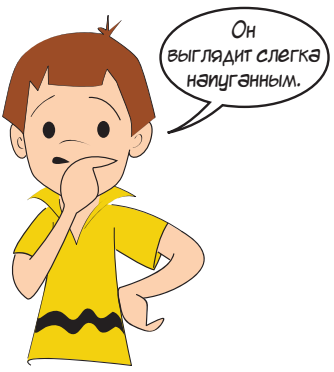
Мы начнем с простой версии для одного игрока. Нашей игре понадобится:

- мяч;
- ракетка для ударов по мячу;
- способ управления ракеткой;
- способ подсчета очков и их вывода на экран;
- способ отслеживания «жизней» – сколько попыток ты получаешь.

Мы выполним эти задачи одну за другой при создании программы.

Мяч

Наш волейбольный мяч слишком велик для такой игры. Нужно что-то поменьше. Мы с Картером остановились на этом безумном теннисном мячике:



Эй, ты бы тоже испугался, если бы тебя собирались гонять ракеткой!

Мы используем спрайты для этой игры, поэтому нам нужно создать спрайт для мяча, а затем и его экземпляр. Используем класс **`Ball`** с методами **`__init__()`** и **`move()`**.

```
class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > width:
            self.speed[0] = -self.speed[0]

        if self.rect.top <= 0 :
            self.speed[1] = -self.speed[1]
```

Отталкивается от сторон окна

Отталкивается от верхнего края окна

Когда мы создадим экземпляр мяча, то укажем ему, какое изображение использовать, его скорость и его начальное расположение:

```
myBall = Ball('wackyball.bmp', ball_speed, [50, 50])
```

Нам также нужно добавить мяч в группу, чтобы мы могли отслеживать столкновения мячика и ракетки. Мы можем создать группу и добавить в нее мяч одновременно:

```
ballGroup = pygame.sprite.Group(myBall)
```

Ракетка

Для ракетки мы последуем старой традиции понга и используем простой прямоугольник. Фон сделаем белым, а прямоугольник ракетки черным. Создадим класс спрайта и экземпляр для ракетки:

```
class Paddle(pygame.sprite.Sprite):
    def __init__(self, location):
        pygame.sprite.Sprite.__init__(self)
        image_surface = pygame.surface.Surface([100, 20])
        image_surface.fill([0,0,0])
        self.image = image_surface.convert()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location

paddle = Paddle([270, 400])
```

Создаем поверхность для ракетки

Заполняем ее черным цветом

Конвертируем поверхность в изображение

Обрати внимание, что для ракетки мы не загружали изображение из файла; мы создали его посредством заливки поверхности прямоугольника черным цветом. Но каждому спрайту нужен атрибут **`image`**, поэтому использовали метод **`Surface.convert()`** для конвертации поверхности в изображение.

Ракетка может перемещаться только вправо или влево, но не вверх-вниз. Мы сделаем положение ракетки по оси *x* (лево-право) зависящим от движений мыши, чтобы пользователь управлял ракеткой с ее помощью. Поскольку мы сделаем это в самом цикле событий, нам не нужен отдельный метод **`move()`** для ракетки.

Управление ракеткой

Как я сказал ранее, мы будем управлять ракеткой с помощью мыши. Мы используем событие **`MOUSEMOTION`**, что значит, что ракетка будет двигаться тогда, когда указатель мыши будет двигаться в окне Pygame. Поскольку Pygame «видит» мышь, когда ее указатель находится в его окне, ракетка будет автоматически ограничена краями окна. Мы сделаем так, чтобы центральная точка ракетки следовала за указателем.

Код должен выглядеть так:

```
elif event.type == pygame.MOUSEMOTION:
    paddle.rect.centerx = event.pos[0]
```

`event.pos` – это список положений мыши со значениями [*x*, *y*]. Итак, **`event.pos[0]`** показывает нам *x*-местоположение мыши всякий раз, когда она переме-

щается. Конечно, если мышь находится на левом или правом краю, ракетка будет выходить за пределы окна, но это нормально.

Последнее, что нам нужно, – это обнаружение столкновения мяча с ракеткой. Так мы «ударяем» по мячу ракеткой. Когда происходит столкновение, мы просто меняем *y*-скорость мяча (поэтому когда он падает и ударяется о ракетку, то отскакивает и начинает подниматься). Код выглядит следующим образом:

```
if pygame.sprite.spritecollide(paddle, ballGroup, False):
    myBall.speed[1] = -myBall.speed[1]
```

Нам также нужно не забыть рисовать элементы заново при каждом выполнении цикла. Если мы сложим все это вместе, получим очень примитивную программу понг. В листинге 18.4 приведен весь код программы (пока что).

Листинг 18.4. Первая версия PyPong

```
import pygame, sys
from pygame.locals import *

class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > screen.get_width():
            self.speed[0] = -self.speed[0]

        if self.rect.top <= 0 :
            self.speed[1] = -self.speed[1]

class Paddle(pygame.sprite.Sprite):
    def __init__(self, location = [0,0]):
        pygame.sprite.Sprite.__init__(self)
        image_surface = pygame.surface.Surface([100, 20])
        image_surface.fill([0,0,0])
        self.image = image_surface.convert()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location

pygame.init()
screen = pygame.display.set_mode([640,480])
clock = pygame.time.Clock()
ball_speed = [10, 5]
myBall = Ball('wackyball.bmp', ball_speed, [50, 50])
ballGroup = pygame.sprite.Group(myBall)
paddle = Paddle([270, 400])
```

Определение класса мяча

Перемещение мяча (отталкивание от верхнего и боковых краев экрана)

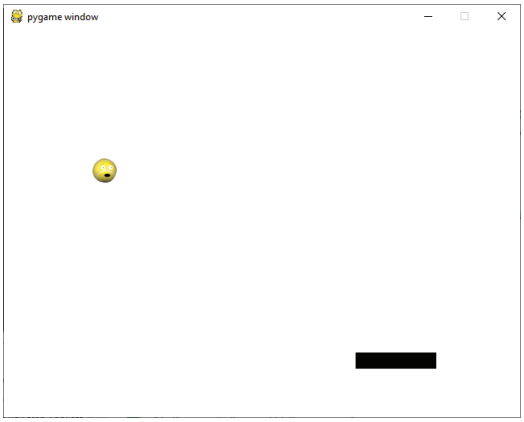
Определение класса ракетки

Инициализация Pygame, часов, мяча и ракетки

```
running = True
while running:                                  Начало основного цикла while
    clock.tick(30)
    screen.fill([255, 255, 255])

    for event in pygame.event.get():
        if event.type == QUIT:
            running = False
        elif event.type == pygame.MOUSEMOTION:          Перемещение ракетки
            paddle.rect.centerx = event.pos[0]          при движении мышью

    if pygame.sprite.spritecollide(paddle, ballGroup, False):   Проверка
        myBall.speed[1] = -myBall.speed[1]                      на попадание
    myBall.move()                                               ракеткой
    screen.blit(myBall.image, myBall.rect)                      по мячу
    screen.blit(paddle.image, paddle.rect)      Обновление
    pygame.display.flip()                       всех элементов
pygame.quit()
```

Перемещение мяча

Ниже представлено, как должна выглядеть программа при запуске:



Хорошо, это не самая захватывающая игра, но мы только начали писать игры с помощью Pygame. Давай добавим еще кое-что.

Подсчет очков и их вывод с помощью `pygame.font`

Нам нужно следить за двумя вещами: количеством жизней и количеством очков. Чтобы все было просто, мы присваиваем игроку одно очко каждый раз, когда мяч ударяется о верхний край окна. Игроку дается три жизни.

Нам также нужно показывать очки. Pygame использует модуль **`font`** для вывода текста. Он используется так:

- создай объект **`font`**, который указывает стиль шрифта и размер;
- *отобразите* текст, передав строку объекту шрифта, который возвращает новую *поверхность* с текстом на ней;
- перенесите эту поверхность на поверхность экрана.

СЛОВАРИК

В компьютерной графике рисование или отображение чего-либо называется *рендерингом*.

Строка в нашем случае будет количеством очков (но нам нужно конвертировать его из **`int`** в **`string`**).

Нам нужен следующий код перед циклом событий (после строки **`paddle = Paddle([270, 400])`**) в листинге 18.4:

```
score_font = pygame.font.Font(None, 50)      <-- Создание объекта шрифта
score_surf = score_font.render(str(score), 1, (0, 0, 0))   <-- Настройка расположения текста
score_pos = [10, 10]
                  Выводит текст на поверхность score_surf
```

`None` в первой строке – это место, в котором мы говорим Pygame о стиле шрифта. Словом **`None`** мы указываем на использование шрифта по умолчанию.

Затем внутри цикла событий нам нужно что-то подобное:

```
screen.blit(score_surf, score_pos)   <-- Отображение текста с количеством очков
```

Этот код будет перерисовывать текст с количеством очков каждый раз в цикле.

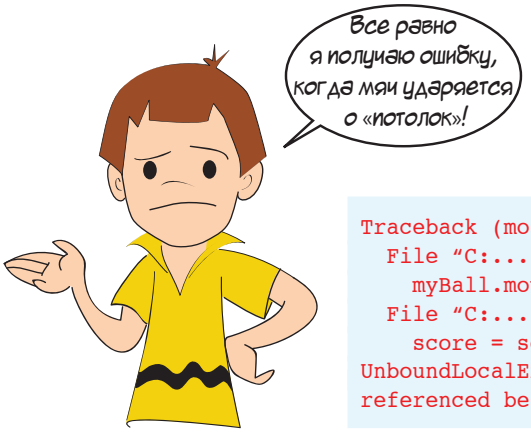
Конечно, Картер, мы же не создали еще переменную **`points`**. (И как раз к этому подводили.) Добавь эту строку перед кодом, который создает объект **`font`**:

```
score = 0
```

Теперь перейдем к отслеживаю количества очков... Мы уже определили с помощью функции **`move()`**, когда мячик ударяется о верхний край окна (чтобы оттолкнуться). Нам нужно добавить всего пару строк:

```
if self.rect.top <= 0 :
    self.speed[1] = -self.speed[1]
    score = score + 1
    score_surf = score_font.render(str(score), 1, (0, 0, 0))
```

Две новые строки




```
Traceback (most recent call last):
  File "C:...", line 59, in <module>
    myBall.move()
  File "C:...", line 24, in move
    score = score + 1
UnboundLocalError: local variable 'score'
referenced before assignment
```

Ой! Мы забыли об *области видимости*. Помнишь мое длинное объяснение в главе 15? Теперь ты видишь реальный пример. Хотя у нас есть переменная **`points`**, мы пытались использовать ее в методе **`move()`** класса **`Ball`**. Класс ищет *локальную* переменную **`score`**, которая не существует. Вместо этого мы хотим использовать уже созданную *глобальную* переменную, поэтому нам просто нужно указать методу **`move()`** использовать глобальную переменную **`score`**:

```
def move(self):
    global score
```

Нам также нужно сделать переменные **`score_font`** (объект для оценки) и **`score_surf`** (поверхность, содержащая визуализированный текст) глобальными, потому что они обновляются с помощью функции **`move()`**. Итак, код должен выглядеть следующим образом:

```
def move(self):
    global score, score_font, score_surf
```

Теперь все должно заработать! Попробуй – и увидишь. Теперь ты увидишь игровой счет в верхнем левом углу окна, и он должен увеличиваться при ударе мяча о верхний край окна.

Учет жизней

Теперь давай проследим за жизнями. Сейчас при пропуске мяча он просто выпадает из окна и никогда не возвращается. Мы хотим предоставить игроку три жизни или попытки, поэтому давай создадим переменную **lives** и присвоим ей значение 3.

```
lives = 3
```

После того как игрок пропускает мячик и тот выпадает из окна, мы вычитаем одну жизнь, ждем пару секунд и начинаем снова с новым мячом:

```
if myBall.rect.top >= screen.get_rect().bottom:
    lives = lives - 1
    pygame.time.delay(2000)
    myBall.rect.topleft = [50, 50]
```

Этот код выполняется внутри цикла **while**. Кстати, причина, по которой мы указали **myBall.rect** для мяча и **get_rect()** для экрана, в следующем:

- **myBall** – это спрайт, а в спрайтах уже включен **rect**;
- **screen** – это поверхность, а поверхность не содержит уже включенного **rect**. Ты можешь найти **rect**, который окружает поверхность с помощью функции **get_rect()**.

Если ты внесешь все эти изменения и запустишь программу, то увидишь, что у игрока теперь три жизни.

Добавление счетчика жизней

Большинство игр, которые предоставляют игроку определенное количество жизней, содержат способ вывода оставшегося их количества. Мы можем сделать то же самое в своей игре.

Легкий способ – показать количество мячей, соответствующее количеству оставшихся жизней. Мы можем поместить эту информацию в верхнем правом углу. Ниже приведена небольшая формула в цикле **for**, которая сформирует счетчик жизней:

```
for i in range (lives):
    width = screen.get_rect().width
    screen.blit(myBall.image, [width - 40 * i, 20])
```

Этот код тоже нужно поместить внутрь основного цикла **while** перед циклом событий (после строки **screen.blit(score_text, textpos)**).

Игра окончена

И последнее, нам нужно добавить сообщение «Игра окончена», когда игрок использует последнюю жизнь. Мы создадим пару объектов шрифтов, которые вклю-

чают наше сообщение и общее количество очков игрока, изобразим их (создадим поверхности с текстом) и перенесем поверхности на экран.

Нам также нужно остановить появление мяча после последней жизни. Для этого мы создадим переменную **done**, которая будет сообщать нам о конце игры. Следующий код выполняет все эти задачи и должен находиться внутри основного цикла **while**.

```
if myBall.rect.top >= screen.get_rect().bottom:
    lives = lives - 1
    if lives == 0:
        final_text1 = "Игра окончена"
        final_text2 = "Ты набрал: " + str(score)
        ft1_font = pygame.font.Font(None, 70)
        ft1_surf = ft1_font.render(final_text1, 1, (0, 0, 0))
        ft2_font = pygame.font.Font(None, 50)
        ft2_surf = ft2_font.render(final_text2, 1, (0, 0, 0))
        screen.blit(ft1_surf, [screen.get_width()//2 - \
                    ft1_surf.get_width()//2, 100])
        screen.blit(ft2_surf, [screen.get_width()//2 - \
                    ft2_surf.get_width()//2, 200])
        pygame.display.flip()
        done = True
    else:
        pygame.time.delay(2000)
        myBall.rect.topleft = [(screen.get_rect().width) - 40*lives, 20]
```

Вычитание жизни при столкновении мяча с нижним краем окна

Помещает текст в окне по центру

Символы продолжения строки

Подожди две секунды, затем возьми другой мяч

Если сложить все вместе, полный код программы будет выглядеть следующим образом:

Листинг 18.5. Финальный вариант игры PyPong

```
import pygame, sys

class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        global score, score_surf, score_font
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > screen.get_width():
            self.speed[0] = -self.speed[0]

        if self.rect.top <= 0 :
            self.speed[1] = -self.speed[1]
            score = score + 1
            score_surf = score_font.render(str(score), 1, (0, 0, 0))
```

Определение класса мяча

```
class Paddle(pygame.sprite.Sprite):
    def __init__(self, location = [0,0]):
        pygame.sprite.Sprite.__init__(self)
        image_surface = pygame.surface.Surface([100, 20])
        image_surface.fill([0,0,0])
        self.image = image_surface.convert()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location

pygame.init()
screen = pygame.display.set_mode([640,480])
clock = pygame.time.Clock()
myBall = Ball('wackyball.bmp', [10,5], [50, 50])
ballGroup = pygame.sprite.Group(myBall)
paddle = Paddle([270, 400])
lives = 3
score = 0
score_font = pygame.font.Font(None, 50)
score_surf = score_font.render(str(score), 1, (0, 0, 0))
score_pos = [10, 10]
done = False
running = True
while running:
    clock.tick(30)
    screen.fill([255, 255, 255])
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEMOTION:
            paddle.rect.centerx = event.pos[0]
    if pygame.sprite.spritecollide(paddle, ballGroup, False):
        myBall.speed[1] = -myBall.speed[1]
    myBall.move()
    if not done:
        screen.blit(myBall.image, myBall.rect)
        screen.blit(paddle.image, paddle.rect)
        screen.blit(score_surf, score_pos)
        for i in range (lives):
            width = screen.get_width()
            screen.blit(myBall.image, [width - 40 * i, 20])
        pygame.display.flip()
    if myBall.rect.top >= screen.get_rect().bottom:
        lives = lives - 1
        if lives == 0:
            final_text1 = "Игра завершена"
            final_text2 = "Ты набрал: " + str(score)
            ft1_font = pygame.font.Font(None, 70)
            ft1_surf = ft1_font.render(final_text1, 1, (0, 0, 0))
            ft2_font = pygame.font.Font(None, 50)
            ft2_surf = ft2_font.render(final_text2, 1, (0, 0, 0))
            screen.blit(ft1_surf, [screen.get_width()//2 - \
```

Определение класса ракетки

Инициализация всех элементов

Создание объекта шрифта

Начало основной программы (цикл **while***)*

Обнаружение движения мыши для передвижения ракетки

Обнаружение столкновений мяча и ракетки

Перемещение мяча

Обновление всех элементов

Уменьшение счетчика жизней при ударе мяча о нижний край экрана

Создание и вывод финального текста со счетом

```
                        ft1_surf.get_width()//2, 100])
            screen.blit(ft2_surf, [screen.get_width()//2 - \
                        ft2_surf.get_width()//2, 200])
            pygame.display.flip()
            done = True
        else:
            pygame.time.delay(2000)
            myBall.rect.topleft = [50, 50]
pygame.quit()
```

Создание и вывод финального текста со счетом

Начало новой жизни после 2-секундной задержки

Если ты выполнишь код из листинга 18.5, то увидишь следующее:

Обрати внимание, в программе около 75 строк кода (плюс пустые строки). Это наша самая большая программа на данный момент, но в ней и происходит многое, хотя выглядит все довольно просто при запуске.

В следующей главе ты узнаешь о звуках в Pygame и добавишь их в игру PyPong.

Что ты узнал?

В этой главе ты узнал:

- о событиях;
- о цикле событий в Pygame;
- об обработке событий;
- о событиях клавиатуры;
- о событиях мыши;
- о событиях таймера (и пользовательских событиях);
- о **`pygame.font`** (для добавления текста в программы Pygame);
- как собрать все это вместе и создать игру!

Проверь свои знания

1. Каковы два вида событий, на которые программа может ответить?
2. Как называется фрагмент кода, который имеет дело с событием?
3. Как называется тип события, который Pygame использует для определения нажатия клавиш?
4. Какой атрибут события `MOUSEMOVE` говорит, в каком месте окна находится указатель мыши?
5. Как узнать следующий доступный номер события в Pygame (например, если ты хочешь добавить пользовательское событие)?
6. Как создать таймер для генерации событий таймера в Pygame?
7. Какой тип объекта используется для вывода текста в окне Pygame?
8. Каковы три шага, необходимые для вывода текста в окне Pygame?

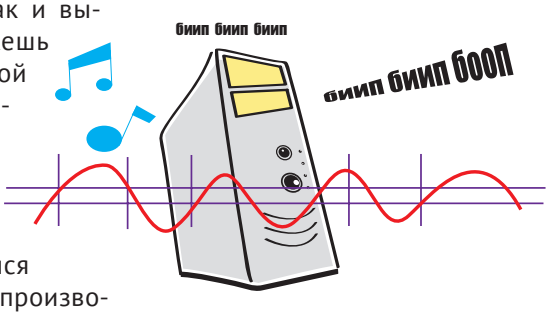
Попробуй самостоятельно

1. Ты не заметил ничего странного, когда мяч попадает по одной из сторон ракетки, а не по ее верхнему краю? Он как бы прыгает по ней до середины ракетки. Ты можешь понять, почему? Можешь исправить это? Попробуй решить задачу сам, прежде чем заглянешь в мои ответы.
2. Попробуй переписать программу (из листинга 18.4 или 18.5) так, чтобы появилась какая-то случайность в ударах мяча. Ты можешь изменить то, как он отскакивает от ракетки или стен, сделать скорость случайной или что-либо другое. (Мы встречали `random.randint()` и `random.random()` в главе 15, поэтому ты знаешь, как генерировать случайные числа, как целые, так и десятичные дроби.)

ГЛАВА 19

Звуковое сопровождение

В предыдущей главе ты написал свою первую игру с графикой, PyPong, используя свои знания о графике, спрайтах, столкновениях, анимации и событиях. В этой главе мы добавим звуковое сопровождение. Каждая видеоигра и множество других программ используют звуковое сопровождение для большего эффекта.



Звук может быть как вводом, так и выводом. В качестве ввода ты можешь подсоединить микрофон или другой источник к компьютеру, а программа будет записывать звук или выполнять другое действие с ним (отправлять в интернет, например). Но звук чаще используется в качестве вывода, и именно этим мы займемся в нашей книге. Ты узнаешь, как воспроизводить звук, например музыку или звуковые эффекты, и добавлять его в программы, такие как PyPong.

Модуль `pygame.mixer`

Звук – это еще один источник возможных сложностей, потому что, как и в случае с графикой, у разных компьютеров разные устройства и программное обеспече-

ние для воспроизведения звука. Чтобы упростить задачу, мы снова обратимся за помощью к Pygame.

У Pygame есть модуль для работы со звуком – **`pygame.mixer`**. В реальном, некомпьютерном мире устройство, которое принимает разные звуки и смешивает их вместе, называется *микшером*, и именно отсюда Pygame взял это имя.

Создание и воспроизведение звуков

Это два основных способа производства звуков для программы. Программа может генерировать или *синтезировать* звуки – создавать их с нуля с помощью звуковых волн разной высоты и громкости. Или программа может *воспроизвести* уже записанный звук. Это может быть музыкальная фраза на диске, файл в формате mp3 или другой вид музыкальных файлов.

В этой книге ты узнаешь только о воспроизведении звуков. Создание звуков с нуля – достаточно объемная тема, а в книге не так много места. Если тебя заинтересовало создание звуков с помощью компьютера, для этого написано много программ.

Воспроизведение звуков

Когда ты слушаешь музыку, ты воспроизводишь звуковой файл, находящийся на жестком диске (или на CD-диске, или в интернете), и превращаешь его в сигналы, которые можно услышать с помощью динамиков или наушников. Существует много разных форматов звуковых файлов, которые можно использовать на компьютере. Ниже приведены самые распространенные:

- *Wave-файлы* – имена файлов заканчиваются расширением *.wav*, например *hello.wav*;
- *Mp3-файлы* – названия файлов заканчиваются расширением *.mp3*, например *mySong.mp3*;
- *WMA-файлы (Windows Media Audio)* – заканчиваются расширением *.wma*, например *someSong.wma*;
- *Ogg Vorbis* – такие файлы заканчиваются расширением *.ogg*, например *yourSong.ogg*.

В примерах мы будем использовать файлы в форматах *.wav* и *.mp3*. Все они находятся в папке *\sounds* каталога с примерами.

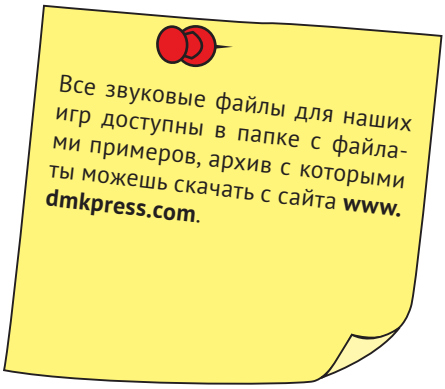
Есть два способа поместить звуковой файл в программу. Ты можешь скопировать его в ту же папку, где сохранена программа. Здесь Python будет искать файл, имя которого ты укажешь в программе:

```
sound_file = "my_sound.wav"
```

Если ты не скопировал звуковой файл в папку с программой, тебе нужно указать Python место дислокации файла, например:

```
sound_file = "C:/Program Files/HelloWorld/sounds/my_sound.wav"
```

Мы настаиваем, чтобы для наших примеров ты копировал звуковые файлы в папку с сохраненными программами. И настаиваем на этом потому, что где бы файл ни был указан в коде, ты увидишь только его имя, а не полное местонахождение. Если ты не скопируешь звуковые файлы в папку с программами, тебе нужно будет указывать путь к ним.



Использование модуля `pygame.mixer`

Как и при написании других программ в Pygame, нам нужно будет импортировать и инициализировать Pygame, прежде чем мы сможем воспроизводить звуки:

```
import pygame
pygame.init()
```

Теперь мы готовы воспроизводить звук. Ты будешь использовать два основных вида звукового сопровождения. Первый – звуковые эффекты. Они обычно короткие и хранятся в файлах формата *.wav*. Для файлов этого формата **`pygame.mixer`** использует объект **`Sound`**:

```
splat = pygame.mixer.Sound("splat.wav")
splat.play()
```

Другой вид звукового сопровождения – музыка. Музыка обычно хранится в файлах с расширением *.mp3*, *.wma* или *.ogg*. Чтобы их воспроизводить, Pygame использует модуль **`music`** в модуле **`mixer`**. Используется он так:

```
pygame.mixer.music.load(«bg_music.mp3»)
pygame.mixer.music.play()
```

Этот код воспроизводит музыку (или что там записано в файле) один раз и останавливается.

Давай попробуем воспроизвести звуки. Сначала звук всплеска.

Нам все еще нужен цикл **`while`**, для того чтобы программа Pygame продолжала выполняться. Также, хотя мы не собираемся ничего рисовать, программы Pygame чувствуют себя неуютно без своего собственного окна. Код приведен в листинге 19.1.

Листинг 19.1. Проба звуков в Pygame

```
import pygame, sys
pygame.init()                                   # Инициализация Pygame и mixer

screen = pygame.display.set_mode([640,480])     # Создание окна Pygame

splat = pygame.mixer.Sound("splat.wav")         # Создание объекта звука
splat.play()                                    # Воспроизведение звука

running = True
while running:                                  # Обычный цикл событий Pygame
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Попробуй выполнить программу.

Теперь давай попробуем воспроизвести музыку с помощью модуля **`mixer.music`**. Нам нужно изменить всего лишь пару строк в листинге 19.1. Новый код приведен в листинге 19.2.

Листинг 19.2. Воспроизведение музыки

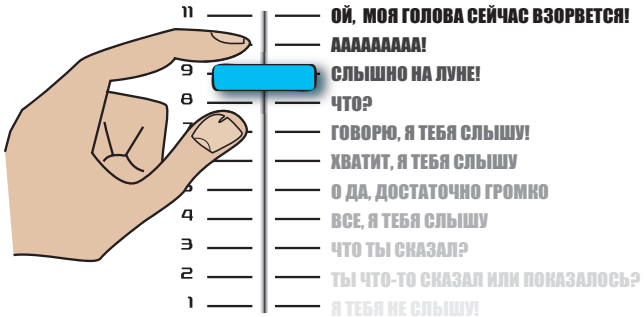
```
import pygame, sys
pygame.init()

screen = pygame.display.set_mode([640,480])

pygame.mixer.music.load("bg_music.mp3")         # Две строки
pygame.mixer.music.play()                       # с изменениями

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Попробуй и убедись, что слышишь музыку.

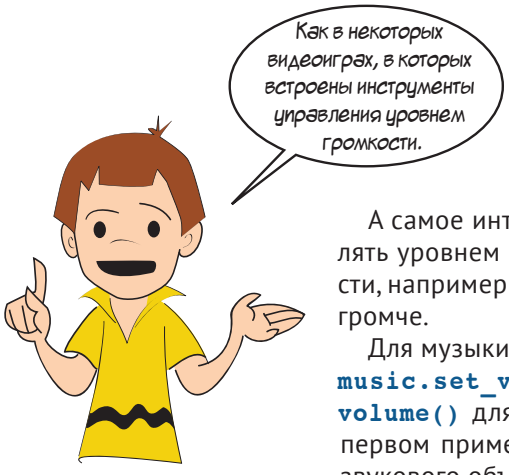
Не знаю, как тебе, а нам она показалась громкой. И пришлось сильно уменьшить уровень громкости динамиков. Рассмотрим далее, как управлять уровнем громкости звука в программе.

Управление уровнем громкости

Ты можешь управлять уровнем громкости звука на компьютере с помощью соответствующих регуляторов. В Windows это маленькое изображение динамика на панели задач. Этот регулятор управляет уровнем громкости всех звуков компьютера. У тебя также может быть ручка регулятора уровня громкости на самих динамиках.



Но еще мы можем контролировать уровень громкости звука, которую Pygame передает аудиосистеме компьютера.



А самое интересное – это то, что мы можем управлять уровнем громкости каждого звука по отдельности, например сделать музыку тише, а звук всплеска – громче.

Для музыки мы используем метод **`pygame.mixer.music.set_volume()`**. Для звука есть метод **`set_volume()`** для каждого звукового объекта. В нашем первом примере слово **`splat`** было именем нашего звукового объекта, поэтому здесь мы используем метод **`splat.set_volume()`**. Уровень громкости представлен десятичной дробью в диапазоне от 0 до 1, например значение **`0.5`** будет 50 % всей громкости.

Теперь давай попробуем поместить звуковые эффекты и музыку в одну программу. Как насчет проигрывания песни, а затем звука всплеска в конце? Мы также немного уменьшим уровень громкости звука. Установим уровень громкости музыки 30 %, а звуковых эффектов – 50 %. Код приведен в листинге 19.3.

Листинг 19.3. Музыка и звуковые эффекты с настройкой уровня громкости

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
pygame.mixer.music.load("bg_music.mp3")
pygame.mixer.music.set_volume(0.30)   <-- Настройка уровня громкости музыки
pygame.mixer.music.play()
splat = pygame.mixer.Sound("splat.wav")
splat.set_volume(0.50)   <-- Настройка уровня громкости звуковых эффектов
splat.play()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Попробуй выполнить код, чтобы увидеть, как он работает.

Картер заметил, что, начав воспроизведение музыки, программа переходит к следующему пункту, то есть проигрыванию звука всплеска. Причина в том, что музыка широко используется в качестве фона, и тебе не всегда нужно, чтобы программа проигрывала всю песню до конца перед следующим действием. В следующем разделе мы все исправим.

Воспроизведение фоновой музыки

Фоновая музыка предназначена для воспроизведения во время игры. Начав воспроизводить музыку, Pygame уже готов выполнять другие задачи, например перемещать спрайты или отслеживать ввод с мыши или клавиатуры. Он не ждет окончания воспроизведения музыки.

Но если тебе нужно знать, когда заканчивается песня? Может быть, тебе нужно начать воспроизведение другой песни и звукового эффекта (как в нашем случае). Как узнать, когда песня закончилась? У Pygame есть для этого способ: ты можешь спросить у модуля **`mixer.music`**, проигрывает ли он музыку до сих пор. Если да, воспроизведение музыки еще не закончилось. Если нет, то воспроизведение уже завершено. Давай попробуем.

Чтобы узнать, занят ли еще модуль воспроизведением музыки, используется функция **`get_busy()`** модуля **`mixer.music`**. Она возвращает значение **`True`**, если модуль еще занят, и значение **`False`**, если нет. В этот раз мы заставим программу воспроизвести песню, а затем звуковой эффект и закрыть окно программы автоматически. Из листинга 19.4 ты узнаешь, как именно это сделать.

Листинг 19.4. Ожидание окончания песни

```
import pygame, sys
pygame.init()

screen = pygame.display.set_mode([640,480])

pygame.mixer.music.load("bg_music.mp3")
pygame.mixer.music.set_volume(0.3)
pygame.mixer.music.play()
splat = pygame.mixer.Sound("splat.wav")
splat.set_volume(0.5)
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    if not pygame.mixer.music.get_busy():
        splat.play()
        pygame.time.delay(1000)
        running = False
pygame.quit()
```

Проверка завершения воспроизведения музыки

Ждем одну секунду для завершения воспроизведения всплеска

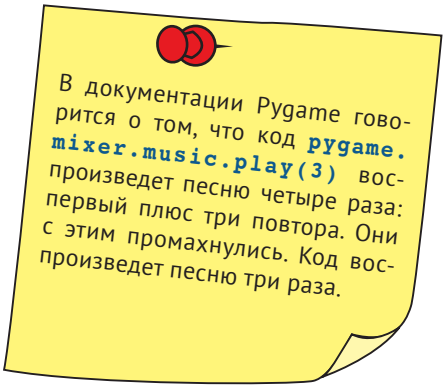
Этот код воспроизводит песню один раз, потом – звуковой эффект, и завершает программу.

Повтор музыки

Если мы собираемся использовать песню в качестве музыкального фона для игры, нам нужно, вероятно, чтобы она играла, пока запущена программа. Модуль **`music`** может сделать и это. Ты можешь *повторить* воспроизведение определенное число раз, например:

```
pygame.mixer.music.play(3)
```

Этот код воспроизведет песню три раза.



Ты можешь заставить песню звучать бесконечно, передав функции особое значение **-1**, например:

```
pygame.mixer.music.play(-1)
```

Этот код заставит песню играть бесконечно, пока запущена программа Pygame. (На самом деле это необязательно должно быть число –1. Любое отрицательное число подойдет.)

Добавление звукового сопровождения в PyPong

Теперь, когда мы усвоили базовые навыки воспроизведения звука, давай добавим его в игру PyPong.

Во-первых, добавим звук удара мячика о ракетку. Мы уже знаем, когда это происходит, потому что используем обнаружение столкновений для изменения направления мяча при ударе ракеткой. Вспомним код из листинга 18.5:

```
if pygame.sprite.spritecollide(paddle, ballGroup, False):
    myBall.speed[1] = -myBall.speed[1]
```

Теперь нам нужно добавить код для воспроизведения звука. Нам нужно добавить **`pygame.mixer.init()`** в начале программы и создать объект звука для использования:

```
hit = pygame.mixer.Sound("hit_paddle.wav")
```

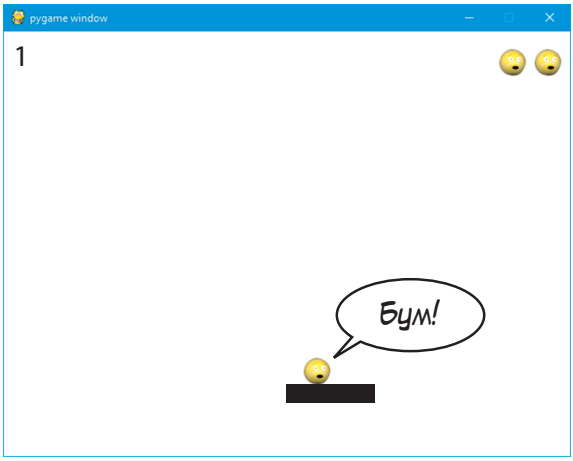
Также настроим уровень громкости звука, чтобы она не была слишком высокой:

```
hit.set_volume(0.4)
```

Затем, когда мяч ударяется о ракетку, мы воспроизводим звук:

```
if pygame.sprite.spritecollide(paddle, ballGroup, False):
    myBall.speed[1] = -myBall.speed[1]
    hit.play()   <-- Воспроизведение звука
```

Попробуй добавить все это в программу PyPong из листинга 18.5. Убедись, что ты скопировал файл *hit_paddle.wav* в папку с программой. Когда ты ее запустишь, ты будешь слышать звук каждый раз при ударе мяча ракеткой.



Больше странных звуков

Теперь у нас есть звук столкновения мяча с ракеткой, но давай добавим еще эффектов. Например, для этих событий:

- когда мяч ударяется о боковые стенки;
- когда мяч ударяется о верхний край и игрок зарабатывает очко;
- когда игрок пропускает мяч и он попадает на дно;
- когда начинается новая жизнь;
- когда игра заканчивается.

Сначала нам нужно создать звуковые объекты для всех этих событий. Ты можешь поместить код в любую позицию после **`pygame.init()`**, но перед циклом **`while`**:

```
hit_wall = pygame.mixer.Sound("hit_wall.wav")
hit_wall.set_volume(0.4)
get_point = pygame.mixer.Sound("get_point.wav")
get_point.set_volume(0.2)
splat = pygame.mixer.Sound("splat.wav")
splat.set_volume(0.6)
new_life = pygame.mixer.Sound("new_life.wav")
new_life.set_volume(0.5)
bye = pygame.mixer.Sound("game_over.wav")
bye.set_volume(0.6)
```

Мы подобрали уровни громкости, просто пробуя на слух, как будет лучше. Ты можешь настроить их на свой вкус. И напомним, что нужно скопировать все звуковые файлы в папку с программой. Все эти звуки находятся в папке *Примеры\sounds*.

Теперь нам нужно добавить методы **`play()`** там, где эти события происходят. Звук **`hit_wall`** должен воспроизводиться при ударе о стенки окна. Мы обнару-

живаем это в методе **`move()`** и меняем скорость мяча по оси *x* (чтобы он отскочил от стен). В оригинальном листинге 18.5 это строка 14:

```
if self.rect.left < 0 or self.rect.right > screen.get_width():
```

Итак, при изменении направления мы воспроизводим звук. Код выглядит так:

```
if self.rect.left < 0 or self.rect.right > screen.get_width():
    self.speed[0] = -self.speed[0]
    hit_wall.play() <-- Воспроизведение звука удара о стену
```

Мы можем сделать то же самое для звука **`get_point`**. Немного далее в методе мяча **`move()`** мы определяем столкновения мяча с потолком. Здесь мы его отталкиваем и причисляем игроку очко. И нам тоже нужно воспроизвести звук. Новый код выглядит так:

```
if self.rect.top <= 0 :
    self.speed[1] = -self.speed[1]
    points = points + 1
    score_text = font.render(str(points), 1, (0, 0, 0))
    get_point.play() <-- Воспроизведение звука получения очка
```

Внеси эти изменения и проверь, работают ли они.

Далее мы можем добавить код для воспроизведения звука при пропуске мяча игроком и потере жизни. Мы обнаруживаем это событие в основном цикле **`while`** в строке 63 листинга 18.5 (**`if myBall.rect.top >= screen.get_rect().bottom:`**). Нам нужно добавить новую строку:

```
if myBall.rect.top >= screen.get_rect().bottom:
    splat.play() <-- Воспроизведение звука пропуска мяча и потери жизни
    # жизнь теряется, если мяч ударяется о нижний край окна
    lives = lives - 1
```

Мы можем добавить звук при начале новой жизни. Это происходит в последних трех строках листинга 18.5 в блоке **`else`**. В этот раз дадим нашему звуковому эффекту немного времени для воспроизведения перед началом новой жизни:

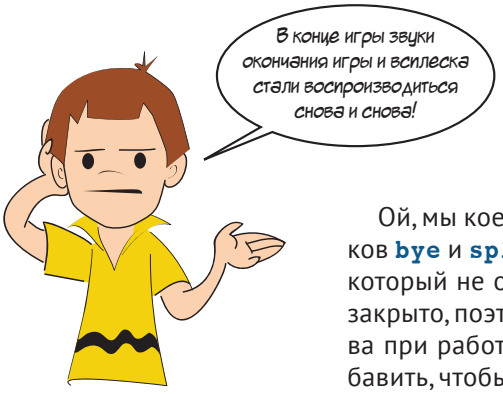
```
else:
    pygame.time.delay(1000)
    new_life.play()
    myBall.rect.topleft = [50, 50]
    screen.blit(myBall.image, myBall.rect)
    pygame.display.flip()
    pygame.time.delay(1000)
```

Вместо того чтобы ждать 2 секунды (как в оригинальной программе), мы ждем одну (1000 миллисекунд), воспроизводим звук и ждем еще секунду перед началом новой жизни. Попробуй – и увидишь, как это звучит.

Есть еще один эффект, который нужно добавить, и это окончание игры. Это происходит в строке 65 листинга 18.5 (**`if lives == 0:`**). Добавь строку для проигрывания звука **`bye`** следующим образом:

```
if lives == 0:
    bye.play()
```

Попробуй выполнить код и проверь, работает ли программа.



Ой, мы кое-что забыли. Код для проигрывания звуков **`bye`** и **`splat`** находится в основном цикле **`while`**, который не остановится, пока окно Pygame не будет закрыто, поэтому звуки воспроизводятся снова и снова при работающем цикле **`while`**! Нужно что-то добавить, чтобы звук воспроизводился только один раз.

Мы можем использовать переменную **`done`**, которая сообщает нам об окончании игры. Мы можем изменить код таким образом:

```
if myBall.rect.top >= screen.get_rect().bottom:
    if not done:
        splat.play()
    lives = lives - 1
    if lives == 0:
        if not done:
            bye.play()
```

Проверка того, что звук воспроизводится только один раз

Попробуй и убедись, что теперь все работает.

Хм... Возможно, об этом стоит задуматься. У нас есть переменная **`done`**, которая сообщает нам об окончании игры, и мы используем ее, чтобы знать, когда проигрывать наш звук **`bye`** и выводить финальное сообщение с количеством очков. Но что делает мяч?

Хотя мяч достиг нижнего края окна, он все еще двигается! Ничто не удерживает мяч от дальнейшего движения вниз, поэтому его значение по оси *y* продолжает возрастать. Он находится «под» экраном, и мы не можем его видеть, но можем *слышать*! Мяч все еще двигается и отталкивается от сторон окна, когда значение по оси *x* становится достаточно большим или достаточно маленьким. Это происходит в методе **`move()`**, и этот метод продолжает выполняться, пока выполняется цикл **`while`**.

Как нам это исправить? Есть несколько способов. Мы можем:

- остановить движение мяча, установив его скорость на **`[0,0]`**, когда игра окончена;
- проверить, находится ли мяч ниже края окна, и не воспроизводить звук **`hit_wall`**, если да;
- проверить переменную **`done`** и не воспроизводить звук **`hit_wall`**, если игра окончена.

Мы выбрали второй, но любой из способов будет работать. Оставляем выбор и изменение кода на твое усмотрение.

Добавление музыки в PyPong

Осталось сделать только одно – добавить музыку. Нам нужно загрузить музыкальный файл, настроить уровень громкости и начать воспроизведение. Нам нужно повторять файл, пока пользователь играет, поэтому мы установим особое значение **`-1`**:

```
pygame.mixer.music.load("bg_music.mp3")
pygame.mixer.music.set_volume(0.3)
pygame.mixer.music.play(-1)
```

Этот код можно поместить в любом месте до основного цикла **`while`**. Он начнет играть музыку. Теперь нам надо только остановить ее в конце, и для этого есть красивый способ.

В **`pygame.mixer.music`** есть метод **`fadeout()`**, который заставит музыку медленно затихать вместо резкого прерывания мелодии. Ты только указываешь, как долго должно происходить затихание:

```
pygame.mixer.music.fadeout(2000)
```

Это 2000 миллисекунд, или 2 секунды. Эту строку можно разместить в том же месте, где находится код **`done = True`**. (Не имеет значения, до или после.)

Теперь программа полностью дополнена музыкой и звуковыми эффектами. Попробуй ее – и узнаешь, как она звучит! В случае если ты не совсем разобрался, как выглядит код программы целиком, я привел финальную версию кода в листинге 19.5. Тебе нужно проверить, что файл *wackyball.bmp* и все аудиофайлы находятся в папке с программой.

Листинг 19.5. PyPong со звуковыми эффектами и музыкой

```
import pygame, sys

class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location = [0,0]):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        global points, score_text
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > screen.get_width():
            self.speed[0] = -self.speed[0]
            if self.rect.top < screen.get_height():
                hit_wall.play()

        if self.rect.top <= 0 :
            self.speed[1] = -self.speed[1]
            points = points + 1
            score_text = font.render(str(points), 1, (0, 0, 0))
            get_point.play()

class Paddle(pygame.sprite.Sprite):
    def __init__(self, location = [0,0]):
        pygame.sprite.Sprite.__init__(self)
        image_surface = pygame.surface.Surface([100, 20])
        image_surface.fill([0,0,0])
        self.image = image_surface.convert()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
pygame.init()
pygame.mixer.init()

pygame.mixer.music.load("bg_music.mp3")
pygame.mixer.music.set_volume(0.3)
pygame.mixer.music.play(-1)
hit = pygame.mixer.Sound("hit_paddle.wav")
hit.set_volume(0.4)
new_life = pygame.mixer.Sound("new_life.wav")
new_life.set_volume(0.5)
splat = pygame.mixer.Sound("splat.wav")
splat.set_volume(0.6)
hit_wall = pygame.mixer.Sound("hit_wall.wav")
hit_wall.set_volume(0.4)

get_point = pygame.mixer.Sound("get_point.wav")
get_point.set_volume(0.2)
bye = pygame.mixer.Sound("game_over.wav")
bye.set_volume(0.6)
```

Воспроизведение звука при ударе мяча о стороны экрана

Воспроизведение звука при ударе мяча о верхний край (игрок получает очко)

Инициализация модуля **sound** *в Pygame*

Загрузка звукового файла

Настройка уровня громкости музыки

Начало воспроизведения музыки, повторяется постоянно

Создание звуковых объектов, загрузка звуков, настройка уровня громкости для каждого

```
screen = pygame.display.set_mode([640,480])
clock = pygame.time.Clock()
myBall = Ball('wackyball.bmp', [12,6], [50, 50])
ballGroup = pygame.sprite.Group(myBall)
paddle = Paddle([270, 400])
lives = 3
points = 0

font = pygame.font.Font(None, 50)
score_text = font.render(str(points), 1, (0, 0, 0))
textpos = [10, 10]
done = False

running = True
while running:
    clock.tick(30)
    screen.fill([255, 255, 255])
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEMOTION:
            paddle.rect.centerx = event.pos[0]

    if pygame.sprite.spritecollide(paddle, ballGroup, False):
        hit.play()
        myBall.speed[1] = -myBall.speed[1]

    myBall.move()

    if not done:
        screen.blit(myBall.image, myBall.rect)
        screen.blit(paddle.image, paddle.rect)
        screen.blit(score_text, textpos)
        for i in range (lives):
            width = screen.get_width()
            screen.blit(myBall.image, [width - 40 * i, 20])
        pygame.display.flip()

    if myBall.rect.top >= screen.get_rect().bottom:
        if not done:
            splat.play()
        lives = lives - 1
        if lives <= 0:
            if not done:
                pygame.time.delay(1000)
                bye.play()
            final_text1 = "Игра окончена"
            final_text2 = "Ты набрал: " + str(points)
            ft1_font = pygame.font.Font(None, 70)
            ft1_surf = font.render(final_text1, 1, (0, 0, 0))
            ft2_font = pygame.font.Font(None, 50)
            ft2_surf = font.render(final_text2, 1, (0, 0, 0))
```

Воспроизведение звука при ударе ракеткой по мячу

Воспроизведение звука, когда игрок теряет жизнь

Ждем одну секунду, затем воспроизводим финальный звук

```
            screen.blit(ft1_surf, [screen.get_width()/2 - \
                        ft1_surf.get_width()/2, 100])
            screen.blit(ft2_surf, [screen.get_width()/2 - \
                        ft2_surf.get_width()/2, 200])
            pygame.display.flip()
            done = True
            pygame.mixer.music.fadeout(2000)  <-- Затухание музыки
        else:
            pygame.time.delay(1000)
            new_life.play()                   <-- Воспроизведение звука начала новой жизни
            myBall.rect.topleft = [50, 50]
            screen.blit(myBall.image, myBall.rect)
            pygame.display.flip()
            pygame.time.delay(1000)
pygame.quit()
```

Довольно длинная программа! (Около ста строк кода плюс пустые строки.) Ее можно сделать меньше, но так ее будет труднее читать и понимать. Мы потратили время на пошаговую разработку программы, поэтому тебе необязательно набирать ее всю за один раз.

Если ты следил за повествованием книги, то должен понимать, что делает каждая часть программы и как они соединяются между собой. И на всякий случай весь листинг находится в папке *Примеры* на твоем компьютере.

В следующей главе мы будем писать программы с кнопками, меню и т. д. – графическим интерфейсом пользователя (GUI).

Что ты узнал?

В этой главе ты узнал:

- как добавить звуковое сопровождение в программу;
- как воспроизводить звуковые файлы (файлы *.wav*);
- как воспроизводить музыку (*.mp3*);
- как узнать, когда воспроизведение звука закончилось;
- как управлять уровнем громкости звуковых эффектов и музыки;
- как сделать так, чтобы музыка повторялась снова и снова;
- как сделать так, чтобы музыка затихала постепенно.

Проверь свои знания

1. Каковы три вида файлов, которые используются для хранения звука?
2. Какой модуль Pygame используется для воспроизведения музыки?
3. Как настроить уровень громкости для звукового объекта Pygame?
4. Как настроить уровень громкости фоновой музыки?
5. Как заставить музыку потихоньку утихнуть?

Попробуй самостоятельно

Попробуй добавить звуки в программу по угадыванию чисел в главе 1. Так как она выполняется в текстовом режиме, тебе понадобится окно Pygame, как в примерах этой главы. Ты можешь использовать аудиофайлы из папки *Примеры\sounds* архива с примерами для этой книги:

- Ahoy.wav;
- TooLow.wav;
- TooHigh.wav;
- WhatYerGuess.wav;
- Avast.Gotlt.wav;
- NoMore.wav.

Или ты можешь записать собственные звуки. Тебе нужно использовать приложение типа Audacity (совместимое со многими операционными системами), доступное по адресу **www.audacityteam.org**.

ГЛАВА 20

Продолжение работы над графическими интерфейсами

В главе 6 мы создали несколько простых графических интерфейсов с помощью EasyGui: диалоговые окна (или *диалоги* для краткости). Но в GUI должно быть что-то еще, помимо диалогов. В большинстве современных программ вся программа выполняется в GUI. В этой главе ты узнаешь о создании GUI с помощью PyQt, который дает больше гибкости и контроля над внешним видом интерфейса.

PyQt – это модуль, который помогает создавать GUI. Сначала мы с его помощью сделаем новую версию программы для конвертации температур.

Работа с PyQt

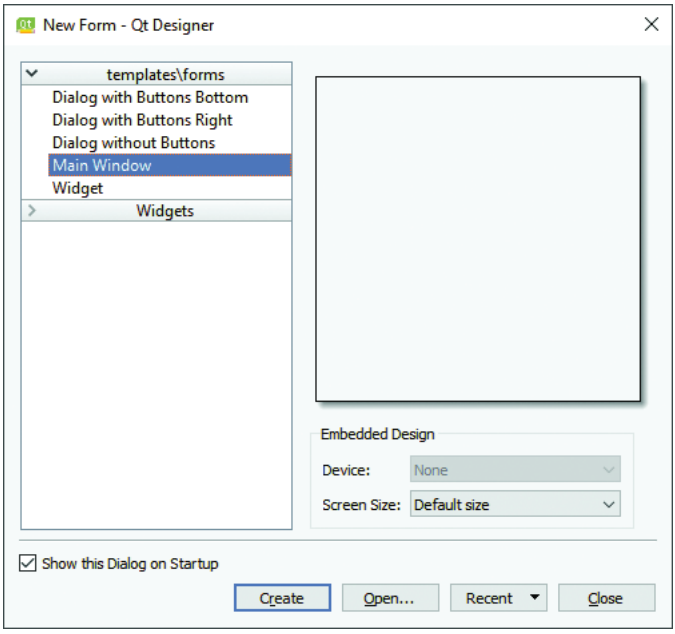
Перед использованием PyQt нам нужно убедиться, что он установлен на твоем компьютере. Если ты установил Python с помощью инсталлятора из архива с примерами, прилагаемого к этой книге, PyQt уже присутствует на твоем компьютере. В противном случае тебе нужно загрузить и установить его. Это можно сделать по адресу **www.riverbankcomputing.com/software/pyqt/download5**. Убедись, что ты скачал подходящую версию для своей операционной системы и версии Python (версия 3.7.3 в нашем случае). Мы используем версию PyQt 5.12.

Процесс разработки GUI программы состоит из двух основных частей. Во-первых, тебе нужно создать сам пользовательский интерфейс (UI), во-вторых, написать код, с помощью которого UI будет делать то, что тебе нужно. Создание пользовательского интерфейса включает в себя размещение в окне таких элементов, как кнопки, текстовые поля, поля выбора и т. д. Затем ты пишешь код, с помощью которого программа реагирует на действия пользователя: нажатие кнопки, введение информации в текстовое поле или выбор команды из предложенных.

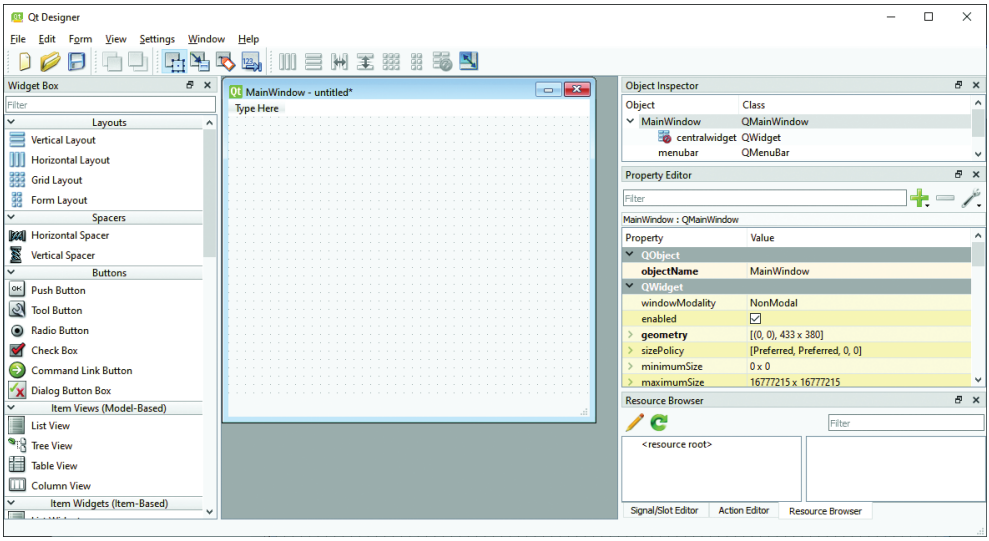
При использовании PyQt ты создаешь пользовательский интерфейс с помощью Qt Designer. Давай посмотрим, как это делается.

Qt Designer

Вместе с PyQt устанавливается программа под названием Qt Designer. Найди ее ярлык (например, в меню «Пуск» в Windows) и запусти. После этого ты увидишь открытое окно Qt Designer, а в центре – диалоговое окно **New Form**, предназначенное для создания формы.



Форма – это термин, обозначающий окно GUI. Поскольку ты собираешься создать новое окно GUI, выбери вариант **Main Window** и нажми кнопку **Create**. Теперь давай посмотрим на остальную часть окна Qt Designer.



Слева расположена панель **Widget Box**, где находятся различные графические элементы, которые ты можешь использовать. Они сгруппированы в несколько различных категорий.

Справа находятся панели **Object Inspector** и **Property Editor**. Именно здесь ты можешь изучить и изменить свойства виджетов. Существует также третья панель, функционал которой выбирается с помощью вкладок внизу. Это может быть **Signal/Slot Editor**, **Action Editor** или **Resource Browser**.

СЛОВАРИК

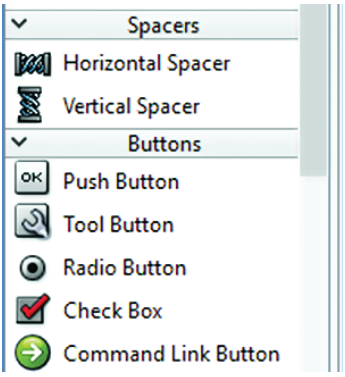
В GUI кнопки, флажки и т. д. называются *виджетами*. Их также называют компонентами, а иногда и элементами управления.

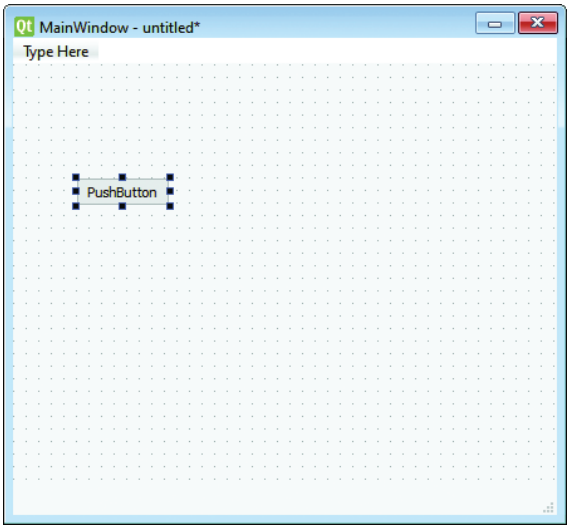
В середине находится новая, пустая форма, которую ты только что создал. В верхней ее части написано **MainWindow – untitled**, потому что ты еще не присвоил ей название. В этом пустом пространстве ты разместишь виджеты, чтобы создать свой пользовательский интерфейс. (В macOS, чтобы увидеть такой вид интерфейса программы, тебе нужно будет выполнить команду меню **Qt Designer** ⇨ **Preferences** и изменить режим пользовательского интерфейса с **Multiple Top-Level Windows** на **Docked Window**. В противном случае все панели будут находиться в отдельных окнах.)

Добавление кнопки

Давай добавим кнопку в GUI. В левой части окна Qt Designer найди раздел **Buttons**, а в нем – виджет **Push Button**.

Перетащи эту кнопку в любое место пустой формы. Теперь у тебя есть кнопка с ярлыком **PushButton**.



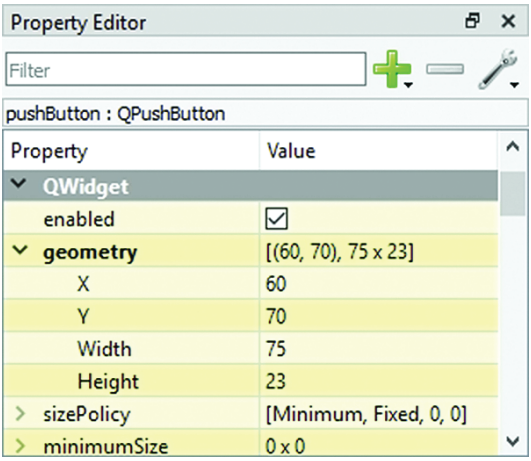


Посмотри вправо на панель **Property Editor**. Если кнопка все еще выделена (если вокруг нее есть маленькие синие квадратики), то ты увидишь ее свойства на панели **Property Editor**. Ты увидишь, что название кнопки – **PushButton**. Если ты прокрутишь список свойств, то увидишь такие пункты, как ширина и высота кнопки, ее положение относительно осей *x* и *y* и т. д.

Изменение кнопки

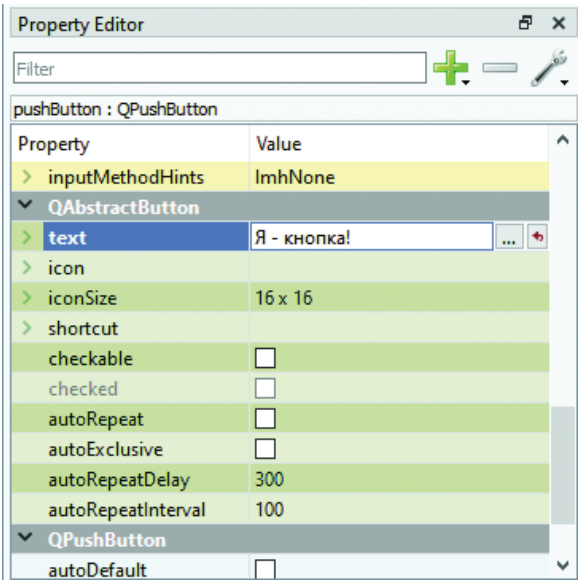
Есть два способа изменения размера и положения кнопки в окне: перетаскивание ее с помощью мыши или изменение соответствующих свойств в редакторе. Попробуй оба способа.

Чтобы переместить кнопку с помощью мыши, нажав и удерживая кнопку мыши в любом месте кнопки, перетащи ее в новую позицию. Чтобы изменить размер кнопки с помощью мыши, нажав и удерживая кнопку мыши на одном из синих квадратиков, находящихся вокруг кнопки (они называются *маркерами*), перетащи мышь, чтобы сделать ее больше или меньше. Чтобы изменить размер кнопки, используя ее свойства, щелкни по маленькому треугольнику рядом со свойством **geometry**; ты увидишь дополнительные свойства кнопки **X** (положение относительно оси *x*), **Y** (положение относительно оси *y*), **Width** (Ширина) и **Height** (Высота). Просто введи необходимые цифры, чтобы переместить кнопку или изменить ее размер.



Ты также можешь изменить текст, написанный на кнопке. Сейчас текст совпадает с названием кнопки, однако это можно изменить. Давай изменим его на «Я кнопка!».

Прокрути вниз свойства на панели **Property Editor**, пока не найдешь пункт **text**. Измени его значение на **Я кнопка!**. Ты также можешь изменить текст кнопки, дважды щелкнув по ней мышью и отредактировав текст непосредственно на кнопке.



Если ты посмотришь на кнопку на форме, то увидишь, что надпись на ней изменилась на «Я кнопка!». Но имя виджета (свойство **objectName**) – по-прежнему **PushButton**. Именно так следует указывать имя этой кнопки, если захочешь что-то с ней сделать.

Сохранение проекта GUI

Давай сохраним результат работы. В программах PyQt описание всех компонентов сохраняется в *файле .ui*. Он содержит всю информацию об окне, меню и виджетах. Это та же информация, которая отображалась в Qt Designer на панели свойств справа, и теперь нам нужно сохранить ее в файл для программы PyQt, чтобы использовать при запуске.

Чтобы сохранить файл в формате UI, выбери команду меню **File ⇨ Save As** и присвой файлу имя. Давай назовем наш интерфейс **MyFirstGui**. Ты заметишь, что расширение файла **.ui** уже введено. Таким образом, пользовательский интерфейс будет сохранен под именем *MyFirstGui.ui*. Убедись, что ты сохранил его в подходящей папке. По умолчанию Designer сохраняет документы в папке с собственными файлами, а это может быть не очень удобным. Перейди в ту папку, где сохраняешь свои программы Python, прежде чем нажать кнопку **Save**.

Ты можешь взглянуть на файл в любом текстовом редакторе, включая IDLE. Если открыть его, то увидишь следующее:

```
<?xml version=»1.0» encoding=»UTF-8»?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>576</width>
    <height>425</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralwidget">
   <widget class="QPushButton" name="pushButton">
    <property name="geometry">
     <rect>
      <x>60</x>
      <y>70</y>
      <width>121</width>
      <height>41</height>
     </rect>
    </property>
    <property name="toolTip">
     <string/>
    </property>
    <property name="text">
     <string>Я кнопка!</string>
    </property>
   </widget>
  </widget>
  <widget class="QMenuBar" name="menubar">
   <property name="geometry">
    <rect>
     <x>0</x>
     <y>0</y>
     <width>576</width>
     <height>21</height>
    </rect>
   </property>
  </widget>
  <widget class="QStatusBar" name="statusbar"/>
 </widget>
 <resources/>
 <connections/>
</ui>
```

Определение окна (фона)

Определение нашей кнопки

Выглядит несколько необычно, но если посмотришь внимательнее, то увидишь часть, которая описывает окно, и часть, которая описывает кнопку, а также некоторые другие части, которые мы еще не обсуждали, например меню и строку состояния.

Как назначить действие элементу GUI

Сейчас у нас есть интерфейс с минимумом базовых функций – окно с кнопкой, которое ничего не делает. Мы не написали код, который указывал бы программе, что ей делать, когда пользователь щелкает по кнопке. Похоже на машину с кузовом и колесами, но без мотора. Выглядит неплохо, но никуда не едет.

Нам нужно немного кода, чтобы запустить программу. Для простейшей программы PyQt потребуется следующее:

Листинг 20.1. Минимальный требуемый код PyQt

```
import sys
from PyQt5 import QtWidgets, uic

form_class = uic.loadUiType("MyFirstGui.ui")[0]

class MyFirstWindow(QtWidgets.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtWidgets.QMainWindow.__init__(self, parent)
        self.setupUi(self)

app = QtWidgets.QApplication(sys.argv)
myWindow = MyFirstWindow()
myWindow.show()
app.exec_()
```

Импортирование необходимых библиотек PyQt

Загрузка пользовательского интерфейса, созданного нами в Designer (1)

Определение класса главного окна

Объект PyQt, запускающий цикл событий

Создание экземпляра класса окна

Запуск программы и отображение окна GUI

Если тебе интересно, что означает `[0]` в конце строки (1), в примечании есть объяснение.

Причина постановки `[0]` в конце строки, которая загружает пользовательский интерфейс, заключается в том, что метод `uic.loadUiType()` возвращает список с двумя элементами: `form_class` и `base_class`. Для наших целей нам нужен только первый элемент, `form_class`, который в списке представлен как `item[0]`.

Как ты мог догадаться, все в PyQt – это объект. Каждое окно – объект, определенный с помощью ключевого слова **class**. Эта программа и все программы PyQt, которые мы создадим, имеют класс, который исходит из класса **QMainWindow**. В листинге 20.1 мы присвоили классу имя **My First Window** (в строке 6), но могли использовать любое имя. Напомним, что определение класса – это всего лишь проект. Нам все еще нужно построить дом – создать экземпляр класса, – и мы делаем это с помощью строки **myWindow = MyFirstWindow()** в нижней части. **myWindow** – это экземпляр класса **MyFirstWindow**.

Набери этот код в редакторе IDLE и сохрани под именем *MyFirstGui.py*.

- Основной код: *MyFirstGui.py*.
- Файл UI: *MyFirstGui.ui*.

Оба файла должны быть сохранены в одном месте, чтобы основная программа могла найти файл пользовательского интерфейса и загрузить его при запуске программы.

Теперь ты можешь запустить программу в IDLE. Ты увидишь окно с кнопкой, по которой можно щелкнуть. Но пока что ничего не происходит. Наша программа запущена, но мы не написали никакого действия для кнопки. Закрой программу, щелкнув по кнопке × в строке заголовка. Давай сделаем что-нибудь простое. Когда мы щелкаем по кнопке, пусть она перемещается в новое месторасположение в окне. Добавь код из строк 10–17 листинга 20.2 к коду из листинга 20.1.

Листинг 20.2. Добавление обработчика событий к кнопке

```
import sys
from PyQt5 import QtWidgets, uic

form_class = uic.loadUiType("MyFirstGui.ui")[0]

class MyFirstWindow(QtWidgets.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtWidgets.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.pushButton.clicked.connect(self.button_clicked)

    def button_clicked(self):
        x = self.pushButton.x()
        y = self.pushButton.y()
        x += 50
        y += 50
        self.pushButton.move(x, y)

app = QtWidgets.QApplication(sys.argv)
myWindow = MyFirstWindow()
myWindow.show()
app.exec_()
```

Соединение обработчика событий с событием

Обработчик событий

Добавь эти строки, чтобы кнопка перемещалась с каждым щелчком мыши

Перемещает кнопку, когда мы на нее нажимаем

Убедись, что ты выделил отступами весь блок **def** относительно утверждения **class**, как показано в листинге. Это нужно сделать, потому что все компоненты *на-*

ходятся внутри или *являются частью* окна. Поэтому обработчик событий кнопки должен быть внутри определения класса.

Попробуй запустить программу. В следующем разделе мы разберем код подробнее.

Возвращение обработчиков событий

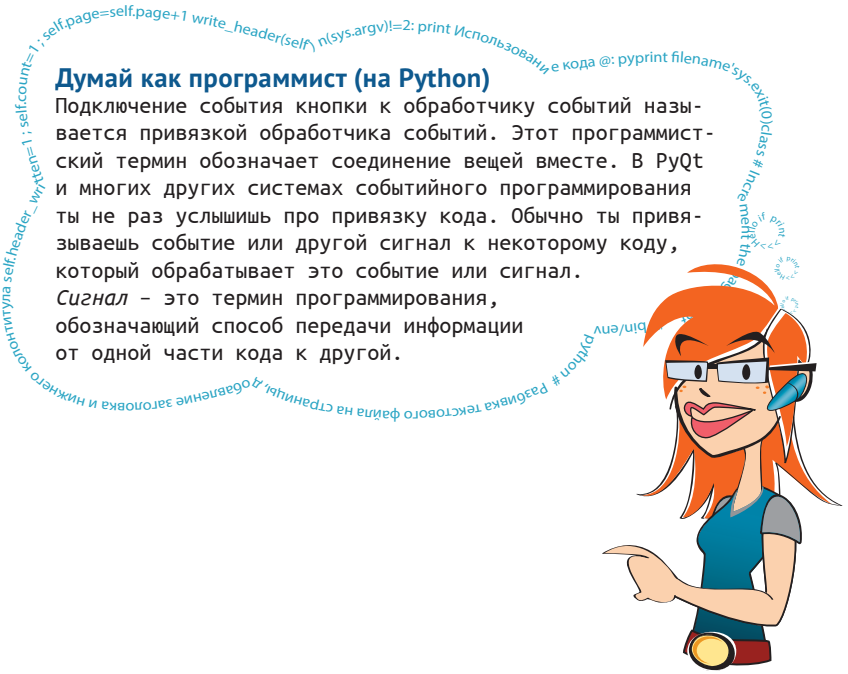
Ранее ты узнал об *обработчиках событий* и о том, как их использовать для поиска событий мыши и клавиатуры. То же самое можно применить в PyQt.

В классе **`MyWindowClass`** мы определяем обработчики события для окна. Поскольку кнопка находится в главном окне, обработчик для ее событий тоже будет здесь.

Сначала мы должны сообщить главному окну, что создаем обработчик событий для конкретного виджета. В листинге 20.2 это происходит в строке 10:

```
self.pushButton.clicked.connect(self.button_clicked)
```

Здесь мы *соединяем* или *связываем событие* (**`self.pushButton.clicked`**) с обработчиком событий (**`self.button_clicked`**). Определение обработчика событий **`button_clicked`** начинается в строке 12. **`clicked`** (щелчок) – это только одно из событий, которые можно применить к кнопке. Также можно использовать **`pressed`** (нажатие) и **`released`** (отпускание).



Думай как программист (на Python)

Подключение события кнопки к обработчику событий называется привязкой обработчика событий. Этот программистский термин обозначает соединение вещей вместе. В PyQt и многих других системах событийного программирования ты не раз услышишь про привязку кода. Обычно ты привязываешь событие или другой сигнал к некоторому коду, который обрабатывает это событие или сигнал. *Сигнал* – это термин программирования, обозначающий способ передачи информации от одной части кода к другой.

Что такое `self`?

В обработчике событий **`button_clicked()`** есть параметр: **`self`**.

Как в нашем первом обсуждении объектов в главе 14, так и здесь **`self`** относится к экземпляру, вызывающему метод. В этом случае все события поступают от фона или главного окна, поэтому именно объект окна обращается к обработчику событий. Здесь **`self`** относится к главному окну. Ты можешь подумать, что **`self`** относится к нажатому *компоненту*, но это не так; оно относится к *окну, содержащему компонент*.

Перемещение кнопки

Когда мы хотим сделать что-то с кнопкой, как нам на нее сослаться? PyQt отслеживает все виджеты в окне. Поскольку ты знаешь, что параметр **`self`** относится к окну, а **`pushButton`** – это имя виджета, то можешь использовать команду **`self.pushButton`**, чтобы получить доступ к этому виджету.

В нашем примере из листинга 20.2 мы заставляли кнопку двигаться каждый раз, когда нажимали на нее. Положение кнопки в окне определяется ее свойством **`geometry`** (геометрия), которое содержит свойства **`x`** (положение относительно оси *x*), **`y`** (положение относительно оси *y*), **`width`** (ширина) и **`height`** (высота). Есть два способа изменить эти свойства.

Один из способов – использовать метод **`setGeometry()`** для изменения свойств геометрии. Другой способ (который мы использовали в листинге 20.2) заключается в применении метода **`move()`**, который изменяет только положение кнопки относительно осей *x* и *y*, а свойства **`width`** и **`height`** оставляет неизменными. Положение относительно оси *x* – это расстояние от левой стороны окна, а положение относительно оси *y* – это расстояние от верхней части окна. Верхний левый угол имеет координаты [0, 0] (как и в Pygame).

Когда ты запустишь эту программу, то увидишь, что после нескольких щелчков мышью кнопка пропадает из правого нижнего угла окна. Если необходимо, можешь изменить размер окна (перетащи мышью край окна), чтобы оно стало больше и ты увидел кнопку. Когда закончишь, можешь закрыть окно с помощью кнопки × вверху окна (или аналогичной кнопки закрытия окна, используемой в твоей операционной системе).

Обрати внимание, что, в отличие от Pygame, нам не нужно беспокоиться о «стирании» кнопки с ее старого места и перерисовывании на новом. Мы просто двигаем ее. PyQt сам заботится обо всех стираниях и перерисовываниях за нас.

Полезные GUI

Наш первый интерфейс с помощью PyQt был хорош в качестве наглядного пособия для базовых компонентов, но больше в нем нет ничего полезного и веселого. Поэтому в остальной части главы и в главе 22 мы будем работать над парой проектов, одним большим и одним маленьким, которые позволят нам узнать больше о возможностях PyQt.

Первый проект будет версией программы для конвертации температур. В главе 22 мы используем PyQt для создания версии игры «Виселица». Позже в этой книге мы снова будем использовать PyQt для создания виртуального питомца.

TempGUI

В главе 3 (раздел «Попробуй самостоятельно») ты написал первую программу по конвертации температур. В главе 5 мы добавили ввод от пользователя, чтобы температура для конвертации не была жестко прописана в программе. В главе 6 мы использовали EasyGui, чтобы получить ввод и показать вывод. Теперь мы применяем PyQt, чтобы сделать графическую версию программы.

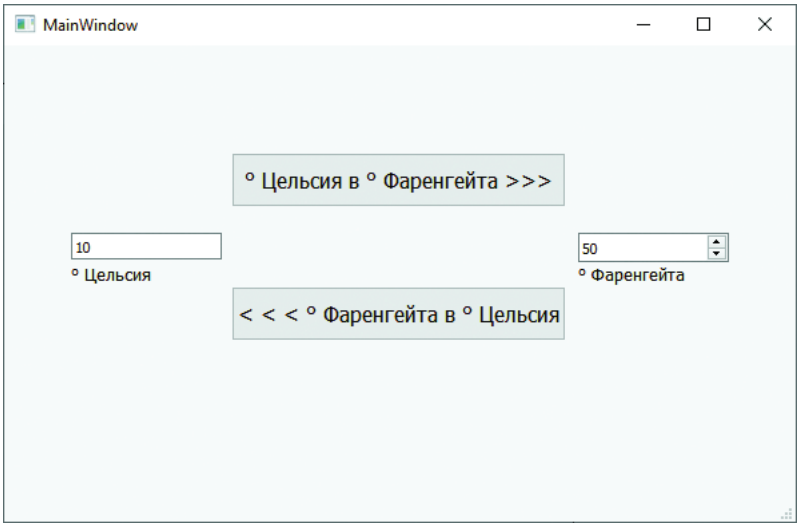
Компоненты TempGUI

Наш интерфейс программы будет простым. Нам нужно всего несколько компонентов:

- поля для ввода/вывода температур (по Цельсию и Фаренгейту);
- кнопки для произведения конвертации;
- метки, для того чтобы показать пользователю, что к чему.

Просто ради интереса, давай используем два разных вида поля ввода для градусов по Цельсию и по Фаренгейту. Ты никогда не сделаешь такое в настоящей программе (это только запутает пользователей), но мы здесь, чтобы учиться!

Когда мы закончим с разметкой GUI, интерфейс должен выглядеть так:



Возможно, ты можешь сделать это самостоятельно, поскольку Qt Designer достаточно прост и понятен. Но в случае если тебе нужна помощь, мы объясним шаги.

Таким образом, мы убедимся, что используем одинаковые имена для компонентов, чтобы тебе было проще разбираться в коде.

Не переживай о точном выравнивании компонентов, такая точность необязательна.

Создание нового GUI

Первый шаг – создание нового проекта PyQt. Когда ты закроешь пользовательский интерфейс, который у тебя открыт (MyFirstGui), Designer снова откроет окно **New Form**. В качестве типа формы выбери вариант **Main Window** и нажми кнопку **Create**.

Теперь начинаем добавлять виджеты: поле ввода градусов по Цельсию – это виджет **Line Edit**, поле ввода градусов по Фаренгейту – это **Spin Box**, метки под каждым полем ввода температуры – это виджеты **Label**, а также есть два компонента **Push Button**. Тебе придется прокрутить вниз список на панели **Widget Box**, чтобы найти некоторые из них. Ниже приведен алгоритм, с помощью которого ты сможешь создать GUI.

1. Найди виджет **Push Button** на панели **Widget Box**. Перетащи его на форму; после чего появится новая кнопка. Затем сделай следующее:
   - сделай кнопку нужного размера, перетаскивая маркеры или вводя новые числа в свойствах группы **geometry** (как ты ты видел на примере MyFirstGui);
   - измени значение свойства **objectName** кнопки на **btnFtoC**;
   - измени значение свойства **text** кнопки на **< < < ° Фаренгейта в ° Цельсия**;
   - измени значение свойства **font size** кнопки на **12**. Если на панели **Property Editor** ты найдешь свойство **font** и нажмешь маленькую кнопку с тремя точками (...), откроется диалоговое окно **Select Font**, похожее на то, которое ты, вероятно, ты видел в своем любимом текстовом редакторе, где можно изменить размер и стиль шрифта.
2. Перетащи другую кнопку на форму, помести ее над первой, сделай ее нужного размера и измени следующие настройки:
   - измени значение свойства **objectName** кнопки на **btnCtoF**;
   - измени значение свойства **text** кнопки на **° Цельсия в ° Фаренгейта >>>**;
   - измени значение свойства **font size** кнопки на **12**.
3. Перетащи виджет **Line Edit** на форму и помести его слева от двух кнопок:
   - измени значение свойства **objectName** виджета **Line Edit** на **editCel**.
4. Перетащи виджет **SpinBox** на форму и помести его справа от двух кнопок:
   - измени значение свойства **objectName** виджета **SpinBox** на **spinFahr**.
5. Перетащи виджет **Label** на форму и помести его под виджетом **Line Edit**:
   - измени значение свойства **text** метки на **° Цельсия**;
   - измени значение свойства **font size** метки на **10**.
6. Перетащи виджет **Label** на форму и помести его под виджетом **SpinBox**:
   - измени значение свойства **text** метки на **° Фаренгейта**;
   - измени значение свойства **font size** метки на **10**.

Теперь у нас есть элементы GUI (компоненты, или элементы управления, или виджеты) с присвоенными именами и метками. Сохрани UI-файл под именем *tempconv.ui* с помощью команды меню **File ⇨ Save As**. Не забудь сохранить файл именно там, где ты сохраняешь свои программы Python.

Далее, открой новый файл в текстовом редакторе IDLE и введи базовый PyQt-код (или скопируй его из первой программы):

```
import sys
from PyQt5 import QtWidgets, uic

form_class = uic.loadUiType("tempconv.ui")[0]

class TemperatureConverterWindow(QtWidgets.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtWidgets.QMainWindow.__init__(self, parent)
        self.setupUi(self)

app = QtWidgets.QApplication(sys.argv)
myWindow = TemperatureConverterWindow()
myWindow.show()
app.exec_()
```

Конвертация градусов из Цельсия в Фаренгейты

Сначала давай сделаем так, чтобы заработала функция конвертации градусов из Цельсия в Фаренгейты. Формула для этого:

```
fahr = cel * 9 / 5 + 32
```

Нам нужно получить значение температуры по Цельсию из текстового поля **`editCel`** виджета **Line Edit**, выполнить расчеты и поместить результат в счетчик градусов по Фаренгейту **`spinFahr`** виджета **SpinBox**. Это все должно происходить, когда пользователь щелкает по кнопке **° Цельсия в ° Фаренгейта >>>**, поэтому весь код будет в обработчике событий этой кнопки.

Во-первых, мы должны привязать событие кнопки **`clicked`** к обработчику событий:

```
self.btnCtoF.clicked.connect(self.btnCtoF_clicked)
```

Этот код использует метод **`__init__()`** класса **`MyWindow`**, точно так же, как это было в нашей первой программе.

Затем нам нужно определить обработчик событий. Чтобы получить значение из поля **Цельсия** (виджета **Line Edit** с именем **`editCell`**), мы используем метод **`self.editCel.text()`**. Это значение будет строкой, которую нужно конвертировать в десятичную дробь:

```
cel = float(self.editCel.text())
```

Затем нужно произвести конвертацию температур:

```
fahr = cel * 9 / 5 + 32
```

Далее нужно поместить значение в поле градусов по Фаренгейту виджета **SpinBox** под названием **`spinFahr`**. Здесь есть одна уловка: виджеты **SpinBox** с изменяемым значением могут использовать только целые числа, не дроби. Итак, нам нужно убедиться, что значение было конвертировано в **`int`** перед его помещением в блок. Число в блоке – это его свойство **`value`**, поэтому код будет выглядеть так:

```
self.spinFahr.setValue(int(fahr))
```

Мы добавляем 0,5 к результату, так что когда мы используем функцию **`int()`** для преобразования числа с плавающей точкой в целое число, оно *округляется* до ближайшего целого числа, а не до следующего наименьшего целого числа. Если сложить все это вместе, то получится вот что:

```
def btnCtoF_clicked(self):
    cel = float(self.editCel.text())      <-- Получение значения в градусах Цельсия
    fahr = cel * 9 / 5 + 32               <-- Конвертирование в градусы Фаренгейта
    self.spinFahr.setValue(int(fahr + 0.5))  <-- Округление значения и его отображение
                                                 в градусах Фаренгейта в поле SpinBox
app = QtGui.QApplication(sys.argv)
myWindow = MyWindowClass()
myWindow.show()
app.exec_()
```

Конвертация градусов из Фаренгейта в Цельсия

Код для конвертации температуры в обратную сторону очень похож. Формула для конвертации:

```
cel = (fahr - 32) * 5 / 9
```

Она находится в обработчике событий для кнопки **< < < ° Фаренгейта в ° Цельсия**. Мы подключаем обработчик событий к кнопке (в методе **`__init__()`** окна):

```
self.btnFtoC.clicked.connect(self.btnFtoC_clicked)
```

Затем в обработчике событий нам нужно получить температуру из поля **SpinBox**:

```
fahr = self.spinFahr.value()
```

Это значение уже является целым числом, поэтому нам не нужно ничего менять. Затем мы применяем формулу:

```
cel = (fahr - 32) * 5 / 9
```

Наконец, мы конвертируем результат в строку и помещаем в поле ввода градусов Цельсия:

```
self.editCel.setText(str(cel))
```

Вся программа приведена в листинге 20.3.

Листинг 20.3. Полная программа по конвертации температур

```
import sys
from PyQt5 import QtWidgets, uic

form_class = uic.loadUiType("tempconv.ui")[0]
class TemperatureConverterWindow(QtWidgets.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtWidgets.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.btnCtoF.clicked.connect(self.btnCtoF_clicked)
        self.btnFtoC.clicked.connect(self.btnFtoC_clicked)

    def btnCtoF_clicked(self):
        cel = float(self.editCel.text())
        fahr = cel * 9 / 5.0 + 32
        self.spinFahr.setValue(int(fahr + 0.5))

    def btnFtoC_clicked(self):
        fahr = self.spinFahr.value()
        cel = (fahr - 32) * 5 / 9.0
        self.editCel.setText(str(cel))

app = QtWidgets.QApplication(sys.argv)
myWindow = TemperatureConverterWindow(None)
myWindow.show()
app.exec_()
```

Загрузка определения пользовательского интерфейса

Привязка обработчика событий к кнопкам

*Обработчик событий для кнопки **btnCtoF***

*Обработчик событий для кнопки **btnFtoC***

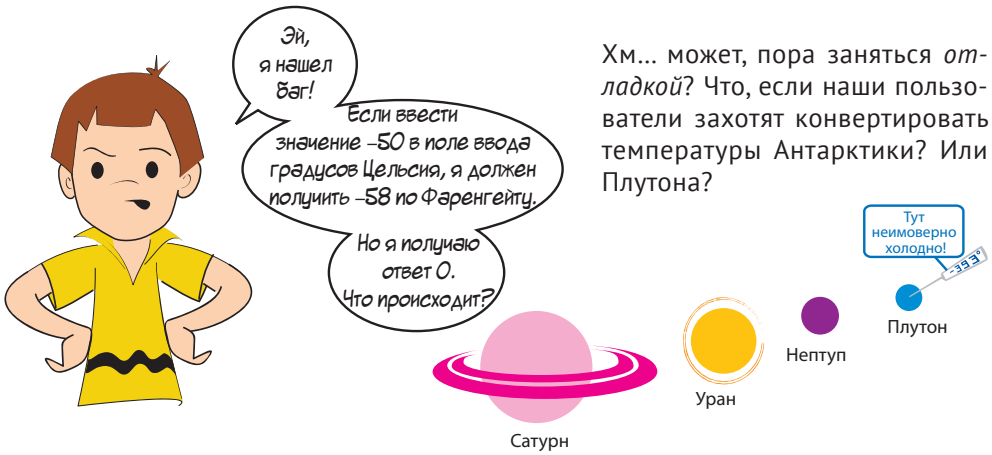
Сохрани код программы в файл с именем *TempGui.py*, запусти и проверь работу интерфейса.

Небольшое улучшение

При запуске ты заметишь, что при конвертации градусов по Фаренгейту в градусы Цельсия в результате отображается много цифр после десятичной точки, а в некоторых случаях часть из них не вмещается в текстовое поле. Есть способ это исправить. Он называется *форматированием вывода*. Ты еще о нем не знаешь, поэтому можешь заглянуть в главу 21, чтобы получить полное объяснение принципов его работы, либо сначала набрать код, приведенный ниже. Измени последнюю строку обработчика событий **`btn_FtoC_clicked`** на две строки, приведенные ниже:

```
cel_text = '%.2f' % cel
self.editCel.setText(cel_text)
```

В результате число будет отображаться с двумя знаками после запятой.



Хм... может, пора заняться *отладкой*? Что, если наши пользователи захотят конвертировать температуры Антарктики? Или Плутона?

Исправление бага

Мы говорили ранее, что одним из способов узнать, что происходит в программе, является вывод значений некоторых переменных во время выполнения программы. Давай попробуем.

Поскольку это значение температуры работает неверно, давай с него и начнем. Добавь следующую строку после строки обработчика событий **`btn_CtoF_clicked`** в листинге 20.3:

```
print('цел = ', cel, ' фар = ', fahr)
```

Теперь при щелчке по кнопке **° Цельсия в ° Фаренгейта >>>** ты увидишь значения переменных **`cel`** и **`fahr`**, выведенные в окне оболочки IDLE. Попробуй разные значения для **`cel`** и посмотри, что происходит. Мы получили следующее:

```
>>>
RESTART: C:/HelloWorld/Примеры/TempGui.py
цел = 50.0 фар = 122.0
цел = 0.0 фар = 32.0
цел = -10.0 фар = 14.0
цел = -50.0 фар = -58.0
```

Кажется, что значение **`fahr`** было посчитано верно. Почему же тогда в поле ввода температуры по Фаренгейту не отображается число меньше 0 (или больше 99, раз уж на то пошло)?

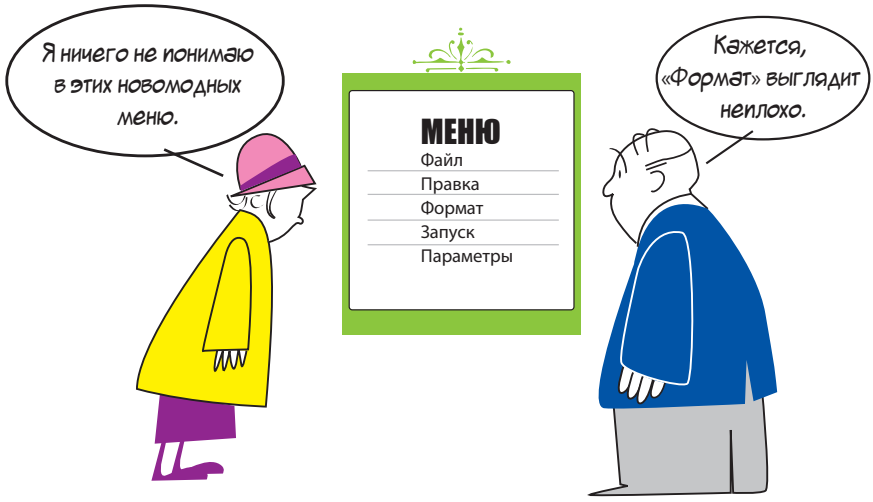
Вернись к Qt Designer и щелкни по виджету `spinFahr`, который мы использовали для температуры по Фаренгейту. Теперь просмотри на панели **Property Editor** разные свойства. Видишь два свойства с именами ***minimum*** и ***maximum*** (в нижней части списка свойств поля **SpinBox**)? Каковы их значения? Теперь сможешь решить проблему?

Содержимое меню

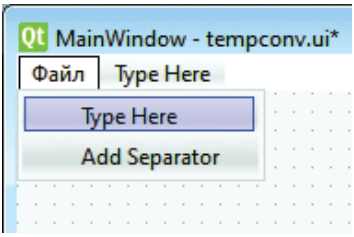
Наш интерфейс «температурной программки» содержит кнопки для проведения конвертации. У многих программ также есть *меню* для выполнения некоторых функций. Иногда это те же функции, которые можно выполнить с помощью кнопок, зачем же тогда нам два разных способа для одного и того же результата?

Ну, некоторым пользователям удобнее использовать меню, чем щелкать по кнопкам. В сложных программах, как правило, много функций, и для них потребовалось бы много кнопок. Это перегрузит графический интерфейс. Поэтому в случае работы со сложной программой лучше использовать меню. Также ты можешь оперировать меню с клавиатуры, а некоторые люди находят более быстрым использовать меню, чем отрывать руки от клавиатуры и использовать мышь.

Давай добавим несколько пунктов меню, чтобы предоставить пользователям нашей программы другой способ преобразования температуры. Мы также можем добавить команду меню **Файл** ⇨ **Выход**, которая есть почти в каждой программе.



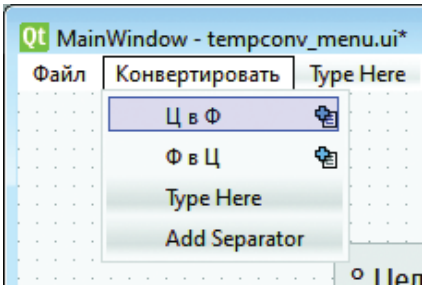
В PyQt есть способ создания и редактирования меню. Если ты взглянешь в верхний левый угол формы в Designer, то увидишь надпись **Type Here**. Именно здесь мы начинаем создавать меню. Во многих программах первое меню – **Файл**, поэтому давай начнем с этого. Щелкни по области с надписью **Type Here**, введи слово **Файл** и нажми клавишу **Enter**. Ты увидишь, как появится меню **Файл**, а также пространство для ввода дополнительных пунктов меню рядом с ним и под ним, например:



Добавление пункта меню

В меню **Файл** мы добавим пункт **Выход**. Там, где написано **Type Here**, под меню **Файл**, введи слово **Выход**, а затем нажми клавишу **Enter**.

Теперь давай добавим пункт меню для конвертации температур (если пользователь не хочет использовать кнопки). В том месте, где написано **Type Here**, справа от меню **Файл**, введи слово **Конвертировать**. Затем, ниже, создай два новых пункта меню: **Ц в Ф** и **Ф в Ц**. По окончании работы твое меню должно выглядеть следующим образом:



Если ты взглянешь на панель **Object Inspector** в правой части окна Qt Designer, то увидишь примерно следующее:

Object Inspector

Object	Class
menubar	QMenuBar
menuFile	QMenu
actionExit	QAction
menuConvert	QMenu
actionC_to_F	QAction
actionF_to_C	QAction

Ты видишь меню **Файл** и **Конвертировать**, а также пункты меню **Выход**, **Ц в Ф** и **Ф в Ц**. В терминологии PyQt пункты меню – экземпляры класса `QAction`. Это имеет смысл, потому что ты хочешь, чтобы при выборе пункта меню произошло какое-то *действие*.

Сохрани отредактированный файл под именем *tempconv_menu.ui*.

Теперь, когда у тебя есть пункты меню (или действия), тебе нужно привязать (или подключить) их события к обработчикам событий. Для пунктов меню **Ц в Ф** и **Ф в Ц** у нас уже есть обработчики событий – те, которые мы сделали для кнопок. Мы хотим, чтобы в меню происходило то же самое, что и при нажатии кнопок. Таким

образом, мы можем просто подключить пункты меню к тем же обработчикам событий.

Для элемента меню (*действия*) это не событие *щелчка*, которое нужно обработать, а событие *тригера*. То, что мы подключаем к обработчику событий, называется **`actionC_to_F`**, а обработчик событий, к которому мы подключаемся, – это обработчик событий кнопки **`btnCtoF_clicked`**. Код для подключения обработчика событий к пункту меню выглядит следующим образом:

> Если ты пользуешься операционной системой macOS, тебе нужно снять флажок **nativeMenuBar** для объекта меню (это последнее свойство в списке на панели **Property Editor**). В противном случае меню твоей операционной системы будет конфликтовать с основным меню Python на рабочем столе, из-за чего меню **Файл** исчезнет из приложения PyQt.

```
self.actionC_to_F.triggered.connect(self.btnCtoF_clicked)
```

То же самое мы проделываем с пунктом меню **Ф в Ц**.

Для пункта меню **Выход** нам нужно создать новый обработчик событий и привязать его к этому событию. Мы назовем этот обработчик событий **`menuExit_selected`**. Код для привязки этого обработчика событий выглядит следующим образом:

```
self.actionExit.triggered.connect(self.menuExit_selected)
```

Обработчик событий для меню **Выход** имеет только одну строку, которая закрывает окно:

```
def menuExit_selected(self):
self.close()
```

Наконец, измени загружаемый файл пользовательского интерфейса (в третьей строке) на файл с меню, который ты сохранил ранее под названием *tempconv_menu.ui*.

После внесения всех этих изменений код должен выглядеть так, как показано в листинге 20.4:

Листинг 20.4. Полная программа конвертации температуры с меню

```
import sys
from PyQt5 import QtWidgets, uic

form_class = uic.loadUiType("tempconv_menu.ui")[0]

class TemperatureConverterWindow(QtWidgets.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtWidgets.QMainWindow.__init__(self, parent)
        self.setupUi(self)
```

Загрузка файла пользовательского интерфейса с меню

```
        self.btnCtoF.clicked.connect(self.btnCtoF_clicked)
        self.btnFtoC.clicked.connect(self.btnFtoC_clicked)
        self.actionC_to_F.triggered.connect(self.btnCtoF_clicked)
        self.actionF_to_C.triggered.connect(self.btnFtoC_clicked)
        self.actionExit.triggered.connect(self.menuExit_selected)

    def btnCtoF_clicked(self):
        cel = float(self.editCel.text())
        fahr = cel * 9 / 5 + 32
        self.spinFahr.setValue(int(fahr + 0.5))

    def btnFtoC_clicked(self):
        fahr = self.spinFahr.value()
        cel = (fahr - 32) * 5 / 9
        self.editCel.setText(str(cel))

    def menuExit_selected(self):
        self.close()

app = QtWidgets.QApplication(sys.argv)
myWindow = TemperatureConverterWindow(None)
myWindow.show()
app.exec_()
```

Подключение обработчика событий меню ***Выход***

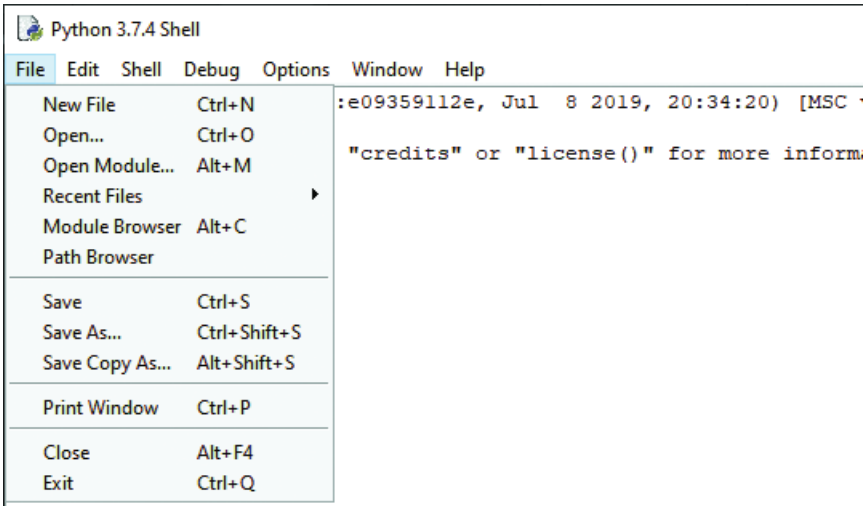
Подключение обработчика событий меню ***Конвертировать***

Новый обработчик событий меню ***Выход***

Горячие клавиши

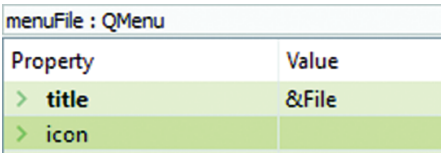
Мы уже говорили, что одна из причин, по которой некоторые люди предпочитают использовать меню, а не кнопки, заключается в том, что они могут взаимодействовать с меню с клавиатуры, не используя мышь. Прямо сейчас наши меню работают при помощи мыши, но пока еще не могут работать при помощи клавиатуры. Чтобы это стало возможным, необходимо добавить горячие клавиши.

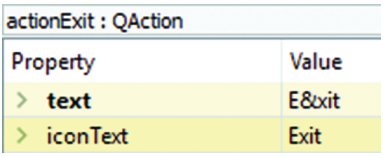
Горячие клавиши (также называемые сочетаниями клавиш) позволяют выбирать пункты меню, используя только клавиатуру. В Windows и Linux ты активируешь систему меню с помощью клавиши **Alt**. (О macOS мы поговорим чуть позже.) При нажатии клавиши **Alt** ты увидишь выделенную определенную букву каждого пункта меню, обычно с подчеркиванием. Подчеркнутая буква – это та, которую нужно нажать для активации этого меню. Так, например, чтобы попасть в меню **File** оболочки Python, нажми сочетание клавиш **Alt**+**F**, то есть удерживай нажатой клавишу **Alt**, а затем нажми клавишу **F**. После этого ты увидишь пункты меню **File**, а также горячие клавиши для каждого из них. Попробуй сделать это с помощью окна IDLE:



Чтобы открыть новое окно, используй сочетание клавиш **Alt**+**F**+**N** (удерживая нажатой клавишу **Alt**, нажми клавишу **F**, а затем – клавишу **N**).

Теперь мы установим горячие клавиши в меню интерфейса преобразования температуры. Чтобы установить горячую клавишу, все, что тебе нужно сделать, – это поставить символ **&** перед буквой, которую ты хочешь использовать в качестве горячей клавиши. Это делается в свойстве **Title** того меню, в котором ты хочешь установить горячие клавиши (например, в меню **Файл**), или в свойстве **text** для элемента меню (например, **Выход**). Для перехода в меню **Файл** мы установим горячую клавишу **Ф**, а для действия **Выход** – горячую клавишу **В**. Таким образом, **Файл** становится **&Файл**, а **Выход** становится **&Выход**.





Нам нужно решить, какие горячие клавиши использовать для пунктов меню **Конвертировать**. Давай создадим горячую клавишу **К** для перехода в меню **Конвертировать**, горячую клавишу **Ц** для конвертации градусов Цельсия в градусы Фаренгейта и горячую клавишу **Ф** для конвертации градусов по Фаренгейту в градусы по Цельсию. Итак, у нас есть **&Конвертировать**, **&Ц в Ф** и **&Ф в Ц**. Теперь мы можем использовать сочетания горячих клавиш **Alt**+**К**+**Ц** (чтобы конвертировать градусы Цельсия в градусы Фаренгейта) и **Alt**+**К**+**Ф** (чтобы конвертировать градусы Фаренгейта в градусы Цельсия).

Как только ты установишь горячие клавиши в Qt Designer, тебе больше ничего не нужно будет делать – не нужно писать никакого кода. PyQt и операционная система сами подчеркнут буквы, соответствующие горячим клавишам, и будут обрабатывать информацию, получаемую от клавиатуры. Просто сохрани файл пользовательского интерфейса. Ты можешь сохранить его под другим именем, например *tempconv_menu_hotkeys.ui*. Также проверь, что изменил третью строку в листинге 20.4, благодаря чему сможешь загрузить этот файл пользовательского интерфейса:

```
form_class = uic.loadUiType("tempconv_menu_hotkeys.ui")[0]
```

Если говорить коротко, то нет. Поскольку все компьютеры Mac поставляются с мышью (или сенсорной панелью) с самого начала, предполагается, что в macOS ты будешь использовать мышь для управления меню. В macOS нет сочетаний клавиш для пунктов меню по аналогии с Windows. Существуют сочетания клавиш для многих функций, и некоторые из них соответствуют пунктам меню. Но ты не можешь напрямую управлять системой меню с помощью горячих клавиш, как это происходит в Windows.

С интерфейсом по конвертации температур закончили. В главе 22 мы сделаем версию игры «Виселица» с помощью PyQt.

Что ты узнал?

В этой главе ты узнал:

- о PyQt;
- о Qt Designer;
- о виджетах – кнопках, полях и т. д., которые составляют интерфейсы GUI;
- об обработчиках событий – как заставить элемент выполнять назначенную команду;
- об элементах меню и горячих клавишах.

Проверь свои знания

1 Каковы три названия для таких элементов, как кнопки, текстовые поля и т. д., в интерфейсе GUI?

2 Как называется клавиша с буквой, которую ты нажимаешь вместе с клавишей **Alt**, для того чтобы попасть в меню?

3 Что должно быть в конце имени файлов ресурсов для Qt Designer?

4 Каковы пять типов виджетов, которые можно включить в GUI с помощью PyQt?

5 Чтобы заставить виджет (например, кнопку) что-то делать, он должен иметь _____.

6 Какой особый символ используется в меню для назначения горячей клавиши?

7 Содержимое поля **SpinBox** в PyQt – это всегда _____.

Попробуй самостоятельно

1 Мы сделали текстовую программу по угадыванию чисел в главе 1 и версию той же игры, но с GUI в главе 6. Попробуй сделать ее GUI-версию с помощью PyQt.

2 Ты разобрался, почему в поле **SpinBox** не отображаются значения ниже 0? (Картер нашел этот баг в листинге 20.2.) Исправь свойства виджета так, чтобы проблема исчезла. Убедись, что ты исправил их так, что виджет может отображать и очень высокие температуры. (Может быть, пользователь будет конвертировать температуры Меркурия и Венеры, а не только Плутона!)

ГЛАВА 21

Форматирование вывода

В самом начале главы 1 ты узнал о команде `print`. Это была наша первая команда в Python. Мы также узнали (в главе 5), что в конце команды `print` можно поставить `end=''`, чтобы Python выводил текст в одной строке. Мы использовали это для приглашения ввода функции `input()`, пока не узнали о сокращении для нее.

В этой главе ты узнаешь о *форматировании* – способах сделать вывод программы таким, как тебе требуется. Мы поговорим о:

- начале новых строк (и когда это делать);
- горизонтальном выделении отступами (и выстраивании в колонки);
- выводе переменных в середине строки;
- форматировании чисел в целом, десятичном формате или в формате E-нотации, настройке количества цифр после запятой.

Мы также узнаем о некоторых встроенных методах Python для работы со строками. Эти методы позволяют выполнять следующие действия:

- разделять строки на меньшие части;
- объединять строки вместе;
- искать строки;
- искать элементы в строках;
- удалять части строк;
- изменять регистр (прописные и строчные буквы).

Все это полезно в текстовых (не имеющих интерфейса GUI) программах, а также многих проектах интерфейса GUI и играх.

Переход строки

Ты уже использовал команду **`print`** много раз. Но что произойдет, если использовать ее не единожды? Попробуй выполнить следующий код:

```
print(«Привет»)
print(«Вам»)
```

При запуске вывод будет выглядеть так:

```
>>>
RESTART: C:/HelloWorld/Примеры/HiThere.py
Привет
Вам
```

Почему эти два слова выведены в разных строках? Почему вывод не выглядит так:

```
ПриветВам
```

Если не запрограммировано иное, Python начинает каждое утверждение **`print`** с новой строки. После слова **`Привет`** Python переходит на строку ниже и обратно к первой колонке, чтобы вывести **`Вам`**. Python вставляет символ *перехода строки* между этими двумя словами. Переход строки похож на нажатие клавиши **Enter** в текстовом редакторе.

Думай как программист

Напомним, что в главе 5 ты узнал о понятиях CR (возврат каретки) и LF (перевод строки), которые используются для окончания строки текста. И помнишь, что я сказал, что некоторые системы используют либо то, либо другое, либо все вместе? *Переход строки* – общий термин для маркера окончания строки в любой системе. В Windows перевод строки равен CR + LF. В Linux перевод строки равен LF, а в macOS – CR. Таким образом, тебе не нужно задумываться об обозначениях системы. Просто укажи переход строки там, где нужно начать текст с новой строки.

Команда `print` и `end=''`

Команда **`print`** автоматически вставляет символ перевода строки в конце вывода, если только не указано обратное. И как это сделать? Добавив **`end=''`** (как в главе 5):

```
print('Привет', end='')
print('Вам')
>>>
RESTART: C:/HelloWorld/Примеры/HiThere2.py
ПриветВам
```

Обрати внимание, что пробел между словами **`Привет`** и **`Вам`** отсутствует.

Ты также можешь вывести несколько аргументов команды **`print`** в одной строке, но Python добавит пробел между ними:

```
print('Привет', 'Вам')
>>>
RESTART: C:/HelloWorld/Примеры/HiThere3.py
Привет Вам
```

Ты также можешь вывести несколько строк в одной с помощью *конкатенации*, о чем ты уже знаешь:

```
print('Привет' + 'Вам')
>>>
RESTART: C:/HelloWorld/Примеры/HiThere4.py
ПриветВам
```

Напомним, что конкатенация *похожа* на сложение строк вместе, но у нее иное название, потому что сложение предназначено только для чисел.

Добавление принудительных переходов строки

А что, если нам нужно добавить принудительный переход строки? Например, нам нужен пробел между словами **`Привет`** и **`Вам`**? Самое простое – добавить еще одно утверждение **`print`**:

```
print("Привет")
print()
print("Вам")
```

При запуске получаем следующий вывод:

```
>>>
RESTART: C:/HelloWorld/Примеры/HiThere5.py
Привет

Вам
```

Особые коды вывода

Существует другой способ добавления перехода строки. В Python есть особые коды, которые можно добавить в строки, чтобы они выглядели по-разному. Эти особые коды вывода начинаются с символа обратного слеша (****).

Код для переноса строки: **\n**. Выполни в интерактивном режиме:

```
>>> print(«Привет Вам»)
Привет Вам
>>> print(«Привет \nВам»)
Привет
Вам
```

Символы **\n** вывели слова **Привет** и **Вам** на разных строках, потому что между ними был добавлен переход строки.

Горизонтальное выделение пробелами – табуляция

Мы только что узнали, как контролировать вертикальные интервалы (добавлением переходов строки или использованием запятых для предотвращения перехода). Теперь поговорим об управлении горизонтальным выделением пробелами – табуляции.

Табуляция полезна для выравнивания содержимого по колонкам. Чтобы разобраться в этой функции, представь, что каждая строка на экране поделена на блоки одинакового размера. Допустим, каждый блок вмещает восемь символов. Когда ты используешь табуляцию, ты перемещаешься к началу следующего блока.

Лучше всего это видно на примере. Особый код для табуляции – **\t**. Выполни в интерактивном режиме:

```
>>> print('АБВ\tЭЮЯ')
АБВ     ЭЮЯ
```

Обрати внимание, что символы **ЭЮЯ** отстоят от символов **АБВ**. Фактически **ЭЮЯ** находятся на расстоянии восьми символов от начала строки. Это произошло потому, что размер блока – 8 символов. Иначе говоря, *шаг табуляции* имеет размер в восемь символов.

Справа представлено несколько примеров разных команд **print**:

```
>>> print('АБВ\tЭЮЯ')
АБВ     ЭЮЯ

>>> print('АБВГД\tЭЮЯ')
АБВГД   ЭЮЯ

>>> print('АБВГДЕ\tЭЮЯ')
АБВГДЕ ЭЮЯ

>>> print('АБВГДЕЖ\tЭЮЯ')
АБВГДЕЖЭЮЯ

>>> print('АБВГДЕЖЗИ\tЭЮЯ')
АБВГДЕЖЗИ       ЭЮЯ
```

Ты можешь представлять себе экран (или каждую строку) разделенным на блоки по восемь символов каждый. Обрати внимание, что последовательность **АБВ** становится длиннее, а **ЭЮЯ** остается на том же месте. Код **\t** говорит Python начинать **ЭЮЯ** в следующем шаге табуляции или в следующем доступном блоке. Но когда последовательность **АБВ** становится достаточно длинной и заполняет первый блок, Python перемещает **ЭЮЯ** в следующий шаг табуляции.

Табуляция хороша для организации содержимого в колонки, чтобы все было красиво выровнено. Давай используем ее, а также наши знания о циклах, чтобы вывести на экран таблицу возведенных в квадрат и в третью степень чисел. Открой новое окно интерпретатора IDLE и введи код программы из листинга 21.1. Сохрани ее и запусти. (Мы назвали нашу программу *squbes.py*.)

Листинг 21.1. Программа для вывода квадратной и третьей степеней чисел

```
print(«Число\t В квадрате\t В кубе»)
for i in range(1, 11):
    print(i, '\t', i**2, '\t', i**3)
```

При запуске вывод будет выглядеть так:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_21-1.py
Число  В квадрате  В кубе
1      1           1
2      4           8
3      9           27
4      16          64
5      25          125
6      36          216
7      49          343
8      64          512
9      81          729
10     100         1000
```

Как вывести на экран обратный слеш?

Поскольку символ обратного слеша (\) используется в специальных кодах для вывода, как нам объяснить Python, что нужно вывести именно сам символ \, а не обозначить часть кода? Нужно указать два обратных слеша рядом:

```
>>> print('привет\\вам')
привет\вам
```

Первый слеш сообщает Python о том, что ожидается нечто особенное, а второй – что это сам слеш.

СЛОВАРИК

Когда ты используешь два обратных слеша для вывода символа, первый обратный слеш называется escape-символом. Мы говорим, что первый обратный слеш скрывает второй, так что второй обратный слеш рассматривается как обычный символ, а не как специальный.

Вставка переменных в строки

До сих пор мы делали следующее, чтобы вставить переменную в середину строки:

```
name = 'Юрий Семецкий'
print('Меня зовут', name, 'и я написал книгу.')
```

При выполнении этого кода мы получали следующее:

```
Меня зовут Юрий Семецкий и я написал книгу.
```

Но есть и другой способ вставить переменную в строку, который дает нам больше возможностей управления их внешним видом, особенно числами. Мы можем использовать *форматирующие строки*, которые включают символ **%**. Допустим, тебе нужно вставить строковую переменную в середину команды **print**, как мы только что делали. С помощью форматирующих строк это можно сделать так:

```
name = 'Юрий Семецкий'
print('Меня зовут %s и я написал книгу.' % name)
```

Символ **%** использован два раза. В середине строки он отмечает позицию, где будет переменная. В конце строки он сообщает Python, какую именно переменную мы хотим там видеть.

Символы **%s** сообщают о том, что мы хотим вставить именно строковую переменную. Для целого числа это будет **%i**, а для десятичной дроби мы используем **%f**.

Ниже приведены дополнительные примеры:

```
age = 13
print('Мне %i лет.' % age)
```

При выполнении этого кода ты увидишь:

```
Мне 13 лет.
```

Ниже представлен еще пример:

```
average = 75.6
print('Средний бал нашего теста по математике %f процентов.' % average)
```

Вывод:

```
Средний бал нашего теста по математике 75.600000 процентов.
```

Символы **%s**, **%i** и **%f** называются *форматирующими строками*, и они определяют внешний вид переменной.

Существует также несколько форматирующих строк, которые можно использовать для получения числа в E-нотации. (Помнишь ее из главы 3?) См. следующие разделы.

Форматирование чисел

Когда мы выводим на экран числа, мы хотим управлять их внешним видом:

- сколько знаков после запятой мы увидим;
- используется ли обычная или E-нотация;
- добавлять ли начальный или конечный ноль;
- выводить ли знаки + или – перед числами.

С помощью форматирующих строк Python дает нам необходимую для этого гибкость.

Например, если ты используешь программу для прогноза погоды, что тебе приятнее видеть:

```
Сегодня ожидается от 15.66569956 до 9.09868934 градусов
```

или

```
Сегодня ожидается от 15 до 9 градусов
```

Правильный внешний вид чисел важен для многих программ.

Давай начнем с примера. Допустим, нам нужно вывести десятичное число с двумя знаками после запятой. Выполни в интерактивном режиме:

```
>>> dec_number = 12.3456
>>> print('Сегодня %.2f градусов тепла.' % dec_number)
Сегодня 12.35 градусов тепла.
```

В середине команды **`print`** находится наша форматирующая строка. Но вместо **`%f`** я использовал **`%.2f`**. Этот код указывает Python использовать две цифры после запятой. (Обрати внимание, что Python правильно округлил число до двух знаков после запятой, а не просто обрубил остальные цифры.)

После строки второй символ **`%`** говорит о том, что число для вывода идет далее. Число выводится на экран с форматированием, которое описано в форматирующей строке. Еще несколько примеров прольют свет на то, как все устроено.

У тебя хорошая память, Картер! Символ % действительно используется для модуля (остаток при делении целых чисел), как мы узнали в главе 3, но также с его помощью обозначает форматирующие строки. Python может по способу использования разобраться, имелся ли в виду модуль или форматирующая строка.

Целые числа: `%d` или `%i`

Чтобы вывести число в виде целого числа, используй форматирующую строку **`%d`** или **`%i`**. (Не знаем, почему их две, но можно использовать любую.)

```
>>> number = 12.67
>>> print('%i' % number)
12
```

Обрати внимание, что в этот раз число не было округлено. Оно было *отсечено*. Если бы оно было округлено, ответ был бы 13, а не 12. Когда ты используешь форматирование целых чисел, знаки после запятой отсекаются, а при использовании десятичных дробей число округляется.

Здесь стоит заметить три момента:

- тебе необязательно наличие другого текста в строке – можно иметь только форматирующую строку;
- даже если наше число было десятичной дробью, мы вывели его на экран как целое число. Это можно сделать с помощью форматирующих строк;
- Python отсек знаки после запятой до более низкого значения целого числа. Тем не менее это работает не так, как функция **`int()`** (о которой мы узнали в главе 4), потому что форматирующие строки не создают новое значение как функция **`int()`** – они изменяют внешний вид значения.

Только что ты вывел число 12.67 в формате целого числа, то есть ты увидел 12. Но значение переменной **`number`** не изменилось. Проверь:

```
>>> print(number)
12.67
```

Значение не изменилось. Мы только показали его по-другому с помощью форматирующей строки.

Десятичные дроби: `%f` или `%F`

С десятичными дробями можно использовать букву *f* в верхнем или нижнем регистре (**`%f`** или **`%F`**):

```
>>> number = 12.3456
>>> print('%f' % number)
12.345600
```

Если ты используешь только **`%f`**, число будет показано с шестью знаками после запятой. Если ты добавишь **`.n`** перед **`f`**, где **`n`** – любое целое число, то дробь будет округлена до указанного количества знаков после запятой:

```
>>> print('%.2f' % number)
12.35
```

Ты можешь видеть, как число 12.3456 было округлено до двух знаков после запятой: 12.35.

Если ты укажешь большее количество знаков, чем на самом деле есть в числе, Python *заполнит* эти места нулями:

```
>>> print('%.8f' % number)
12.34560000
```

Здесь число содержит только четыре знака после запятой, но нам нужно 8, поэтому остальные четыре знака были заполнены нулями.

Если число отрицательное, **`%f`** всегда отображает знак отрицания **`-`**. Если ты хочешь всегда видеть знак (даже положительный) возле числа, используй символ **`+`** сразу после символа **`%`** (хорошо для выравнивания списков положительных и отрицательных чисел):

```
>>> print('%+f' % number)
+12.345600
```

Если тебе нужно выровнять список положительных и отрицательных чисел, но ты не хочешь видеть знак **`+`** возле положительных, используй пробел вместо знака **`+`** после символа **`%`**:

```
>>> number2 = -98.76
>>> print('% .2f' % number2)
-98.76
>>> print('% .2f' % number)
12.35
```

Обрати внимание, что перед цифрой 12 в выводе есть пробел, поэтому числа 12 и 98 находятся прямо друг под другом, хотя рядом с одним указан его знак, а рядом с другим – нет.

E-нотация: %e и %E

Когда мы говорили о E-нотации (в главе 3), я пообещал, что покажу тебе, как выводить на экран число с ее помощью. Время пришло.

```
>>> number = 12.3456
>>> print('%e' % number)
1.234560e+01
```

Форматирующая строка **%e** используется для вывода числа в E-нотации. Она всегда выводит шесть знаков после запятой, если не указано иное.

Ты можешь вывести больше или меньше знаков после запятой с помощью **.n** после знака **%**, как и в случае с десятичными дробями:

```
>>> number = 12.3456
>>> print('%.3e' % number)
1.235e+01
>>> print('%.8e' % number)
1.23456000e+01
```

Строка **%.3e** округлила число до трех знаков после запятой, а **%.8e** добавила нули, чтобы показать лишние знаки.

Ты можешь использовать нижний или верхний регистр для буквы *e*, и вывод будет в том же регистре, который ты выбрал:

```
>>> print('%E' % number)
1.234560E+01
```

Автоматический выбор десятичной дроби или E-нотации: %g и %G

Если ты хочешь, чтобы Python автоматически выбрал вывод числа в виде десятичной дроби или E-нотации, используй форматирующую строку **%g**. При использовании верхнего регистра ты получишь верхний регистр в выводе.

```
>>> number1 = 12.3
>>> number2 = 456712345.6
>>> print('%g' % number1)
12.3
>>> print('%g' % number2)
4.56712e+08
```

Заметил, как Python выбрал E-нотацию для длинного числа и десятичную дробь для короткого?

Как мне вывести на экран знак процента?

Ты можешь задуматься, если знак процента (`%`) – это особый символ для форматирующих строк, как вывести сам знак на экран?

Ну, порой Python достаточно умен, чтобы понять, когда ты используешь знак `%` для начала форматирующей строки, а когда ты просто хочешь вывести его на экран:

```
>>> print('Я выполнил тест по математике на 90%!')
Я выполнил тест по математике на 90%!
```

В данном случае не было второго знака % за пределами строки и не было переменной для форматирования, поэтому Python решил, что знак % был просто символом в строке.

Но если ты печатаешь в форматирующей строке и хочешь вывести знак процента, то используй два знака процента, точно так же, как ты использовал два обратных слеша для вывода одного обратного слеша. Мы говорим, что первый знак процента скрывает второй знак процента, точно так же, как ранее в этой главе:

```
>>> math = 75.4
>>> print('Я выполнил тест по математике на %.1f%%!' % math)
Я выполнил тест по математике на 75.4%!
```

Первый знак процента запускает форматирующую строку. Два знака процента, стоящих рядом, сообщают Python, что ты действительно хочешь вывести символ процента. Затем у тебя есть знак процента вне кавычек, который сообщает Python, что переменная, которую ты хочешь напечатать, будет идти следом.

Несколько форматирующих строк

Что делать, если ты хочешь вывести несколько форматирующих строк? Вот как это можно сделать:

```
>>> math = 75.4
>>> science = 82.1
>>> print('Тест по математике я выполнил на %.1f, а по физике на %.1f'
          % (math, science))
```

Ты можешь поместить столько форматирующих строк, сколько захочешь, в инструкцию **`print`**, а затем в кортеж переменных, которые хочешь вывести. Напомним, что кортеж подобен списку, за исключением того, что он использует круглые скобки вместо квадратных, и кортеж неизменяем. Это один из примеров, когда Python придирчив, – ты должен использовать кортеж; а не список. Единственное исключение – это когда у тебя есть только одна переменная для форматирования; тогда она не обязательно должна находиться в кортеже. (Ты видел это в большинстве наших примеров.) Убедись, что количество форматирующих строк (внутри кавычек) и количество переменных (вне кавычек) совпадают, иначе ты получишь сообщение об ошибке.

Хранение форматированных чисел

Иногда тебе не нужно выводить форматированные числа на экран сразу же, но ты хочешь сохранить их для дальнейшего использования. Это просто. Вместо вывода присвой их переменной, например:

```
>>> my_string = '%.2f' % 12.3456
>>> print(my_string)
12.35
>>> print('Ответ', my_string)
Ответ 12.35
```

Вместо того чтобы сразу вывести форматированное число, мы присвоили его переменной **my_string**. Затем мы совместили переменную с текстом и вывели на экран предложение.

Хранение форматированного числа в виде строки очень полезно для использования в GUI и графических программах, например в играх. Когда у тебя есть имя переменной для форматированной строки, ты можешь вывести его, когда и как захочешь: в текстовом поле, диалоге или на экране игры.

Новый способ форматирования

Способ форматирования строк, о котором ты только что узнал, работает во всех версиях Python. Но есть и другой способ сделать то же самое в Python 3.6 и более поздних версиях. Поскольку в этой книге мы используем Python 3.7, я подумал, что стоит упомянуть об этом новом способе. Ты можешь встретить этот способ Python и, по крайней мере, будешь знать, что это такое. Решай сам, какой способ – старый или новый – будешь использовать.

f-строки

В Python (в версиях 3.6 и более поздних) есть особый способ для форматирующих строк: постановка буквы **f** перед кавычками. Это работает аналогично форматирующим строкам со знаком процента, которые ты видел ранее. Фактически спецификаторы формата – **f**, **g**, **e** и т. д. – одинаковы. Ты просто используешь их немного по-другому. Лучший способ понять это – увидеть на примере.

Вот старый способ:

```
print('Тест по математике я выполнил на %.1f, а по физике на %.1f'
      % (math, science))
```

А вот новый:

```
print(f'Тест по математике я выполнил на {math:.1f}, а по физике
      на {science:.1f}')
```

Пользуясь новым способом, ты, вместо того чтобы ставить знак процента в начале форматирующей строки, заключаешь ее в фигурные скобки. Ты пишешь имя переменной или любое другое выражение. Затем ты добавляешь двоеточие и используешь спецификатор формата (например, .1f) Точно так же, как и в старом методе.

Вот и все. Ты можешь поместить отформатированную строку в переменную, так же как и при старом форматировании с помощью знака процента:

```
distance = 149597870700
myString = f'Солнце находится на расстоянии {distance:.4e} метров
           от Земли'
```

И поскольку ты больше не используешь знак **%** для обозначения форматирующих строк, тебе не нужно делать ничего дополнительного, чтобы напечатать знак процента:

```
>>> print(f'Я выполнил тест по математике на {math:.1f}%')
Я выполнил тест по математике на 87%
```

Есть еще одно различие между форматирующими строками, начинающимися с буквы **f**, и строками, начинающимися со знака **%**. Вместо того чтобы использовать спецификатор формата **i**, ты просто пишешь выражение, которое хочешь вывести, без спецификатора формата:

```
>>> print(f'Солнце находится на расстоянии {distance} метров от Земли')
Солнце находится на расстоянии 149597870700 метров от земли
```

Некоторые программисты скажут, что лучше использовать букву **f**, особенно в Python 3. Но ты можешь использовать любой способ, который захочешь. Во всех примерах этой книги применяется знак **%**.

Думай как программист (на Pygame)

На самом деле некоторые программисты Python предпочитают *другой* способ форматирования строк. Метод **.format()** аналогичен строкам, начинающимся с буквы **f**, за исключением того, что вместо включения имени переменной непосредственно в строку ты используешь индекс и передаешь **.format()** значения, которые хочешь вставить.

Это выглядит примерно так:

```
>>> print(«Тест по математике я выполнил на
      {0:.1f}, а по физике на {1:.1f}».format
      (math, science))
```

К тому времени, когда эта книга будет напечатана, Python, вероятно, будет иметь еще более новые, захватывающие способы вставки значений в строки!

Форматирование строк

Когда ты впервые узнал о строках (в главе 2), мы говорили, что можно совместить две строки с помощью знака **+**:

```
>>> print('кот' + 'пес')
котпес
```

Сейчас ты узнаешь больше о возможностях строк.

Строки в Python на самом деле являются *объектами* (видишь, все является объектом...), и у них есть свои собственные *методы* для поиска, разделения и совмещения. Они известны как *строковые методы*. Способ **format()**, о котором ты читал выше, тоже к ним относится.

Разделение строк

Иногда тебе нужно разбить длинную строку на несколько коротких. Обычно тебе нужно это сделать в определенном месте строки, например там, где появляется особый символ. Допустим, общепринятый способ хранения данных в текстовом файле – разделение элементов с помощью запятой. У тебя может быть похожий список имен:

```
>>> name_string = 'Даша,Артем,Борис,Максим,Паша,Саша,Ваня,Ол
```

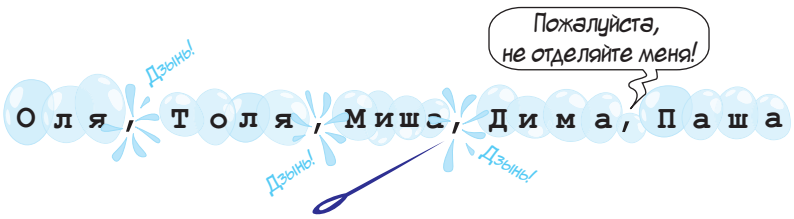
Предположим, ты хочешь поместить эти имена в список так, чтобы каждый элемент содержал одно имя. Тебе нужно разбить строку в тех местах, где встречается запятая. Метод Python для выполнения этой задачи называется **split()**, и работает он так:

```
>>> names = name_string.split(',')
```

Ты указываешь, какой символ использовать в качестве маркера разбиения, и метод выдает тебе список, являющийся оригинальной строкой, разбитой на части. Если мы выведем на экран результат этого действия, большая строка из имен будет разделена на отдельные элементы в списке:

```
>>> print(names)
['Оля','Толя','Миша','Дима','Паша','Саша','Ваня','Федя']
>>> for name in names:
        print(name)

Оля
Толя
Миша
Дима
Паша
```

```
Саша
Ваня
Федя
```



Ты можешь указать более одного символа в качестве маркера разбиения. Например, можно использовать слово **'Паша,'** в качестве маркера, и у тебя получится следующее:

```
>>> parts = name_string.split('Паша,')
>>> print(parts)
['Оля, Толя, Миша, Дима', 'Саша,Ваня,Федя']
>>> for part in parts:
        print(part)

Оля, Толя, Миша, Дима
Саша,Ваня,Федя
```

В этот раз строка была разделена на две части: все имена до Паши и все имена после Паши. Обрати внимание, что Паши нет в списке, потому что маркер разбиения отбрасывается.

Тебе нужно знать еще кое-что. Если ты не укажешь маркер разбиения, Python разобьет строку по *пробельным символам*:

```
>>> names = name_string.split()
```

Пробелы – это все промежутки, отступы и переходы строки.

Объединение строк

Мы только что узнали, как делить строки на меньшие части. А как насчет объединения двух и более строк в большую строку? Мы уже знаем об этом из главы 2, строки можно объединять с помощью оператора `+`. Похоже на сложение строк вместе, но называется это *конкатенацией*.

Есть еще один способ объединения строк. Можно использовать функцию **`join()`**. Ты указываешь, какие строки объединить и какие символы (если требуется) вставить между частями при объединении. Это функция, обратная функции **`split()`**. Ниже приведен пример в интерактивном режиме:

```
>>> word_list = ['Меня', 'зовут', 'Александра']
>>> long_string = ' '.join(word_list)
>>> long_string
'Меня зовут Александра'
```

Признаем, выглядит несколько странно. Символы, которые будут находиться между элементами объединенной строки, пишутся *перед* функцией **`join()`**. В этом случае нам нужны были пробелы между словами, поэтому мы использовали **`' '.join()`**. Это отличается от того, что ты ожидал, но данный метод работает именно так.

Взгляни на следующий пример:

```
>>> long_string = ' ГАВ ГАВ '.join(word_list)
>>> long_string
'Меня ГАВ ГАВ зовут ГАВ ГАВ Александра'
```

То есть строка перед функцией **`join()`** использована в качестве «клея», чтобы соединить остальные строки.

Поиск строк

Допустим, ты хочешь написать программу для мамы, которая выбирает рецепты и выводит их на экране в GUI. Тебе нужно поместить ингредиенты в одном месте, а инструкции – в другом. Давай представим, что рецепт выглядит примерно так:

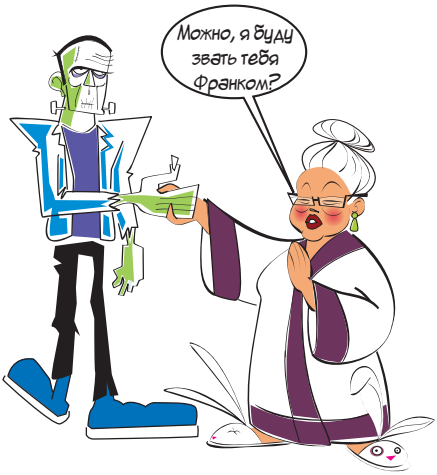
```
Шоколадный торт
Ингредиенты:
1 яйцо
180 г муки
200 г сахара
2 ст. л. какао

Инструкции:
Разогреть духовку до 200°
Смешать все ингредиенты вместе
Выпекать 30 минут
```

Допустим, все строки рецепта находятся в *списке* и каждая строка – отдельный элемент в списке. Как ты найдешь раздел с инструкциями? В Python есть пара методов, которые тебе помогут.

Метод **`startswith()`** определяет, начинается ли строка с определенного символа или символов. Проще всего показать на примере. Попробуй в интерактивном режиме:

```
>>> name = «Франкенштейн»
>>> name.startswith('Ф')
True
```

```
>>> name.startswith("Франк")
True
>>> name.startswith(«Фил»)
False
```



Имя «Франкенштейн» начинается с буквы «Ф», поэтому первое утверждение было верным. Имя «Франкенштейн» начинается с букв «Франк», поэтому второе утверждение тоже было верным. Имя «Франкенштейн» *не* начинается с букв «Фил», поэтому третье утверждение было ложным.

Поскольку метод **`startwith()`** возвращает значение **`True`** или **`False`**, можно использовать его в сравнениях или утверждениях **`if`**, например так:

```
>>> if name.startswith(«Франк»):
        print(«Можно я буду звать тебя Франком?»)
```

Существует похожий метод – **`endswith()`**, который определяет именно то, что заложено в его названии, работает с окончаниями слов:

```
>>> name = «Франкенштейн»
>>> name.endswith('н')
True
>>> name.endswith('штейн')
True
>>> name.endswith('штайн')
False
```

Теперь вернемся к нашей имеющейся задаче... Если ты хочешь найти начало инструкций в рецепте, можно сделать следующее:

```
i = 0
while not lines[i].startswith("Инструкции"):
    i = i + 1
```

Этот цикл выполняется, пока не найдет строку, которая начинается со слова «Инструкции». Напомним, что **`lines[i]`** означает, что **`i`** – это *индекс* для **`lines`**. Поэтому ты начинаешь с **`lines[0]`** (первая строка), затем идет **`lines[1]`** (вторая) и т. д. Когда цикл **`while`** закончится, **`i`** будет равно индексу строки, которая начинается со слова «Инструкции», именно та, что ты ищешь.

Поиск в любом месте строки: `in` и `index()`

Методы **`startswith()`** и **`endswith()`** особенно хороши в поиске символов в начале и конце строки. Но что, если тебе нужно найти что-то в середине?

Допустим, у тебя есть несколько строк с адресами:

```
657 Мира улица
46 Ленина улица
95 Кирова проспект
```

Может быть, ты хочешь найти адрес со словом «улица». Ни одна строка не начинается и не заканчивается им, но две из них содержат это слово. Как их найти?

На самом деле мы уже знаем, как это сделать. Когда мы говорили о *списках* (глава 12), то упоминали, как проверить, содержит ли список элемент:

```
if someItem in my_list:
    print(«Нашел!»)
```

Мы использовали ключевое слово **in**, чтобы проверить, есть ли в списке определенный элемент. Это ключевое слово работает и со строками. Строка на самом деле – это список символов, поэтому можно сделать следующее:

```
>>> addr1 = '657 Мира улица'
>>> if 'Мира' in addr1:
        print(«В этом адресе есть улица 'Мира'.»)
```

СЛОВАРИК

Когда ты ищешь меньшую строку, например «улица», в большей строке, например «657 Мира улица», меньшая строка называется *подстрокой*.

Ключевое слово **in** сообщает нам, существует ли подстрока *где-то* в строке, которую мы проверяем. Оно не говорит нам, где. Чтобы узнать это, нам нужен метод **index()**. Как и в случае со списками, **index()** сообщает нам, где начинается меньшая строка в большей строке.

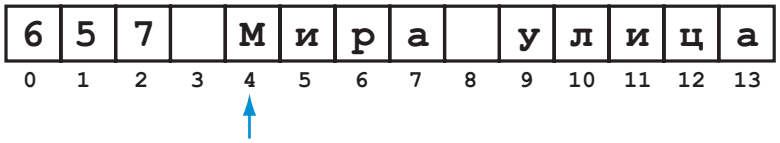
Ниже приведен пример:

```
>>> addr1 = '657 Мира улица'
>>> if 'Мира' in addr1:
        position = addr1.index('Мира')
        print(«найдена улица 'Мира' с индексом», position)
```

Если выполнить этот код, ты получишь следующее:

```
найдена улица 'Мира' с индексом 4
```

Слово «улица» начинается на 4-й позиции строки «657 Мира улица». Как и в списках, индексы (или позиции) букв в строке начинаются с 0, поэтому «у» имеет индекс 4.



Обрати внимание, что перед использованием **`index()`** мы проверили наличие подстроки «улица» в большей строке с помощью **`in`**. Потому что если использовать **`index()`** в строке, где этого элемента *нет*, ты получишь сообщение об ошибке. Проверка с помощью **`in`** исключает возможность ошибки. То же самое мы делали со списками в главе 12.

Удаление части строки

Довольно часто тебе будет требоваться удалить или *вырезать* часть строки. Обычно тебе нужно вырезать что-то в конце ее, например символ перехода строки или некоторые лишние пробелы. В Python строковый метод **`strip()`** служит именно для этого. Ты просто указываешь, что нужно удалить:

```
>>> name = 'Юрий Семецкий'
>>> short_name = name.strip('кий')
>>> short_name
'Юрий Семец'
```

В этом случае мы вырезали «кий» в конце имени. Если бы такого слога в конце не было, ничего не было бы обрезано:

```
>>> name = 'Барт Симпсон'
>>> short_name = name.strip('кий')
>>> short_name
'Барт Симпсон'
```

Если ты не указываешь функции **`strip()`**, что именно обрезать, она удалит все пробелы. Как мы и говорили ранее, это включает в себя табуляцию, пробелы между словами и переходы строк. Поэтому если у нас есть лишние пробелы, можно сделать так:

```
>>> name = "Юрий Семецкий      "
>>> short_name = name.strip()
>>> short_name
'Юрий Семецкий'
```

Дополнительные пробелы в конце имени

Обрати внимание, что дополнительные пробелы после имени были удалены. Хорошо, что не нужно указывать, сколько именно пробелов удалить. Метод удалит все пробелы в конце строки.

Изменение регистра

Мы хотим показать тебе еще два строковых метода. Они предназначены для изменения регистра строки с верхнего на нижний и наоборот. Иногда тебе нужно сравнить две строки, например «Привет» и «привет», и ты хочешь знать, написаны ли они одними буквами, даже если регистр разный. Первый способ это сделать – перевести все буквы в обеих строках в нижний регистр и выполнить сравнение.

Для этого в Python есть строковый метод. Он называется **`lower()`**. Выполни код в интерактивном режиме:

```
>>> string1 = "Привет"
>>> string2 = string1.lower()
>>> print(string2)
привет
```

Существует такой же метод **`upper()`**:

```
>>> string3 = string1.upper()
>>> print(string3)
ПРИВЕТ
```

Ты можешь сделать копии оригинальных строк полностью в верхнем или нижнем регистре и сравнить, одинаковы ли они, игнорируя регистр.

Что ты узнал?

В этой главе ты узнал:

- как настроить вертикальное выделение пробелами (добавлением или удалением переходов строк);
- как настроить горизонтальное выделение пробелами с помощью табуляции;
- как вывести на экран разные форматы чисел с помощью форматирующих строк;
- три способа форматирования строк: знак **`%`**, буква **`f`** и метод **`format()`**;
- как разбивать строки с помощью метода **`split()`** и объединять с помощью функции **`join()`**;
- как искать строки с помощью **`startswith()`**, **`endswith()`**, **`in`** и **`index()`**;
- как удалять символы в конце строки с помощью метода **`strip()`**;
- как превратить все буквы в строке в прописные и строчные с помощью методов **`upper()`** и **`lower()`**.

Проверь свои знания

1. Если у тебя есть два разных утверждения **`print`**, например:

```
print(«Как»)
print(«тебя зовут?»)
```

   как ты сделаешь так, чтобы текст был выведен в одной строке?
2. Как добавить дополнительные пустые строки при выводе?
3. Какой особый код вывода используется для выравнивания элементов по колонкам?
4. Какая форматирующая строка используется для вывода числа в E-нотации?

Попробуй самостоятельно

1. Напиши код программы, которая спрашивает имя, возраст и любимый цвет пользователя, затем выводит все в одном предложении. Вывод программы должен выглядеть так:

```
>>>
RESTART: C:/HelloWorld/Примеры/sentence.py
Как тебя зовут? Саша
Сколько тебе лет? 12
Какой твой любимый цвет? зеленый
Тебя зовут Саша, тебе 12 лет, и тебе нравится зеленый цвет.
```

2. Помнишь нашу программу с таблицей умножения из главы 8 (листинг 8.5)? Напиши улучшенную версию, которая использует табуляцию для выравнивания содержимого.
3. Напиши код программы, которая вычисляет дробную часть числа 8 (1/8, 2/8, 3/8...до 8/8) и выводит ее на экран с тремя знаками после запятой.

ГЛАВА 22

Ввод и вывод файла

Ты когда-нибудь задумывался, как твоя любимая игра помнит игровой счет даже после выключения компьютера? А как браузер запоминает твои любимые сайты? В данной главе ты узнаешь об этом.

Мы несколько раз говорили о том, что у программ есть три основных аспекта: ввод, обработка и вывод. До сих пор ввод поступал в основном от пользователя, с клавиатуры или мыши. Вывод напрямую отправлялся на экран (или на динамики в случае со звуком). Но иногда нам нужно использовать ввод из других источников. Довольно часто программам нужно использовать ввод информации, хранящейся в другом месте, а не введенной пользователем при выполнении. Некоторые программы требуют ввод *файла* с жесткого диска компьютера.

Например, если ты сделаешь игру «Виселица», твоей программе нужен будет список слов для выбора загаданного слова. Этот список слов должен где-то храниться, возможно в файле, который будет идти вместе с программой. Программа должна будет открывать этот файл, читать список и выбирать слово.

То же самое верно и для вывода. Иногда вывод программы должен где-то храниться. Все переменные, которые она использует, временные – они теряются, когда программа завершает свое выполнение. Если ты хочешь сохранить какую-то информацию для дальнейшего использования, тебе нужно сохранить ее в чем-то более постоянном, например на жестком диске. К примеру, список рекордов игры – ты должен хранить его в файле, чтобы при следующем запуске программы она могла прочитать файл и показать очки.

В этой главе ты узнаем, как открывать файлы и как *читать* и *записывать* информацию в них (*получать* информацию и *хранить* ее).

Что такое файл?

Перед тем как приступить к открытию, чтению и записи в файлы, нужно знать, что же такое файл.

Мы говорили, что компьютеры хранят информацию в *двоичном* формате, который использует единицы и нули. Каждая единица или ноль называется *битом*, а группа из восьми битов называется *байтом*. *Файл* – это коллекция байтов, которая имеет имя и хранится на жестком диске, CD, DVD, Blu-ray-диске, Flash-накопителе или другом виде запоминающего устройства.

Файлы могут хранить много разной информации. Файл может содержать текст, изображение, музыку, компьютерную программу, список телефонных номеров и т. д. Все, что хранится на жестком диске компьютера, хранится в файлах. Программы состоят из одного и более файлов. Операционная система твоего компьютера (Windows, macOS или Linux) требует очень много файлов для выполнения.

У файлов есть следующие свойства:

- имя;
- тип, который определяет, какой вид данных в файле (изображение, звук, текст);
- расположение (где он хранится);
- размер (сколько байтов в файле).

Имена файлов

В большинстве операционных систем (включая Windows) часть имени файла используется, чтобы указать, какой тип данных в нем. Имена файлов обычно имеют, по крайней мере, одну «точку» (.). Часть именно после точки и сообщает тип файла. Эта часть называется *расширением*.

Ниже представлено несколько примеров:

- в файле *my_letter.txt* расширением является *.txt*, что значит «text» (текст), поэтому файл наверняка содержит текст;
- в файле *my_song.mp3* расширением является *.mp3*, поэтому это тип звукового файла;
- в файле *my_program.exe* расширение *.exe*, что значит «executable» (исполняемый или выполнимый). Как я упомянул в главе 1, «выполнение» – это запуск программы. Поэтому этот тип файла – программа, которую можно запустить;
- у файла *my_cool_game.py* расширение *.py*, что означает программу на языке Python.

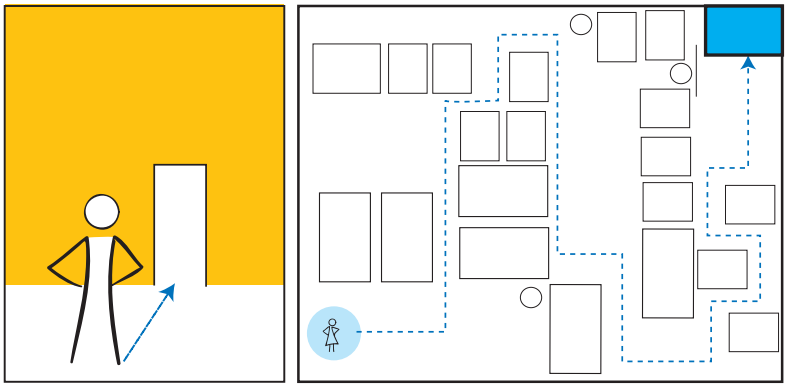
В macOS программные файлы (файлы, которые содержат программы) имеют расширение *.app*, что значит «application» (приложение), синоним слова «программа».

Важно знать, что ты можешь присвоить файлу любое имя и использовать любое расширение. Ты можешь создать текстовый файл (в программе «Блокнот», например) и назвать его *my_notes.mp3*. Это не сделает файл звуковым. В нем все равно есть только текст, поэтому это текстовый файл. Ты просто присвоил ему расширение, с которым он *выглядит* как звуковой файл, что может ввести в заблуждение компьютер и людей. Такой файл попросту не откроется в музыкальной программе и выведет ошибку. При присвоении имени файлу нужно использовать то расширение, которое ему подходит.

Расположение файлов

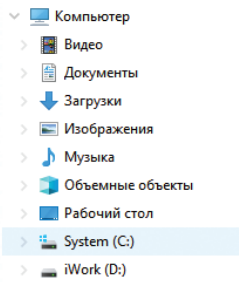
До сих пор мы работали с файлами, которые хранились в той же папке, где и сама программа. Мы не беспокоились о том, как найти файл, потому что он был под рукой.

Представь, что ты находишься в своей квартире и тебе не нужно волноваться, как найти ванную – она здесь. Но если ты в другом помещении, доме, городе, то поиск ванной становится более сложным!

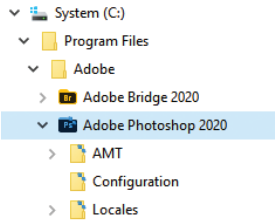


Каждый файл нужно *где-то* хранить, поэтому вместе с именем у каждого файла есть и расположение. Жесткие диски и другие накопители организуют файлы в *папки*, или *каталоги*. Это два названия одного и того же. Они группируют файлы вместе. Способ организации папок или каталогов называется *структурой папок*, или *структурой каталогов*.

В Windows каждый раздел носителя информации имеет свою букву, например *C* для жесткого диска или *E* для Flash-накопителя. В macOS и Linux каждый раздел имеет имя (например, *hda* или *FLASH DRIVE*). Каждый раздел может быть разделен на папки, например *Музыка*, *Картинки*, *Программы*. Если взглянуть на них с помощью средства просмотра файлов, например Проводника Windows, увидим следующее:



Папки могут содержать другие папки, а те, в свою очередь, другие папки, и т. д. Справа представлен пример трех уровней папок:



Первый уровень – *C:*. Следующий уровень – *Program Files*. Затем папка *Adobe*. Внутри нее папки *Adobe Bridge 2020* и *Adobe Photoshop 2020*. И последний уровень – *AMT*, *Configuration* и другие папки внутри *Adobe Photoshop 2020*.

СЛОВАРИК

Папки, находящиеся в других папках, называются *подпапками*, или *вложенными* папками. Если ты используешь термин «каталог», то *подкаталогами*.

Когда ты пытаешься найти папку в Проводнике Windows (или другом обозревателе файлов), папки выглядят как ветки деревьев. «Корень» – это сам диск, например *C:* или *E:*. Каждая главная папка похожа на толстую ветвь дерева. Каталоги в каждой главной папке похожи не маленькие ветки и т. д.

Но когда тебе нужно получить доступ к файлам из программы, идея с деревьями не совсем подходит. Твоя программа не может щелкать по папкам мышью и переходить от дерева к дереву в поиске отдельных файлов. Ей нужен более точный путь. К счастью, есть другой способ представить структуру дерева. Посмотри на адресную строку в Проводнике, когда щелкаешь по разным файлам и подкаталогам, и увидишь что-то подобное:

```
E:\Music\Old Music\Really old music\my_song.mp3
```

Это *путь*. Путь файла – это описание его местонахождения в системе папок. Именно этот путь читается так:

- начать с диска *E:*;
- перейти в папку *Music*;
- в папке *Music* перейти в подкаталог *Old Music*;
- в подкаталоге *Old Music* перейти в подкаталог *Really old music*;
- в подкаталоге *Really old music* найти файл *my_song.mp3*.

Ты можешь добраться до любого файла на компьютере с помощью такого пути. Таким образом программы находят и открывают файлы. Например:

```
image_file = «C:/program files/HelloWorld/examples/beachball.png»
```

Ты всегда можешь найти файл с помощью его имени пути. Это имя включает в себя все названия папок до корневого каталога (диск, например, *C:*). Имя в этом примере – это полное путевое имя файла.

Слеш или обратный слеш?

Очень важно, чтобы слеши (`\` и `/`) были указаны в нужном направлении. Операционная система Windows принимает и обычный слеш (*/*), и обратный (\) в именах путей, но если использовать что-то, подобное *c:\test_results.txt*, в программе на Python, символы `\t` приведут к проблеме. Напомним, что в главе 21 мы говорили об особых символах для форматирования вывода, например табуляции с помощью `\t`. Поэтому нужно избегать символа `\` в именах путей. Python (и Windows) отнесутся к `\t` как к символу табуляции, а не как к части имени файла. Используй вместо него `/`.

Или можно использовать двойной обратный слеш:

```
image_file = «C:\\program files\\HelloWorld\\images\\beachball.png»
```

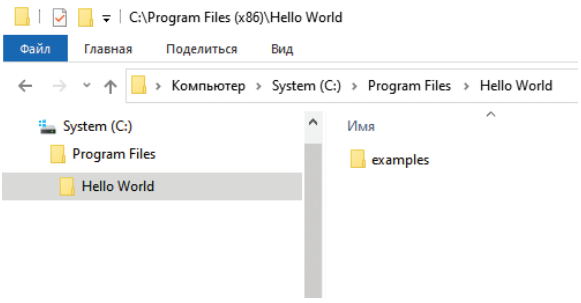
Напомним, что если ты хочешь вывести символ `\`, тебе нужно указать еще один символ слеша перед ним. То же касается и имен файлов. Но я рекомендую использовать `/`.

Иногда тебе не потребуется указывать все имя пути целиком. В следующем разделе я расскажу о поиске файла, если ты уже находишься на полпути к нему.

Как узнать, где ты находишься

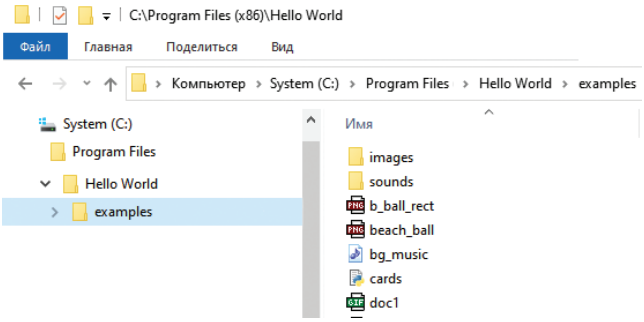
Большинство операционных систем (включая Windows) имеют понятие *рабочего каталога*, или, иногда, *текущего рабочего каталога*. Это папка в дереве папок, с которой ты в данный момент работаешь.

Представим, что ты начал работать в корневом каталоге (*C:*) и перешел к ветке *Hello World*. Твой текущий каталог – *C:/Program Files/Hello World*.



Теперь, чтобы найти файл *beach_ball.png*, тебе нужно перейти к ветке *Examples*. То есть путь – *Examples/beach_ball.png*. Поскольку ты уже был на полпути к файлу, тебе нужна только оставшаяся часть пути, чтобы попасть по назначению.

Помнишь, как в главе 19 мы открывали звуковые файлы *splat.wav* и т. д.? Мы не использовали весь путь. Потому что сказали тебе скопировать звуковые файлы в папку с программой. Если бы ты заглянул в Проводник, то увидел бы что-то похожее:



Обрати внимание, что у нас файлы Python (расширение *.py*) находятся в той же папке, что и звуковые файлы (расширение *.wav*). Когда программа на Python выполняется, ее *рабочим каталогом* является папка с файлом *.py*.

Если ты сохранил программу в папке *E:/programs* и запустил ее, то *рабочим каталогом* программы будет *E:/programs*. Если звуковой файл сохранен в той же папке, твоей программе потребуется только его имя, чтобы использовать файл. Ей не нужен путь, чтобы найти его, потому что файл уже здесь. Ты можешь сделать так:

```
my_sound = pygame.mixer.Sound(«splat.wav»)
```

Обрати внимание, что нам не нужно целое путевое имя звукового файла (которое представляет собой *E:/programs/splat.wav*). Мы используем только имя файла без его пути, потому что файл находится в той же папке, что и программа, его использующая.

Хватит говорить о путях!

Это все, что я хотел сказать о путях и расположении файлов. Саму тему папок, каталогов, путей, рабочих каталогов и т. д. некоторые пользователи находят несколько сложной, и нужно гораздо больше страниц, чтобы охватить ее целиком. Но эта книга о программировании, а не об операционных системах, файловых структурах или путях, поэтому, если ты испытываешь потребность, спроси родителей, учителя или любого знающего человека.

Во всех остальных примерах с использованием файлов они находятся в одной папке с программой, поэтому нам не нужно беспокоиться об их пути.

Открытие файла

Перед тем как открыть файл, тебе нужно понимать, что ты будешь с ним делать:

- если ты будешь использовать файл в качестве *ввода* (просматривать информацию, не меняя ее), ты открываешь его для *чтения*;
- если ты будешь *создавать* новый файл или *заменять* существующий файл новым, ты открываешь его для *записи*;

- если ты будешь *добавлять* информацию в существующий файл, ты открываешь его для *дополнения*. (Напомним, что в главе 12 мы говорили, что *дополнение* значит добавление чего-то.)

Когда ты открываешь файл, то создаешь *объект файла* в Python. (Видишь, я же говорил, что многое в Python является объектом.) Объект файла создается с помощью функции **`open()`** с именем самого файла:

```
my_file = open('my_filename.txt','r')
```

Имя файла – это *строка*, поэтому вокруг него нужны кавычки. Символы **`'r'`** означают, что мы открываем файл для чтения. Далее ты узнаешь об этом больше.

Важно понимать разницу между *объектом файла* и *именем файла*. *Объект файла* – это то, что мы используем в программе для доступа к нему. *Имя файла* – это то, как операционная система Windows (Linux, macOS) называет файл на диске.

То же самое происходит и с людьми. У нас есть разные имена, которые мы используем в разных местах. Если твоего учителя зовут Федор Борисович Сидоров, то ты его наверняка называешь Федором Борисовичем. Его друзья называют его Борей, а именем его компьютера может быть Ф. Б. Сидоров. У файлов есть имя, которое используется операционной системой для его хранения на диске (имя файла), и есть имя, используемое программой при работе с ним (объект файла).

Оба имени – имя объекта и имя файла – могут не совпадать. Ты можешь назвать объект как угодно. Например, у нас есть текстовый файл, названный *notes.txt*. Мы можем сделать следующее:

```
notes = open('notes.txt', 'r')
```

Объект файла *Имя файла*

Или следующее:

```
some_crazy_stuff = open("notes.txt", 'r')
```

Объект файла *Имя файла*

Когда мы открыли файл и создали его объект, нам не нужно его имя. Мы выполняем действия с ним в программе с помощью объекта файла.

Чтение файла

Как я говорил ранее, мы открываем файл и создаем его объект с помощью функции **`open()`**. Это одна из встроенных функций Python. Чтобы открыть файл для чтения, используй в качестве второго аргумента значение **`'r'`**:

```
my_file = open('notes.txt', 'r')
```

Если ты попробуешь открыть для чтения несуществующий файл, получишь сообщение об ошибке. (Ты же не можешь прочесть то, чего нет, верно?)

В Python есть еще несколько встроенных функций для получения информации файла и ее использования в программе, когда файл уже открыт. Чтобы прочитать строки текста из файла, можно использовать метод **`readlines()`**:

```
lines = my_file.readlines()
```

Этот метод проанализирует весь файл и составит список, в котором каждая строка текста будет отдельным элементом. Допустим, файл *notes.txt* содержит небольшой список дел:

```
Помыть машину
Заправить кровать
Собрать деньги
```

Можно использовать программу, например «Блокнот», для создания этого файла. Фактически почему бы тебе не создать подобный файл прямо сейчас? Назови его *notes.txt* и сохрани в папке с программами Python.

Чтобы открыть этот файл с помощью программы и прочитать его, код будет следующим.

Листинг 22.1. Открытие и чтение файла

```
my_file = open('notes.txt', 'r')
lines = my_file.readlines()
print(lines)
```

Вывод будет примерно таким (в зависимости от того, что ты написал):

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_22-1.py
['Помыть машину\n', 'Заправить кровать\n', 'Собрать деньги']
```

Строки текста были прочитаны в файле и помещены в список, который мы назвали **`lines`**. Каждый элемент списка – строка, содержащая одну строку файла. Обрати внимание на символы **`\n`** в конце первых двух строк. Это символы *перехода строки*, которые разделяют строки в файле. Когда мы нажимали клавишу **Enter** в процессе создания файла, они появились. Если ты нажал клавишу **Enter** и после последней строки, то увидишь третий символ **`\n`** после третьего элемента.

В программу нужно добавить еще кое-что. Когда мы закончили работу с файлом, его нужно закрыть:

```
my_file.close()
```

Картер, если другая программа захочет использовать файл, а он будет открыт в нашей программе, то она не сможет получить к нему доступ. Лучше закрывать файлы, когда заканчиваешь работу с ними.

Когда файл в нашей программе стал списком строк, мы можем делать с ним все, что угодно. Этот список такой же, как и любой другой список Python, поэтому мы можем создать цикл по нему, сортировать его, добавлять элементы, удалять элементы и т. д. Строки такие же, как и любые другие строки Python, мы можем выводить их на экран, конвертировать в **`int`** или **`float`** (если они содержат числа), использовать в качестве меток в GUI или совершать любое другое действие.

Чтение одной строки единовременно

Метод **`readlines()`** читает все строки файла с начала до конца. Если ты хочешь прочесть только одну строку единовременно, можно использовать метод **`readline()`**:

```
first_line = my_file.readline()
```

Этот код читает только первую строку файла. Если использовать метод **`readline()`** снова в той же программе, Python запомнит, где он остановился. Поэтому при повторном использовании ты получишь вторую строку файла. В листинге 22.2 показан пример.

Листинг 22.2. Использование метода **`readline()`** неоднократно

```
my_file = open('notes.txt', 'r')
first_line = my_file.readline()
second_line = my_file.readline()
print("первая строка = ", first_line)
print("вторая строка = ", second_line)
my_file.close()
```

Вывод программы будет выглядеть так:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_22-2.py
первая строка = Помыть машину

вторая строка = Заправить кровать
```

Метод **`readline()`** читает по одной строке, поэтому он не помещает результат в список. Каждый раз при его использовании ты получаешь одну строку.

Возвращение к началу

Если ты использовал метод **`readline()`** несколько раз и хочешь начать снова с начала файла, можно использовать метод **`seek()`**:

```
first_line = my_file.readline()
second_line = my_file.readline()
my_file.seek(0)

first_line_again = my_file.readline()
```

Метод **`seek()`** позволяет перейти к тому месту файла, которое ты укажешь. Число в скобках – количество байтов от начала файла. Поэтому, написав в скобках 0, мы вернулись в начало.

Текстовые и двоичные файлы

Все примеры открытия и чтения строк текста до сих пор предполагали одно: *в файле действительно был текст*! Напомним, что текст – это только один из видов содержимого файлов. Программисты объединяют все остальные типы файлов вместе и называют их *двоичными*. Есть два основных вида файлов, которые можно открыть:

- *текстовые файлы* – в них есть текст, буквы, числа, пунктуационные знаки и другие особые символы, например переход строки;
- *двоичные файлы* – в них нет текста. В них может быть музыка, изображения или данные, но поскольку в них нет текста, в них нет и строк, а следовательно, и переходов строк.

Это значит, что нельзя использовать методы **`readline()`** или **`readlines()`** с двоичным файлом. Если попробовать прочесть «строку» из файла *.wav*, ты получишь, вероятнее всего, целый абзац абракадабры:

```
>>> f = open('splat.wav', 'r')
>>> print(f.readline())
RIFFö▲ WAVEfmt ► ☺ ☺ "V "V datap?
ÇÇÇÇÇÇÇÇüÇÇÇÇÇÇÇÇÇÇÇÇÇÇÇ?ÇÇ⌂⌂⌂⌂⌂Ç⌂⌂⌂⌂Ç
ÇÇÇ⌂⌂ÇÇÇÇÇÇÇÇÇÇÇÇÇÇÇÇ⌂Ç⌂Ç⌂ÇÇÇüÇÇÇÇÇÇÇÇÇÇÇÇÇÇÇÇ⌂Ç⌂ÇÇÇÇüüÇÇÇ⌂ÇÇ⌂Ç⌂ÇÇÇÇ⌂⌂ÇÇüéééÇ
zvvy{|Çâçïê}trv|äëïîèå~ut|⌂yrqrtxÇîÖÆæäàütvçÆÄ|mlfWR]jnmpxüêÅ °ƒârà6«¼Ö}`
ORj⌂{hZZgwàëy{äæá⌐¿ÿézåÿèmWLISjÇàzrvÇüytv~üÇ}yrifjt}äêèêëÄöÉémSCFZlrtyéïö
¥ñ¬½⌐ñ¢ÆÄìÅô⌂ÆÄÅæ|åÜ⌐¬ÿüpd\UME@;99:>EJMW]YTZfuçòƒ⌐▓╢└──┼────└┐⌐¢ðë~{|{yxzz
uiZNGHLSbs⌂~wrnf\TPQU]`jvàæÉ⌂osÇïôæä}üàëÆ┼
```

Что-то в самом начале отдаленно напоминает текст, но потом все становится еще хуже. Так происходит потому, что в файле *.wav* нет текста, в нем содержится звук. Методы **`readline()`** и **`readlines()`** читают только текстовые файлы.

В большинстве случаев при необходимости использования двоичных файлов ты сделаешь это с помощью Pygame или другого модуля для загрузки файла, например как мы делали в главе 19:

```
pygame.mixer.music.load('bg_music.mp3')
```

В этом случае Pygame заботится об открытии файла и чтении двоичных данных (которые в этом случае создают звук). Но если ты хочешь открыть двоичный файл самостоятельно, то можешь сделать это, добавив *b* к режиму файла:

```
my_music_file = open('bg_music.mp3', 'rb')
```

Символы **`'rb'`** означают, что мы открываем файл для *чтения* в *двоичном* режиме.

В нескольких предыдущих разделах мы узнали, как получить информацию из файла в нашей программе, что называется *чтением* файлов. Далее ты узнаешь о помещении информации из программы в файл. Это называется *записью* в файл.

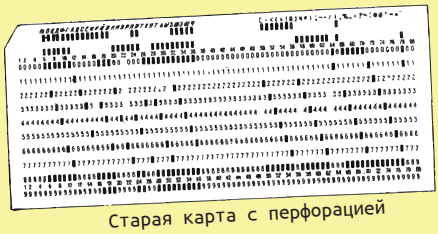
Запись в файл

Если тебе нужно сохранить информацию из программы на более долгий срок, можно, конечно, записать ее на бумаге. Но этот способ уничтожает весь смысл использования компьютера!

Лучше сохранить информацию на жестком диске, чтобы даже при незапущенной программе – даже при выключенном компьютере – твои данные сохранились и ты мог получить к ним доступ позднее. Ты уже делал это много раз. Когда ты сохраняешь реферат для школы, рисунок, песню или программу на языке Python, ты сохраняешь данные на жестком диске.

В СТАРЫЕ ДОБРЫЕ ВРЕМЕНА

Раньше все, что у нас было, – это бумага! Не было мониторов, принтеров и даже клавиатур. Код «оформлялся» перфорацией в картах. Потом эта пачка перфокарт скармливалась большой машине, которая превращала дырочки в электрические сигналы, которые компьютер мог понять. Иногда требовались дни, чтобы получить ответ! Это было ужасно!



Старая карта с перфорацией

Как я говорил ранее, есть два способа поместить информацию в файл:

- *запись* – создание нового файла или перезапись существующего;
- *дополнение* – добавление информации в существующий файл и сохранение уже бывшей в нем информации.

Чтобы записать или дополнить файл, тебе нужно его сначала открыть. Для этого используется, как и ранее, функция **`open()`**, но второй параметр становится другим:

- для чтения используй **`'r'`** в качестве режима файла:

```
my_file = open('new_notes.txt', 'r')
```

- для записи – **`'w'`**:

```
my_file = open('new_notes.txt', 'w')
```

- для дополнения – **`'a'`**:

```
my_file = open('notes.txt', 'a')
```

Если ты используешь режим дополнения, файл уже должен существовать на жестком диске, иначе ты получишь сообщение об ошибке. Это произойдет потому, что *дополнение* осуществляется в уже имеющийся файл.

Картер *снова* прав!

Если ты используешь режим записи, есть два варианта:

- файл уже существует, и информация в нем будет потеряна и заменена новой;
- файл не существует, будет создан новый файл с указанным именем, и информация будет записана в него.

Давай рассмотрим примеры.

Дополнение файла

Сначала используем созданный нами файл *notes.txt* и *дополним* его чем-нибудь. Давай добавим еще одну строку «Потратить деньги». Если ты был внимателен в последнем примере с методом **`readlines()`**, ты мог заметить, что в нем не было **`\n`**, не было *перехода строки* в конце последней строки. Поэтому нам нужно добавить его и нашу новую строку. Чтобы записать строки в файл, мы используем метод **`write()`**, как показано в листинге 22.3.

Листинг 22.3. Использование режима дополнения

```
todo_list = open('notes.txt', 'a')      ← Открывает файл в режиме добавления
todo_list.write('\nПотратить деньги')   ← Закрывает файл
todo_list.close()                       ← Добавляет строку в конец
```

Когда мы читали файлы, то говорили, что нужно закрывать файл после работы с ним. Но еще важнее использовать функцию **`close()`** при *записи* в файл. Изменения в файле не должны быть сохранены, пока ты его не закроешь.

После запуска программы из листинга 22.3 открой файл *notes.txt* с помощью программы «Блокнот» (или другого текстового редактора) и посмотри, что в нем изменилось. Не забудь закрыть программу «Блокнот» по завершении.

Запись в файл с помощью режима записи

Теперь давай попробуем выполнить пример записи в файл с помощью соответствующего режима. Мы откроем файл, которого нет на диске. Набери код программы из листинга 22.4.

Листинг 22.4. Использование режима записи в файл

```
new_file = open("my_new_notes.txt", 'w')
new_file.write("Поужинать\n")
new_file.write(«Поиграть в футбол\n»)
new_file.write("Пойти спать")
new_file.close
```

Как узнать, сработал ли код? Проверь папку с сохраненной программой из листинга 22.4. Ты увидишь в ней файл *my_new_notes.txt*. Можешь открыть этот файл в программе «Блокнот» и проверить его содержимое. Ты увидишь следующее:

```
Поужинать
Поиграть в футбол
Пойти спать
```

Ты создал текстовый файл в этой программе и сохранил в нем текст. Этот текст находится на жестком диске и будет там вечно – пока работает жесткий диск, – если ты его не удалишь. У нас есть способ длительного хранения данных из наших программ. Теперь они могут оставить след в мире (или хотя бы на твоем диске). Все, что тебе нужно сохранить после завершения программы и выключения компьютера, можно поместить в файл.

Давай посмотрим, что случится с файлом, который уже был на диске, после применения режима записи. Помнишь файл *notes.txt*? Если запустить программу из листинга 22.3, увидишь следующее:

```
Помыть машину
Заправить кровать
Собрать деньги
Потратить деньги
```

Давай откроем этот файл в режиме записи и посмотрим, что будет. Код приведен в листинге 22.5.

Листинг 22.5. Использование режима записи в существующий файл

```
the_file = open('notes.txt', 'w')
the_file.write("Проснуться\n")
the_file.write(«Посмотреть мультики»)
the_file.close()
```

Запусти программу, а затем открой файл *notes.txt* в программе «Блокнот», чтобы увидеть его содержимое. Ты увидишь:

```
Проснуться
Посмотреть мультики
```

Предыдущий список в файле *notes.txt* был удален. Он был заменен новым содержимым из листинга 22.5.

Запись в файл с помощью команды `print()`

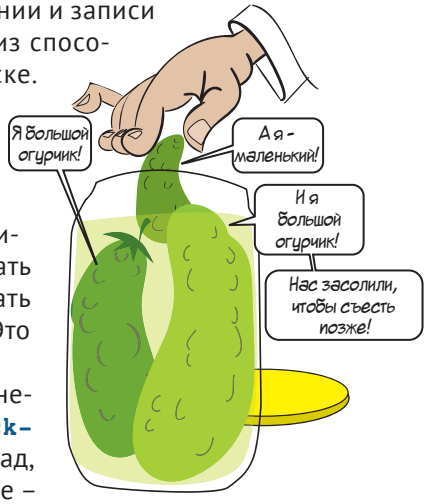
Выше мы записывали информацию в файл с помощью функции **`write()`**. Но также можно использовать команду **`print`** для записи в файл. Тебе все равно нужно открыть его в режиме записи или дополнения, но потом можно использовать команду **`print`**:

```
my_file = open("new_file.txt", 'w')
print("Привет, сосед!", file=my_file)
my_file.close()
```

Иногда удобнее использовать **`print()`** вместо **`write()`**, потому что **`print()`** выполняет некоторые действия автоматически, например конвертацию чисел в строки и т. д. Ты можешь сам решить, использовать **`print()`** или **`write()`** для записи текста в файл.

Сохранение данных в файл

В первой части этой главы мы говорили о чтении и записи текстовых файлов. Текстовые файлы – один из способов хранения информации на жестком диске. Но что, если тебе нужно сохранить список или объект?



Иногда элементы списка бывают строками, но не всегда. А как насчет хранения объектов? Ты должен иметь возможность конвертировать все свойства объекта в строки и записать их в текстовый файл, но тебе придется делать и обратное, чтобы вернуть объект из файла. Это может быть сложно.

К счастью, в Python есть способ сделать хранение списков и объектов проще. Это модуль **`pickle`**. Это забавное имя (от англ. *pickle* – маринад, соленье, соленые огурцы), но подумай: соленье – это способ хранения еды длительное время. В Python можно «засолить» данные и сохранить их на диске для дальнейшего использования. Имеет смысл!

Использование модуля `pickle`

Допустим, у нас есть список разных элементов:

```
my_list = ['Федор', 73, 'Привет', 81.9876e-13]
```

Чтобы использовать модуль `pickle`, его нужно сначала импортировать:

```
import pickle
```

Затем, чтобы «засолить» что-то, например список, используй функцию **`dump()`**. (Подумай об укладывании огурчиков в банку.) Функции **`dump()`** требуется объект файла, и мы знаем, как его создать:

```
pickle_file = open('my_pickled_list.pkl', 'wb')
```

Мы открываем его для записи с помощью **`'w'`**, потому что мы будем *хранить* данные в этом файле; символ `'b'` нужен для того, чтобы сказать Python, что мы будем хранить двоичные данные, а не текст. Ты можешь выбрать любое имя и расширение. Мы выбрали расширение *.pkl*, сокращение от слова pickle.

А затем мы укладываем наш список в файл:

```
pickle.dump(my_list, pickle_file)
```

Весь процесс показан в листинге 22.6.

Листинг 22.6. Использование модуля `pickle` для сохранения списка в файл

```
import pickle
my_list = ['Федор', 73, 'Привет', 81.9876e-13]
pickle_file = open('my_pickled_list.pkl', 'wb')
pickle.dump(my_list, pickle_file)
pickle_file.close()
```

Можно использовать этот же метод для сохранения любого вида данных в файл. Но как насчет обратного получения информации? Идем дальше.

Обратный процесс

В реальной жизни при засаливании продукта он остается соленым навсегда. Ты не можешь отменить это действие. Но в Python при «консервации» данных с помощью **`pickle`** можно обратить процесс и получить свои данные в начальном виде.

Для этого есть функция **`load()`**. Ты предоставляешь ей объект файла, который хранит консервированную информацию, и она отдает тебе данные в их начальном виде. Давай попробуем. Если ты запускал программу из листинга 22.6, тебе необходимо, чтобы файл *my_pickled.list.pkl* располагался в одной папке с программой. Теперь давай выполним код из листинга 22.7.

Листинг 22.7. Обратный процесс с помощью `load()`

```
import pickle
pickle_file = open('my_pickled_list.pkl', 'rb')
recovered_list = pickle.load(pickle_file)
pickle_file.close()

print(recovered_list)
```

Ты должен получить следующий список в качестве вывода:

```
['Федор', 73, 'Привет', 8.19876e-012]
```

Кажется, сработало! Мы вернули те же элементы, которые законсервировали. E-нотация выглядит несколько иначе, но это то же самое число, по крайней мере в нем 16 знаков после запятой. Разница состоит в *ошибке округления*, о которой мы говорили в главе 4.

Далее мы используем свои новые знания для создания новой игры.

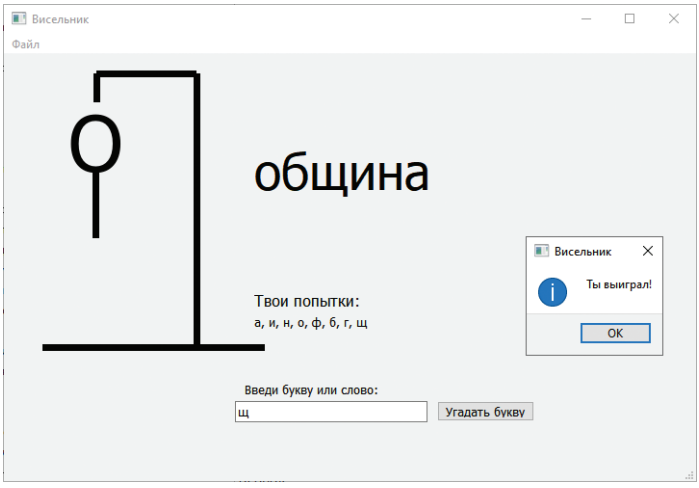
И снова игра – «Виселица»

Почему у нас в главе о файлах? Ну, единственное, что делает игру «Виселица» интересной, – это длинный список слов, которые будут загаданы игроку. Самый простой способ это реализовать – читать список из файла. Мы также используем PyQt для того, чтобы ты понял, что Pygame – не единственный способ создания графических игр.

Не будем вдаваться в детали игры, как в предыдущих программах. Сейчас ты уже должен уметь читать код и понимать, как работает большинство деталей. Мы укажем только направление.

Интерфейс Виселицы

Основной интерфейс нашей игры выглядит так:



В нем показаны все части висящего человека, но при запуске программы мы спрячем все части. Когда игрок называет неверную букву, мы показываем следующую часть человека. Когда нарисован весь человек, игрок получает еще одну попытку, и затем *игра окончена*!

Когда игрок угадывает букву, программа проверяет ее наличие в загаданном слове. Если она в нем есть, буква выводится на экран. Внизу окна игрок может видеть все уже названные буквы. Игрок также может попытаться угадать все слово в любой момент.

Далее я кратко изложу принципы работы программы.

В начале она выполняет эти действия:

- загружает список слов из файла;
- убирает символы перехода строки в конце каждой строки;
- делает все части висящего человека невидимыми;
- выбирает случайное слово из списка;
- показывает такое количество дефисов, сколько букв в загаданном слове.

Когда игрок щелкает по кнопке **Угадать букву**, программа делает следующее:

- проверяет, что представляет из себя догадка: букву или целое слово;
- если это буква, то:
  - проверяет наличие этой буквы в секретном слове;
  - если игрок угадал букву, заменяет дефисы на эту букву в тех местах, где она встречается в слове;
  - если игрок не угадал, показывает следующую часть человека;
  - добавляет угаданную букву к списку **Твои попытки**;
  - проверяет, не угадано ли слово полностью (угаданы ли все буквы);
- если это слово, то:
  - проверяет, угадал ли игрок слово;
  - если да, показывает диалоговое окно со словом **Правильно!** и начинает новую игру;
  - проверяет, не закончились ли попытки у игрока, – если да, показывает диалоговое окно со словами **Ты проиграл** и загаданное слово.

Получение слов из списка

Это глава о файлах, поэтому давай посмотрим на ту часть программы, в которой мы получаем список слов. Код выглядит так:

```
f = open("words.txt", 'r')
self.lines = f.readlines()
for line in self.lines:
line.strip()
f.close()
```

Удаляет символы переноса строки в каждой строке

Файл *words.txt* текстовый, поэтому мы можем прочесть его с помощью функции **`readlines()`**. Затем, чтобы выбрать слово из списка, используем функцию **`random.choice()`**:

```
self.currentword = random.choice(self.lines)
```

Показываем человека

Есть несколько способов отследить, какие части тела человека уже показаны и какую часть показать следующей. Мы решили использовать вложенные циклы. Выглядит это так:

```
def wrong(self):
    self.pieces_shown += 1
    for i in range(self.pieces_shown):
        self.pieces[i].setHidden(False)
    if self.pieces_shown == len(self.pieces):
        message = «Ты проиграл. Слово было « + self.currentword
        QtWidgets.QMessageBox.warning(self,"Висельник",message)
        self.new_game()
```

Мы используем функцию `self.pieces_shown`, чтобы отслеживать, сколько частей висельника отображается. Если отражены все части, мы выводим диалоговое окно, сообщающее игроку, что он проиграл.

Проверка букв

Одна из самых сложных частей программы – проверка буквы, названной игроком, на ее наличие в загаданном слове. Сложность заключается в том, что буква может встречаться в слове не один раз. Например, если загаданное слово – «молоко» и игрок угадывает «о», то тебе нужно открыть вторую, четвертую и шестую буквы, потому что все они – «о».

Есть несколько функций, которые помогут нам. Функция **`find_letters()`** находит все случаи определенной буквы в слове и возвращает список их позиций. Например, для буквы «о» и слова «молоко» она вернет значение **`[1,3]`**, потому что буква «о» встречается под индексами 1 и 3 в строке. (Напомним, что индексация начинается с 0.) Ниже приведен код:

```
def find_letters(letter, a_string):
    locations = []
    start = 0
    while a_string.find(letter, start, len(a_string)) != -1:
        location = a_string.find(letter, start, len(a_string))
        locations.append(location)
        start = location + 1
    return locations
```

Проверяет, где появляется буква

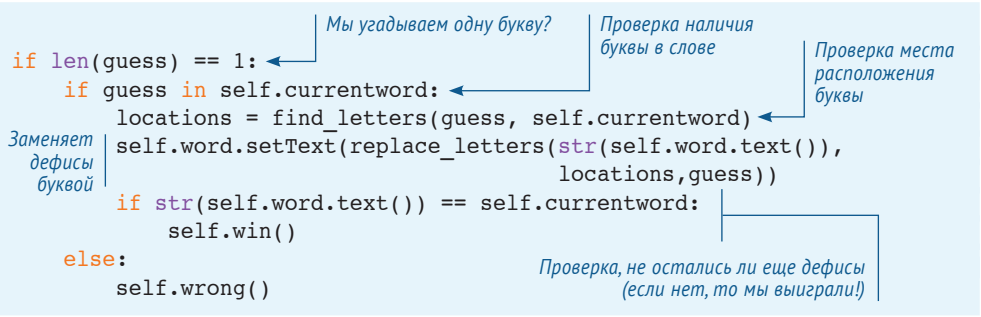
Добавляет местоположение в список

Функция **`replace_letters()`** берет список из **`find_letters()`** и заменяет дефисы правильными буквами на нужных позициях. В нашем примере (с буквой «о») она заменит **`------`** на **`—o-o-o`**. Она показывает игроку, где находятся угаданные буквы. Ниже приведен код:

```
def replace_letters(string, locations, letter):
    new_string = ''
```

```
    for i in range(0, len(string)):
        if i in locations:
            new_string = new_string + letter
        else:
            new_string = new_string + string[i]
    return new_string
```

Затем, когда игрок угадывает букву, мы используем две только что определенные функции:




```
if len(guess) == 1:
    if guess in self.currentword:
        locations = find_letters(guess, self.currentword)
        self.word.setText(replace_letters(str(self.word.text()),
                                          locations,guess))
        if str(self.word.text()) == self.currentword:
            self.win()
    else:
        self.wrong()
```

Вся программа состоит из 95 строк кода плюс пустые строки. В листинге 22.8 приведен весь код с некоторыми комментариями. Код находится в папке *Примеры* на твоем компьютере. Он включает в себя файлы *Висельник.py*, *hangman.ui* и *words.txt*.

Напомним, что в главе 20 мы говорили, что если ты работаешь на Mac, тебе нужно будет открыть файл *hangman.ui* в Qt Designer и снять флажок свойства **nativeMenuBar** для объекта **menubar**.

Листинг 22.8. Полный код программы «Виселица»

```
import sys
from PyQt5 import QtWidgets, uic
import random
form_class = uic.loadUiType("hangman.ui")[0]
def find_letters(letter, a_string):
    locations = []
    start = 0
    while a_string.find(letter, start, len(a_string)) != -1:
        location = a_string.find(letter, start, len(a_string))
        locations.append(location)
        start = location + 1
    return locations

def replace_letters(string, locations, letter):
    new_string = ''
    for i in range (0, len(string)):
        if i in locations:
            new_string = new_string + letter
        else:
            new_string = new_string + string[i]
    return new_string
```

Нахождение букв

Замена дефисов буквами, которые игрок угадал

```
def dashes(word):
    letters = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя"
    new_string = ''
    for i in word:
        if i in letters:
            new_string += "-"
        else:
            new_string += i
    return new_string
class HangmanGame(QtWidgets.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtWidgets.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.btn_guess.clicked.connect(self.btn_guess_clicked)
        self.actionExit.triggered.connect(self.menuExit_selected)
        self.pieces = [self.head, self.body, self.leftarm, self.leftleg,
                       self.rightarm, self.rightleg]
        self.gallows = [self.line1, self.line2, self.line3, self.line4]
        self.pieces_shown = 0
        self.currentword = ""
        f=open("words.txt", 'r')
        self.lines = f.readlines()
        f.close()
        self.new_game()

    def new_game(self):
        self.guesses.setText("")
        self.currentword = random.choice(self.lines)
        self.currentword = self.currentword.strip()
        for i in self.pieces:
            i.setFrameShadow(QtWidgets.QFrame.Plain)
            i.setHidden(True)
        for i in self.gallows:
            i.setFrameShadow(QtWidgets.QFrame.Plain)
        self.word.setText(dashes(self.currentword))
        self.pieces_shown = 0

    def btn_guess_clicked(self):
        guess = str(self.guessBox.text())
        if str(self.guesses.text()) != "":
            self.guesses.setText(str(self.guesses.text())+", "+guess)
        else:
            self.guesses.setText(guess)
        if len(guess) == 1:
            if guess in self.currentword:
                locations = find_letters(guess, self.currentword)
                self.word.setText(replace_letters(str(self.word.text()),
                                                  locations,guess))
                if str(self.word.text()) == self.currentword:
                    self.win()
            else:
```

Замена букв дефисами при запуске программы 1

Присоединение обработчика событий

Части человека

Части виселицы

Получаем список слов

Случайный выбор слова из списка

Прячем человека

Вызов функции, заменяющей буквы дефисами

Позволяет игроку угадать букву или слово

Отгадывание буквы

```
                self.wrong()
        else:
            if guess == self.currentword:
                self.win()
            else:
                self.wrong()
        self.guessBox.setText("")

    def win(self):
        QtWidgets.QMessageBox.information(self,"Висельник","Ты выиграл!")
        self.new_game()

    def wrong(self):
        self.pieces_shown += 1
        for i in range(self.pieces_shown):
            self.pieces[i].setHidden(False)
        if self.pieces_shown == len(self.pieces):
            message = «Ты проиграл. Слово было « + self.currentword
            QtWidgets.QMessageBox.warning(self,"Висельник", message)
            self.new_game()
    def menuExit_selected(self):
        self.close()

app = QtWidgets.QApplication(sys.argv)
myapp = HangmanGame(None)
myapp.show()
app.exec_()
```

Отгадывание слова

Позволяет игроку угадать букву или слово

Отображение диалогового окна, если игрок выиграл

Отображение следующей части человека

Игрок проиграл

Неправильная догадка

Чтобы не усложнять нашу программу «Виселица», мы используем в ней только строчные буквы. Наш список слов состоит только из строчных букв, соответственно, пользователь тоже должен вводить только строчные буквы.

Функция **`dashes()`** (1) заменяет буквы дефисами в начале новой игры.

Советуем тебе попробовать написать эту программу самому. Ты можешь создать интерфейс с помощью Qt Designer. Не имеет значения, выглядит ли он так же, как и наша версия. Просто убедись, что имена компонентов совпадают. Имена в коде должны совпадать с именами в файле *.ui*.

Набери код. Запусти программу и посмотри, как она работает. Если ты хочешь попробовать что-то еще, вперед! Развлекайся, играй с кодом и экспериментируй. Это одна из самых интересных частей в программировании, и именно так ты всему научишься.

Что ты узнал?

В этой главе ты узнал:

- что такое файл;
- как открывать и закрывать файлы;

- разные способы открытия файла: чтение, запись и дополнение;
- разные способы записи в файл: **`write()`** и **`print()`**;
- как использовать модуль **`pickle`** для хранения списков и объектов (и других видов данных) в файле;
- много нового о папках (каталогах), расположении файлов и путях.

Мы также создали игру «Виселица» и использовали в ней файл для получения списка слов.

Проверь свои знания

1. Тип объекта в Python, который работает с файлами, – это _____.
2. Как создать объект файла?
3. В чем разница между объектом файла и именем файла?
4. Что нужно сделать с файлом, когда ты закончил его чтение или запись?
5. Что случится, если ты откроешь файл в режиме дополнения, а потом запишешь в него что-то?
6. Что случится, если ты откроешь файл в режиме записи и запишешь в него что-то?
7. Как снова начать читать начало файла, если ты уже прочитал некоторую его часть?
8. Какая функция модуля **`pickle`** используется для сохранения объекта Python в файл?
9. Какой метод модуля **`pickle`** используется для «расконсервации» объекта – его получения из файла и помещения обратно в переменную Python?

Попробуй самостоятельно

1. Напиши программу для создания глупых предложений. Каждое предложение должно иметь по крайней мере четыре части:

```
_____________ _____________ ___________________ _____________
прилагательное существительное глагольная конструкция обстоятельство
```

Например:

```
Сумасшедшая  мартышка  играла на гавайской гитаре  на столе.
прилагательное существительное глагольная конструкция обстоятельство
```

Программа должна составлять предложения, выбирая эти части в случайном порядке. Слова хранятся в файлах, их можно создать с помощью программы «Блокнот». Самый простой способ – иметь отдельный файл для каждой группы слов, но ты можешь поступать по-своему. Ниже представлены некоторые идеи, но я уверен, что ты сделаешь все на свой вкус:

- прилагательные: сумасшедший, глупый, застенчивый, чокнутый, злой, ленивый, упрямый, сиреневый;

- существительные: обезьяна, слон, байкер, учитель, автор, хоккеист;
- глагольные конструкции: играл на гавайской гитаре, танцевал джигу, причесывался, хлопал ушами;
- обстоятельства: на столе, в магазине, в душе, после завтрака, с веником.

Еще один пример вывода: Ленивый автор причесывался веником.

2 Напиши программу, которая просит пользователя ввести свое имя, возраст, любимый цвет и блюдо. Пусть программа сохраняет все элементы в текстовый файл, каждый в отдельной строке.

3 Сделай то же самое, что и в задании 2, но используй модуль **`pickle`** для сохранения данных в файл. (Подсказка: будет проще, если поместить данные в список.)

ГЛАВА 23

Аспект случайности

Самое смешное в играх – ты никогда не знаешь, что именно может произойти. Игры непредсказуемы. Они *случайны*. Именно случайный порядок делает их интересными.

Как мы уже знаем, компьютеры могут симулировать случайное поведение. В нашей программе по угадыванию чисел (глава 1) мы использовали модуль **`random`**, чтобы генерировать случайное целое число, которое пользователь должен был угадать. Мы также использовали модуль **`random`** для подбора слов для глупых предложений в разделе «Попробуй сам» предыдущей главы.

Компьютеры также могут симулировать случайное поведение игральных костей или колоды карт. Это дает возможность создавать компьютерные игры с картами и костями (или другими предметами со случайным поведением). Например, всем известная игра «Солитер» в Windows – карточная игра, в которой программа перемешивает карты в случайном порядке перед каждой игрой. Компьютерный триктрак тоже очень популярен, в нем используются две кости.

В этой главе ты узнаешь о том, как использовать модуль **`random`** для создания компьютерных костей и колоды карт для игр. Мы также посмотрим, как можно использовать генерированные компьютером случайные события для исследования *вероятности,* то есть возможности случиться для какого-то события.

Что такое случайный порядок?

Перед тем как начать писать программы со случайным поведением, нужно понять, что же значит «случайный».

Возьмем пример подброшенной монетки. Если подбросить ее в воздух и позволить ей упасть, ты увидишь или орел, или решку. Для нормальной монеты шансы показать решку равны шансам показать орел. Иногда ты получаешь решку, иногда орел. При каждом броске ты не знаешь, что выпадет. Поскольку результат броска нельзя предугадать, мы говорим, что он *случаен*. Подбрасывание монеты – пример *случайного события*.

Если подбросишь монетку несколько раз, ты, возможно, получишь одинаковое количество орлов и решек. Но ты никогда не можешь быть уверен. Если ты подбросишь монету 4 раза, ты можешь получить 2 раза орел и 2 раза решку. Но можешь получить и 3 раза орел и 1 раз решку, 1 раз орел и 3 раза решку или даже 4 раза орел (или решку). Если ты подбросишь монету 100 раз, ты можешь получить 50 раз решку. Но также можешь получить ее 20, 44, 67 или даже 100 раз! Это почти невероятно, но возможно.

Смысл в том, что каждое событие случайно. Хотя ты можешь выработать схему подбрасывания, но каждый отдельный бросок имеет такой же шанс показать решку или орел. Другим образом, монета не имеет *памяти*. Поэтому даже если ты только что получил 99 раз подряд решку и думаешь, что почти невозможно получить ее 100 раз подряд, все равно при следующем броске существует вероятность 50 % получить решку. Вот что значит *случайность*.

Случайное событие – это событие с двумя и более возможными результатами, которые ты не можешь предсказать. Результатом может быть порядок карт в перемешанной колоде, количество точек после броска костей или сторона монеты после броска.

Бросаем кости

Почти все знают игры с использованием костей. Монополия, нарды, триктрак и другие игры используют кости как один из самых известных способов генерации случайного события в игре.

Кости легко симулировать в программе, и модуль **`random`** допускает несколько способов это реализовать. Один из них – функция **`randint()`**, которая выбирает случайное целое число. Поскольку количество точек на грани кости – целое число (1,2,3,4,5 и 6), бросок одной кости можно симулировать так:

```
import random
die_1 = random.randint(1, 6)
```

Это дает нам число от 1 до 6 с одинаковой вероятностью получения любого из них. Как и в случае с настоящей костью.

Второй способ сделать то же самое – создать список возможных результатов и использовать функцию **`choice()`**, чтобы выбрать один из них. Код выглядит так:

```
import random
sides = [1, 2, 3, 4, 5, 6]
die_1 = random.choice(sides)
```

Этот код делает то же самое, что и предыдущий. Функция **`choice()`** выбирает в случайном порядке элемент из списка. В этом случае в списке находятся числа от 1 до 6.

Если кость не одна

Что, если мы хотим симулировать бросок двух костей? Если ты просто собираешься добавить вторую кость так, чтобы получить только общее количество точек, можно сделать так:

```
two_dice = random.randint(2, 12)
```

Суммой точек двух костей может быть число от 2 до 12, правильно? И да, и нет. Ты будешь получать случайное число от 2 до 12, но не тем же способом, что при сложении двух случайных чисел от 1 до 6. Этот код как будто бросает одну большую 11-стороннюю кость, а не две 6-гранные. Но в чем же разница? Мы переходим к теме *вероятности*. Самый простой способ – попробовать самому и понять.

Давай бросим кости несколько раз и посмотрим, сколько раз выпадет каждая сумма. Сделаем это с помощью цикла и списка. Цикл будет бросать кости, а список – следить за количеством выпадений каждой суммы. Начнем с одной 11-гранной кости, как показано в листинге 23.1.

Листинг 23.1. Бросание одной 11-гранной кости 1000 раз

```
import random

totals = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  ← 1 В списке есть 13 элементов с индексами от 0 до 12
for i in range(1000):
    dice_total = random.randint(2, 12)
    totals[dice_total] += 1  ← 2 Добавляет 1 к счету этой суммы

for i in range (2, 13):
    print("сумма", i, "выпала", totals[i], "раз")
```

1 Список содержит индексы от 0 до 12, но мы не используем первые два, потому что суммы 0 и 1 в костях никогда не бывает. 2 Когда мы получаем результат, то добавляем 1 к этому элементу списка. Если сумма равна 7, добавляем единицу к **`totals[7]`**. Итак, **`totals[2]`** – количество двоек, **`total[3]`** – количество полученных троек и т. д.

Если запустить этот код, получим примерно следующее:

```
сумма 2 выпала 95 раз
сумма 3 выпала 81 раз
сумма 4 выпала 85 раз
сумма 5 выпала 86 раз
сумма 6 выпала 100 раз
сумма 7 выпала 85 раз
сумма 8 выпала 94 раз
сумма 9 выпала 98 раз
сумма 10 выпала 93 раз
сумма 11 выпала 84 раз
сумма 12 выпала 99 раз
```

Если взглянуть на сумму, заметно, что все числа выпали примерно одинаковое количество раз между 80 и 100. Не одинаковое, потому что числа случайные, но приблизительно, и нет очевидно заметной частоты выпадения одних чисел больше, чем других. Попробуй запустить программу несколько раз, чтобы убедиться. Или попробуй увеличить количество циклов до 10 000 или 100 000.

Теперь попробуем то же самое с двумя 6-гранными костями. Код приведен в листинге 23.2.

Листинг 23.2. Бросание двух 6-гранных костей 1000 раз

```
import random

totals = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
for i in range(1000):
    die_1 = random.randint(1, 6)
    die_2 = random.randint(1, 6)
    dice_total = die_1 + die_2
    totals[dice_total] += 1

for i in range (2, 13):
    print("сумма", i, "выпала", totals[i], "раз")
```

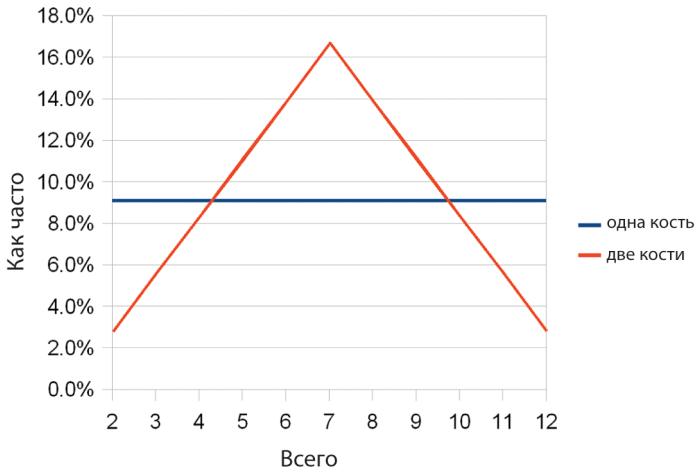
Если запустить код из листинга 23.2, ты должен получить такой вывод:

```
сумма 2 выпала 22 раз
сумма 3 выпала 61 раз
сумма 4 выпала 93 раз
сумма 5 выпала 111 раз
сумма 6 выпала 141 раз
сумма 7 выпала 163 раз
сумма 8 выпала 134 раз
сумма 9 выпала 117 раз
сумма 10 выпала 74 раз
сумма 11 выпала 62 раз
сумма 12 выпала 22 раз
```

Обрати внимание, что самые высокие и самые низкие значения выпадали меньше всего раз, а средние, например 6 и 7, выпадали чаще всего. С одной 11-гранной костью была другая картина. Если выполнить цикл еще несколько раз и затем посчитать процент выпадения каждой суммы, получим примерно такую таблицу:

Результат	Одна 11-гранная кость	Две 6-гранные кости
2	9.1 %	2.8 %
3	9.1 %	5.6 %
4	9.1 %	8.3 %
5	9.1 %	11.1 %
6	9.1 %	13.9 %
7	9.1 %	16.7 %
8	9.1 %	13.9 %
9	9.1 %	11.1 %
10	9.1 %	8.3 %
11	9.1 %	5.6 %
12	9.1 %	2.8 %

Если мы создадим график этих чисел, он будет выглядеть так:



Почему они отличаются? Причина включает в себя большую тему, касающуюся *вероятности*. Средние числа встречаются гораздо чаще при двух костях, потому что есть много способов получить среднюю сумму при них.

Когда ты бросаешь две кости, могут случиться разные комбинации. Ниже приведен их список вместе с суммами:

```
1+1 = 2   1+2 = 3   1+3 = 4   1+4 = 5   1+5 = 6   1+6 = 7
2+1 = 3   2+2 = 4   2+3 = 5   2+4 = 6   2+5 = 7   2+6 = 8
3+1 = 4   3+2 = 5   3+3 = 6   3+4 = 7   3+5 = 8   3+6 = 9
4+1 = 5   4+2 = 6   4+3 = 7   4+4 = 8   4+5 = 9   4+6 = 10
5+1 = 6   5+2 = 7   5+3 = 8   5+4 = 9   5+5 = 10  5+6 = 11
6+1 = 7   6+2 = 8   6+3 = 9   6+4 = 10  6+5 = 11  6+6 = 12
```

Есть 36 возможных комбинаций. А теперь давай посмотрим, сколько раз получается одна и та же сумма:

- сумма 2 выпала 1 раз;
- сумма 3 выпала 2 раза;
- сумма 4 выпала 3 раза;
- сумма 5 выпала 4 раза;
- сумма 6 выпала 5 раз;
- сумма 7 выпала 6 раз;
- сумма 8 выпала 5 раз;
- сумма 9 выпала 4 раза;
- сумма 10 выпала 3 раза;
- сумма 11 выпала 2 раза;
- сумма 12 выпала 1 раз.

Это значит, что у нас больше шансов выкинуть 7, чем 2. Для того чтобы получить 7, можно выкинуть 6+1, 2+5, 3+4, 4+3, 5+2, 6+1. Чтобы получить 2, нужно выкинуть только 1+1. Итак, имеет смысл ожидать больше семерок, чем двоек, при бросании костей много раз. Именно это мы получили в нашей программе с двумя костями.

Использование компьютерных программ для генерации случайных событий – хороший способ поэкспериментировать с вероятностью и посмотреть, что происходит при большом количестве попыток. Тебе понадобится много времени, чтобы бросить реальные кости 1000 раз и записать результаты. А компьютер может сделать это за долю секунды!

Десять раз подряд

Давай выполним еще один эксперимент по вероятности, перед тем как двигаться дальше. Несколькими страницами выше мы говорили о подбрасывании монетки и вероятности получить решку несколько раз подряд. Почему бы не провести эксперимент и не узнать это? Такое событие происходит не очень часто, поэтому нам придется кинуть монетку очень много раз. Давай попробуем 1 000 000 раз! С реальной монетой это займет... много времени.

Если ты бросаешь монету 1 раз в 5 секунд, это будет 12 раз в минуту, или 720 раз в час. Если ты можешь заниматься этим 12 часов в день (ну, тебе нужно есть и спать), ты можешь делать 8640 подбрасываний в день. Итак, нужно около 115 дней (4 месяца), чтобы подбросить монету миллион раз. Но компьютер может сделать это за секунды! (Ну ладно, несколько минут, потому что сначала надо написать программу.)

В этой программе вместе с подбрасыванием монеты нам нужно следить за тем, когда выпадает

10 решек подряд. Первый способ – использовать счетчик. Счетчик – это переменная для подсчета чего-либо.

Нам понадобится два счетчика. Первый будет считать количество выпавших подряд решек. Назовем его **heads_in_row**. Второй – количество раз, когда выпало 10 решек подряд. Назовем его **ten_heads_in_row**. Программа будет выполнять следующие действия:

- при выпадении решки счетчик **heads_in_row** будет увеличиваться на 1;
- при выпадении орла счетчик **heads_in_row** будет возвращаться к 0;
- когда счетчик **heads_in_row** достигнет значения 10, мы увеличиваем значение счетчика **ten_heads_in_row** на 1 и возвращаем счетчику **heads_in_row** значение 0;
- в конце мы выводим сообщение о том, сколько раз у нас выпало подряд 10 решек.

Код приведен в листинге 23.3.

Листинг 23.3. Поиск 10 решек подряд

```
from random import *
coin = [“Решка”, “Орел”]
heads_in_row = 0
ten_heads_in_row = 0
for i in range (1000000):
    if choice(coin) == “Решка”: ←—— Подбрасывание монетки
        heads_in_row += 1
    else:
        heads_in_row = 0
    if heads_in_row == 10:
        ten_heads_in_row += 1 ←—— Получение 10 решек подряд, счетчик с инкрементом
        heads_in_row = 0

print(“Выпало 10 решек подряд”, ten_heads_in_row, “раз.”)
```

Когда мы запустилм эту программу, у нас получилось:

```
Выпало 10 решек подряд 510 раз.
```

Мы запустили программу несколько раз, и число всегда было около 500. Значит, если подбросить монетку миллион раз, решка может выпасть 10 раз подряд примерно 500 раз, или 1 раз на 2000 бросков (1 000 000 / 500 = 2000).

Создание колоды карт

Еще один вид случайных событий, часто используемых в играх, – это вытягивание карты. Оно случайное, потому что колода перемешана, и ты не знаешь, какая карта выпадет. И при каждом перемешивании колоды порядок меняется.

В случае с монетой и костями мы говорили, что у каждого броска одинаковая вероятность, потому что у монеты (или кости) нет памяти. Но это неверно для карт. Когда ты вытягиваешь карту из колоды, в ней остается все меньше и меньше карт (в большинстве игр, по крайней мере). Это меняет вероятность вытаскивания каждой из оставшихся карт.

Например, когда ты начинаешь с полной колоды, шансы вытянуть четверку червей – 1/52, или около 2 %. Потому что в колоде 52 карты и только одна четверка червей. Если ты продолжишь вытаскивать карты (и еще не вытянул четверку червей), на половине колоды шансы вытащить нужную карту равны 1/26, или около 4 %. По достижении последней карты, если ты еще не вытащил четверку червей, шансы будут 1/1, или 100 %. Ты точно вытащишь нужную четверку, потому что осталась только одна карта.

Мы говорим тебе это все потому, что если мы будем делать игру с колодой карт, нам нужно следить за тем, какие карты были вытащены из колоды. Это можно сделать с помощью списка. Мы можем начать с полного списка всех 52 карт в колоде и использовать **`random.choice()`** для случайного выбора карты из списка. При вытаскивании каждой карты из списка (колоды) мы ее оттуда удаляем с помощью функции **`remove()`**.

Перемешивание колоды

В реальной карточной игре мы перемешиваем колоду, чтобы карты находились в случайном порядке. Так мы можем взять верхнюю карту, и она будет случайной. Но с помощью функции **`random.choice()`** мы все равно выбираем из списка случайную карту. Нам не нужно брать «верхнюю», поэтому нет смысла в «перемешивании». Мы просто выбираем случайную карту из любой части колоды. Это похоже на раскладывание карт веером и предложение выбрать любую карту. Это займет довольно много времени в настоящей игре, но очень легко для компьютерной программы.

Объект карты

Мы используем список, который будет вести себя как «колода» карт. А как насчет самих карт? Как мы будем хранить каждую из них? Как строку? Как целое число? Что нам нужно знать о каждой карте?

В карточных играх нам нужно знать три вещи о каждой карте:

- *масть* – бубны, червы, пики или трефы;
- *достоинство* – туз, 2, 3,... 10, валет, дама, король;
- *значение* – для нумерованных карт (от 2 до 10), то же самое, что и достоинство. У валета, дамы и короля это 10, у туза может быть 1, 11 или другое значение, в зависимости от игры.

Достоинство	Значение
Туз	1, 11
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
Валет	10
Дама	10
Король	10

Итак, нам нужно следить за этими тремя параметрами и держать их вместе в каком-то контейнере. Список подойдет, но нам надо помнить, какой элемент был чем. Другой способ – создать объект «карты» с такими атрибутами:

```
card.suit
card.rank
card.value
```

Так мы и поступим. Мы также добавим еще пару атрибутов **suit_id** и **rank_id**:

- **suit_id** – это число от 1 до 4, где
  1 = бубны, 2 = червы, 3 = пики, 4 = трефы;
- **rank_id** – число от 1 до 13, где
  1 = туз,
  2 = 2,
  3 = 3,
  ...
  10 = 10,
  11 = валет,
  12 = дама,
  13 = король.

Мы добавляем два этих атрибута, чтобы легко использовать вложенный цикл **for** для создания колоды из 52 карт. Можно сделать внутренний цикл для достоинств (1–13) и внешний цикл для масти (1–4). Метод **__init__()** для объекта карты будет использовать **suit_id** и **rank_id** и создавать другие атрибуты масти, достоинства и значения. Также будет проще сравнивать достоинство двух карт.

Нам нужно добавить еще два атрибута, чтобы наш объект карты было проще использовать в программе. Когда программе нужно вывести карту на экран, мы будем видеть что-то наподобие «4H» или «4 of Hearts». Для карт с фигурами это будет «JD» или «Jack of Diamonds». Мы добавим атрибуты **short_name** и **long_name**, чтобы программа могла легко выводить короткое или длинное описание

карты. В игре мы используем английские названия мастей и достоинств; подумай, как ты можешь выводить и использовать их на русском языке.

Давай создадим класс для игровой карты. Код показан в листинге 23.4.

Листинг 23.4. Класс карты

```
class Card:
    def __init__(self, suit_id, rank_id):
        self.rank_id = rank_id
        self.suit_id = suit_id

        if self.rank_id == 1:
            self.rank = "Ace"
            self.value = 1
        elif self.rank_id == 11:
            self.rank = "Jack"
            self.value = 10
        elif self.rank_id == 12:
            self.rank = "Queen"
            self.value = 10
        elif self.rank_id == 13:
            self.rank = "King"
            self.value = 10
        elif 2 <= self.rank_id <= 10:
            self.rank = str(self.rank_id)
            self.value = self.rank_id
        else:
            self.rank = "RankError"
            self.value = -1

        if self.suit_id == 1:
            self.suit = "Diamonds"
        elif self.suit_id == 2:
            self.suit = "Hearts"
        elif self.suit_id == 3:
            self.suit = "Spades"
        elif self.suit_id == 4:
            self.suit = "Clubs"
        else:
            self.suit = "SuitError"
        self.short_name = self.rank[0] + self.suit[0]
        if self.rank == '10':
            self.short_name = self.rank + self.suit[0]
        self.long_name = self.rank + " of " + self.suit
```

Создание атрибутов **rank** *и* **value**

Создание атрибута **suit**

1 *Проверка некоторых ошибок*

Проверка на ошибку 1 позволяет убедиться, что значения **rank_id** и **suit_id** находятся в нужном диапазоне и являются целыми числами. Если нет, ты увидишь что-то наподобие «7 SuitError» или «RankError of Clubs» при отображении карты в программе.

Строка, содержащая атрибут **`short_name`**, просто берет число или первую букву достоинства (6 или Jack) и первую букву масти (Diamonds) и складывает их вместе. Таким образом, сокращенное наименование короля червей – KH (от англ. *King of Hearts*), шестерки пик – 6S (от англ. *6 of Spades*).

Это не вся программа. Это только определение для класса **`Card`**. Поскольку мы сможем использовать его в разных программах, имеет смысл сделать класс *модулем*. Сохрани код из листинга 23.4 в файл *cards.py*.

Теперь нужно создать несколько экземпляров карт – на самом деле неплохо это сделать для всей колоды! Чтобы проверить наш класс **`Card`**, давай напишем программу для создания колоды из 52 карт и выберем 5 случайных карт с их атрибутами. В листинге 23.5 показан код.

Листинг 23.5. Создание колоды карт

```
import random
from cards import Card  <-- Импорт модуля cards

deck = []
for suit_id in range(1, 5):
    for rank_id in range(1, 14):
        deck.append(Card(suit_id, rank_id))

hand = []
for cards in range(0, 5):
    a = random.choice(deck)
    hand.append(a)
    deck.remove(a)

print()
for card in hand:
    print(card.short_name, '=' ,card.long_name, " Value:", card.value)
```

1 Использование вложенных циклов **for** для создания колоды

2 Выбор 5 карт из колоды для создания руки

1 Внутренний цикл выполняется для каждой карты каждой масти, а внешний – для каждой масти (13 карт * 4 масти = 52 карты). 2 Затем в коде выбираются пять карт из колоды и кладутся в руку. Карты удаляются из колоды.

Если выполнить программу из листинга 23.5, получим примерно следующее:

```
7D = 7 of Diamonds Value: 7
9H = 9 of Hearts Value: 9
KH = King of Hearts Value: 10
6S = 6 of Spades Value: 6
KC = King of Clubs Value: 10
```

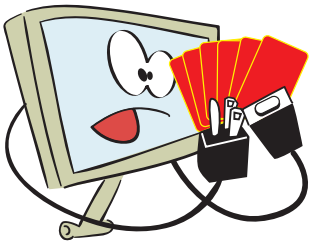
Если запустить программу сначала, то ты получишь пять других карт. И не имеет значения, сколько раз ты запустишь программу, ты каждый раз получишь разные карты.

Теперь мы можем создавать колоду карт и вытаскивать случайную карту из нее, добавляя ее в руку. Кажется, у нас есть базовые компоненты для карточной игры! Далее мы напишем игру и сможем сыграть в нее с компьютером.

Сумасшедшие восьмерки

Ты мог слышать о карточной игре «Сумасшедшие восьмерки». Ты даже, возможно, играл в нее.

Единственная сложность в карточных играх на компьютере – трудно реализовать игру с несколькими участниками. Потому что в большинстве карточных игр ты не должен видеть карты противников. Если все смотрят на экран одного и того же компьютера, они видят карты друг друга. Поэтому лучше всего играть с двумя участниками – тобой и компьютером. Сумасшедшие восьмерки – одна из таких игр, в которые весело играть и вдвоем, поэтому мы ее и напишем с двумя игроками.



Ниже представлены правила нашей программы. Это игра для двоих участников. Каждый игрок получает пять карт. Остальные карты кладутся лицом вниз, одна из них переворачивается, чтобы начать отбой. Цель игры – избавиться от всех карт быстрее всех и до того, как закончится колода.

1. При каждом ходе игрок должен выполнить одно из действий:
   - положить карту той же масти, что и верхняя карта;
   - положить карту того же достоинства, что и верхняя карта;
   - положить восьмерку.
2. Если игрок положил восьмерку, он может «заказать масть», что значит, он выбирает масть, которую должен положить противник.
3. Если игрок не может сыграть ни одной картой, он должен взять карту из колоды и оставить ее себе.
4. Если игрок избавляется от всех карт, он побеждает и получает количество очков, равное тому, какие карты остались у противника:
   - 50 очков за каждую восьмерку;
   - 10 очков за каждую фигурную карту;
   - соответствующее значение каждой карты с цифрой;
   - 1 очко за каждый туз.
5. Если колода заканчивается и никто не выиграл, игра окончена. В этом случае оба игрока получают очки за карты, оставшиеся в руках у противника.
6. Можно играть до определенного количества очков или пока не надоест, игрок с большим количеством очков побеждает.

Первое, что нам нужно сделать, – немного изменить некоторые объекты карт. Значения карт в игре почти совпадают с теми, что были у нас раньше, кроме восьмерок, за которые игрок получает 50 очков, а не 8. Мы можем изменить метод **`__init__()`** в нашем классе **`Card`**, но это повлияет на все остальные игры, использующие модуль **`cards`**. Лучше внести изменения в основную программу и не трогать определение класса. В листинге 23.6 приведен один из способов это сделать, введя функцию **`init_cards()`**, которая внесет изменения в колоду карт специально для «Сумасшедших восьмерок».

Листинг 23.6. Создание колоды карт для «Сумасшедших восьмерок»

```
def init_cards():
    global deck, p_hand, c_hand, up_card, active_suit, active_rank
    deck = []
    for suit in range(1, 5):
        for rank in range(1, 14):
            new_card = Card(suit, rank)
            # Назначение восьмеркам 50 очков
            if new_card.rank == 8:          <-- Изменение значения восьмерки на 50 очков
                new_card.value = 50
            deck.append(new_card)
    p_hand = []
    c_hand = []
    for i in range(5):
        p_card = random.choice(deck)
        deck.remove(p_card)
        p_hand.append(p_card)
        c_card = random.choice(deck)
        deck.remove(c_card)
        c_hand.append(c_card)
up_card = random.choice(deck)
deck.remove(up_card)
active_suit = up_card.suit
active_rank = up_card.rank
```

Здесь перед добавлением новой карты в колоду мы проверяем, не является ли она восьмеркой. Если да, то мы меняем ее значение на 50. Обрати внимание, что в этом листинге игрокам раздается по 5 карт и «переворачивается» одна из карт для начала игры.

Теперь мы готовы начинать писать игру. Ниже представлены еще некоторые действия нашей программы:

- отслеживание карт, которые лежат рубашкой вниз;
- получение выбора действия игрока (сыграть или тянуть карту);
- проверка правильности действия:
  - карта должна быть правильной;
  - карта должна быть в руке игрока;
  - карта должна подходить либо по масти, либо по достоинству к карте, лежащей лицом вверх, или быть восьмеркой;

- если игрок кладет 8, программа должна спросить, какова будет новая масть (и убедиться, что выбрана существующая масть);
- программа должна сыграть свой ход;
- определение окончания игры;
- подсчет очков.

В оставшейся части главы мы рассмотрим каждый из этих пунктов. Для некоторых будет достаточно одной-двух строк кода, а некоторые будут длинными. Для длинных задач мы создадим функции, которые можно будет вызывать из основного цикла.

Основной цикл

Прежде чем вдаваться в детали, давай разберемся, как будет выглядеть основной цикл программы. Нам нужно чередовать ходы компьютера и игрока, пока один из них не выиграет или оба не будут заблокированы. Код приведен в листинге 23.7.

Листинг 23.7. Основной цикл сумасшедших восьмерок

```
game_done = False
init_cards()
while not game_done:
    blocked = 0
    player_turn()                          # Ход игрока
    if len(p_hand) == 0:                   # Рука игрока (p_hand) не содержит
        game_done = True                   # больше карт, поэтому игрок побеждает
        print()
        print("Ты выиграл!")
    if not game_done:
        computer_turn()                    # Ход компьютера
    if len(c_hand) == 0:                   # В руке компьютера (c_hand) не осталось
        game_done = True                   # больше карт, поэтому он побеждает
        print()
        print("Компьютер выиграл!")
    if blocked >= 2:                       # (1) Оба игрока проиграли, игра окончена
        game_done = True
        print(«У обоих игроков нет возможности хода. ИГРА ОКОНЧЕНА.»)
```

Часть основного цикла определяет, когда игра окончена. Она может быть окончена, когда у одного из игроков заканчиваются карты. Она может быть окончена, когда у обоих еще есть карты, но нет вариантов хода. Переменная **`blocked`** настроена в коде на очередь игрока (когда у него нет ходов) и на очередь компьютера (когда у него нет ходов). (1) Мы ждем появления значения **`blocked = 2`**, позволяющего убедиться, что и у компьютера, и у игрока нет вариантов хода.

Обрати внимание, что листинг 23.6 – это не вся программа, и если ты попробуешь ее запустить, то получишь сообщение об ошибке. Это только основной цикл. Нам еще нужны остальные части программы.

Это код для одной игры. Если мы хотим продолжать играть дальше, то можем поместить его в другой, внешний цикл **`while`**:

```
done = False p_total = c_total = 0
while not done:
   [запуск игры... см. Листинг 23.6]
    play_again = input(«Сыграть снова (д/н)? “)
    if play_again.lower().startswith(‘д’):
        done = False
    else:
        done = True
```

Это основной каркас программы. Теперь нужно добавить отдельные фрагменты, выполняющие нужные действия.

Думай как программист

Подход, описанный выше, называется «*нисходящим*» программированием. В нем ты начинаешь с описания того, что надо сделать, и постепенно добавляешь детали.

Другой подход называется «*восходящим*». В нем ты сначала создаешь отдельные части, например ход игрока, ход компьютера и т. д., и потом соединяешь их вместе как кирпичики.

У обоих подходов есть свои преимущества и недостатки. Как решить, какой из них использовать, – тема не для этой книги. Но мы думаем, тебе стоит знать о разных подходах к написанию программ.

Карта рубашкой вниз

Когда карты розданы, одна карта из колоды переворачивается рубашкой вниз (лицом вверх), чтобы начать отбой. Когда игрок делает ход, его карта отходит в отбой лицом вверх. Верхняя карта в отбое так и называется – *верхней*. Можно следить за этим с помощью списка отбоя, так же, как мы и поступили со списком «руки» в нашем тестовом коде из листинга 23.5. Но нам не особо интересны карты в отбое. Нас интересует лишь последняя сыгранная карта. Поэтому можно использовать один экземпляр объекта **`Card`**, чтобы следить за ней.

Когда компьютер или игрок делает ход, мы в свою очередь действуем так:

```
hand.remove(chosen_card)
up_card = chosen_card
```

Действующая масть

Обычно активная масть (та, которой карта компьютера или игрока должна соответствовать) такая же, как масть верхней карты. Но есть исключение. Когда сыграна восьмерка, игрок называет желаемую масть. Поэтому если он сыграл восьмеркой бубен, то может заказать трефовую масть. Значит, следующий ход должен соответствовать трефовой масти, хотя сверху лежит бубновая (восьмерка).

Следовательно, нам надо отслеживать активную масть, потому что она может отличаться от масти верхней карты. Можно использовать переменную **`active_suit`**:

```
active_suit = card.suit
```

Когда карта сыграна, мы обновляем переменную, а когда игрок ходит восьмеркой, он выбирает новую активную масть.

Ход игрока

Когда наступает очередь игрока играть, нам нужно получить от него ввод хода. Он может сыграть картой из руки (если есть такая возможность) или вытащить ее из колоды. Если бы мы делали версию этой программы с интерфейсом, игрок щелкал бы мышью по нужной карте или по колоде. Но мы начнем с текстовой версии программы, поэтому игроку придется набрать свой ход на клавиатуре, а нам нужно проверить ввод и понять, что он хочет сделать и верно ли действие.

Чтобы ты понял, каков должен быть ввод игрока, взгляни на пример игры. Ввод игрока выделен **жирным шрифтом**.

```
Сумасшедшие восьмерки

В твоей руке:  QD JC 3C 9D 6C    Верхняя карта:  KD
Что ты хочешь сделать?  Введи карту или команду 'тянуть', чтобы тянуть
карту: 9D
Ты разыграл 9D
 Компьютер разыграл  4D
У компьютера осталось 4 карт

В твоей руке:  QD JC 3C 6C    Верхняя карта:  4D
Что ты хочешь сделать?  Введи карту или команду 'тянуть', чтобы тянуть
карту: JC
Неправильный ход. Попробуй снова: QD
Ты разыграл QD
 Компьютер разыграл  QC
У компьютера осталось 3 карт

В твоей руке:  JC 3C 6C    Верхняя карта:  QC
Что ты хочешь сделать?  Введи карту или команду 'тянуть', чтобы тянуть
карту: JC
Ты разыграл JC
 Компьютер вытянул карту
У компьютера осталось 4 карт
```

```
В твоей руке:  3C 6C    Верхняя карта:  JC
Что ты хочешь сделать?  Введи карту или команду ‘тянуть’, чтобы тянуть
карту: 6C
Ты разыграл 6C
 Компьютер разыграл  6D
У компьютера осталось 3 карт

В твоей руке:  3C    Верхняя карта:  6D
Что ты хочешь сделать?  Введи карту или команду ‘тянуть’, чтобы тянуть
карту: 3C
Неправильный ход. Попробуй снова: тянуть
Ты вытянул 2H
 Компьютер разыграл  AD
У компьютера осталось 2 карт

В твоей руке:  3C 2H    Верхняя карта:  AD
Что ты хочешь сделать?  Введи карту или команду ‘тянуть’, чтобы тянуть
карту: тянуть
Ты вытянул JS
 Компьютер разыграл  AS
У компьютера осталось 1 карт

В твоей руке:  3C 2H JS    Верхняя карта:  AS
Что ты хочешь сделать?  Введи карту или команду ‘тянуть’, чтобы тянуть
карту: JS
Ты разыграл JS
 Компьютер разыграл  3S
У компьютера осталось 0 карт

Компьютер выиграл!
Компьютер получил 5 очков за твои карты
Сыграть снова (д/н)?
```

Надеюсь, ты уловил смысл. Игрок должен вводить названия карт или команду **тянуть**, чтобы сделать ход. Программа должна проверять, что введенное имеет смысл. Мы используем строковые методы (из главы 21), чтобы реализовать это.

Показ руки

Перед тем как спросить у игрока, что он собирается делать, ты должен показать ему его карты и верхнюю карту:

```
print(“\nВ твоей руке: “, end=’’)
for card in p_hand:
    print(card.short_name, end=’ ‘)
print(“ Верхняя карта: “, up_card.short_name)
```

Если была сыграна восьмерка, нужно указать активную масть. Давай добавим еще несколько строк в наш код, как показано в листинге 23.8.

Листинг 23.8. Показ карт игрока

```
print(«\nВ твоей руке: «, end='')
for card in p_hand:
    print(card.short_name, end=' ')
print(" Верхняя карта: ", up_card.short_name)
if up_card.rank == '8':
    print(« Масть:», active_suit)
```

Как и листинг 23.7, листинг 23.8 не является всей программой. Мы все еще строим части для окончательного варианта. Но при запуске кода из листинга 23.8 (как части всей программы) ты видишь следующее:

```
В твоей руке: 4S, QS, 3C Верхняя карта: 8C
 Масть: Spades
```

Если ты хочешь использовать длинные имена карт, вывод будет выглядеть так:

```
В твоей руке: 4 of Spades, Queen of Spades, 3 of Clubs
Верхняя карта: 8 of Clubs Масть: Spades
```

В примерах я использовал короткие имена.

Получение хода игрока

Теперь нам нужно спросить игрока, что он хочет сделать, и обработать его ответ. У него есть два варианта:

- сыграть карту;
- вытащить карту из колоды.

Если он решит сыграть карту, нужно убедиться, что его ход правильный. Выше мы говорили, что нужно проверить три условия:

- правильную ли карту выбрал игрок? (Не попробовал ли он сыграть несуществующей картой?)
- находится ли эта карта у него в руке?
- можно ли сыграть выбранной картой? (Подходит ли она по масти или достоинству к верхней карте или является восьмеркой?)

Но если подумать, в руке игрока могут быть только правильные карты. Если мы проверим наличие карты в руке, нам не нужно проверять ее валидность. У него не может быть четверки пастилы в руке, потому что ее никогда не было в колоде.

Давай взглянем на код для получения хода игрока и его проверки.

СЛОВАРИК

Валидация позволяет убедиться, что проверяемое является правильным, то есть допустимым и имеющим смысл.

Листинг 23.9. Получение хода игрока

```
print(«Что ты хочешь сделать? «, end='')
response = input(«Введи карту или команду 'тянуть', чтобы тянуть карту: « )
valid_play = False
while not valid_play:                          # Продолжаем, пока игрок не введет верный вариант
    selected_card = None                       # Получаем карту из руки
    while selected_card == None:               # игрока или из колоды
        if response.lower() == 'тянуть':
            valid_play = True
            if len(deck) > 0:
                card = random.choice(deck)
                p_hand.append(card)
                deck.remove(card)
                print("Ты вытянул", card.short_name)   # (1) При вводе слова тянуть тянем карту из колоды и кладем ее в руку игрока
            else:
                print(«В колоде не осталось карт»)     # После вытаскивания карты возвращаемся в основной цикл
                blocked += 1
            return
        else:
            for card in p_hand:
                if response.upper() == card.short_name:   # Проверка наличия выбранной карты в руке игрока – проверяем, пока она там не появится (или он ее не вытянет)
                    selected_card = card
            if selected_card == None:
                response = input(«У тебя нет такой карты. Попробуй снова:")

if selected_card.rank == '8':                  # Ход восьмеркой всегда допустим
    valid_play = True
    is_eight = True
elif selected_card.suit == active_suit:        # Проверка, подходит ли выбранная карта масти верхней карты
    valid_play = True
elif selected_card.rank == up_card.rank:       # Подходит ли выбранная карта значению верхней карты
    valid_play = True

if not valid_play:
    response = input(«Неправильный ход. Попробуй снова: «)
```

(Опять же, это еще не полная программа)

(1) Сейчас у нас есть правильный ход: вытаскивание карты или игра правильной картой. Если игрок тянет карту, мы добавляем ее в его руку, пока в колоде остаются карты.

Если он играет картой, нам нужно убрать ее из его руки и сделать верхней:

```
p_hand.remove(selected_card)
up_card = selected_card
active_suit = up_card.suit
print("Ты разыграл", selected_card.short_name)
```

Если сыгранная карта была восьмеркой, игрок должен указать следующую масть. Поскольку функция **`player_turn()`** становится довольно длинной, мы сделаем получение новой масти отдельной функцией **`get_new_suit()`**. Код приведен в листинге 23.10.

Листинг 23.10. Получение новой масти от игрока, после того как он сыграл восьмеркой

```
def get_new_suit():
    global active_suit
    got_suit = False
    while not got_suit:
        suit = input("Выбери масть: ")
        if suit.lower() == 'd':
            active_suit = "Diamonds"
            got_suit = True
        elif suit.lower() == 's':
            active_suit = "Spades"
            got_suit = True
        elif suit.lower() == 'h':
            active_suit = "Hearts"
            got_suit = True
        elif suit.lower() == 'c':
            active_suit = "Clubs"
            got_suit = True
        else:
            print("Неверная масть. Попробуй снова. «, end='')
    print(«Ты выбрал», active_suit)
```

Продолжаем, пока игрок не введет верную масть

Это все, что нам нужно для хода игрока. Далее мы сделаем компьютер достаточно умным для этой игры.

Ход компьютера

После хода игрока должен сыграть компьютер, поэтому мы должны рассказать программе, как играть в «Сумасшедшие восьмерки». Она должна следовать тем же правилам, что и игрок, но программа сама решает, какой картой ходить. Нужно точно рассказать ей, как действовать в разных ситуациях:

- ход восьмеркой (и выбор новой масти);
- ход другой картой;
- вытягивание карты из колоды.

Чтобы было проще, мы укажем компьютеру всегда играть восьмеркой, если таковая имеется. Это не лучшая стратегия, но сама простая.

Если компьютер ходит восьмеркой, он должен выбрать новую масть. Проще всего это сделать – посчитать количество карт каждой масти в руке компьютера, выбрать масть, карт которой у него больше всего. Снова это не самая лучшая стратегия, но один из самых простых подходов.

Если восьмерки у компьютера нет, программа пересмотрит его карты и узнает, какими можно сыграть. Из этих карт она выберет одну с самым высоким значением и сыграет ею.

Если нет возможности сыграть картой, компьютер вытянет ее из колоды. Если компьютер попробует вытянуть карту и в колоде их больше нет, он блокируется, как и игрок.

В листинге 23.11 приведен код.

Листинг 23.11. Ход компьютера

```
def computer_turn():
    global c_hand, deck, up_card, active_suit, blocked
    options = []
    for card in c_hand:
        if card.rank == '8': ←——— Ход восьмеркой
            c_hand.remove(card)
            up_card = card
            print(" Компьютер разыграл ", card.short_name)
            # suit totals:  [diamonds, hearts, spades, clubs]
            suit_totals = [0, 0, 0, 0] ←——— Считаем карты каждой масти; масть
            for suit in range(1, 5):        с самым большим количеством карт –
                for card in c_hand:         длинная масть
                    if card.suit_id == suit:
                        suit_totals[suit-1] += 1
            long_suit = 0
            for i in range (4):
                if suit_totals[i] > long_suit:
                    long_suit = i
            if long_suit == 0:  active_suit = "Diamonds"
            if long_suit == 1:  active_suit = "Hearts"      Делает длинную
            if long_suit == 2:  active_suit = "Spades"      масть активной
            if long_suit == 3:  active_suit = "Clubs"
            print(« Компьютер меняет масть на «, active_suit)
            return ←——— Заканчивает ход игрока; возвращение в основной цикл
        else:
            if card.suit == active_suit:
                options.append(card)
            elif card.rank == up_card.rank:     Проверка, какими картами
                options.append(card)            можно сыграть

    if len(options) > 0:
        best_play = options[0]
        for card in options:
            if card.value > best_play.value:    Проверка, какой ход будет лучшим
                best_play = card                (самое высокое значение карты)

        c_hand.remove(best_play)
        up_card = best_play
        active_suit = up_card.suit                          Ход картой
        print(" Компьютер разыграл ", best_play.short_name)
    else:
        if len(deck) > 0:
            next_card = random.choice(deck)
            c_hand.append(next_card)                Тянет карту, потому что
            deck.remove(next_card)                  нет подходящих
            print(" Компьютер вытянул карту")
```

```
        else:
            print(« У компьютера нет вариантов хода»)
            blocked += 1
    print(«У компьютера осталось %i карт» % (len(c_hand)))
```

Нет карт в колоде – компьютер проиграл

Мы почти закончили – осталось добавить еще несколько действий. Ты мог заметить, что очередь компьютера определена как функция и мы использовали глобальные переменные в ней. Мы могли передать переменные функции, но использование глобальных переменных работает отлично и больше похоже на реальность, где колода «глобальна» – любой может вытянуть из нее карту.

Очередь игрока также является функцией, но мы не показали первую часть ее определения:

```
def player_turn():
    global deck, p_hand, blocked, up_card, active_suit
    valid_play = False
    is_eight = False
    print("\nВ твоей руке: ", end='')
    for card in p_hand:
        print(card.short_name, end=' ')
    print(" Верхняя карта: ", up_card.short_name)
    if up_card.rank == '8':
        print(« Масть:», active_suit)
    print(«Что ты хочешь сделать? «, end='')
    response = input(«Введи карту или команду 'тянуть', чтобы тянуть
                     карту: «)
```

Осталось последнее. Узнать, кто победил!

Подсчет очков

Последнее, что необходимо нашей игре, – подсчет очков. Когда игра заканчивается, нам нужно отследить, сколько очков получил победитель за карты, оставшиеся в руке проигравшего. Нужно показать очки, полученные за игру, и суммарные очки всех игр. При добавлении этих действий основной цикл будет выглядеть как в листинге 23.12.

Листинг 23.12. Основной цикл с подсчетом очков

```
done = False
p_total = c_total = 0
while not done:
    game_done = False
    blocked = 0
    init_cards()
    while not game_done:
        player_turn()
        if len(p_hand) == 0:
            game_done = True
```

1 *Настройка колоды и рук компьютера и игрока*

Игрок побеждает

```
            print()
            print("Ты выиграл!")
            # отображение игрового счета
            p_points = 0
            for card in c_hand:                    # Добавление очков за карты компьютера
                p_points += card.value
            p_total += p_points                    # Добавление очков этой игры к общему счету
            print("Ты получил %i очков за карты компьютера" % p_points)
        if not game_done:
            computer_turn()
        if len(c_hand) == 0:                       # Компьютер побеждает
            game_done = True
            print()
            print("Компьютер выиграл!")
            # отображение игрового счета
            c_points = 0
            for card in p_hand:                    # Добавление очков за карты игрока
                c_points += card.value
            c_total += c_points
            print("Компьютер получил %i очков за твои карты" % c_points)
        if blocked >= 2:                           # Оба заблокированы, оба получают очки
            game_done = True
            print("У обоих игроков нет возможности хода. ИГРА ОКОНЧЕНА.")
            player_points = 0
            for card in c_hand:
                p_points += card.value
            p_total += p_points
            c_points = 0
            for card in p_hand:
                c_points += card.value
            c_total += c_points
        print("Ты получаешь %i очков за карты компьютера" % p_points)   # Вывод счета игры
        print("Компьютер получает %i очков за твои карты" % c_points)
    play_again = input("Играть снова (д/н)? ")
    if play_again.lower().startswith('д'):
        done = False
        print("\nИтак, у тебя %i очков," % p_total)                     # Вывод общего счета
        print("а у компьютера %i очков.\n" % c_total)
    else:
        done = True
print("\nОкончательный результат:")
print("Ты: %i Компьютер: %i" % (p_total, c_total))                     # Вывод всех очков
```

1 Функция **`init_cards()`** (не показанная здесь) просто настраивает колоду, создает руку игрока (5 карт), руку компьютера (5 карт) и открывает первую карту.

В листинге 23.12 содержится не вся программа, поэтому при запуске ты получишь сообщение об ошибке. Но если ты был внимателен, у тебя есть уже почти целая программа в редакторе. Полный листинг слишком длинный (около 200 строк кода плюс пробелы и комментарии), но ты можешь найти его в папке *Примеры* с файлами для этой книги.

Ты можешь использовать редактор IDLE, чтобы отредактировать и запустить эту программу.

Что ты узнал?

В этой главе ты узнал:

- что такое случайный порядок и случайные события;
- немного о вероятности;
- как использовать модуль **`random`** для генерации случайных событий в программе;
- как симулировать подбрасывание монеты или костей;
- как симулировать вытаскивание карт из перемешанной колоды;
- как играть в «Сумасшедшие восьмерки» (если ты раньше не знал).

Проверь свои знания

1. Опиши, что такое «случайное событие». Приведи два примера.
2. Почему бросок одной 11-гранной кости с числами от 2 до 12 отличается от броска двух 6-гранных костей, которые производят суммы от 2 до 12?
3. Каковы два способы симуляции бросания кости в Python?
4. Какую переменную Python мы использовали для одной карты?
5. Какую переменную Python мы использовали для колоды карт?
6. Какой метод мы использовали, чтобы удалить карту из колоды, когда она вытащена, или из руки, когда она сыграна?

Попробуй самостоятельно

Проведи эксперимент «десять раз подряд» с помощью программы из листинга 23.3, но попробуй разные значения «подряд». Как часто тебе попалась пятерка? А шестерка, семерка и восьмерка? Ты видишь закономерность?

ГЛАВА 24

Компьютерное моделирование

Ты когда-нибудь видел тамагочи – маленькие игрушки с экраном и несколькими кнопками, чтобы кормить виртуального питомца, класть спать, играть с ним и т. д.? У виртуального питомца есть некоторые характеристики настоящего домашнего животного. Это пример компьютерного моделирования – устройство с небольшим экраном представляет собой маленький компьютер.

В предыдущей главе ты узнал о случайных событиях и о том, как их генерировать в программе. Это был тоже вид моделирования. Моделирование – это создание компьютерной модели предмета из реального мира. Мы создали компьютерные модели монет, костей и колоды карт.

В этой главе ты узнаешь об использовании компьютерных программ для симуляции реального мира.

Моделирование реального мира

Есть много причин использования компьютера для моделирования реального мира. Иногда непрактично проводить эксперимент из-за времени, расстояния, опасности или других причин. Например, в предыдущей главе мы моделировали бросание монеты миллион раз. У большинства из нас нет времени делать это с настоящей монетой, а компьютер справился за секунды.

Иногда ученые хотят узнать, «а что, если…?». Что будет, если астероид столкнется с Луной? Мы не можем заставить настоящий астероид врезаться в Луну, а с по-

мощью компьютерного моделирования можно предсказать, что произойдет. Улетит ли Луна в космос? Столкнется ли с Землей? Как изменится ее орбита?

Когда пилоты и астронавты учатся пилотировать самолеты и космические ракеты, они не всегда имеют возможность практиковаться в реальной жизни. Это было бы очень дорого! (А ты хотел бы стать пассажиром настоящего самолета, пилотируемого стажером?) Они используют симуляторы с такими же элементами управления, как у настоящего самолета или ракеты.

С помощью моделирования можно делать многое:

- провести эксперимент или выработать навык без настоящего оборудования (кроме твоего компьютера) и без какой-либо опасности для людей;
- ускорить или замедлить время;
- провести несколько экспериментов одновременно;
- попробовать дорогие, опасные или невозможные в реальном мире вещи.

Первый наш опыт моделирования будет включать в себя гравитацию. Мы попробуем посадить космический аппарат на Луну, но у нас будет ограниченное количество топлива, поэтому нам нужно быть аккуратными с использованием двигателей. Это очень простая версия аркадной игры Lunar Lander, которая была популярна много лет назад.

Игра «Луноход»

Начнем с того, что наш космический аппарат для посадки на Луну находится на некотором расстоянии от ее поверхности. Гравитация Луны начнет его притягивать, и мы используем двигатели, чтобы замедлить его снижение и совершить мягкую посадку.

Программа будет выглядеть так: →

Небольшой ползунок слева – это управление двигателями. Его можно перетаскивать с помощью мыши вверх или вниз, чтобы управлять торможением двигателей. Датчик топлива показывает, сколько топлива у нас осталось, а текст сообщает о векторной скорости, ускорении, высоте и тяге.

Для нашей модулируемой ситуации допустим, что сила притяжения постоянна. Это не совсем правда, но если космический аппарат не очень далеко от Луны, гравитация почти постоянна – довольно близкое значение для нашего моделирования.

СЛОВАРИК

Слово «векторная скорость» (англ. *velocity*) значит почти то же самое, что и слово «скалярная скорость» (англ. *speed*), но включает в себя направление, в то время как скалярная скорость – нет. Например, выражение «50 километров в час» описывает скалярную скорость, а «50 километров в час на север» описывает векторную скорость. Многие используют просто слово «скорость», когда имеют в виду и векторную, и скалярную скорости. В нашей программе нам нужно знать, летит ли аппарат вверх или вниз, поэтому мы используем векторную скорость.
Ускорение означает, как быстро изменяется скорость. Положительное ускорение означает ее увеличение, а отрицательное – уменьшение.

Сила двигателей зависит от того, сколько топлива ты сжигаешь. Иногда она будет больше силы тяжести, иногда меньше. Когда двигатели выключены, их сила равна 0, и на корабль влияет только гравитация.

Чтобы узнать общую, или *результирующую*, силу космического аппарата, мы складываем две силы вместе. Поскольку они действуют в разных направлениях, одна будет положительной, а другая отрицательной.

Когда у нас есть результирующая сила космического аппарата, мы можем посчитать его скорость и расположение с помощью формулы.

В этом моделировании мы будем отслеживать следующее:

- высоту космического аппарата над поверхностью Луны, его векторную скорость и ускорение;
- массу космического аппарата (которая изменяется с использованием топлива);
- тягу или силу двигателей. Чем больше тяги мы используем, тем быстрее сгорает топливо;
- количество топлива в баках. При его сжигании двигателями космический аппарат становится легче, но если у нас закончится топливо, то тяги больше не будет;
- силу притяжения на космическом аппарате. Она зависит от размера Луны и массы аппарата с топливом.

Возвращение Pygame

Мы снова используем Pygame для этого моделирования. Тиканье часов Pygame будет нашей временной единицей. При каждом тике мы будем проверять результирующую силу космического аппарата, обновлять высоту, векторную скорость, ускорение и количество топлива. Затем мы используем эту информацию, чтобы обновить графику и текст.

Поскольку анимация очень простая, мы не будем использовать спрайты для космического аппарата. Но используем один для тормоза (серый прямоугольник), потому что так будет легче перетаскивать его с помощью мыши. Датчик топлива – просто пара прямоугольников, нарисованных с помощью метода **`draw.rect()`**. Текст создан с помощью объектов **`pygame.font`**, как и в случае с PyPong.

В коде будут части, выполняющие следующие действия:

- инициализация игры – настройка окна Pygame, загрузка изображений, настойка некоторых начальных значений переменных;
- определение класса спрайта для тормоза;
- подсчет высоты, векторной скорости, ускорения и потребления топлива;
- вывод информации;
- обновление датчика топлива;
- показ ракетных огней (их размер меняется в зависимости от тяги);
- обновление экрана, проверка событий мыши, обновление позиции тормоза и проверка, приземлился ли аппарат, – это основной цикл Pygame;
- вывод сообщения «Игра окончена» и финальных данных статистики.

В листинге 24.1 показан код для «Лунохода», и ты можешь найти его в файле *Листинг_24-1.py* в папке *Примеры*. Графические файлы (лунный пейзаж и космический аппарат) находятся там же. Взгляни на код и комментарии и убедись, что ты понимаешь, как программа работает. Не переживай о формулах высоты, векторной скорости и ускорения. Ты узнаешь о них из курса физики в старших классах, сдашь экзамен и забудешь (если только не пойдешь работать в «Роскосмос»). Или эта программа поможет тебе запомнить их!

Листинг 24.1. Луноход

```
import pygame, sys

pygame.init()
screen = pygame.display.set_mode([400,600])
screen.fill([0, 0, 0])
ship = pygame.image.load('lunarlander.png')
moon = pygame.image.load('moonsurface.png')
ground = 540
start = 90
clock = pygame.time.Clock()
ship_mass = 5000.0
fuel = 5000.0
velocity = -100.0
gravity = 10
height = 2000
thrust = 0
delta_v = 0
y_pos = 90
held_down = False

class ThrottleClass(pygame.sprite.Sprite):
    def __init__(self, location = [0,0]):
        pygame.sprite.Sprite.__init__(self)
        image_surface = pygame.surface.Surface([30, 10])
        image_surface.fill([128,128,128])
        self.image = image_surface.convert()
```

Посадочная площадка – y равно 540

Инициализация программы

Класс спрайта для тормоза

```
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.centery = location
```

Класс спрайта для тормоза

Считает высоту, векторную скорость, ускорение, топливо

```
def calculate_velocity():
    global thrust, fuel, velocity, delta_v, height, y_pos
    delta_t = 1/fps
    thrust = (500 - myThrottle.rect.centery) * 5.0
    fuel -= thrust /(10 * fps)
    if fuel < 0: fuel = 0.0
    if fuel < 0.1: thrust = 0.0
    delta_v = delta_t * (-gravity + 200 * thrust / (ship_mass + fuel))
    velocity = velocity + delta_v
    delta_h = velocity * delta_t
    height = height + delta_h
    y_pos = ground - (height * (ground - start) / 2000) - 90
```

Один кадр (тик) цикла Pygame

Вычитает топливо в зависимости от силы тяги

Превращает поворот спрайта тормоза по оси y в силу тяги

Формула из физики

Конвертирует высоту в значение по оси y

```
def display_stats():
    v_str = "скорость: %i m/s" % velocity
    h_str = "высота:   %.1f" % height
    t_str = "тяга:     %i" % thrust
    a_str = "ускорение: %.1f" % (delta_v * fps)
    f_str = "топливо:  %i" % fuel
    v_font = pygame.font.Font(None, 26)
    v_surf = v_font.render(v_str, 1, (255, 255, 255))
    screen.blit(v_surf, [10, 50])
    a_font = pygame.font.Font(None, 26)
    a_surf = a_font.render(a_str, 1, (255, 255, 255))
    screen.blit(a_surf, [10, 100])
    h_font = pygame.font.Font(None, 26)
    h_surf = h_font.render(h_str, 1, (255, 255, 255))
    screen.blit(h_surf, [10, 150])
    t_font = pygame.font.Font(None, 26)
    t_surf = t_font.render(t_str, 1, (255, 255, 255))
    screen.blit(t_surf, [10, 200])
    f_font = pygame.font.Font(None, 26)
    f_surf = f_font.render(f_str, 1, (255, 255, 255))
    screen.blit(f_surf, [60, 300])
```

Выводит данные с помощью шрифтов

Рисует треугольники огней

```
def display_flames():
    flame_size = thrust / 15
    for i in range (2):
        startx = 252 - 10 + i * 19
        starty = y_pos + 83
        pygame.draw.polygon(screen, [255, 109, 14], [(startx, starty),
                            (startx + 4, starty + flame_size),
                            (startx + 8, starty)], 0)
```

Вывод огней ракеты с помощью двух треугольников

```
def display_final():
    final1 = "Игра окончена"
    final2 = «Скорость %.1f м/с» % velocity
    if velocity > -5:
        final3 = "Хорошая посадка!"
```

Вывод финальных данных при окончании игры

```
        final4 = «Тебе пора в Роскосмос!»
    elif velocity > -15:
        final3 = «Ой! Жестковато, но ты выжил.»
        final4 = «В следующий раз получится лучше.»
    else:
        final3 = «Ой! Ты разбил корабль!»
        final4 = «Как полетишь домой?»
    pygame.draw.rect(screen, [0, 0, 0], [5, 5, 350, 280],0)
    f1_font = pygame.font.Font(None, 70)
    f1_surf = f1_font.render(final1, 1, (255, 255, 255))
    screen.blit(f1_surf, [20, 50])
    f2_font = pygame.font.Font(None, 40)
    f2_surf = f2_font.render(final2, 1, (255, 255, 255))
    screen.blit(f2_surf, [20, 110])
    f3_font = pygame.font.Font(None, 26)
    f3_surf = f3_font.render(final3, 1, (255, 255, 255))
    screen.blit(f3_surf, [20, 150])
    f4_font = pygame.font.Font(None, 26)
    f4_surf = f4_font.render(final4, 1, (255, 255, 255))
    screen.blit(f4_surf, [20, 180])
    pygame.display.flip()

myThrottle = ThrottleClass([15, 500])
running = True
while running:
    clock.tick(30)
    fps = clock.get_fps()
    if fps < 1: fps = 30
    if height > 0.01:
        calculate_velocity()
        screen.fill([0, 0, 0])
        display_stats()
        pygame.draw.rect(screen, [0, 0, 255], [80, 350, 24, 100], 2)
        fuelbar = 96 * fuel / 5000
        pygame.draw.rect(screen, [0,255,0],
            [84,448-fuelbar,18, fuelbar], 0)
        pygame.draw.rect(screen, [255, 0, 0],
            [25, 300, 10, 200],0)
        screen.blit(moon, [0, 500, 400, 100])
        pygame.draw.rect(screen, [60, 60, 60],
            [220, 535, 70, 5],0)
        screen.blit(myThrottle.image, myThrottle.rect)
        display_flames()
        screen.blit(ship, [230, y_pos, 50, 90])
        instruct1 = «Аккуратно, не расходуя топливо»
        instruct2 = «Хорошо: < 15 м/с Отлично: < 5м/с»
        inst1_font = pygame.font.Font(None, 24)
        inst1_surf = inst1_font.render(instruct1, 1, (255, 255, 255))
        screen.blit(inst1_surf, [50, 550])
        inst2_font = pygame.font.Font(None, 24)
        inst2_surf = inst1_font.render(instruct2, 1, (255, 255, 255))
        screen.blit(inst2_surf, [20, 575])
```

Вывод финальных данных при окончании игры

Создание объекта тормоза

Начало основного цикла Pygame

Рисуем датчик топлива

Количество топлива

Рисуем все элементы

Рисуем тормоз

Рисуем луну

Рисуем посадочную площадку

Рисуем ручку тяги

Рисуем корабль

```
        pygame.display.flip()

    else:                                    Игра закончена – выводим окончательный счет
        display_final()
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            held_down = True                 Проверка управления тормозом
        elif event.type == pygame.MOUSEBUTTONUP:   с помощью мыши
            held_down = False
        elif event.type == pygame.MOUSEMOTION:
            if held_down:
                myThrottle.rect.centery = event.pos[1]
                if myThrottle.rect.centery < 300:
                    myThrottle.rect.centery = 300
                if myThrottle.rect.centery > 500:          Обновление позиции
                    myThrottle.rect.centery = 500          тормоза
pygame.quit()
```

Попробуй запустить программу. Возможно, ты окажешься хорошим космическим пилотом! Если думаешь, что это легко, можешь изменить код, чтобы увеличить гравитацию, сделать корабль тяжелее (более массивным), залить меньше топлива или установить другую стартовую высоту либо скорость. Ты программист, можешь сам решать, как будет работать игра.

Моделирование игры в основном касается гравитации. В остальной части главы мы поговорим о другом важном факторе моделирования – времени. И проведем моделирование с отслеживанием времени.

Следим за временем

Время – важный фактор в разных моделируемых ситуациях. Иногда нам нужно ускорить время или сделать так, чтобы события происходили быстрее, чем в реальном мире, дабы нам не пришлось ждать результата слишком долго. Иногда нам нужно замедлить время, чтобы мы могли лучше взглянуть на вещи, которые обычно происходят очень быстро. А иногда нам нужно, чтобы в программе было *реальное время* – и действие происходило так же, как в реальном мире. Во всех случаях нам нужны часы для измерения времени в программе.

В каждом компьютере есть встроенные часы. Ты уже видел примеры измерения времени:

- в главе 8 мы использовали функцию **`time.sleep()`** для обратного отсчета времени;
- в наших программах на Pygame мы использовали функции **`time.delay`** и **`clock.tick`** для управления скоростью анимации или частотой кадров. Мы также использовали функцию **`get_fps()`** для проверки скорости анимации, это тоже способ измерения времени (среднее время для каждого кадра).

Итак, мы всегда следили за временем при выполнении программы, но иногда нам нужно это делать, даже когда программа не запущена. Если ты создал виртуального питомца на Python, его программа не должна быть всегда запущена. Ты будешь играть с ним некоторое время, затем закрывать ее и возвращаться к ней позже. Пока тебя не было, твой питомец должен устать, или проголодаться, или уснуть. Программа должна знать, сколько времени прошло с ее последнего запуска.

Один из способов это сделать – сохранить небольшое количество информации – текущее время – в файл перед выключением. Затем при следующем запуске программа может прочесть файл, узнать о предыдущем времени, проверить текущее время и сравнить их, чтобы узнать, сколько времени прошло с последнего запуска.

СЛОВАРИК

Когда ты сохраняешь текущее время в файл для дальнейшего использования, это называется *меткой времени*.

В Python есть особые объекты для работы со временем и датами. Ты узнаешь о них далее.

Временны́е объекты

Классы объектов времени и даты в Python определены в своем собственном модуле **`datetime`**. В этом модуле есть классы для работы с датами, временем и разницей, или *дельтой*, между двумя датами, или метками времени.

СЛОВАРИК

Слово *дельта* означает «разница». Это буква греческого алфавита, и выглядит она так: Δ (треугольник). Буквы греческого алфавита часто используют в науке и математике как сокращение определенного количества. Дельта используется для разницы между двумя величинами.

Сначала мы используем объект **`datetime`**. (Да, у класса такое же имя, как и у модуля.) Объект **`datetime`** включает в себя год, месяц, день, час, минуту и секунду. Создать его можно так (в интерактивном режиме):

```
>>> import datetime
>>> when = datetime.datetime(2019, 10, 24, 10, 45, 56)
```

Имя модуля *Имя класса*

Давай посмотрим, что у нас получилось:

```
>>> print(when)
2019-10-24 10:45:56
```

Мы создали объект **`datetime`** по имени **`when`**, который содержит значения даты и времени.

При создании объекта **datetime** порядок параметров (чисел в скобках) следующий: год, месяц, день, час, минуты, секунды. Но если ты не можешь запомнить его, порядок можно изменить, указав в Python, что является чем:

```
>>> when = datetime.datetime(hour=10, year=2019, minute=45, month=10,
second=56, day=24)
```

С объектами **datetime** можно делать многое. Можно получать отдельные части, например год, день или минуты. Можно получить форматированную строку с датой и временем. Попробуй в интерактивном режиме:

```
>>> print(when.year)
2019
>>> print(when.day)
24
>>> print(when.ctime())
Thu Oct 24 10:45:56 2019
```

Получает отдельные части объекта datetime

Выводит строковые версии даты и времени

Объект **datetime** содержит как дату, так и время. Если тебя интересует только дата, есть класс **date**, который содержит только год, месяц и день. Если только время, то есть класс **time**, который содержит только часы, минуты и секунды. Ниже представлено, как они выглядят:

```
>>> today = datetime.date(2019, 10, 24)
>>> some_time = datetime.time(10, 45, 56)
>>> print(today)
2019-10-24
>>> print(some_time)
10:45:56
```

Как и в случае с объектом **datetime**, можно передать параметры в другом порядке, если указать, в каком именно:

```
>>> today = datetime.date(month=10, day=24, year=2019)
>>> some_time = datetime.time(second=56, hour=10, minute=45)
```

Есть также способ разбиения объекта **datetime** на объект **date** и объект **time**:

```
>>> today = when.date()
>>> some_time = when.time()
```

И можно совместить объекты **date** и **time**, чтобы создать объект **datetime** с помощью метода **combine()** класса **datetime** в модуле **datetime**:

```
>>> when = datetime.datetime.combine(today, some_time)
```

Имя модуля *Имя класса* *Метод*

Теперь, когда мы познакомились с объектом **datetime** и некоторыми его свойствами, ты узнаешь, как их сочетать, чтобы узнать о разнице между ними (сколько времени прошло между двумя временными точками).

Разница между двумя точками времени

Довольно часто при моделировании нам нужно знать, сколько прошло времени. Например, в программе «Тамагочи» нам нужно знать, сколько прошло времени с того момента, когда питомца последний раз кормили, чтобы узнать, не голоден ли он.

В модуле **datetime** есть класс объекта, который поможет нам узнать разницу между двумя датами, или точками времени. Класс называется **timedelta**. Напомним, что *дельта* значит «разница». Итак, **timedelta** – это разница между двумя точками времени.

Чтобы создать **timedelta** и узнать разницу, вычитаем одну точку времени из другой:

```
>>> import datetime
>>> yesterday = datetime.datetime(2019, 10, 23)
>>> tomorrow = datetime.datetime(2019, 10, 25)
>>> difference = tomorrow - yesterday  <-- Получает разницу между двумя датами
>>> print(difference)
2 days, 0:00:00  <-- Между завтра и вчера – два дня разницы
>>> print(type(difference))
<class 'datetime.timedelta'>  <-- Разница – это объект timedelta
```

Обрати внимание, когда мы выполнили вычитание с двумя объектами **datetime**, мы получили не еще один объект **datetime**, а объект **timedelta**. Python делает это автоматически.

Небольшие промежутки времени

До сих пор мы измеряли время в долях секунды. Но объекты времени (**date**, **time**, **datetime** и **timedelta**) более точные. Они могут измерять время в микросекундах.

Для примера попробуй метод **now()**, который показывает текущее время компьютера:

```
>>> print(datetime.datetime.now())
2020-08-20 13:29:31.381481
```

Обрати внимание, во времени есть не только секунды, но и доли секунды:

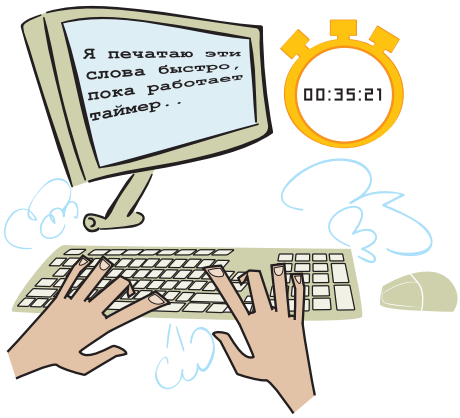
```
44.343000
```

На нашем компьютере последние три цифры всегда будут нулями, потому что часы нашей операционной системы считают только миллисекунды (тысячные доли секунды). Но для нас их точности достаточно.

Важно знать, что хотя число и выглядит как десятичная дробь, секунды на самом деле хранятся как количество секунд (целое число) и количество микросекунд (целое число): 44 секунды и 343 000 микросекунд. Чтобы сделать этот число десятичной дробью, нужна формула. Допустим, у тебя есть объект времени **some_time**, и тебе нужно количество секунд в виде дроби. Формула выглядит так:

```
seconds_float = some_time.second + some_time.microsecond / 1000000
```

Можно использовать метод **now()** и объект **timedelta**, чтобы проверить твою скорость печати. Программа в листинге 24.2 выводит случайное сообщение, а пользователь должен его набрать на клавиатуре. Программа засекает, сколько времени требуется пользователю для этого, и считает скорость печати. Попробуй.



Листинг 24.2. Измерение разницы между временными точками – проверка скорости печати

```
import time, datetime, random ←— Использует модуль time для функции sleep()

messages = [
    «Из всех деревьев, которые мы могли ударить, мы выбрали то, что
    ударит в ответ.»,
    «Если он не перестанет пытаться спасти твою жизнь, он убьет тебя.»,
    «Наш выбор показывает нашу сущность лучше, чем наши способности.»,
    «Я волшебник, а не бабуин, размахивающий палочкой.»,
    «Величие порождает зависть, зависть вызывает злобу, злоба приводит
    ко лжи.»,
    «В мечтах мы посещаем мир, принадлежащий только нам.»,
    «Я верю, что правда предпочтительнее лжи.»,
    «Рассвет, казалось, следовал за полночью с неприличной скоростью.»
    ]

print(«Тест на скорость печати. Введи следующее сообщение.
      Я засеку время.»)
time.sleep(2)
print("\nНа старт...")
time.sleep(1)
print("\nВнимание...")
time.sleep(1)
print("\nНачали:")
message = random.choice(messages) ←— Выбирает сообщение из списка
print("\n " + message)
```

Вывод инструкций

```
start_time = datetime.datetime.now()
typing = input('>')
end_time = datetime.datetime.now()
diff = end_time - start_time
typing_time = diff.seconds + diff.microseconds / 1000000
cps = len(message) / typing_time
wpm = cps * 60 / 5
print(«\nТы набрал %i знаков за %.1f секунд.» % (len(message),
                    typing_time))
print(«Это %.2f знаков в секунду, или %.1f слов в минуту» % (cps, wpm))
if typing == message:
    print("Ты не наделал ошибок.")
else:
    print(«Но ты сделал по крайней мере одну ошибку.»)
```

Старт часов

Остановка часов

Подсчитывает истекшее время

Скорость печати 1 слово = 5 символов

Вывод результатов с форматированием печати

Тебе нужно знать кое-что еще об объектах **timedelta**. В отличие от объектов **datetime**, которые содержат год, месяц, день, час, минуты и секунды (и микросекунды), объект **timedelta** содержит только дни, секунды и микросекунды. Если тебе нужны месяцы или годы, тебе нужно посчитать по количеству дней. Если тебе нужны минуты или часы, тебе нужно посчитать их по количеству секунд.

Сохранение значения времени в файл

Как мы упомянули в начале главы, иногда нам нужно сохранить значение времени в файл (на жестком диске), чтобы оно не было утеряно с завершением работы программы. Если сохранить время **now()** при завершении программы, можно проверить время при следующем ее запуске и вывести подобное сообщение:

```
Прошло 2 дня, 7 часов, 23 минуты с момента последнего использования
программы.
```

Конечно, большинство программ не делают подобного, но некоторым из них нужно знать, сколько времени они бездействовали или не были запущены. Один из примеров – программа с виртуальным питомцем. Как и в случае с тамагочи, которые были популярны несколько лет назад, можно заставить программу отслеживать время, когда ты ее не используешь. Например, ты завершил программу и вернулся к ней два дня спустя, твой виртуальный питомец будет очень голоден! Единственный способ для программы узнать, насколько голодным должен быть питомец, – знать, сколько времени прошло с последнего кормления. Это включает в себя и то время, когда программа была не запущена.

Есть несколько способов сохранения времени в файл. Мы можем записать в файл строку

```
timeFile.write(«2019-10-24 14:23:37»)
```

Затем, когда мы хотим прочитать временную метку, мы используем какой-нибудь строковый метод, например **split()**, чтобы разбить строку на разные части, например на день, месяц, год и час, минуты, секунды. Так будет хорошо.

Другой способ – использовать модуль **pickle**, о котором мы узнали в главе 22. Модуль **pickle** позволяет сохранить любой вид переменной в файл, включая объекты. Поскольку мы будем использовать объекты **datetime**, чтобы следить за временем, будет легче использовать **pickle** для их хранения и чтения.

Давай посмотрим на очень простой пример, который просто выводит сообщение о том, когда программа была запущена в последний раз. Он должен выполнять следующие действия:

- искать файл хранения и открывать его. В Python есть модуль **os** (сокращение от англ. *operating system* – операционная система), который может сказать нам, существует ли файл. Метод, который мы используем, называется **isfile()**;
- если файл существует, мы допустим, что программа раньше запускалась, и узнаем, когда это произошло в последний раз (по времени в файле);
- затем мы запишем новый файл для хранения с текущим временем;
- если программа была запущена впервые, файла со временем нет, и мы выведем сообщение о том, что он был создан.

Код показан в листинге 24.3. Попробуй.

Листинг 24.3. Сохранение времени в файл с помощью модуля **pickle**

```
import datetime, pickle
import os

first_time = True
if os.path.isfile("last_run.pkl"):
    pickle_file = open("last_run.pkl", 'rb')
    last_time = pickle.load(pickle_file)
    pickle_file.close()
    print(«Последний раз программа была запущена «, last_time)
    first_time = False

pickle_file = open("last_run.pkl", 'wb')
pickle.dump(datetime.datetime.now(), pickle_file)
pickle_file.close()
if first_time:
print(«Создан новый файл с текущим временем.»)
```

Импорт модулей **datetime**, **pickle** *и* **os**

Проверка существования файла **pickle**

Открытие файла для чтения (если он существует)

Распаковка объекта **datetime**

Открывает (или создает) файл для записи

Сохраняет объект **datetime** *с текущим временем*

Теперь у нас есть все фрагменты для создания простой программы с виртуальным питомцем, что мы и сделаем далее.

Игра «Тамагочи»

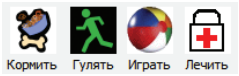
Мы напишем упрощенную программу по моделированию виртуального питомца. Можно купить игрушки с подобными питомцами (например, брелок с маленьким экраном). Есть сайты, допустим Neopetz и Webkinz, которые являются формой виртуальных питомцев. Все это, конечно же, тоже моделирование. Они подражают поведению живых существ, им бывает скучно, они устают, хотят кушать и т. д.

Чтобы они были счастливы и здоровы, их нужно кормить, играть с ними или водить к ветеринару.

Наш виртуальный питомец будет намного проще и менее реалистичным, чем те, которых можно купить или играть в онлайн-режиме, потому что я хочу донести до тебя базовую идею и не хочу, чтобы код был слишком сложным. Но ты можешь взять нашу простую версию и расширить ее или улучшить так, как тебе захочется.

Наша программа будет иметь такие функции:

- у питомца будет четыре вида активности: его можно кормить, выгуливать, играть с ним или водить к ветеринару:



- у него будет три типа статистики, за которыми нужно следить: голод, счастье и здоровье:

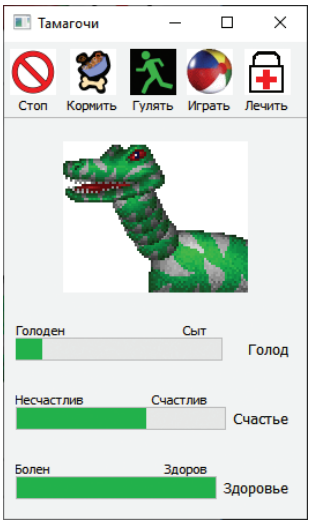
- питомец может бодрствовать или спать;

- уровень голода будет увеличиваться с течением времени. Уменьшить голод можно, покормив питомца;
- уровень голода увеличивается медленнее, когда питомец спит;
- если питомец спит, а ты производишь с ним какое-то действие, он просыпается;
- если питомец слишком голоден, его счастье уменьшается;
- если питомец очень сильно голоден, его здоровье уменьшается;
- выгуливание животного увеличивает здоровье и счастье;
- игра с питомцем увеличивает счастье;
- поход с животным к ветеринару увеличивает его здоровье;
- у питомца будет шесть разных изображений:
  - для сна;

- для бодрствования в состоянии покоя;
- для прогулки;
- для игры;
- для поглощения пищи;
- для похода к ветеринару.

Изображения будут использовать простую анимацию. Далее ты узнаешь, как это все совместится в программе.

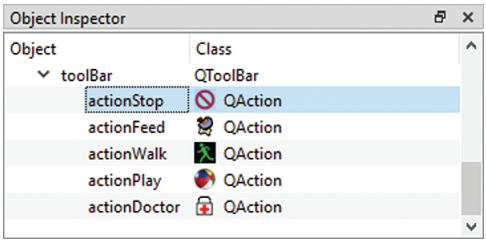
Интерфейс

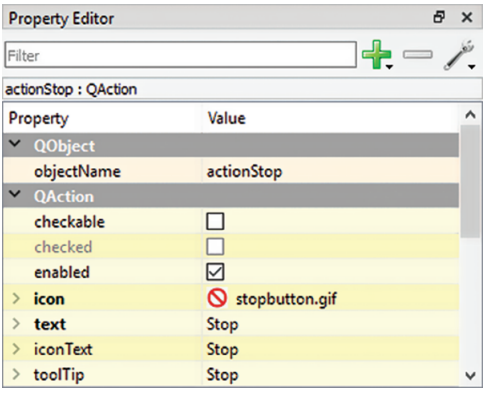


Мы с Картером сделали интерфейс для нашей программы на PyQt. В нем есть кнопки (значки на панели инструментов) действий и индикаторы для сбора статистики жизненно важных показателей. Есть область для показа графики действий питомца. Справа показано, как выглядит интерфейс.

Обрати внимание, что на рисунке в строке заголовка окна написано «Тамагочи». Чтобы установить заголовок окна, создай новую форму в приложении Qt Designer и щелкни по объекту `MainWindow` на панели **Object Inspector**. Затем в разделе на панели **Property Editor** найди свойство **windowTitle** и измени его значение на **Тамагочи** (или на любое другое).

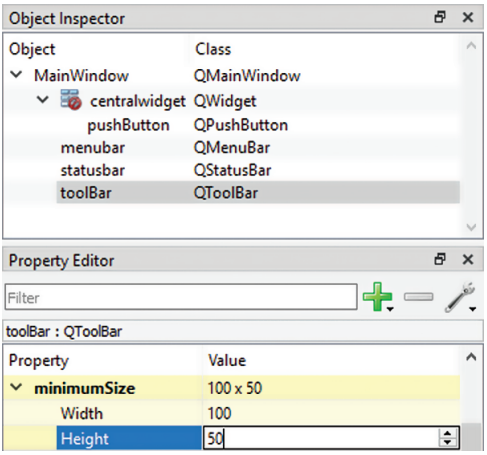
Группа кнопок действий – это тип виджета PyQt под названием *панель инструментов*. Панель инструментов содержит действия точно так же, как и меню, но разница заключается в том, что у каждого действия есть соответствующий *значок*.



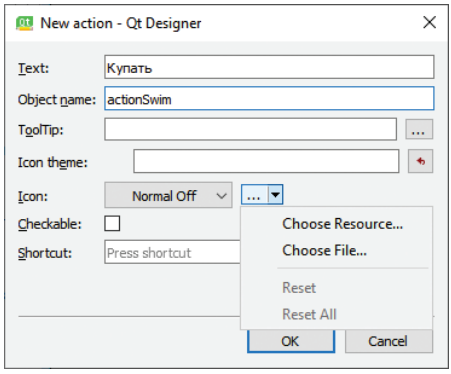


Чтобы добавить панель инструментов, щелкни правой кнопкой мыши по главному окну и выбери пункт **Add Tool Bar**. Так ты создашь панель инструментов в верхней части окна, но она будет очень маленькой. Найди пункт **toolBar** на па-

нели **Object Inspector** и щелкни по нему мышью. Затем найди на панели **Property Editor** свойство **minimumSize**. Установи параметры минимального размера: **100** в ширину и **50** в высоту.



Чтобы добавить кнопки действий на панель инструментов, перейди на вкладку **Action Editor** в правом нижнем углу окна программы Qt Designer. Щелкни правой кнопкой мыши по любому месту панели **Action Editor** и выбери пункт **New**. Появится диалоговое окно для добавления нового действия. Единственное поле, которое тебе нужно заполнить, – это **text**, а поле Qt Designer заполнит **Object name** автоматически. Затем найди в центре маленький квадратик с тремя точками (**...**) и нажми стрелку справа, указывающую вниз. Выбери пункт **Choose File**, а потом выбери файл изображения, который хочешь использовать для кнопки действия на панели инструментов.



У нас остался последний шаг к добавлению нового значка на панель инструментов. Создав новое действие, ты увидишь его на панели **Action Editor**. Теперь тебе нужно перетащить его на панель инструментов. Когда ты это сделаешь, выбранное тобой изображение нового действия появится в виде нового значка на панели инструментов. Qt Designer автоматически масштабирует изображение так, чтобы оно поместилось на панели инструментов.

Индикатор здоровья – это тип виджета, называемый **Progress Bar**. Основная графика – это **Push Button** (мы уже использовали ее) с определенными свойствами, поэтому она выглядит не как обычная кнопка, а как изображение.

Дополнительные фрагменты текста – это виджеты **Label**.

Ты можешь создать подобный графический интерфейс, используя PyQt Designer. Или ты можешь загрузить тот, который мы сделали (из папки примеров), в Qt Designer и изучить его виджеты и их свойства.

Алгоритм

Чтобы написать код для виртуального питомца, нам нужно точно определить его поведение. Мы будем использовать следующий алгоритм.

- Мы поделим «день» питомца на 60 частей, которые назовем «тиками». Каждый «тик» представляет собой 5 секунд реального времени, поэтому «виртуальный день» питомца длится 5 минут.

- Питомец будет бодрствовать 48 тиков, а затем будет спать 12 тиков. Ты можешь его разбудить, но он будет зол!
- Голод, счастье и здоровье будут оцениваться по шкале от 0 до 8.
- Когда питомец бодрствует, голод увеличивается на 1 единицу за каждый тик, счастье уменьшается на 1 единицу за 2 тика (если питомец не гуляет и не играет).
- Во сне голод увеличивается на 1 единицу за 3 тика.
- При кормлении голод уменьшается на 2 единицы за 1 тик.
- Во время игры с питомцем его счастье увеличивается на 1 единицу за 1 тик.
- На прогулке счастье и здоровье увеличиваются на 1 единицу каждые 2 тика.
- У ветеринара здоровье увеличивается на 1 единицу за 1 тик.
- Если голод равен 7, здоровье уменьшается на 1 единицу каждые 2 тика.
- Если голод равен 8, здоровье уменьшается на 1 единицу за 1 тик.
- Если питомца разбудили во время сна, счастье уменьшается на 4 единицы.
- Когда программа не запущена, питомец или бодрствует и ничего не делает, или спит.
- Когда программа снова запускается, мы считаем, сколько тиков прошло, и обновляем статистику соответственно.

Список правил кажется внушительным, но их очень легко внести в код. Фактически ты даже можешь добавить свои варианты поведения, чтобы сделать игру более интересной. Код с объяснениями следует далее.

Простая анимация

Тебе не всегда нужен Pygame для создания анимации. Мы можем создавать простую анимацию с помощью PyQt и так называемого *таймера*. Таймер создает *событие* периодически. Затем ты пишешь *обработчик событий*, для того чтобы что-то происходило при срабатывании таймера. Похоже на написание обработчика событий для действия пользователя, например для нажатия кнопки, но событие таймера генерируется программой, а не пользователем. Тип события, которое таймер генерирует по истечении времени, – это событие *истечения времени*.

В интерфейсе нашего питомца будет два таймера: для анимации и для тиков. Анимация будет обновляться каждые 0,5 секунды, а тик будет происходить каждые 5 секунд.

Когда срабатывает таймер анимации, мы меняем изображение питомца. Каждое действие (кормление, игра и т. д.) будет иметь свой набор изображений для анимации, и каждый набор будет храниться в списке. Анимация будет проходить по всем изображениям в списке. Программа будет определять, какой список использовать в зависимости от текущего действия.

Экспериментируй снова и снова

В этой программе мы используем еще кое-что новое. Называется блоком **`try-except`**.

Если программа собирается сделать что-то, что может привести к ошибке, нужно иметь способ локализовать ошибку и обработать ее, а не просто останавливать программу. Блок **try-except** делает именно это.

Например, если попробовать отрыть файл, который не существует, то ты получаешь сообщение об ошибке. Если ты не обработаешь ее, программа просто остановится в этой точке. Но ты можешь попросить пользователя ввести имя файла снова, если он просто опечатался. Блок **try-except** позволяет локализовать ошибку и продолжить выполнение программы.

Ниже представлено, как выглядит вывод при попытке открыть файл:

```
try:
    file = open(“somefile.txt”, “r”)
except:
    print(«Невозможно открыть файл. Хочешь ввести имя файла снова?»)
```

То, что ты хочешь сделать еще раз (что могло привести к ошибке), находится в блоке **try**. В этом случае это попытка открыть файл. Если действие происходит без ошибки, часть с **except** пропускается.

Если код в блоке **try** приводит к ошибке, код в блоке **except** выполняется. Он говорит программе, что делать, если произошла ошибка. Можно представлять себе это так:

```
try:
    сделать то-то (и не делать ничего иного...)
except:
    если произошла ошибка, сделать это
```

Утверждения **try-except** – это способ Python сделать то, что обычно называется *обработкой ошибки*. Обработка ошибок позволяет писать код, при котором что-то может пойти не так – то, что обычно приводит к остановке программы, – а программа все равно будет работать. Мы не будем углубляться в обработку ошибок, но хотим показать тебе основы, потому что ты встретишь их в коде питомца.

Давай взглянем на код, который приведен в листинге 24.4. Комментарии объясняют большинство происходящего. Программа довольно длинная, поэтому если не хочешь набирать ее код сам, ты найдешь листинг в папке *Примеры*. Файл ресурсов PyQt и вся графика находятся в том же каталоге. Попробуй запустить программу, изучи код и разберись, как программа работает.

Листинг 24.4. Код игры «Тамагочи»

```
import sys, pickle, datetime
from PyQt5 import QtCore, QtGui, QtWidgets, uic

formclass = uic.loadUiType(“virtualpet.ui”)[0]

class VirtualPetWindow(QtWidgets.QMainWindow, formclass):
    def __init__(self, parent=None):
        QtWidgets.QMainWindow.__init__(self, parent)
```

```
        self.setupUi(self)
        self.doctor = False
        self.walking = False
        self.sleeping = False
        self.playing = False
        self.eating = False
        self.time_cycle = 0
        self.hunger = 0
        self.happiness = 8
        self.health = 8
        self.forceAwake = False
        self.sleepImages = ["sleep1.gif","sleep2.gif","sleep3.gif",
                            "sleep4.gif"]
        self.eatImages = ["eat1.gif", "eat2.gif"]
        self.walkImages = ["walk1.gif", "walk2.gif", "walk3.gif",
                           "walk4.gif"]
        self.playImages = ["play1.gif", "play2.gif"]
        self.doctorImages = ["doc1.gif", "doc2.gif"]
        self.nothingImages = ["pet1.gif", "pet2.gif", "pet3.gif"]

        self.imageList = self.nothingImages
        self.imageIndex = 0

        self.actionStop.triggered.connect(self.stop_Click)
        self.actionFeed.triggered.connect(self.feed_Click)
        self.actionWalk.triggered.connect(self.walk_Click)
        self.actionPlay.triggered.connect(self.play_Click)
        self.actionDoctor.triggered.connect(self.doctor_Click)

        self.myTimer1 = QtCore.QTimer(self)
        self.myTimer1.start(500)
        self.myTimer1.timeout.connect(self.animation_timer)
        self.myTimer2 = QtCore.QTimer(self)
        self.myTimer2.start(5000)
        self.myTimer2.timeout.connect(self.tick_timer)

        filehandle = True
        try:
            file = open("savedata_vp.pkl", "rb")
        except:
            filehandle = False
        if filehandle:
            save_list = pickle.load(file)
            file.close()
        else:
            save_list = [8, 8, 0, datetime.datetime.now(), 0]
        self.happiness = save_list[0]
        self.health = save_list[1]
        self.hunger = save_list[2]
        timestamp_then = save_list[3]
        self.time_cycle = save_list[4]
```

Инициализация значений

Список изображений для анимации

Соединяет обработчики событий с кнопками панели инструментов

Настройка таймеров

Попытка открыть файл

Чтение файла, если он открылся

Использование настроек по умолчанию, если файл не открыт

Вытаскивает отдельные значения из списка

```
        difference = datetime.datetime.now() - timestamp_then
        ticks = int(difference.seconds / 50)
        for i in range(0, ticks):
            self.time_cycle += 1
            if self.time_cycle == 60:
                self.time_cycle = 0
            if self.time_cycle <= 48:
                self.sleeping = False
                if self.hunger < 8:
                    self.hunger += 1
            else:
                self.sleeping = True
                if self.hunger < 8 and self.time_cycle % 3 == 0:
                    self.hunger += 1
            if self.hunger == 7 and (self.time_cycle % 2 ==0) \
                                and self.health > 0:
                self.health -= 1
            if self.hunger == 8 and self.health > 0:
                self.health -=1
        if self.sleeping:
            self.imageList = self.sleepImages
        else:
            self.imageList = self.nothingImages
```

Проверка, сколько прошло времени с последнего запуска

Питомец проснулся

Моделирование всех тиков, которые прошли за время простоя

Питомец спит

Использование правильной анимации – бодрствование или сон

```
    def sleep_test(self):
        if self.sleeping:
            result = (QtWidgets.QMessageBox.warning(self, 'ВНИМАНИЕ',
«Ты уверен, что хочешь разбудить питомца? Он будет недоволен!»,
                    QtWidgets.QMessageBox.Yes | QtWidgets.QMessageBox.No,
                    QtWidgets.QMessageBox.No))

            if result == QtWidgets.QMessageBox.Yes:
                self.sleeping = False
                self.happiness -= 4
                self.forceAwake = True
                return True
            else:
                return False
        else:
            return True
```

Пример диалога

Отображающиеся кнопки

Кнопка по умолчанию

Проверка сна питомца перед совершением действия

```
    def doctor_Click(self):
        if self.sleep_test():
            self.imageList = self.doctorImages
            self.doctor = True
            self.walking = False
            self.eating = False
            self.playing = False

def feed_Click(self):
        if self.sleep_test():
            self.imageList = self.eatImages
```

Обработчик событий для кнопки лечения

Обработчик событий для кнопки кормления

```
            self.eating = True
            self.walking = False
            self.playing = False
            self.doctor = False

def play_Click(self):
        if self.sleep_test():
            self.imageList = self.playImages
            self.playing = True
            self.walking = False
            self.eating = False
            self.doctor = False

def walk_Click(self):
        if self.sleep_test():
            self.imageList = self.walkImages
            self.walking = True
            self.eating = False
            self.playing = False
            self.doctor = False

def stop_Click(self):
        if not self.sleeping:
            self.imageList = self.nothingImages
            self.walking = False
            self.eating = False
            self.playing = False
            self.doctor = False

    def animation_timer(self):
        if self.sleeping and not self.forceAwake:
            self.imageList = self.sleepImages
        self.imageIndex += 1
        if self.imageIndex >= len(self.imageList):
            self.imageIndex = 0
        icon = QtGui.QIcon()
        current_image = self.imageList[self.imageIndex]
        icon.addPixmap(QtGui.QPixmap(current_image),
                       QtGui.QIcon.Disabled, QtGui.QIcon.Off)
        self.petPic.setIcon(icon)
        self.progressBar_1.setProperty("value", (8-self.hunger)*(100/8.0))
        self.progressBar_2.setProperty("value", self.happiness*(100/8.0))
        self.progressBar_3.setProperty("value", self.health*(100/8.0))

    def tick_timer(self):
        self.time_cycle += 1
        if self.time_cycle == 60:
            self.time_cycle = 0
        if self.time_cycle <= 48 or self.forceAwake:
            self.sleeping = False
        else:
            self.sleeping = True
```

Обработчик событий для кнопки кормления

Обработчик событий для кнопки игры

Обработчик событий для кнопки прогулки

Обработчик событий для кнопки прерывания действия

Обработчик событий для таймера анимации (каждые 0.5 секунды)

Обновление изображения питомца (анимация)

Начало основного обработчика событий для 5-секундного таймера

Проверка состояния питомца (сон или бодрствование)

```
        if self.time_cycle == 0:
            self.forceAwake = False
        if self.doctor:
            self.health += 1
            self.hunger += 1
        elif self.walking and (self.time_cycle % 2 == 0):
            self.happiness += 1
            self.health += 1
            self.hunger += 1
        elif self.playing:
            self.happiness += 1
            self.hunger += 1
        elif self.eating:
            self.hunger -= 2
        elif self.sleeping:
            if self.time_cycle % 3 == 0:
                self.hunger += 1
        else:
            self.hunger += 1
            if self.time_cycle % 2 == 0:
                self.happiness -= 1
        if self.hunger > 8:  self.hunger = 8
        if self.hunger < 0:  self.hunger = 0
        if self.hunger == 7 and (self.time_cycle % 2 ==0) :
            self.health -= 1
        if self.hunger == 8:
            self.health -=1
        if self.health > 8:  self.health = 8
        if self.health < 0:  self.health = 0
        if self.happiness > 8:  self.happiness = 8
        if self.happiness < 0:  self.happiness = 0
        self.progressBar_1.setProperty("value", (8-self.hunger)*(100/8.0))
        self.progressBar_2.setProperty("value", self.happiness*(100/8.0))
        self.progressBar_3.setProperty("value", self.health*(100/8.0))

    def closeEvent(self, event):
        file = open("savedata_vp.pkl", "wb")
        save_list = [self.happiness, self.health, self.hunger, \
                     datetime.datetime.now(), self.time_cycle]
    pickle.dump(save_list, file)
        event.accept()

    def menuExit_selected(self):
        self.close()

app = QtWidgets.QApplication(sys.argv)
myapp = VirtualPetWindow()
myapp.show()
app.exec_()
```

Проверка состояния питомца (сон или бодрствование)

Добавление или вычитание единиц в зависимости от активности

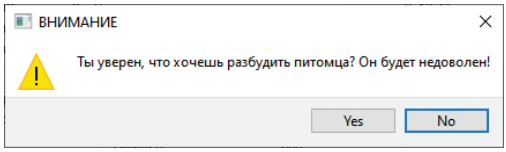
Убеждаемся, что значения не вышли за пределы диапазона

Обновляем графики

Сохраняем состояние и время в файл

Символ продления строки

Функция `sleep_test()` использует диалоговое окно PyQt «предупреждающее сообщение». Определенные параметры подсказывают ему, какие кнопки показывать и какая из этих кнопок установлена по умолчанию. Примечания к листингу 24.4 объясняют это. Это диалоговое окно (оно появляется, когда ты пытаешься разбудить своего питомца) выглядит следующим образом:



Не волнуйся, если ты не понимаешь код. Ты можешь узнать больше о PyQt, если захочешь, самостоятельно. Лучше всего начать с сайта **python-scripts.com/pyqt5**.

В этой главе мы только слегка коснулись того, что можно сделать с помощью компьютерного моделирования. Ты увидел базовые идеи моделирования условий реального мира, например гравитацию и время, но компьютерное моделирование широко используется в науке, инженерном деле, медицине, финансах и других областях. Многие из примеров моделирования очень сложные и требуют дней и недель для выполнения даже на супербыстрых компьютерах. Но и небольшие игры, которые мы создали в этой главе,– это вид моделирования, и иногда самые простые его примеры – самые интересные.

Что ты узнал?

В этой главе ты узнал:

- что такое компьютерное моделирование и почему оно используется;
- как моделировать гравитацию, ускорение и силу;
- как отслеживать и моделировать время;
- как сохранять временные метки в файл с помощью `pickle`;
- немного об обработке ошибок (`try-except`);
- как использовать таймеры для генерации периодических событий.

Проверь свои знания

1. Перечисли три причины, по которым используется компьютерное моделирование.
2. Перечисли три вида компьютерного моделирования, которые ты видел или знаешь.
3. Какой тип объекта используется для хранения разницы между двумя датами или временными точками?

Попробуй самостоятельно

1. Добавь тест «вне орбиты» в программу с луноходом. Если аппарат вылетает за пределы окна и скорость увеличивается до +100 м/с, останови программу и выведи сообщение: «Ты преодолел гравитацию Луны и вылетел в открытый космос!»
2. Добавь возможность для пользователя сыграть снова в «Луноход» после приземления и без необходимости перезапускать программу.
3. Добавь кнопку «Пауза» в интерфейс «Тамагочи». Она должна останавливать (или «замораживать») время для него вне зависимости от того, запущена программа или нет. (Подсказка: возможно, тебе придется сохранить состояние питомца при паузе в файл.)

ГЛАВА 25

А теперь подробнее о «Лыжнике»

В главе 10 ты написал игру «Лыжник» (мы надеемся!) и запустил ее. Мы сделали в коде кое-какие комментарии, но не привели подробного объяснения. Даже если ты не до конца понимаешь код, не ленись его вводить и запускать – это отличный способ изучения как программирования в целом, так и конкретно Python.

Теперь, когда ты больше узнал о Python, возможно, тебе будет интересно более детально разобраться в работе программы «Лыжник». В этой главе мы подробно обсудим это.

«Лыжник»

Сначала мы «запрограммируем» лыжника. Во время игры в «Лыжника» ты, вероятно, заметил, что герой движется только взад и вперед по экрану. Он не двигается ни вверх, ни вниз. Иллюзию скатывания по склону создают пейзажи (деревья и флажки), прокручивающиеся мимо него.

Существует пять различных образов лыжника, спускающегося с холма: один для прямого спуска, два для поворота налево (для плавного и резкого поворотов) и два для поворота направо (также для плавного и резкого поворотов). При запуске программы мы составили список этих изображений и расположили их в определенном порядке:

```
skier_images = ["skier_down.png",
                "skier_right1.png", "skier_right2.png",
                "skier_left2.png", "skier_left1.png"]
```

Позже ты узнаешь, почему этот порядок так важен.

Чтобы отслеживать, в какую сторону смотрит лыжник, мы используем переменную, называемую **`angle`**. Ее значения колеблются от –2 до +2:

- –2 = резкий поворот влево;
- –1 = плавный поворот влево;
- 0 = движение прямо вниз;
- +1 = плавный поворот вправо;
- +2 = резкий поворот вправо.

(Обрати внимание, что лыжник поворачивает в правую и в левую стороны экрана, то есть вправо и влево с нашей стороны, а не со своей.)

Значение **`angle`** подсказывает нам, какое изображение использовать. Фактически мы можем непосредственно использовать значение **`angle`** в качестве индекса к списку изображений:

- **`skier_images[0]`** – лыжник едет прямо вниз:

- **`skier_images[1]`** – лыжник совершает плавный поворот вправо:

- **`skier_images[2]`** – лыжник резко поворачивает вправо:

А теперь самое сложное. Напомним, что в главе 12, когда мы говорили о списках, сказали, что отрицательный индекс списка обернется и начнется обратный отсчет с конца списка. Итак, в данном случае:

- **`skier_images[-1]`** – лыжник совершает плавный поворот влево (это то же самое, что и **`skier_images[4]`**):

- **`skier_images[-2]`** – лыжник резко поворачивает влево (это то же самое, что и **`skier_images[3]`**):

Теперь ты понимаешь, почему мы расположили изображения в списке именно в таком порядке:

- angle = +2 (резкий поворот вправо) = **`skier_images[2]`**
- angle = +1 (плавный поворот вправо) = **`skier_images[1]`**
- angle = 0 (прямой спуск вниз) = **`skier_images[0]`**

- angle = –1 (плавный поворот влево) = **`skier_images[-1]`** (также известный как **`skier_images[4]`**)
- angle = –2 (резкий поворот влево) = **`skier_images[-2]`** (также известный как **`skier_images[3]`**)

Мы создаем класс для лыжника, который является спрайтом Pygame. Лыжник всегда находится в 100 пикселах от верхней части окна. Он начинает двигаться из центра окна, слева направо, что соответствует значению *x* = 320, потому что ширина окна – 640 пикселей. Таким образом, координаты его исходного положения – [320, 100]. Первая часть определения класса лыжника выглядит следующим образом:

```
class SkierClass(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load("skier_down.png")
        self.rect = self.image.get_rect()
        self.rect.center = [320, 100]
        self.angle = 0
```

Мы создаем метод класса для поворотов лыжника, который изменяет значение **`angle`**, загружает правильное изображение, а также устанавливает скорость лыжника. У скорости есть как *x*-, так и *y*-значения. Мы только перемещаем спрайт лыжника влево и вправо (*x*-скорость). Показатель *y*-скорости определяет, как быстро прокручивается пейзаж (как быстро лыжник спускается со склона). Когда он едет прямо вниз, его нисходящая скорость больше; когда лыжник поворачивает, его нисходящая скорость меньше. Формула скорости выглядит следующим образом:

```
speed = [self.angle, 6 - abs(self.angle) * 2]
```

Значение **`abc`** в этой строке кода получает *абсолютное значение угла*. То есть мы игнорируем знак (+ или –). Для нисходящей скорости нам все равно, в каком направлении поворачивает лыжник, нам просто нужно знать, насколько он поворачивается.

Весь код для поворота выглядит так:

```
def turn(self, direction):
    self.angle = self.angle + direction
    if self.angle < -2: self.angle = -2
    if self.angle > 2: self.angle = 2
    center = self.rect.center
    self.image = pygame.image.load(skier_images[self.angle])
    self.rect = self.image.get_rect()
    self.rect.center = center
    speed = [self.angle, 6 - abs(self.angle) * 2]
    return speed
```

Нам также нужен метод перемещения лыжника вперед и назад. Благодаря нему лыжник не выйдет за край окна:

```
def move(self, speed):
    self.rect.centerx = self.rect.centerx + speed[0]
    if self.rect.centerx < 20: self.rect.centerx = 20
    if self.rect.centerx > 620: self.rect.centerx = 620
```

Мы используем клавиши со стрелками, чтобы направить лыжника влево и вправо, поэтому добавим наш код инициализации Pygame из цикла событий. Он даст нам рабочую программу, в которой есть только лыжник. Это представлено в листинге 25.1.

Листинг 25.1. Создание игры «Лыжник», в которой есть только лыжник

```
import pygame, sys, random

skier_images = ["skier_down.png",
                "skier_right1.png", "skier_right2.png",
                "skier_left2.png", "skier_left1.png"]

class SkierClass(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load("skier_down.png")
        self.rect = self.image.get_rect()
        self.rect.center = [320, 100]
        self.angle = 0

    def turn(self, direction):
        self.angle = self.angle + direction
        if self.angle < -2: self.angle = -2
        if self.angle > 2: self.angle = 2
        center = self.rect.center
        self.image = pygame.image.load(skier_images[self.angle])
        self.rect = self.image.get_rect()
        self.rect.center = center
        speed = [self.angle, 6 - abs(self.angle) * 2]
        return speed

    def move(self, speed):
        self.rect.centerx = self.rect.centerx + speed[0]
        if self.rect.centerx < 20: self.rect.centerx = 20
        if self.rect.centerx > 620: self.rect.centerx = 620

def animate():
    screen.fill([255, 255, 255])
    screen.blit(skier.image, skier.rect)
    pygame.display.flip()

pygame.init()
screen = pygame.display.set_mode([640,640])
clock = pygame.time.Clock()
skier = SkierClass()
```

Различные изображения лыжника в зависимости от направления его движения

Не позволяй лыжнику поворачиваться больше чем на ±2

Перемещение лыжника вправо и влево

Обновление экрана

```
speed = [0, 6]

running = True
while running:
    clock.tick(30)
    for event in pygame.event.get():
        if event.type == pygame.QUIT: running = False
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                speed = skier.turn(-1)
            elif event.key == pygame.K_RIGHT:
                speed = skier.turn(1)
    skier.move(speed)
    animate()

pygame.quit()
```

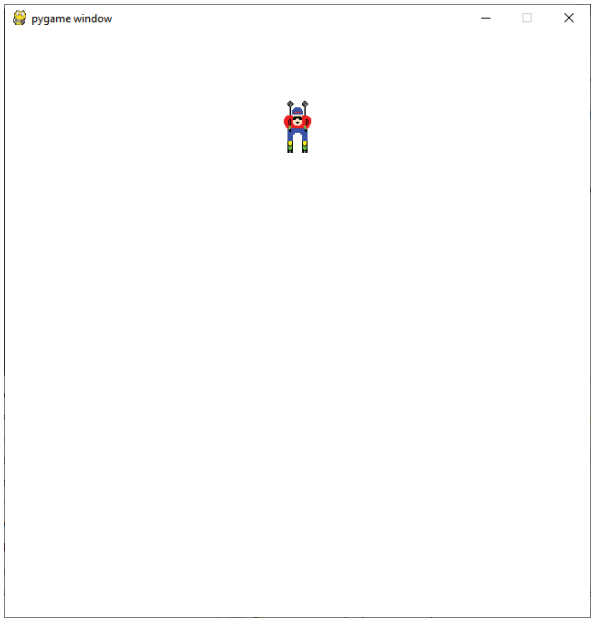
Проверка нажатия клавиш

Основной цикл событий Pygame

При нажатии клавиши ← лыжник поворачивает влево

При нажатии клавиши → лыжник поворачивает вправо

Если ты запустишь программу из листинга 25.1, то увидишь только лыжника (без игрового счета и препятствий). Ты сможешь поворачивать его влево и вправо.



Препятствия

Следующее, что мы сделаем, – это выясним, как создать препятствия – деревья и флажки. Чтобы все было просто и понятно, мы снова начнем с нуля – никаких лыжников, только препятствия. Мы соединим код лыжника с кодом препятствий в конце.

Окно для игры «Лыжник» составляет 640×640 пикселей. Чтобы все было просто, а препятствия не находились слишком близко друг к другу, мы разделим окно на сетку размером 10×10. Сетка состоит из 100 квадратов размером в 64×64 пикселей каждый. Поскольку наши спрайты препятствий не так велики, между ними будет некоторое пространство, даже если они находятся в соседних квадратах сетки.

Создание индивидуальных препятствий

Сначала нам нужно создать индивидуальные препятствия. Мы создадим для этого класс **`ObstacleClass`**. Как и сам лыжник, препятствия – это **`Sprite`**:

```
class ObstacleClass(pygame.sprite.Sprite):
    def __init__(self, image_file, location, obs_type):
        pygame.sprite.Sprite.__init__(self)
        self.image_file = image_file
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.center = location
        self.obs_type = obs_type
        self.passed = False
```

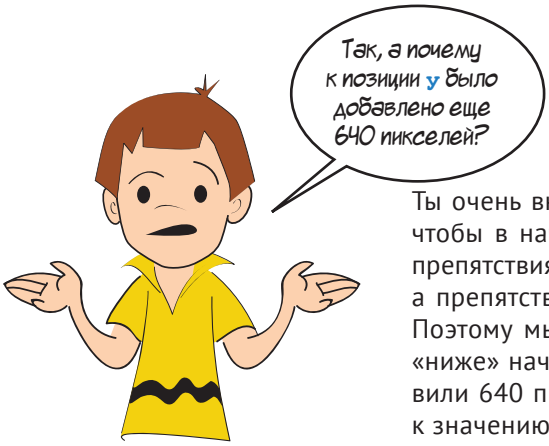
Создание карты препятствий

Теперь давай создадим несколько препятствий. Мы создадим 10 препятствий (флажков и деревьев), которые будут распределены случайным образом по 100 квадратам сетки. Препятствия могут быть двух видов: флажки или деревья. Совершенно необязательно, что у нас будет одинаковое количество флажков и деревьев, у нас может быть два флажка и восемь деревьев, семь флажков и три дерева и т. д. Они выбираются случайным образом. Их расположение на сетке также случайно.

Единственное важное условие – на одном месте не должно быть больше одного препятствия. Таким образом, мы отслеживаем, какие места использовали. Переменная **`locations`** – это список местоположений, которые мы уже использовали.

Когда мы устанавливаем новое препятствие в определенном месте, то сначала проверяем, не стоит ли уже в этом месте другое препятствие:

```
def create_map():
    global obstacles
    locations = []
    for i in range(10):                          # 10 препятствий на экране
        row = random.randint(0, 9)
        col = random.randint(0, 9)
        location = [col * 64 + 32, row * 64 + 32 + 640]   # Местоположение (координаты x и y) препятствия
        if not (location in locations):          # Убеждаемся, что два препятствия не находятся на одном месте
            locations.append(location)
            obs_type = random.choice(["tree", "flag"])
            if obs_type == "tree": img = "skier_tree.png"
            elif obs_type == "flag": img = "skier_flag.png"
            obstacle = ObstacleClass(img, location, obs_type)
            obstacles.add(obstacle)
```



Ты очень внимательный, Картер! Мы не хотим, чтобы в начале игры на экране уже были все препятствия. Вначале экран должен быть пустым, а препятствия должны появляться постепенно. Поэтому мы создаем экраны с препятствиями «ниже» начального экрана. Для этого мы добавили 640 пикселей (к высоте начального окна) к значению каждого препятствия по оси *y*.

После начала игры препятствия должны двигаться вверх (ведь лыжник едет вниз). Для этого мы меняем положение каждого препятствия по оси *y*. То, насколько сильно мы изменим его, зависит от скорости лыжника, спускающегося с холма. За это отвечает метод под названием **`update()`**, который является частью **`ObstacleClass`**:

```
def update(self):
    global speed
    self.rect.centery -= speed[1]
```

Переменная **`speed`** – скорость лыжника – глобальная переменная. У переменной **`speed`** есть как *x*-, так и *y*-координаты, поэтому, чтобы получить скорость спуска (скорость *y*), мы используем индекс [1].

Одного экрана с препятствиями будет недостаточно, нам нужно создать еще один, внизу. Как ты узнаешь, когда это нужно сделать? У нас есть переменная под названием **`map_position`**, которая отслеживает, как далеко прокрутился пейзаж. Мы делаем это в основном цикле:

```
running = True
while running:
    clock.tick(30)
    for event in pygame.event.get():
        if event.type == pygame.QUIT: running = False

    map_position += speed[1]  <-- Отслеживаем, насколько прокрутилась карта

    if map_position >= 640:
        create_map()
        map_position = 0
```

Если первый экран с препятствиями прокрутился до конца, создаем новый

Здесь у нас также есть функция **`animate()`** (как и в коде с одним лыжником), благодаря которой все обновляется.

Итак, полный код с одними препятствиями представлен в листинге 25.2.

Листинг 25.2. Создание игры «Лыжник», в которой есть только препятствия

```
import pygame, sys, random

class ObstacleClass(pygame.sprite.Sprite):
    def __init__(self, image_file, location, obs_type):
        pygame.sprite.Sprite.__init__(self)
        self.image_file = image_file
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.center = location
        self.obs_type = obs_type
        self.passed = False

    def update(self):
        global speed
        self.rect.centery -= speed[1]

def create_map():
    global obstacles
    locations = []
    for i in range(10):
        row = random.randint(0, 9)
        col = random.randint(0, 9)
        location = [col * 64 + 32, row * 64 + 32 + 640]
        if not (location in locations):
            locations.append(location)
            obs_type = random.choice(["tree", "flag"])
            if obs_type == "tree": img = "skier_tree.png"
            elif obs_type == "flag": img = "skier_flag.png"
            obstacle = ObstacleClass(img, location, obs_type)
            obstacles.add(obstacle)

def animate():
    screen.fill([255, 255, 255])
    obstacles.draw(screen)
    pygame.display.flip()

pygame.init()
screen = pygame.display.set_mode([640,640])
clock = pygame.time.Clock()
speed = [0, 6]
obstacles = pygame.sprite.Group()
map_position = 0
create_map()

running = True
while running:
    clock.tick(30)
    for event in pygame.event.get():
        if event.type == pygame.QUIT: running = False
    map_position += speed[1]
```

Класс спрайта препятствий (флажки или деревья)

Создание одного «экрана» препятствий: 640×640

На экране 10 препятствий

Избегаем создания двух препятствий на одном месте

Обновление всех элементов

Инициализация всех элементов

Основной цикл

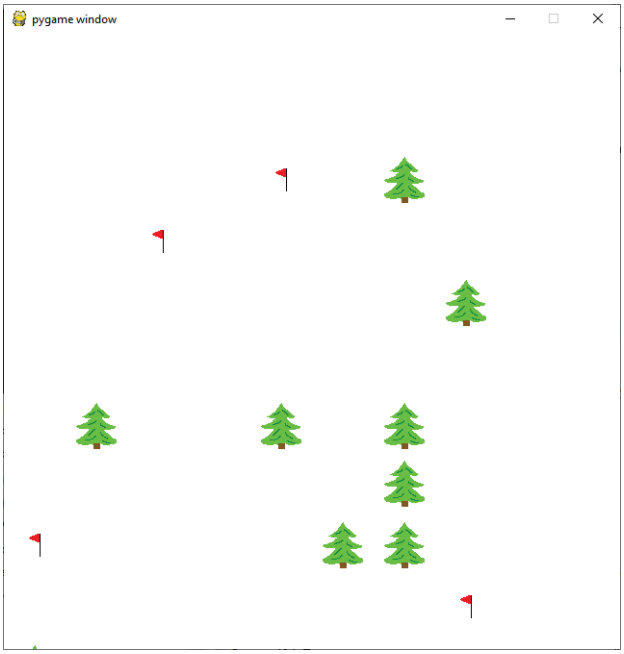
Отслеживаем, насколько прокрутилась карта препятствий

```
    if map_position >= 640:
        create_map()
        map_position = 0

    obstacles.update()
    animate()

pygame.quit()
```

Создаем новую карту препятствий ниже

Основной цикл

Если ты запустишь программу из листинга 25.2, то увидишь деревья и флажки, прокручивающиеся вверх по экрану.



Хороший вопрос, Картер! В написанном нами коде они просто продолжают прокручиваться вверх, за пределы верхней части окна, а их координаты по оси *y* становятся все более и более отрицательными. Если игра работает в течение длительного времени, мы в конечном итоге создаем большое количество спрайтов препятствий. Это может привести к тому, что наша программа в какой-то момент затормозит или закончится память. Так что нам нужно навести порядок в нашем коде.

Применяя метод **`update()`** для класса препятствий, мы проверяем, исчезло ли препятствие в верхней части экрана. Если исчезло, мы избавляемся от него. У Pygame есть для этого встроенный метод, называемый **`kill()`**. Новый метод **`update()`** выглядит следующим образом:

```
def update(self):
    global speed
    self.rect.centery -= speed[1]
    if self.rect.centery < -32:          <-- Проверяем, исчез ли спрайт в верхней части экрана
        self.kill()                      <-- Избавляемся от него
```

Теперь мы готовы объединить лыжника и препятствия:

- нам нужны **`SkierClass`** и **`ObstacleClass`**;
- наша функция **`animate()`** должна «оживлять» как лыжника, так и препятствия;
- наш код инициализации должен создать лыжника и начальную карту;
- основной цикл должен включать в себя как обработку ключевых событий лыжника, так и создание новых блоков препятствий.

В основном это комбинация листингов 25.1 и 25.2. Результат показан в листинге 25.3.

Листинг 25.3. Лыжник и препятствия в одном листинге

```
import pygame, sys, random

skier_images = ["skier_down.png", "skier_right1.png",
                "skier_right2.png", "skier_left2.png",
                "skier_left1.png"]

class SkierClass(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load("skier_down.png")
        self.rect = self.image.get_rect()
        self.rect.center = [320, 100]
        self.angle = 0

    def turn(self, direction):
        self.angle = self.angle + direction
        if self.angle < -2: self.angle = -2
        if self.angle > 2: self.angle = 2
        center = self.rect.center
        self.image = pygame.image.load(skier_images[self.angle])
        self.rect = self.image.get_rect()
        self.rect.center = center
        speed = [self.angle, 6 - abs(self.angle) * 2]
        return speed

    def move(self, speed):
```

Код лыжника

```
        self.rect.centerx = self.rect.centerx + speed[0]
        if self.rect.centerx < 20: self.rect.centerx = 20
        if self.rect.centerx > 620: self.rect.centerx = 620

class ObstacleClass(pygame.sprite.Sprite):
    def __init__(self, image_file, location, obs_type):
        pygame.sprite.Sprite.__init__(self)
        self.image_file = image_file
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.center = location
        self.obs_type = obs_type
        self.passed = False

    def update(self):
        global speed
        self.rect.centery -= speed[1]
        if self.rect.centery < -32:
            self.kill()

def create_map():
    global obstacles
    locations = []
    for i in range(10):
        row = random.randint(0, 9)
        col = random.randint(0, 9)
        location = [col * 64 + 32, row * 64 + 32 + 640]
        if not (location in locations):
            locations.append(location)
            obs_type = random.choice(["tree", "flag"])
            if obs_type == "tree": img = "skier_tree.png"
            elif obs_type == "flag": img = "skier_flag.png"
            obstacle = ObstacleClass(img, location, obs_type)
            obstacles.add(obstacle)

def animate():
    screen.fill([255, 255, 255])
    obstacles.draw(screen)
    screen.blit(skier.image, skier.rect)
    pygame.display.flip()

pygame.init()
screen = pygame.display.set_mode([640,640])
clock = pygame.time.Clock()
points = 0
speed = [0, 6]
skier = SkierClass()
obstacles = pygame.sprite.Group()
create_map()
map_position = 0
```

Код лыжника

Код препятствий

Обновление лыжника и препятствий

Создание лыжника

Создание препятствий

Инициализация всех элементов

```
running = True
while running:
    clock.tick(30)
    for event in pygame.event.get():
        if event.type == pygame.QUIT: running = False

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                speed = skier.turn(-1)
            elif event.key == pygame.K_RIGHT:
                speed = skier.turn(1)

    skier.move(speed)

    map_position += speed[1]

    if map_position >= 640:
        create_map()
        map_position = 0

    obstacles.update()
    animate()

pygame.quit()
```

Основной цикл

Если ты запустишь код из листинга 25.3, то сможешь направить лыжника вниз по склону. Также ты увидишь препятствия, прокручивающиеся мимо него. Ты заметишь, что скорость движения лыжника зависит от направления его движения (прямо вниз, влево или вправо).

Итак, мы уже почти закончили. У нас остались последние две задачи:

- определить, когда лыжник врезается в деревья или собирает флажки;
- следить за счетом и отображать его.

В главе 16 ты узнал, как обнаруживать столкновения. В коде из листинга 25.3 мы уже поместили спрайты препятствий в группу спрайтов, поэтому можем использовать функцию **`spritecollide()`**, чтобы определить, когда спрайт лыжника ударяется о спрайт дерева или флажка. Затем нам нужно установить, с чем именно столкнулся лыжник (с деревом или флажком), и отреагировать следующим образом:

- если лыжник врезался в дерево, выводим изображение аварии и вычитаем у игрока 100 очков:

- если лыжник взял флажок, добавляем 10 очков к счету игрока и удаляем флажок с экрана.

Код, необходимый для того, чтобы сделать это, находится в основном цикле. Он выглядит следующим образом:

```
hit = pygame.sprite.spritecollide(skier, obstacles, False)
if hit:
    if hit[0].obs_type == "tree" and not hit[0].passed:
        points = points - 100

        skier.image = pygame.image.load("skier_crash.png")
        animate()
        pygame.time.delay(1000)
        skier.image = pygame.image.load("skier_down.png")
        skier.angle = 0
        speed = [0, 6]
        hit[0].passed = True

    elif hit[0].obs_type == "flag" and not hit[0].passed:
        points += 10
        hit[0].kill()
```

Проверка столкновений

Лыжник врезался в дерево

Выведение изображения аварии на 1 секунду

Продолжение спуска

Запоминаем, что лыжник уже врезался в это дерево

Лыжник взял флаг

Удаление флажка

Переменная **hit** сообщает нам, со спрайтом какого препятствия столкнулся лыжник. Вообще, это список, но в нашем случае в нем содержится только один пункт, потому что лыжник может столкнуться лишь с одним препятствием за раз. Таким образом, препятствие, с которым столкнулся лыжник, – это **hit[0]**.

Переменная **passed** указывает на то дерево, с которым столкнулся лыжник. Это гарантирует, что когда лыжник продолжит скатываться вниз после столкновения с деревом, он не ударится сразу о то же самое дерево.

Теперь нам нужно отобразить счет. Для этого нам понадобится всего три строки кода. В разделе инициализации мы создадим объект **font** – часть класса **Font** в Pygame:

```
font = pygame.font.Font(None, 50)
```

В основном цикле мы выводим объект шрифта с новым счетом:

```
score_text = font.render("Счет: " + str(points), 1, (0, 0, 0))
```

А с помощью функции **animate()** мы выводим счет в левом верхнем углу:

```
screen.blit(score_text, [10, 10])
```

Итак, мы сделали все, что нужно. Если ты сложишь все это вместе, то получишь код из листинга 10.1 в главе 10. Только теперь он стал гораздо понятнее. Понимание принципов работы игры «Лыжник» должно помочь тебе при создании собственных игр.

Что ты узнал?

В этой главе ты узнал:

- как работают все части программы «Лыжник»;
- как создать прокручивающийся фон.

Попробуй самостоятельно

1. Попробуй изменить игру «Лыжник» так, чтобы она усложнялась по ходу игры. Вот что ты можешь попробовать:
   - сделать так, чтобы скорость увеличивалась по ходу игры;
   - сделать так, чтобы количество деревьев увеличивалось по ходу игры;
   - добавить «лед», чтобы лыжнику было труднее поворачивать.
2. В программе Ski Free, которая была прототипом «Лыжника», был отвратительный снеговик, который неожиданно появлялся и начинал преследовать лыжника. Если ты готов к настоящему испытанию, попробуй добавить что-то подобное в свою программу. Тебе нужно будет найти или нарисовать изображение спрайта и выяснить, как изменить код, чтобы герой вел себя необходимым образом.

ГЛАВА 26

Создание сетевых соединений с помощью сокетов

В этой главе ты узнаешь о том, как передавать данные между программами, работающими на разных компьютерах, используя *компьютерные сети*. Это довольно продвинутая тема для начинающих, но раз ты дошел до этого раздела, то, думаю, вполне можешь справиться.

Всякий раз, когда ты соединяешь несколько компьютеров вместе (с помощью кабеля или беспроводного соединения), ты создаешь компьютерную сеть. Самая известная компьютерная сеть – *интернет*, он позволяет компьютерам по всему миру обмениваться данными друг с другом. Компьютерные сети используются для самых разных целей, от заказа пиццы до координации глобальной финансовой системы.

Когда две программы хотят обменяться данными друг с другом, одна из них открывает *соединение* с другой. Как только соединение будет установлено, обе программы смогут отправлять или получать байты данных. Сетевые подключения немного похожи на файлы, потому что тебе нужно выбрать формат, или *протокол*, который будут использовать машины. Протокол определяет, как будет кодироваться информация: какие байты будут отправлены, в каком порядке и т. д.

Python поставляется со многими модулями, которые работают с сетевыми соединениями. В главе 5 мы использовали запрос **`urllib.request`** для загрузки небольшого фрагмента данных с веб-сервера. Точно так же, как модуль **`pickle`**,

о котором мы говорили в главе 22, использовал специальный формат для хранения данных в файле, модуль **`urllib.request`** использует протокол *HTTP* для получения данных из интернета.

Есть еще один способ сделать то же самое – использовать модуль **`socket`** «нижнего уровня» для прямой отправки байтов, необходимых для выполнения запроса. В листинге 26.1 показано, как это выглядит.

Листинг 26.1. Создание протокола HTTP с помощью модуля `socket`

```
import socket
connection = socket.create_connection(('helloworldbook3.com', 80))
connection.sendall('GET /data/message.txt HTTP/1.0\r\n'.encode('utf-8'))
connection.sendall(b'Host: helloworldbook3.com\r\n\r\n')
response = bytes()
while True:
    new_data = connection.recv(4096)
    if not new_data:
        break
    response += new_data
print(response.decode('utf-8'))
connection.close()
```

Открываем соединение сокета с сервером

Просим сервер прислать нам секретное сообщение

Получаем ответное сообщение от сервера

Останавливаемся, когда больше нечего скачивать

Закрываем соединение сокета

Хороший вопрос! Буква **`b`** сообщает Python, что в этой строке содержится не обычный текст, а необработанные байты. Это подводит нас к очень важной теме.

В чем разница между текстом и байтами?

В большинстве программ, которые мы уже написали, использовались строки текста: буквы, цифры, знаки препинания и пробелы. Но компьютеры мыслят по-другому: они хранят данные в памяти в виде двоичных чисел. Поэтому люди придумали способы представления текста в виде двоичных байтов. (Напомним, что байт – это всего лишь группа из восьми двоичных цифр.)

Буква	Двоичное	Десятичное
A	01000001	65
B	01000010	66
C	01000011	67
D	01000100	68
E	01000101	69
F	01000110	70
G	01000111	71
H	01001000	72

К сожалению, программисты долго спорили о том, какой способ кодировки лучше. Таким образом, существует множество различных способов кодировки символов, которые можно использовать для преобразования текста в байты и наоборот. Различные сетевые протоколы используют различные способы кодировки, поэтому при отправке текста через сокет в Python (в байтах) ты должен указать, какой способ кодировки хочешь использовать. Ты можешь сделать это с помощью метода **`encode()`**, указав название способа кодировки. (В этой книге мы всегда будем использовать кодировку UTF-8, самую популярную на сегодняшний день.)

В СТАРЫЕ ДОБРЫЕ ВРЕМЕНА

Способ кодировки под названием UTF-8 появился совсем недавно! Раньше все пользовались кодировкой ASCII (от англ. *American Standard Code for Information Interchange*). ASCII кодировала только 128 различных символов, но этого было достаточно для англоязычного текста: люди могли кодировать прописные и строчные буквы, а также цифры и знаки препинания.

Если же ты хотел кодировать текст, написанный на другом языке, тебе пришлось бы создать свою собственную систему кодирования. Многие так и поступали. Из-за этого был полный бардак! Число 169 соответствовало символу Й в русском языке, а в японском – символу ウ. В наши дни все гораздо проще. Почти все используют метод кодировки UTF-8, который включает в себя тысячи символов для каждого языка на планете.

Метод **`encode()`** вызывает специальные объекты, называемые байтами (**`bytes`**). Они работают как строки (**`strings`**), но, вместо того чтобы хранить символы в каждом индексе, они хранят целые числа. Ты сможешь увидеть различия между строками и байтами в интерактивной оболочке:

```
>>> hello_str = "Привет!"
>>> hello_bytes = hello_str.encode('utf-8')
>>> type(hello_str)
<class 'str'>
>>> type(hello_bytes)
<class 'bytes'>
>>> list(hello_str)
['П', 'р', 'и', 'в', 'е', 'т', '!']
>>> list(hello_bytes)
[208, 159, 209, 128, 208, 184, 208, 178, 208, 181, 209, 130, 33]
>>>
```

К объектам **bytes** можно применить метод **decode()**, который преобразует байты обратно в строку:

```
>>> secret_word = bytes([112, 105, 122, 122, 97])
>>> secret_word.decode('utf-8')
'pizza'
>>>
```

В Python также есть горячие клавиши для создания объектов **bytes**. Если строка содержит только символы ASCII (буквы и символы, встречающиеся на большинстве стандартных клавиатур США), ты можешь поставить перед ней букву **b**, чтобы превратить ее в объект **bytes**:

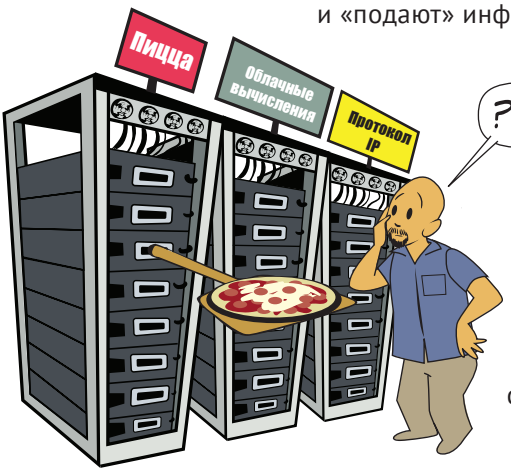
```
>>> some_bytes = b"pepperoni"
>>> type(some_bytes)
<class 'bytes'>
>>> list(some_bytes)
[112, 101, 112, 112, 101, 114, 111, 110, 105]
>>>
```

Ты можешь сделать это в Python, потому что символы ASCII соответствуют одним и тем же байтам в большинстве способов кодировки символов.

Ох! Это был довольно длинный ответ на вопрос Картера, но зато мы узнали об объектах **bytes** и кодировке символов. Далее это нам пригодится.

Серверы

В листинге 26.1 мы отправили запрос на веб-сервер. Но что же такое «сервер»? Довольно часто, когда люди говорят о серверах, они имеют в виду специальные компьютеры, которые по существу управляют интернетом. Они принимают соединения от *клиентских* программ, таких как веб-браузеры, и «подают» информацию.



Но на самом деле сервер – это просто программа, которая принимает соединения. Ты можешь запустить сервер на любом компьютере, в том числе и у себя дома! Возможно, он не сможет обрабатывать столько соединений, сколько специальные машины, которые использует большинство веб-сайтов, но главное, что он будет работать.

Ниже представлен алгоритм действий по созданию сервера.

1. Нам нужно создать сокет. В листинге 26.1 мы использовали команду **`socket.create_connection`** для создания сокета соединения, но нашему серверу понадобится собственный сокет, чтобы он мог принять это соединение.
2. Затем нам необходимо сообщить Python, порт под каким номером нужно использовать серверу. Это называется *привязкой*.

ЧТО ТАМ ПРОИСХОДИТ?

Номера портов – это способ, благодаря которому несколько программ на компьютере могут совместно использовать сетевое соединение. Разные протоколы используют разные номера портов. Например, протокол HTTP, который мы использовали в листинге 26.1, использует порт 80. Сообщения электронной почты обычно используют порт 25. Номера портов варьируются от 1 до 65 535, и обычно ты можешь использовать любой свободный порт. (Как правило, тебе придется выбрать порт, чей номер состоит минимум из четырех цифр, потому что большинство портов, чей номер состоит менее чем из четырех цифр, уже заняты.)

3. Как только мы привяжем сокет к номеру порта, нам нужно сообщить серверу, чтобы он проверял этот порт на предмет входящих соединений.
4. Когда поступит соединение, мы должны принять его и, как правило, прочитать и отправить какой-то ответ.
5. Наконец, мы должны закрыть сокеты нашего сервера, когда закончим.

Алгоритм создания сервера на Python представлен в листинге 26.2.

Листинг 26.2. Простой серверный сокет

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Создаем сокет для сервера
s.bind(('', 12345))  # Привязываем сокет к порту 12345

s.listen(1)  # Начинаем проверять порт на предмет входящих сообщений
connection, from_address = s.accept()  # Ждем, пока кто-нибудь подключится

connection.sendall(b"Hi there--oops, sorry, gotta go!\r\n")  # Отвечаем на сообщение

connection.shutdown(socket.SHUT_WR)  # Прерываем связь
connection.close()
s.close()  # Закрываем сокет сервера
```

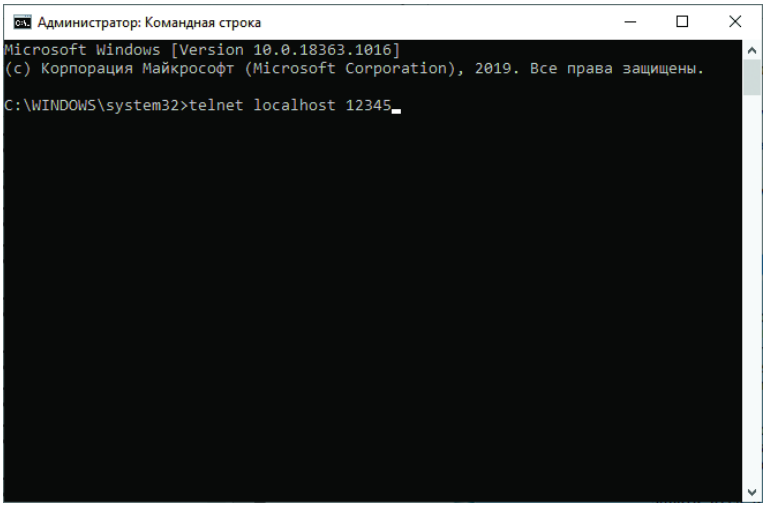
Когда ты запустишь эту программу, может показаться, что ничего не происходит. Это потому, что сервер ждет, когда кто-то подключится к нему. Мы могли бы написать свою собственную программу для подключения к этому серверу, но будет

проще, если мы используем чью-то другую. У большинства компьютеров есть встроенная программа под названием Telnet, которая позволяет подключаться к серверу очень простым способом – отправкой текстового сообщения.

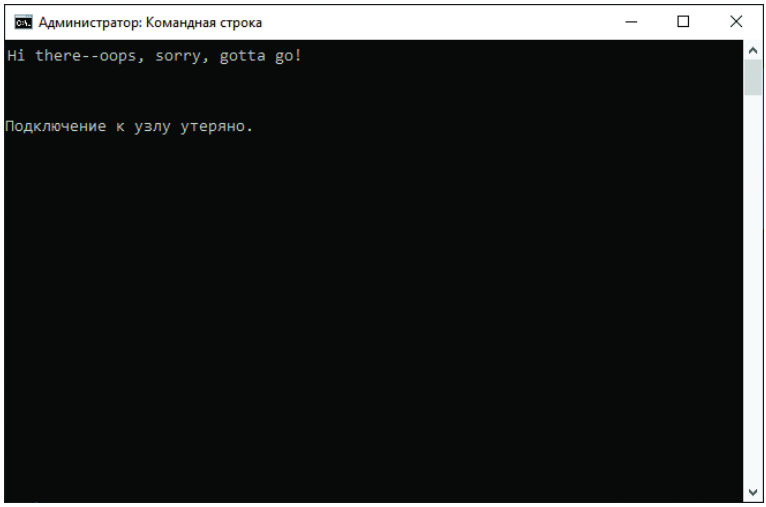
Telnet – это текстовая программа, поэтому для ее использования тебе нужно будет открыть окно оболочки. В Windows это можно сделать, найдя приложение **Командная строка** в меню **Пуск**. В macOS и Linux эта программа обычно называется **Терминал**.

Если ты используешь операционную систему Windows, тебе может потребоваться включить Telnet: открой окно **Панель управления**, найди в нем пункт **Программы и компоненты**, а затем щелкни мышью по строке **Включение или отключение компонентов Windows**. Установи флажок **Telnet Client** и нажми кнопку **OK**.

Оболочка работает почти так же, как интерактивная консоль Python. После того как у тебя есть открытая оболочка, ты сможешь запустить Telnet, введя `telnet`, затем пробел, а потом адрес машины, с которой ты хочешь соединиться. Во многих случаях это будет IP-адрес типа 127.0.0.1, но поскольку наш сервер работает на том же компьютере, что и клиент Telnet, мы будем использовать локальный адрес `localhost`. Затем введи еще один пробел, а потом – номер порта, который ты хочешь использовать. Твоя команда должна выглядеть примерно так:



Когда ты нажмешь клавишу **Enter**, Telnet подключится к твоему серверу, и ты увидишь следующее сообщение:



Если это не сработает, убедись, что программа Python все еще работает. Тебе также может потребоваться отключить брандмауэр на твоем компьютере, поскольку брандмауэры и антивирусные приложения могут мешать установке программ или получению информации.

ЧТО ТАМ ПРОИСХОДИТ?

Возможно, тебе будет интересно узнать, что такое **socket.AF_INET** и **socket.SOCK_STREAM** из листинга 26.2. Эти значения говорят модулю **socket** предоставить сокет, который использует протоколы IPv4 (**AF_INET**) и TCP (**SOCK_STREAM**).

IPv4, или *интернет-протокол версии 4*, используется для определения того, куда должны передаваться данные и как они должны туда попасть. TCP, или *протокол управления передачей*, позволяет создавать соединения и обеспечивать передачу данных через интернет без ошибок. Эти два протокола идут рука об руку, и программисты часто называют их TCP/IP. TCP/IP – это основа современного интернета.

Некоторые компьютерные сети используют протокол IPv6 – более новую версию IP с более длинными адресами. Ты можешь обновить листинг 26.2 и использовать в нем протокол IPv6, заменив **socket.AF_INET** на **socket.AF_INET6**. Тебе также может понадобиться изменить строку **s.bind(('', 12345))** на **s.bind((':', 12345))**.

Получение данных от клиента

Сервер, который мы создали в листинге 26.2, не очень полезен. Мы подключились к нему, он отправил ответное сообщение и немедленно прекратил соединение. Большинству полезных программ необходимо получать входные данные от пользователя, а большинству серверов необходимы некоторые данные от клиента. В листинге 26.3 показано, как это делается.

Листинг 26.3. Реагирование серверного сокета на ввод данных пользователем

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 12345))
s.listen(1)
connection, from_address = s.accept()
connection.sendall(b"Hi there! Welcome to my server!\r\nWhat's your
                   name? ")
name = bytes()
while True:
    next_character = connection.recv(1)
    if next_character in [b'', b'\r', b'\n']:
        break
    else:
        name += next_character
connection.sendall(b"Nice to meet you, " + name + b"! Goodbye for now!\
                   r\n")
connection.shutdown(socket.SHUT_WR)
connection.close()
s.close()
```

Запускаем сервер и подтверждаем соединение

Ждем, пока клиент отправит нам сообщение

*Прекращаем читать после нажатия клавиши **Enter** или если больше нет доступных данных*

Считывание сообщения клиента по одному байту за раз

Создание чат-сервера

Теперь, когда мы знаем основы сокетов, клиентов, серверов и отправки данных по сети, давай используем их для создания простого чат-сервера.

Наши первые серверы могли принимать только одно соединение. Они принимали его, отвечали, а затем закрывали соединение и выходили. Наш чат-сервер должен обрабатывать несколько подключений одновременно, чтобы несколько пользователей могли подключиться к нему и обмениваться сообщениями друг с другом. Когда один пользователь отправляет нашему серверу сообщение, мы должны отправить его всем остальным.

Обычно, когда ты вызываешь функцию `.recv()` на сокете, Python ждет, пока поступят некоторые данные; в это время ты не можешь делать ничего другого. Это не подойдет для нашего чат-сервера, потому что во время ожидания сообщения от одного из клиентов мы не сможем узнать, отправил ли сообщение другой клиент. Поэтому нам нужно изменить поведение сервера – мы должны ждать сообщения не от кого-то конкретно, нам нужно сосредоточиться на первом сообщении не-

зависимо от того, от кого именно оно придет. Алгоритм создания такого сервера состоит из двух частей:

- нам нужно перевести сокет в *неблокирующий режим* – это означает, что пока сокет ждет ввода, наша программа может делать и что-то другое. Мы можем сделать это с помощью функции **`.setblocking(False)`**;
- мы можем использовать модуль **`select`** для одновременного ожидания данных от нескольких сокетов. Функция **`select.select()`** вернет значение, когда у любого из сокетов появятся новые данные.

Чтобы упорядочить эту программу, мы решили создать класс под названием **`Client`**. Мы создадим его для каждого клиентского соединения. Мы будем вести список открытых клиентских сокетов, а также отслеживать экземпляр **`Client`**, соответствующий каждому сокету.

Листинг 26.4. Чат-сервер

```
import select, socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind(('', 12345))
server_socket.listen()
server_socket.setblocking(False)

client_sockets = []
client_objects = {}

class Client:
    def __init__(self, socket):
        self.socket = socket
        self.text_typed = b""
        self.username = None

        socket.setblocking(False)
       msg = b'Welcome to my server!\r\n'
        msg += b'Whats your name:\r\n'
        socket.send(msg)

    def receive_data(self):
        data = self.socket.recv(2048)
        if not data:
            self.close_connection()
            return
        for char in data:
            char = bytes([char])
            if char == b'\n':
                self.handle_command(self.text_typed.strip())
                self.text_typed = b""
            else:
                self.text_typed += char
```

Запускаем сервер и ждем соединения

Переводим сокеты в неблокирующий режим

Чтение до 2048 байтов за раз

Когда нет новых данных для чтения, это означает, что клиент закрыл соединение

Мы вызовем это, когда будет больше данных для чтения или клиент закроет соединение

*Нажимая клавишу **Enter**, клиент отправляет сообщение, которое ввел*

```
    def handle_command(self, command):
        global client_objects
        if self.username == None:
            self.username = command
            msg = b'Hi, ' + self.username + b'!'
            msg += b' Type a message and press Enter to send it.\r\n'
            self.socket.send(msg)
        elif command == b'/quit':
            self.close_connection()
        else:
            msg = b'[' + self.username + b']: ' + command + b'\r\n'
            for client_object in client_objects.values():
                if client_object == self or client_object.username == None:
                    continue
                client_object.socket.send(msg)

    def close_connection(self):
        global client_sockets, client_objects
        client_sockets.remove(self.socket)
        del client_objects[self.socket.fileno()]
        self.socket.close()

while True:
    ready_to_read = select.select([server_socket] + client_sockets, [],
                                  [])[0]
    for sock in ready_to_read:
        if sock == server_socket:
            new_connection, address = sock.accept()
            client_sockets.append(new_connection)
            client_objects[new_connection.fileno()] =
                Client(new_connection)
            server_socket.listen()
        else:
            client_objects[sock.fileno()].receive_data()
```

Отправляем сообщение всем, кто подключен к серверу

Ждем, пока что-нибудь произойдет с одним из наших сокетов

Запускается, когда на сервере появляется новое соединение

Продолжаем ждать новые соединения

Запускается, когда клиент отправляет сообщение или прерывает соединение

Возможно, ты заметил, что на этот раз мы внесли некоторые изменения в настройки сокета нашего сервера. Ранее, если сервер переставал работать, не закрыв сокет (возможно, произошел сбой или ты нажал сочетание клавиш **Ctrl**+**C**), ты, вероятно, не смог бы сразу запустить другой сервер с тем же номером порта. Строка

```
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

помогает предотвратить это.

Ты можешь протестировать эту программу, открыв несколько окон оболочки на своем компьютере и запустив **`telnet localhost 12345`** в каждом из них. Если ты наберешь сообщение в одном окне и нажмешь клавишу **Enter**, оно должно появиться во всех остальных.

Да, Картер. Разумеется, это неудобно. Чтобы наш чат-сервер был полезным, люди должны иметь возможность подключаться к нему с нескольких разных компьютеров. Для этого нам нужно узнать о том, как работают IP-адреса.

Несколько слов об IP-адресах

Когда ты подключаешься к сети с помощью кабеля или Wi-Fi, сеть присваивает твоему компьютеру IP-адрес. IPv4-адреса обычно состоят из четырех чисел, разделенных точками, например **`192.168.27.86`**. IPv6-адреса намного длиннее и могут содержать буквы, цифры и двоеточия, например **`fe80::fa5d:8468:4ce2:c681`**.

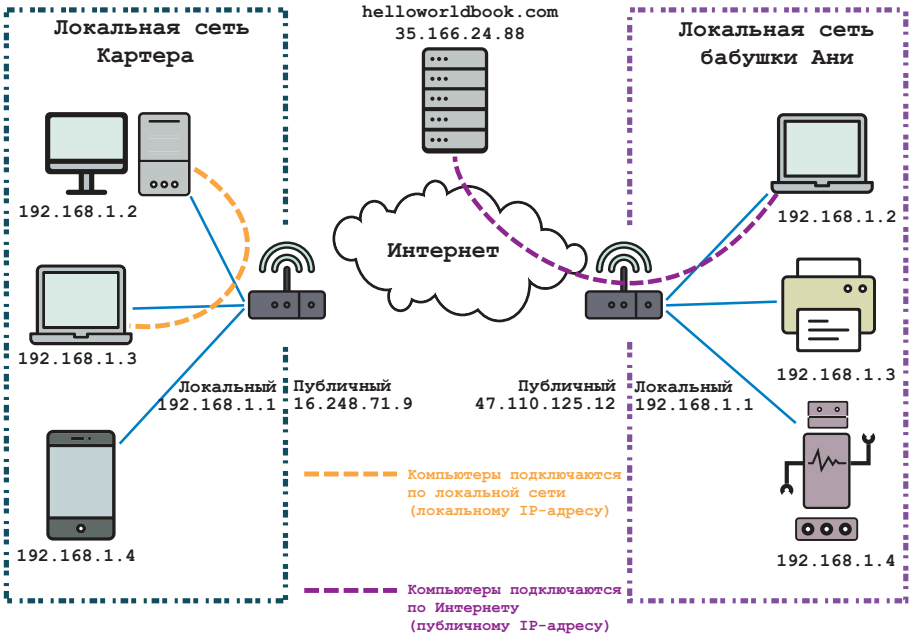
Другие компьютеры в сети могут подключаться к тебе, используя тот же IP-адрес, – это немного похоже на то, как другие люди используют твой номер телефона, чтобы позвонить тебе. Однако в большинстве случаев IP-адрес, который предоставляет тебе сеть, может использоваться только компьютерами в той же самой сети, например в школе или дома. Адрес, который ты используешь в своей собственной сети, называется твоим *локальным адресом*.

Поэтому если ты хочешь создать сервер чата, к которому можно будет подключиться с разных компьютеров, ты должен соблюсти два условия: во-первых, пользователи, которые хотят подключиться со своих компьютеров, должны знать IP-адрес того компьютера, на котором запущен сервер; во-вторых, все компьютеры должны находиться в одной сети.

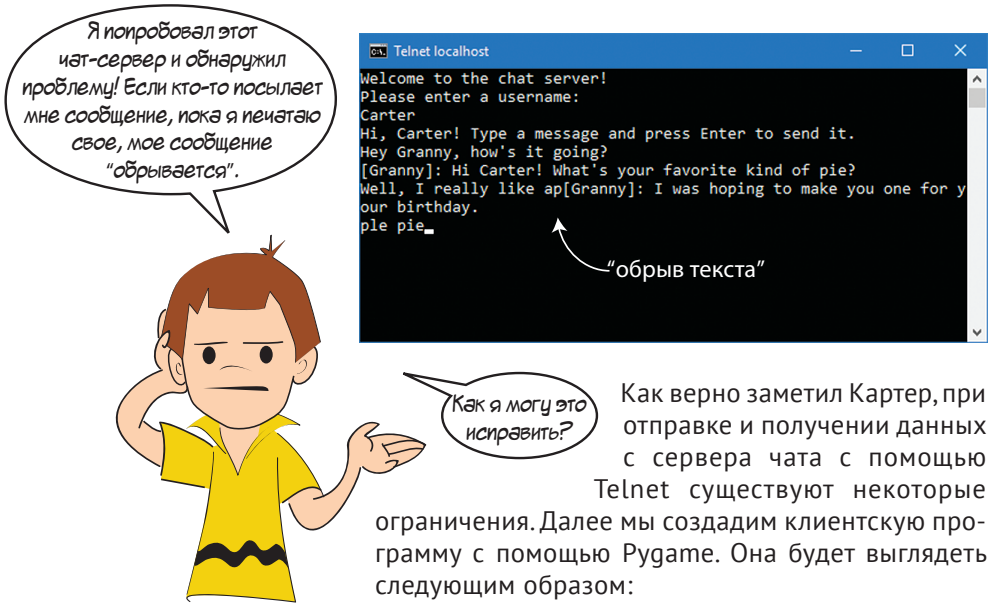
Узнать локальный адрес компьютера можно по-разному, это зависит от используемой операционной системы. Загляни в сетевые настройки своего компьютера, введи команду **`ipconfig`** или **`ifconfig`** в оболочку или введи в поисковой системе запрос «Какой локальный IP-адрес у моего компьютера». Затем ты можешь ввести этот адрес в Telnet на другом компьютере для подключения к серверу; команда будет выглядеть примерно так: **`telnet 192.168.1.38 12345`**.

Если компьютер подключен к интернету, он также имеет *глобальный*, или *публичный*, *адрес*, который может использовать для подключения к серверам за пре-

делами твоей локальной сети. Однако обычно все в локальной сети будет иметь один и тот же глобальный адрес, поэтому посторонние люди, скорее всего, не смогут использовать его для подключения к серверам на твоем компьютере.



Создание клиентской программы чата



Как верно заметил Картер, при отправке и получении данных с сервера чата с помощью Telnet существуют некоторые ограничения. Далее мы создадим клиентскую программу с помощью Pygame. Она будет выглядеть следующим образом:

Это больше похоже на обычную программу чата, где ты можешь вводить сообщения в нижней части экрана, не прерываясь. Итак, сначала давай напишем программу Pygame, которая позволит нам ввести сообщение!

Листинг 26.5. Ввод сообщения

```
import pygame

pygame.init()
screen_width, screen_height = screen_size = (640, 320)
font = pygame.font.Font(None, 50)
bg_color = (0, 0, 0)
text_color = (255, 255, 255)

screen = pygame.display.set_mode(screen_size)
pygame.key.set_repeat(300, 100)
typing_text = ""          <-- Начинаем без текста на экране
running = True
clock = pygame.time.Clock()

while running:
    clock.tick(60)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
```

```
            if event.key == pygame.K_BACKSPACE:
                if typing_text:
                    typing_text = typing_text[:-1]
            elif event.key == pygame.K_RETURN:
                typing_text = ""
            else:
                typing_text += event.unicode

        screen.fill(bg_color)
        typing_surf = font.render(typing_text, True, text_color, bg_color)
        screen.blit(typing_surf, (0, screen_height -
                                  typing_surf.get_height()))
        pygame.display.flip()

pygame.quit()
```

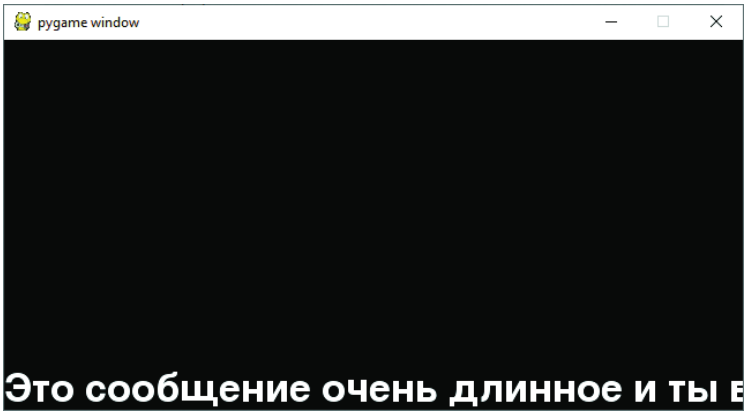
Удаляем последний символ отображаемого текста

Добавляем букву, набранную пользователем, в отображаемый текст

Обновляем текст внизу экрана

Большая часть этого кода Pygame должна быть тебе знакома из предыдущих глав. Событие **`event.unicode`** – это свойство ключевых событий Pygame, которое содержит строку с введенными буквой или символом.

Если ты запустишь эту программу, то сможешь ввести сообщение, которое появится в нижней части окна. Если ты наберешь достаточно длинное сообщение, оно выйдет за пределы окна:



Это происходит потому, что функция **`pygame.Font.render()`** всегда помещает текст в одну строку. Она не знает ширину окна, поэтому не может автоматически выполнять перенос слов. Давай это исправим.

СЛОВАРИК

Перенос слов, также называемый *разрывом строк*, – это разбивка текста на несколько строк, цель которой – избежать слишком длинной строки. Для всего текста в этой книге настроен перенос – иначе страницы были бы очень, очень широкими!

И в EasyGui, и в PyQt есть встроенный перенос слов. К сожалению, в Pygame его нет, поэтому нам придется настроить его самостоятельно.

Листинг 26.6. Ввод сообщения с переносом слов

```
import pygame
pygame.init()
screen_width, screen_height = screen_size = (640, 320)
font = pygame.font.Font(None, 50)
bg_color = (0, 0, 0)
text_color = (255, 255, 255)
space_character_width = 8 ← Обрати внимание, что мы добавили эту константу
screen = pygame.display.set_mode(screen_size)
pygame.key.set_repeat(300, 100)
def message_to_surface(message):
    words = message.split(' ')
    word_surfs = []
    word_locations = []
    word_x = 0
    word_y = 0
    text_height = 0
    for word in words:
        word_surf = font.render(word, True, text_color, bg_color) ← Создаем поверхность с одним словом
        if word_x + word_surf.get_width() > screen_width:
            word_x = 0
            word_y = text_height
        word_surfs.append(word_surf)
        word_locations.append((word_x, word_y))
        word_x += word_surf.get_width() + space_character_width ← Ставим пробел между этим словом и следующим
        if word_y + word_surf.get_height() > text_height:
            text_height = word_y + word_surf.get_height()
    surf = pygame.Surface((screen_width, text_height))
    surf.fill(bg_color)
    for i in range(len(words)):
        surf.blit(word_surfs[i], word_locations[i])
    return surf
```

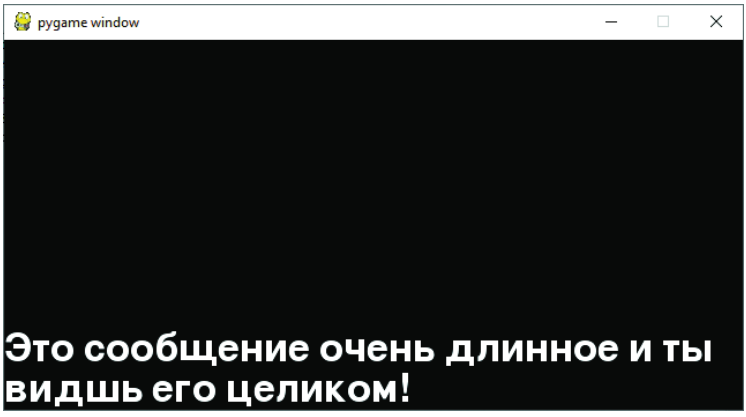
Если слово не помещается в одну строку, переходим на другую строку

Отображаем все поверхности со словами на одной большой поверхности

```
typing_text = ""
running = True
clock = pygame.time.Clock()
while running:
    clock.tick(60)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_BACKSPACE:
                if typing_text:
                    typing_text = typing_text[:-1]
            elif event.key == pygame.K_RETURN:
                typing_text = ""
            else:
                typing_text += event.unicode
    screen.fill(bg_color)
    typing_surf = message_to_surface(typing_text)
    screen.blit(typing_surf, (0, screen_height -
                              typing_surf.get_height()))
    pygame.display.flip()
pygame.quit()
```

Изменяем эту строку, чтобы вызвать нашу новую функцию отображения текста

Глядя на этот код, ты можешь заметить, что теперь мы отображаем каждое слово на его собственной поверхности. Таким образом, мы можем контролировать, в каком месте отображается каждое слово. Затем мы объединяем все поверхности слов в одну большую поверхность и выводим ее на экран.



Теперь мы готовы подключить эту программу к чат-серверу!

Наш клиент будет постоянно ждать сообщения с сервера. Иногда будут появляться новые сообщения, а иногда нет. Если сообщения нет, метод **`recv()`** обычно приостанавливает работу программы и ждет появления нового сообщения. Но мы хотим делать другие вещи, пока ждем сообщений, поэтому нам нужно использовать неблокирующий режим.

В неблокирующем режиме вызов функции **`recv()`** при отсутствии данных вызывает ошибку блокировки (**`BlockingIOError`**) вместо приостановки работы программы. Если мы обработаем эту ошибку с помощью блока **`try-except`** (как в главе 24), наша программа будет продолжать работать, пока сокет ожидает ввода.

Листинг 26.7. Сетевой клиентский чат

```
import pygame
import socket ◄──── Добавляем этот импорт
pygame.init()
screen_width, screen_height = screen_size = (640, 640) ◄──┐
font = pygame.font.Font(None, 50)                          │ Делаем экран выше, чтобы в нем
bg_color = (0, 0, 0)                                       │ поместилось больше сообщений
text_color = (255, 255, 255)
space_character_width = 8
message_spacing = 8
connection = socket.create_connection(('localhost', 12345))  Подключаемся
connection.setblocking(False)                                к серверу
screen = pygame.display.set_mode(screen_size)
pygame.key.set_repeat(300, 100)

def message_to_surface(message):
    words = message.split(' ')
    word_surfs = []
    word_locations = []
    word_x = 0
    word_y = 0
    text_height = 0
    for word in words:
        word_surf = font.render(word, True, text_color, bg_color)
        if word_x + word_surf.get_width() > screen_width:
            word_x = 0
            word_y = text_height
        word_surfs.append(word_surf)
        word_locations.append((word_x, word_y))
        word_x += word_surf.get_width() + space_character_width
        if word_y + word_surf.get_height() > text_height:
            text_height = word_y + word_surf.get_height()
    surf = pygame.Surface((screen_width, text_height))
    surf.fill(bg_color)
    for i in range(len(words)):
        surf.blit(word_surfs[i], word_locations[i])
    return surf
message_surfs = []

def add_message(message):                                  Новый код
    if len(message_surfs) > 50:                            для отслеживания
        message_surfs.pop(0)                               старых сообщений
    message_surfs.append(message_to_surface(message))
```

```
text_from_socket = b''

def read_from_socket():
    global connection, text_from_socket, running
    try:
        data = connection.recv(2048)
    except BlockingIOError:
        return

    if not data:
        running = False
    for char in data:
        char = bytes([char])
        if char == b'\n':
            add_message(text_from_socket.strip().decode('utf-8'))
            text_from_socket = b''
        else:
            text_from_socket += char

def redraw_screen():
    screen.fill(bg_color)

    typing_surf = message_to_surface("> " + typing_text)
    y = screen_height - typing_surf.get_height()
    screen.blit(typing_surf, (0, y))

    message_index = len(message_surfs) - 1
    while y > 0 and message_index >= 0:
        message_surf = message_surfs[message_index]
        message_index -= 1
        y -= message_surf.get_height() + message_spacing
        screen.blit(message_surf, (0, y))
    pygame.display.flip()
running = True
typing_text = ""
clock = pygame.time.Clock()
while running:
    clock.tick(60)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_BACKSPACE:
                if typing_text:
                    typing_text = typing_text[:-1]
            elif event.key == pygame.K_RETURN:
                add_message('Ты: ' + typing_text)
                connection.send(typing_text.encode('utf-8') + b"\r\n")
                typing_text = ""
            else:
                typing_text += event.unicode
```

Обрабатываем ошибку, вызванную неблокирующим режимом

Останавливаем программу, когда закрывается соединение

Новый код для чтения из сокета

Преобразуем сообщение с сервера из байтов в строку и отображаем его

Новая функция для отображения всех сообщений на экране

Новый код для отправки сообщения на сервер

```
    read_from_socket()
    redraw_screen()
pygame.quit()
connection.close()
```

Обновляем основной цикл, чтобы вызвать новые функции (к строкам read_from_socket() и redraw_screen())

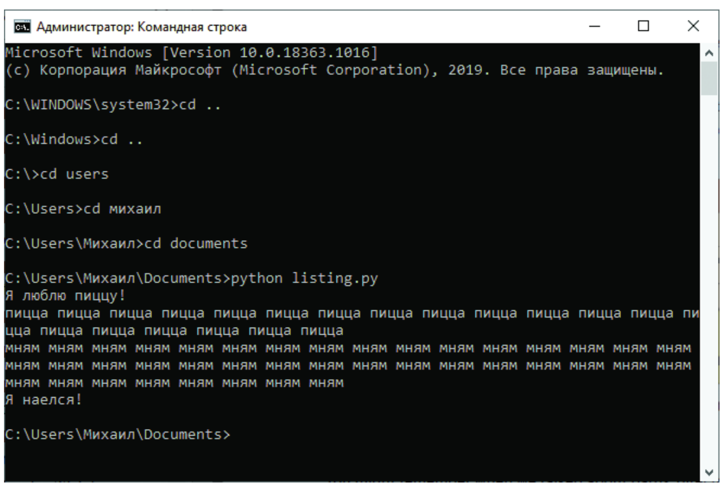
Когда закончим, закрываем соединение (к строке connection.close())

Итак, теперь у нас есть чат-сервер и клиент. Запусти их и посмотри, что получится! Если ты запускаешь сервер на другом компьютере, нежели клиента, тебе нужно будет изменить слово «**localhost**» на IP-адрес сервера.

Ну, Картер, похоже, нам придется придумать другой способ, как запустить одну из этих программ. До сих пор для запуска программ мы использовали IDLE. Однако можно запускать программы Python из оболочки, как Telnet.

На самом деле это очень даже несложно. Тебе просто нужно ввести слово **python**, пробел, затем имя твоей программы, и она будет работать. Самое сложное – убедиться, что твоя оболочка находится в нужном каталоге, чтобы она могла найти твою программу и запустить ее.

Возможно, ты помнишь, что мы говорили о путях, каталогах и подкаталогах в главе 22. В оболочке ты перемещаешься по дереву каталогов с помощью команды **cd**. Чтобы перейти в подкаталог, ты вводишь **cd**, пробел, а затем имя подкаталога. Чтобы перейти «вверх» на один уровень в дереве каталогов, ты вводишь **cd**. Ниже представлен пример перехода к нужному каталогу для запуска одного из наших листингов с помощью оболочки:



Возможно, тебе придется приложить усилия, чтобы поместить оболочку в нужный каталог, но как только ты это сделаешь, остальное будет легко. Ты можешь запустить клиента в оболочке, а сервер – в IDLE, или наоборот.

Что ты узнал?

В этой главе ты узнал:

- почему компьютеры должны кодировать текст, перед тем как отправить его через интернет;
- как использовать сокеты для подключения к другому компьютеру;
- как отправлять и получать данные через сокеты;
- как принимать соединения сокета от других компьютеров;
- о неблокирующем режиме, который позволяет тебе заниматься другими вещами во время ожидания данных от сокета;
- о том, как работает перенос слов;
- как запускать программы в оболочке.

Проверь свои знания

1. Что такое сервер?
2. Что тебе нужно сделать со строками, прежде чем передавать их в **`socket.sendall()`**?
3. Почему ты не можешь подключиться непосредственно со своего домашнего компьютера к домашнему компьютеру твоего друга?
4. Какую команду ты будешь использовать для перемещения по дереву каталогов в оболочке?

Попробуй самостоятельно

1. Используя команду **`telnet`**, попробуй вручную отправить HTTP-запрос непосредственно на веб-сервер. Если ты не знаешь, что можно отправить, обратись к листингу 26.1.
2. На многих серверах можно использовать команду **`/me`** для отправки специальных сообщений. Например, если Картер напечатает

   ```
   /me голодный
   ```

   другие пользователи получат следующее сообщение:

   ```
   * Картер голодный
   ```

Попробуй сделать так, чтобы в твоем чате также можно было использовать эту команду. (Тебе нужно будет изменить только сервер, а не клиента.)

3 Во многих чатах, когда присоединяется новый пользователь, всем приходит оповещение. Сделай это и в нашем чате.

4 Добавь смайлики в программу чата. Смайлики заменяют определенные слова картинками. Например, ты можешь заменить слово `:pizza:` на изображение куска пиццы. (Тебе нужно будет изменить только клиента, а не сервер.)

ГЛАВА 27

Что дальше?

Это конец нашей книги! Если ты прочел ее полностью и попробовал все примеры, у тебя теперь есть хорошая база по программированию, и ты можешь делать многое со своими знаниями.

Здесь мы расскажем тебе, где искать информацию по программированию. Есть источники по общему программированию, по Python, по игровому программированию и другим разделам.

То, что именно тебе нужно еще узнать, зависит от твоих целей. Ты уже познакомился с Python, и многие вещи, которые ты узнал из этой книги, – это общие идеи и концепции программирования, они пригодятся тебе при работе и с другими компьютерными языками. Как и чему тебе нужно научиться, зависит от того, в каком направлении ты хочешь дальше развиваться. Игры? Веб-программирование? Робототехника? (Для работы с роботами нужно программное обеспечение, которое говорит им, что нужно делать.)

Юным программистам

Наши юные читатели! Если вам понравилось изучать программирование на Python, вы также можете попробовать писать программы на другом языке. Scratch – это «язык программирования» для детей, он почти полностью визуальный. Программируя на Scratch, вы в основном не пишите код, а создаете программы, перетаскивая спрайты и упорядочивая кодовые «блоки», чтобы контролировать то, что они делают. Scratch доступен по адресу **www.scratch.mit.edu**.

Еще один язык, похожий на Scratch, – это Squeak Etoys. Программируя на Squeak, вы также можете создавать программы, перетаскивая спрайты (в этом они похожи

со Scratch). Язык Squeak сам превращает графические объекты в код на языке Smalltalk. Вы можете узнать больше об Etoys на сайте **www.squeakland.org**.

Python

Существует много ресурсов, на которых можно узнать еще больше о Python. Документация Python во Всемирной паутине достаточно полная, но она может быть тяжелой для восприятия. Найти ее можно здесь: **docs.python.org**.

Многие книги посвящены более сложному программированию на Python. На самом деле их так много, что мы не можем порекомендовать какие-то конкретные. Выбор книг зависит от твоих предпочтений, стиля обучения и конкретных вещей, которые ты хочешь сделать на Python. Но мы уверены, если ты захочешь продолжить работу на Python, то сможешь подобрать себе подходящие книги.

Разработка игр и Pygame

Если ты хочешь создавать игры, по этой теме тоже есть много книг – слишком много, чтобы их перечислять. Ты можешь узнать о такой вещи, как OpenGL, что является сокращением от Open Graphics Language. Эту графическую систему используют многие игры. OpenGL доступна на Python с помощью модуля `PyOpenGL`.

Если ты заинтересовался Pygame, есть несколько дополнительных источников информации. Сайт Pygame, **www.pygame.org**, содержит много примеров и руководств.

Если ты хочешь, чтобы в твоих играх было больше физической точности, воспользуйся физическим движком. Один из них – PyMunk, основанный на Chipmunk Physics. Он позволяет создавать круги, линии и фигуры для 2D-игр. Также он имитирует основные физические силы, такие как гравитация и трение, действующие на эти фигуры. PyMunk доступен на сайте **www.pymunk.org**.

Создание игр на других языках (не на Python)

Если ты хочешь создавать игры, обрати внимание на игровой движок Unity. Unity включает в себя несколько полезных элементов, в том числе игровой 3D-движок, физический движок и способы написания скриптов. Скрипты для Unity написаны на языке C#, который произносится как «си-шарп» (от англ. *see-sharp*).

Возможно, ты уже играл в некоторые игры, которые можно расширить с помощью дополнительного кода. Например, платформа Roblox (**www.roblox.com**) позволяет писать игры на языке Lua. Ты можешь изучить способы моддинга игры Minecraft (**www.minecraft.com**), разрабатывая код на языках Lua, Forth или Java. (Кстати, популярная игра Angry Birds написана на языке Lua.)

Да будет BASIC

При поиске книг ты, возможно, заметишь, что довольно много книг по программированию для детей были написаны в 1980-х годах; во многих из них используется язык под названием BASIC, который был очень популярен в то время. (Существует также версия BASIC для современных компьютеров.) В этих книгах, как правило, много игр. Попробуй взять игру, написанную на BASIC, и переписать ее на Python. Это может быть интересно! Для работы с графикой воспользуйся Pygame или PyQt, если это необходимо. Мы гарантируем, что ты многому научишься, делая это!

Создание веб-сайтов

Чтобы научиться создавать веб-сайты, тебе нужно изучить *три* языка: HTML (который отвечает за содержимое и структуру страницы), CSS (который отвечает за внешний вид страницы) и JavaScript (который делает веб-страницу интерактивной). Можно начать с сайта Mozilla Developer Network: **developer.mozilla.org**.

Конечно, существует также много других инструментов, которые помогут тебе создавать веб-сайты без написания какого-либо кода вообще. Но если ты читаешь эту книгу, тебе, вероятно, захочется научиться создавать веб-сайт полностью с нуля.

Создание мобильных приложений

Если ты хочешь создавать приложения для смартфонов под управлением операционной системы iOS или Android, есть несколько способов сделать это. Один из них – написать приложения, которые будут работать только в iOS или только в Android. Ты можешь писать приложения для iOS на языке программирования Swift с помощью библиотеки под названием UIKit. Приложения для Android можно создавать на языке Kotlin.

Существуют также *кросс-платформенные* инструменты разработки приложений, которые облегчают написание программ, работающих как в iOS, так и в Android. Таких инструментов очень много. Более того, продолжают создаваться все новые и новые инструменты, поэтому мы не будем рекомендовать никакие из них, ведь, возможно, к моменту выхода книги они уже устареют.

Оглянись вокруг

Есть еще много других тем и ресурсов, которые могут тебе помочь в разных областях программирования и Python. Ты всегда можешь заглянуть в соседнюю библиотеку или книжный магазин. Можно поискать во Всемирной паутине интересующие тебя вопросы и необходимые руководства по Python.

Python может многое, но для некоторых особых задач тебе может понадобиться другой язык, например C, C++, Java. Если такое случится, тебе понадобится книга или иной источник, обучающий особенностям языка. Их есть великое множество, поэтому мы не можем советовать ничего конкретного.

Что бы ты ни делал, развлекайся, программируя! Учись, узнавай новое и экспериментируй. Чем больше ты узнаешь о программировании, тем интереснее оно становится!

ПРИЛОЖЕНИЕ А

Правила присвоения имен переменным

Ниже представлены правила присвоения имен переменным (также называемые признаками).

- Имя должно начинаться с буквы или знака подчеркивания. Следовательно, ты можешь использовать неограниченную последовательность букв, чисел и знаков подчеркивания.
- Буквы могут быть как строчными, так и прописными, и регистр имеет значение. Другими словами, `Ax` – не то же самое, что `aX`.
- Числа могут содержать любые цифры от 0 до 9.
- Имя не может быть *ключевым словом* Python.

Помимо букв, чисел и знака подчеркивания другие символы не используются. Пробелы, знаки пунктуации и другие символы нельзя использовать в именах переменных:

```
~ ` ! @ # $ % ^ & * ( ) ; - : « ‘ < > , . ? / { } [ ] + = /
```

Единственный особый символ – нижнее подчеркивание. В случае если ты его не знаешь, ниже представлено несколько примеров:

- `first_number = 15`
- `student_name = «Ваня»`

Символ между словами **first** и **number** – это нижнее подчеркивание. Он же присутствует между словами **student** и **name**. Программисты иногда используют подчеркивание для разделения двух слов в имени переменной. Поскольку пробелы запрещены, они используют подчеркивание.

Не рекомендуем использовать подчеркивание в начале или конце имени переменной, если только ты наверняка не знаешь, зачем его используешь. В некоторых случаях использование подчеркивания в начале или конце имеет особое значение. Поэтому избегай следующего:

- **_first_number = 15**
- **student_name_ = «Ваня»**

Мы уже упоминали, что ключевое слово не может быть использовано в качестве имени переменной. Ключевые слова, такие как **if** или **while**, имеют особое значение в Python, поэтому ты не можешь использовать их в качестве имен переменных – они «зарезервированы» для самого Python. Вот список ключевых слов Python:

```
and as assert async await break class continue def del elif else except
False finally for from global if import in is lambda nonlocal None not
or pass raise return True try while with yield
```

Ниже представлены примеры правильных имен переменных:

- **my_answer**
- **answer23**
- **answer_23**
- **YourAnswer**
- **Your2ndAnswer**

Ниже представлены примеры неправильно составленных имен переменных:

- **23answer** (имя переменной не может начинаться с цифры);
- **your-answer** (дефис нельзя использовать);
- **my answer** (нельзя использовать пробел);
- **while** (**while** – ключевое слово).

ПРИЛОЖЕНИЕ Б

Разница между Python 3 и Python 2

На протяжении всей книги мы упоминали о некоторых различиях между Python 3 и Python 2. В книге используется Python 3, но мы также хотим, чтобы ты знал, как распознать код Python 2, и при необходимости мог сделать свой код совместимым с Python 2. В этом приложении говорится о различиях только в тех частях Python 3 и Python 2, о которых мы говорили в этой книге.

Итак, вот несколько отличий Python 2 от Python 3.

`print`

В Python 2 **`print`** – это ключевое слово, а не функция. Это значит, что вместо того, чтобы написать

```
print(«Привет, мир!»)
```

ты написал бы

```
print «Привет, мир!»
```

Есть и другие отличия, связанные с этим. Вместо того чтобы использовать аргумент **`end`** для вывода следующей команды **`print`** в той же строке (как в Python 3), в Python 2 ты бы поставил запятую в конце. Так что вместо того, чтобы писать

```
print(«Привет,», end=» «)
print(«мир!»)
```

ты написал бы

```
print «Привет,»,
print «мир!»
```

input()

Функция, которая в Python 3 называется **input()**, в Python 2 называлась **raw_input()**. Функция **input()** в Python 2 оценивает входные данные (возможно ли преобразовать их в число).

Это означает, что вместо того, чтобы писать (как в Python 3)

```
your_name = input(«Введи свое имя: «)
```

ты написал бы следующее (в Python 2):

```
your_name = raw_input(“Введи свое имя: “)
```

Целочисленное деление

Третье важное отличие заключается в целочисленном делении. Python 2 по умолчанию использует именно его, в то время как Python 3 по умолчанию использует деление с плавающей точкой. Таким образом, в Python 3 ты получаешь

```
>>> print(5/2)
2.5
```

а в Python 2:

```
>>> print 5/2
2
```

Оператор деления по модулю (**%**) для получения остатка в целочисленном делении работает одинаково в Python 2 и в Python 3.

В Python 3:

```
>>> print(5%2)
1
```

В Python 2:

```
>>> print 5%2
1
```

range()

В Python 3 функция **range()** вызывает объект **range**, который работает как список:

```
>>> range(0,3)
range(0, 3)
>>> type(range(0,3))
<class 'range'>
```

В Python 2 функция **range()** просто вызывает обычный список:

```
>>> range(0,3)
[0, 1, 2]
>>> type(range(0,3))
<type 'list'>
```

В Python 2 также есть функция **xrange()**, которая работает почти так же, как функция **range()** в Python 3.

Байты и кодировка символов

В главе 26 мы обсуждали различные способы, которые компьютеры используют для преобразования текста в байты. Строка Python 3 (тип **str**) может хранить символы из любого языка, но ты должен использовать команду **.encode('utf-8')**, чтобы преобразовать их в объект **bytes**, который можно отправить по сети.

В Python 2 строки работают немного по-другому. Вместо того чтобы хранить текст, как строки Python 3, они хранят последовательность байтов вроде объекта **bytes** в Python 3. Это означает, что ты можешь поместить символы из ASCII в Python 2 только в кавычках, точно так же, как можешь поместить символы из ASCII в Python 3 только в кавычках в команде **b»»**.

Если ты хочешь использовать символы, отличные от ASCII, в Python 2, тебе нужно использовать объект **unicode**. Ты можешь создать объект **unicode**, поставив **u** перед строкой или вызвав с помощью команды **str.decode('utf-8')**. (Ты можешь превратить объект **unicode** обратно в строку с помощью команды **unicode.encode('utf-8')**.)

Преобразование Python 2 в Python 3

Существует инструмент под названием `2to3`, который может преобразовать код Python 2 в код Python 3, а также инструмент `3to2`, который совершает обратное действие. Кроме того, существует онлайн-инструмент по адресу **www.pythonconverter.com**. Мы не проверяли эти инструменты ни на одном из кодов в этой книге, поэтому не можем гарантировать, что они будут работать.

ПРИЛОЖЕНИЕ В

Ответы на вопросы

Ниже приведены ответы на вопросы разделов «Проверь свои знания» и «Попробуй самостоятельно». Конечно, иногда существует несколько вариантов правильного ответа, особенно это касается разделов «Попробуй самостоятельно», но ты можешь использовать ответы, чтобы понять, в верном ли направлении ты двигаешься.

Глава 1 «Приступая к работе»

Проверь свои знания

1. В операционной системе Windows запусти интерпретатор IDLE из меню **Пуск** (Start), выбрав в папке **Python 3.7** пункт **IDLE (Python 3.7 64-bit)** или **IDLE (Python 3.7 32-bit)**. В macOS щелкни мышью по ярлыку **IDLE** в доке или дважды щелкни мышью по приложению **IDLE.app** в папке Python 3.7, находящейся в папке **Приложения** (Applications). В Linux, в зависимости от используемого файлового менеджера, есть меню **Приложения** (Applications) или **Программы** (Programs). Обрати внимание, что в Linux многие не используют IDLE – они просто запускают Python и используют редактор, такой как vi или emacs, для редактирования своего кода.
2. Команда `print` выводит текст в окне вывода (окно оболочки IDLE в наших первых примерах).
3. Символ умножения в Python – звездочка (`*`).
4. Когда ты запускаешь программу, IDLE показывает слово RESTART, а затем местоположение и имя выполняемого скрипта Python, например:

```
>>>
RESTART: C:/HelloWorld/Примеры/Листинг_1-2.py
```

5 «Выполнение» программы – другое название для «запуска» программы.

Попробуй самостоятельно

1 **`>>> print(7 * 24 * 60)`** (7 дней в неделе, 24 часа в сутках, 60 минут в часе). Ты должен получить значение 10 080 в качестве ответа.

2 Твоя программа должна выглядеть примерно так:

```
print(«Меня зовут Петя Иванов.»)
print(«Я родился 1 марта 1998 года.»)
print(«Мой любимый цвет — синий.»)
```

Глава 2 «Память и переменные»

Проверь свои знания

1 Ты указываешь Python, что переменная является строкой, заключая ее в кавычки.

2 Вопрос был таким: «Можно ли изменить значение, присвоенное переменной?» Это зависит от того, что ты понимаешь под словом «изменить». Если можно сделать так:

```
myAge = 10
```

то можно сделать и так:

```
myAge = 11
```

Ты изменил то, что присвоено переменной **`myAge`**. Ты переместил метку **`myAge`** и присвоил ее другому значению: с 10 на 11. Но ты не изменил само значение 10 на 11. Поэтому более правильным будет говорить, что можно «присвоить заново имя другому значению» или «присвоить новое значение переменной», а не «изменить значение переменной».

3 Нет, **`TEACHER`** – это не то же самое, что **`TEACHEr`**. Поскольку имена переменных чувствительны к регистру, последняя буква этих переменных делает их разными.

4 Да, **`'Бах'`** и **`«Бах»`** – это одно и то же. Оба слова являются строками, и интерпретатору Python безразлично, какие именно кавычки ты используешь, пока открывающие и закрывающие кавычки одинаковы.

5 Нет, `«4»` – это не то же самое, что `4`. Первый элемент является строкой (хотя в ней всего один символ), потому что вокруг него стоят кавычки. А второй – числом.

6 Ответ б. `2Teacher` – неверное имя для переменной. Имена переменных в Python не могут начинаться с цифры.

7 `«10»` – это строка, потому что вокруг нее есть кавычки.

Попробуй самостоятельно

1 В интерактивном режиме ты должен сделать нечто похожее:

```
>>> temperature = 25
>>> print(temperature)
25
```

2 Можно сделать так:

```
>>> temperature = 40
>>> print(temperature)
40
```

или так:

```
>>> temperature = temperature + 15
>>> print(temperature)
40
```

3 Ты должен сделать примерно так:

```
>>> firstName = “Федя”
>>> print(firstName)
Федя
```

4 При использовании переменных твоя программа подсчета минут в дне будет выглядеть так:

```
>>> DaysPerWeek = 7
>>> HoursPerDay = 24
>>> MinutesPerHour = 60
>>> print(DaysPerWeek * HoursPerDay * MinutesPerHour)
10080
```

5 Чтобы узнать, что случится, если в дне будет 26 часов, нужно сделать следующее:

```
>>> HoursPerDay = 26
>>> print(DaysPerWeek * HoursPerDay * MinutesPerHour)
10920
```

Глава 3 «Основы математики»

Проверь свои знания

1. В языке Python используется звездочка (`*`) для обозначения умножения.
2. Интерпретатор Python выведет ответ **`9 / 5 = 1.8`**.
3. Чтобы получить ответ без остатка, используй двойной слеш: **`9//5`**.
4. Чтобы получить остаток, используй оператор модуля: **`8 % 3`**.
5. Каков другой способ посчитать ответ на пример **`6 * 6 * 6 * 6`** в Python? **`6 ** 4`**.
6. 17 000 000 в E-нотации будет записано как **`1.7e7`**.
7. Число **`4.56e-5`** – то же самое, что **`0.0000456`**.

Попробуй самостоятельно

1. Ниже приведено несколько способов решения проблем. Ты можешь выбрать свой вариант решения.
   - Посчитать, сколько каждый человек должен заплатить в ресторане:

     ```
     >>> print(3527 * 1.15 / 3)
     >>> 1352.0166666666667
     ```

     Если округлить результат, получится, что каждый человек должен заплатить по 1352 рубля.
   - Посчитать площадь и периметр прямоугольника:

     ```
     length = 16.7
     width = 12.5
     Perimeter = 2 * length + 2 * width
     Area = length * width
     print('Длина = ', length, ' Ширина = ', width)
     print('Площадь = ', Area)
     print('Периметр = ', Perimeter)
     ```

     Пример результата выполнения программы:

     ```
     Длина = 16.7 Ширина = 12.5
     Площадь = 208.75
     Периметр = 58.4
     ```

2 Ниже представлена программа для конвертации градусов по Фаренгейту в градусы Цельсия:

```
fahrenheit = 75
celsius = 5/9 * (fahrenheit - 32)
print(«Градусов Фаренгейта = «, fahrenheit, «Градусов Цельсия =»,
      celsius)
```

3 Посчитать время, необходимое, чтобы проехать данное расстояние при данной скорости:

```
distance = 200
speed = 80
time = distance / speed
print(“время =”, time)
```

Глава 4 «Типы данных»

Проверь свои знания

1 Функция **`init()`** всегда округляет с уменьшением (до целого числа в сторону уменьшения).

2 Поскольку интерактивная оболочка отображает **`‘4’`** с кавычками, мы понимаем, что **`thing1`** – это строка.

3 Можно «обмануть» **`int()`** и округлить значение правильно, а не с уменьшением, добавив 0.5 к числу, которое ты передаешь **`int()`**. Ниже приведен пример в интерактивном режиме:

```
>>> a = 13.2
>>> roundoff = int(a + 0.5)
>>> roundoff
13
>>> b = 13.7
>>> roundoff = int(b + 0.5)
>>> b
14
```

Если оригинальное число меньше 13.5, **`int()`** получает число меньше 14, которое она округляет до 13.
Если оригинальное число равно 13.5 или больше, **`int()`** получает число, равное или больше 14, которое округляет до 14.

Попробуй самостоятельно

1 Можно использовать **`float()`**, чтобы конвертировать строку в десятичную

дробь:

```
>>> a = float('12.34')
>>> print a
12.34
```

Но откуда мы знаем, что это число, а не строка? Давай проверим тип:

```
>>> type(a)
<class 'float'>
```

2 Можно использовать **int()** для конвертации десятичного числа в целое:

```
>>> print(int(56.78))
56
```

Ответ был округлен с уменьшением.

3 Можно использовать **int()** для конвертации строки в целое число:

```
>>> a = int('75')
>>> print(a)
75
>>> type(a)
<class 'int'>
```

Глава 5 «Ввод»

Проверь свои знания

1 При коде

```
answer = input()
```

если пользователь введет **12**, **answer** содержит строку. Потому что **input()** всегда дает строку.
Попробуй исполнить небольшую программу – и увидишь:

```
print("Введи число: ", end="")
answer = input()
print(type(answer))
>>> ============== RESTART ==============
>>>
enter a number: 12
<type 'str'>
>>>
```

Итак, функция **`input()`** выдает в ответ строку.

2 Чтобы с помощью функции **`input()`** вывести сообщение для ввода, помести текст в кавычки внутри скобок:

```
answer = input("Введи число: ")
```

3 Чтобы получить целое число с помощью **`input()`**, используй **`int()`** для конвертации строки, которую ты получишь. Можно сделать это в два шага:

```
something = input()
answer = int(something)
```

или в один шаг:

```
answer = int(input())
```

4 Этот вопрос очень похож на предыдущий, кроме того что нужно использовать **`float()`** вместо **`int()`**.

Попробуй самостоятельно

1 Твои инструкции в интерпретаторе должны выглядеть примерно так:

```
>>> first = 'Ваня'
>>> last = 'Сидоров'
>>> print(first + last)
ВаняСидоров
```

Ой! Нет пробела. Можно добавить пробел в конце имени:

```
>>> first = 'Ваня '
```

Или попробовать следующее:

```
>>> print(first + ' ' + last)
Ваня Сидоров
```

Или просто использовать запятую:

```
>>> first = 'Ваня'
>>> last = 'Сидоров'
>>> print(first, last)
Ваня Сидоров
```

2 Программа должна выглядеть примерно так:

```
first = input('Введи свое имя: ')
last = input('Введи свою фамилию: ')
print('Привет,', first, last, ' как дела?')
```

3 Программа должна выглядеть примерно так:

```
length = float(input ('длина комнаты в метрах: '))
width = float(input ('ширина комнаты в метрах: '))
area = length * width
print('Площадь комнаты равна ', area, ' квадратных метров.')
```

4 Можно добавить несколько строк в предыдущую программу:

```
length = float(input ('длина комнаты в метрах: '))
width = float(input ('ширина комнаты в метрах: '))
cost_per_meter = float(input ('цена ковролина за м2: '))
area = length * width
total_cost = area * cost_per_meter
print('Площадь комнаты равна ', area, ' квадратных метров.')
print('Стоимость ковролина: ', total_cost)
```

5 Программа должна выглядеть примерно так:

```
ten  = int(input(«Сколько у тебя 10-рублевых монет? «))
five = int(input(«Сколько у тебя 5-рублевых монет? «))
two = int(input(«Сколько у тебя 2-рублевых монет? «))
one = int(input(«Сколько у тебя 1-рублевых монет? "))
total = 10 * ten + 5 * five + 2 * two + 1 * one
print("Всего у тебя: ", total)
```

Глава 6 «Графические пользовательские интерфейсы (GUI)»

Проверь свои знания

1 Чтобы вывести окно с сообщением с помощью EasyGui, используй **`msgbox()`**:

```
easygui.msgbox("Привет!")
```

2 Чтобы получить строку в качестве ввода с помощью EasyGui, используй **`enterbox`**.

3 Чтобы получить целое число в качестве ввода, можно использовать **`enterbox`** (который получает от пользователя строку), затем конвертировать ее с помощью функции **`int()`**. Или использовать **`integerbox`**.

4 Чтобы получить десятичную дробь от пользователя, можно использовать **enterbox** (который получает строку), затем использовать функцию **float()** для конвертации строки в десятичную дробь.

5 Значение по умолчанию похоже на «автоматический ответ». Вот способ использования: если ты пишешь программу, где все ученики в твоем классе должны указать свое имя и адрес, то можешь сделать название своего города значением по умолчанию в адресе. Таким образом, ученикам не надо будет вводить его, если только они не живут в другом городе.

Попробуй самостоятельно

1 Ниже представлена версия программы по конвертации температур с использованием EasyGui:

```
# tempgui1.py
# версия программы по конвертации температур с использованием EasyGui
# конвертирует градусы Фаренгейта в градусы Цельсия
import easygui
easygui.msgbox('Эта программа конвертирует температуру по Фаренгейту
               в градусы Цельсия')
temperature = easygui.enterbox('Введи температуру по Фаренгейту:')
Fahr = float(temperature)
Cel = (Fahr - 32) * 5 / 9
easygui.msgbox('Получится ' + str(Cel) + ' градусов Цельсия.')
```

2 Ниже приведен код программы, которая запрашивает твое имя и адрес, а затем выводит информацию. Она содержит нечто новое: начало с новой строки посредством **\n**. См. также главу 21.

```
# address.py
# Ввод частей адреса и вывод всей информации
import easygui
name = easygui.enterbox("Как тебя зовут?")
addr = easygui.enterbox("Название твоей улицы, номер дома и квартиры?")
city = easygui.enterbox("В каком городе ты живешь?")
state = easygui.enterbox("В какой области и стране ты живешь?")
code = easygui.enterbox("Какой индекс у твоего адреса?")
whole_addr = name + "\n" + addr + "\n" + city + ", " + state + "\n" + code
easygui.msgbox(whole_addr, "Вот твой адрес:")
```

Глава 7 «Вычисления»

Проверь свои знания

1 Вывод будет таким:

```
Меньше 20
```

Потому что значение переменной **my_number** меньше 20, проверка в утверждении **if** является истинной, и блок за **if** (в этому случае – одна строка) выполняется.

2 Вывод будет таким:

```
20 или больше
```

Поскольку значение переменной **my_number** больше 20, проверка в утверждении **if** ложна, поэтому код в блоке, следующем за **if**, не выполняется. Код из блока **else** выполняется вместо него.

3 Чтобы проверить, больше ли число 30, но меньше или равно 40, используем что-то подобное:

```
if number > 30 and number <= 40:
    print('Число находится в диапазоне между 30 и 40')
```

или можно сделать так:

```
if 30 < number <= 40:
print(«Число находится в диапазоне между 30 и 40»)
```

4 Чтобы проверить регистр буквы, делаем так:

```
if answer == 'Ы' or answer == 'ы':
    print(«ты ввел букву 'Ы' «)
```

Обрати внимание, что мы использовали двойные кавычки для строки, а для буквы – одинарные. Если ты сомневаешься, какими же кавычками пользоваться, правило простое: выделяй строку другими кавычками.

Попробуй самостоятельно

1 Ниже приведен один вариант ответа:

```
# программа для расчета скидки
# 10% скидки на покупки суммой 1000 рублей и меньше, 20% скидки на покупки
item_price = float(input('введи цену товара: '))
if item_price <= 1000.0:
    discount = item_price * 0.10
else:
    discount = item_price * 0.20
final_price = item_price - discount
print('Ты получаешь скидку ', discount, ' и окончательная цена составит ',
      final_price)
```

Мы не округляли ответ до двух знаков после запятой (копеек) и не выводили название денежных единиц.

2 Ниже приведен один вариант решения:

```
# программа для проверки пола и возраста игроков в футбол
# принимаются девочки от 10 до 12 лет
gender = input(«Ты - мальчик или девочка? (‘м’ или ‘д’) “)
if gender  == ‘д’:
    age = int(input(‘Сколько тебе лет? ‘))
    if age >= 10 and age <= 12:
        print(‘Ты можешь играть в команде.’)
    else:
        print(‘Ты не подходишь по возрасту.’)
else:
    print(‘В эту команду принимаются только девочки.’)
```

3 Ниже приведен один из вариантов ответа:

```
# программа для проверки уровня топлива.
# Следующая АЗС через 200 км
tank_size = int(input(‘Объем бензобака машины в литрах? ‘))
full = int(input(‘Насколько заполнен бензобак (в процентах)? ‘))
mileage = int(input(‘Сколько км проезжает машина на 1 л бензина? ‘))
range = tank_size * (full / 100) * mileage
print(‘Машина проедет еще ‘, range, ‘км.’)
print(‘Следующая АЗС через 200 км.’)
if range <= 200:
    print(‘Заправься сейчас!’)
else:
    print(‘Ты можешь подождать до следующей АЗС.’)
```

Чтобы добавить 5 л погрешности, измени строку

```
range = tank_size * (full / 100) * mileage
```

на

```
range = (tank_size - 5) * (full / 100) * mileage
```

4 Ниже приведена простая программа с паролем:

```
password  = “bigsecret”
guess  = input(“Введи свой пароль: “)
if guess == password:
    print(“Пароль угадан. Ты победил!»)
    # здесь введи остальной код программы
else:
    print(«Пароль неверный. Чао!»)
```

Глава 8 «Циклы»

Проверь свои знания

1. Цикл будет выполнен 5 раз.
2. Цикл будет выполнен 3 раза, значения будут следующими: `i = 1`, `i = 3`, `i = 5`.
3. Если ты хочешь увидеть, какие числа функция `range()` отобразит при запуске, проверь это в интерактивной оболочке:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

   Функция `range(1, 8)` выдаст `[1, 2, 3, 4, 5, 6, 7]`.
4. Функция `range(8)` выдаст `[0, 1, 2, 3, 4, 5, 6, 7]`.
5. Функция `range(2, 9, 2)` выдаст `[2, 4, 6, 8]`.
6. Функция `range(10, 0, -2)` выдаст `[10, 8, 6, 4, 2]`.
7. Для остановки текущей итерации цикла и перехода к следующей используется утверждение `continue`.
8. Цикл `while` заканчивается тогда, когда условие становится ложным.

Попробуй самостоятельно

1. Ниже представлен код программы для вывода таблицы умножения по выбору пользователя с использованием цикла `for`:

```
# программа для вывода таблицы умножения для чисел до 10
number = int(input('Какая таблица тебе нужна? '))
print('Вот твоя таблица:')
for i in range(1, 11):
    print(number, 'x', i, '=', number * i)
```

2. Та же таблица умножения с использованием цикла `while`:

```
# программа для вывода таблицы умножения (с циклом while)
number = int(input('Какая таблица тебе нужна? '))
print('Вот твоя таблица:')
i = 1
while i <= 10:
    print(number, 'x', i, '=', number * i)
    i = i + 1
```

3. Ниже представлен код таблицы умножения с диапазоном, определяемым пользователем:

```
# программа для вывода таблицы умножения
```

```
# пользователь определяет, какие числа ему нужны в таблице
number = int(input('Какая таблица тебе нужна? '))
limit = int(input('До какого множителя считать? '))
print('Вот твоя таблица:')
for i in range(1, limit + 1):
    print(number, 'x', i, '=', number * i)
```

Обрати внимание, что в строке **for** второе значение функции **range()** включает в себя переменную, а не просто число. Подробнее об этом рассказано в главе 11.

Глава 9 «Комментарии»

Проверь свои знания

Ниже представлены примеры комментариев, которые я добавил бы в программу по конвертации температур:

```
# tempconv1.py
# программа конвертирует градусы Фаренгейта в градусы Цельсия
Fahr = 75
Cel = (Fahr - 32) * 5 / 9 #вычисление градусов Цельсия по формуле
print(«градусов Фаренгейта = «, Fahr, «градусов Цельсия = «, Cel)
```

Глава 10 «Время поиграть»

Попробуй самостоятельно

Ты попробовал набрать программу и запустить ее? Не забудь поместить графические файлы в папку с программой.

Глава 11 «Вложенные циклы и циклы с переменной»

Проверь свои знания

1 Цикл с переменной в Python можно сделать, поместив переменную в функцию **range()**:

```
for i in range(numberOfLoops)
```

или

```
for i in range(1, someNumber)
```

2 Чтобы сделать вложенный цикл, помести его в тело другого цикла:

```
for i in range(5):
    for j in range(8):
        print(«привет», end=» «)
    print()
```

Этот код выводит слово «привет» 8 раз в строку (внутренний цикл) и создает пять таких строк (внешний цикл).

3 Будет выведено 15 звездочек.

4 Вывод этого кода будет выглядеть так:

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

5 Для четырехуровневого дерева решений есть 2**4, или 2 * 2 * 2 * 2, возможных вариантов выбора. Это 16 вариантов, или 16 ветвей дерева.

Попробуй самостоятельно

6 Ниже представлена программа таймера, которая спрашивает пользователя, когда ей начинать отсчет:

```
# Таймер отсчета спрашивает пользователя, когда ему начинать
import time
start = int(input("Таймер: Сколько секунд? "))
for i in range(start, 0, -1):
    print(i)
    time.sleep(1)
print("СТАРТ!")
```

7 Эта версия выводит полосу звездочек рядом с каждым числом:

```
# Таймер отсчета спрашивает пользователя, когда ему начинать,
# и выводит звездочки рядом с каждым числом
import time
start = int(input("Таймер: Сколько секунд? "))
for i in range (start, 0, -1):
    print(i, end=' ')
    for star in range(i):
```

```
        print('*', end='')
    print()
    time.sleep(1)
print("СТАРТ!")
```

Ниже представлена альтернативная версия без вложенного цикла:

```
import time
start = int(input("Таймер: Сколько секунд? "))
for i in range (start, 0, -1):
    print(i, '*' * i)
    time.sleep(1)
print(«СТАРТ!»)
```

Глава 12 «Списки»

Проверь свои знания

1. Можно добавить элемент в список с помощью функции **`append()`**, **`insert()`** или **`extend()`**.
2. Можно удалить элемент из списка с помощью функции **`remove()`**, **`pop()`** или **`del()`**.
3. Чтобы сортировать копию списка, можно выполнить одно из следующих действий:
   - создать копию списка с помощью нарезки: **`new_list = my_list[:]`**, и сортировать новый список: **`new_list.sort()`**;
   - использовать функцию **`sorted()`**: **`new_list = sorted(my_list)`**.
4. Узнать, находится ли определенное значение в списке, можно с помощью ключевого слова **`in`**.
5. Узнать расположение значения в списке можно с помощью метода **`index()`**.
6. Кортеж – это коллекция, похожая на список, но ее нельзя изменить. Кортежи неизменны, а списки изменяемы.
7. Можно создать список списков несколькими способами:
   - с помощью вложенных квадратных скобок:

```
>>> my_list = [[1, 2, 3], ['а', 'б', 'в'], ['красный', 'зеленый',
'синий']]
```

   - с помощью функции **`append()`**:

```
>>> my_list = []
>>> my_list.append([1, 2, 3])
>>> my_list.append(['а', 'б', 'в'])
```

```
>>> my_list.append(['красный', 'зеленый', 'синий'])
>>> print(my_list)
[[1, 2, 3], ['а', 'б', 'в'], ['красный', 'зеленый', 'голубой']]
```

- создав отдельные списки и соединив их:

```
>>> list1 = [1, 2, 3]
>>> list2 = ['а', 'б', 'в']
>>> list3 = ['красный', 'зеленый', 'синий']
>>> my_list = [list1, list2, list3]
>>> print(my_list)
[[1, 2, 3], ['а', 'б', 'в'], ['красный', 'зеленый', 'голубой']]
```

8 Отдельное значение из списка можно получить с помощью двух индексов:

```
my_list = [[1, 2, 3], ['а', 'б', 'в'], ['красный', 'зеленый',
'синий']]
my_color = my_list[2][1]
```

Ответом будет **`'зеленый'`**.

9 Словарь – это набор пар ключ-значение.

10 Ты можешь добавить элемент в словарь, указав ключ и значение:

```
phone_numbers['Ваня'] = '555-1234'
```

11 Чтобы найти элемент словаря по его ключу, можно использовать индекс:

```
print(phone_numbers['Ваня'])
```

Попробуй самостоятельно

1 Ниже приведен код программы, которая получает 5 имен, помещает их в список и выводит:

```
nameList = []
print(«Введи пять имен (нажимай клавишу Enter после каждого имени):»)
for i in range(5):
    name = input()
    nameList.append(name)
print("Имена:", nameList)
```

2 Ниже приведен код программы, которая выводит оригинальный список и его сортированную версию:

```
nameList = []
print(«Введи пять имен (нажимай клавишу Enter после каждого имени):»)
```

```
for i in range(5):
    name = input()
    nameList.append(name)
print("Имена:", nameList)
print("Отсортированные имена:", sorted(nameList))
```

3 Ниже приведен код программы для вывода только третьего имени в списке:

```
nameList = []
print(«Введи пять имен (нажимай клавишу Enter после каждого имени):»)
for i in range(5):
    name = input()
    nameList.append(name)
print("Третье введенное имя:", nameList[2])
```

4 Ниже приведен код программы, которая позволяет пользователю заменять имя в списке:

```
nameList = []
print(«Введи пять имен (нажимай клавишу Enter после каждого имени):»)
for i in range(5):
    name = input()
    nameList.append(name)
print("Имена:", nameList)
print(«Замени одно имя. Какое? (1-5):", end=' ')
replace = int(input())
new = input("Новое имя: ")
nameList[replace - 1] = new
print("Имена:", nameList)
```

5 Программа ниже позволяет создать словарь со словами и определениями:

```
user_dictionary = {}
while 1:
    command = input(«Добавить (д) слово, найти (н) слово или выйти
                    (в):»)
    if command == "д":
        word = input("Введи слово: ")
        definition = input("Введи определение: ")
        user_dictionary[word] = definition
        print(«Слово добавлено!»)
    elif command == «н»:
        word = input("Введи слово: ")
        if word in user_dictionary.keys():
            print(user_dictionary[word])
        else:
            print(«Данного слова нет в словаре.»)
     elif command == 'в':
        break
```

Глава 13 «Функции»

Проверь свои знания

1. Для создания функции используется ключевое слово **def**.
2. К функции можно обратиться с помощью ее имени в круглых скобках.
3. Аргументы функции можно передать, поместив их в круглые скобки при обращении к функции.
4. Нет ограничения количества аргументов функции.
5. Функция отправляет информацию обратно обратившемуся с помощью ключевого слова **return**.
6. После выполнения функции все локальные переменные уничтожаются.

Попробуй самостоятельно

1. Функция – это просто набор команд **print**:

```
def printMyNameBig():
    print(« PPPPP    OOOO    M       M        A       H    H «)
    print(« P    P  O    O  M  M   M  M      A A      H    H «)
    print(« P    P  O    O  M   M M   M     A   A     H    H «)
    print(« PPPPP   O    O  M    M    M    AAAAAAA    HHHHHH «)
    print(« P       O    O  M         M   A       A   H    H «)
    print(« P        OOOO   M         M  A         A  H    H «)
```

Программа, которая к ней обращается, выглядит так:

```
for i in range(5):
    printMyNameBig()
```

2. Ниже представлен мой пример для вывода адреса с семью аргументами:

```
# определение функции с семью аргументами
def printAddr(name, num, street, city, prov, pcode, country):
    print(name)
    print(num, end=' ')
    print(street)
    print(city, end=" ")
    if prov !="":
        print(", "+prov)
    else:
        print ("")
    print(pcode)
    print(country)
    print()
# вызов функции и передача ей семи аргументов
printAddr(«Макс», «45», «улица Гагарина», «Улан-Удэ», «Бурятия»,
```

```
          «670034», «Россия»)
printAddr(«Хо», «64», «улица Ки», «Пекин», «», «235643», «Китай»)
```

3 Нет ответа, просто попробуй.

4 Функция для прибавления сдачи выглядит так:

```
def addUpChange(ten, five, two, one):
    total = 10 * ten + 5 * five + 2 * two + 1 * one
    return total
```

Программа, которая к ней обращается, выглядит так:

```
ten  = int(input(«Сколько у тебя 10-рублевых монет? «))
five = int(input(«Сколько у тебя 5-рублевых монет? «))
two = int(input(«Сколько у тебя 2-рублевых монет? «))
one = int(input(«Сколько у тебя 1-рублевых монет? “))
total = addUpChange(ten, five, two, one)
print(«Всего у тебя: «, total)
```

Глава 14 «Объекты»

Проверь свои знания

1 Чтобы определить новый объект, используй ключевое слово **`class`**.

2 Атрибуты – это «то, что ты знаешь» об объекте. Это переменные, содержащиеся в объекте.

3 Методы – это «действия», которые можно выполнить с объектом. Это функции, содержащиеся в объекте.

4 Класс – это просто определение или план метода. Экземпляр – то, что ты получаешь при создании объекта по плану.

5 Имя **`self`** обычно используется как ссылка на экземпляр в методе объекта.

6 Полиморфизм – это способность иметь два и больше методов с тем же именем при разных объектах. Методы могут вести себя по-разному в зависимости от того, какому объекту они принадлежат.

7 Наследование – это способность объектов принимать атрибуты и методы от своих «предков». «Дочерний» класс (который называется подклассом или производным классом) получает все атрибуты и методы родителя и может иметь свои атрибуты и методы, отдельные от родителя.

Попробуй самостоятельно

1 Класс для банковской учетной записи будет выглядеть так:

```
class BankAccount:
    def __init__(self, acct_number, acct_name):
```

```
        self.acct_number = acct_number
        self.acct_name = acct_name
        self.balance = 0.0
    def displayBalance(self):
        print(«Баланс учетной записи:», self.balance)
    def deposit(self, amount):
        self.balance = self.balance + amount
        print("Начислено:", amount)
        print("Обновленный баланс:", self.balance)
    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance = self.balance - amount
            print("Списано:", amount)
            print("Обновленный баланс:", self.balance)
        else:
            print("Попытка снятия:", amount)
            print(«Баланс учетной записи:», self.balance)
            print(«Операция отклонена. Недостаточно средств.»)
```

Ниже приведен код для проверки программы:

```
myAccount = BankAccount(234567, «Степан Ромашкин»)
print(«Имя учетной записи:», myAccount.acct_name)
print(«Номер учетной записи:», myAccount.acct_number)
myAccount.displayBalance()
myAccount.deposit(5700)
myAccount.withdraw(650)
myAccount.withdraw(130)
```

2 Чтобы создать учетную запись с процентами, сделай подкласс **`BankAccount`** и создай метод для добавления процентов:

```
class InterestAccount(BankAccount):
    def __init__(self, acct_number, acct_name, rate):
        BankAccount.__init__(self, acct_number, acct_name)
        self.rate = rate
    def addInterest(self):
        interest = self.balance * self.rate
        print("начислено процентов,", self.rate * 100, "%")
        self.deposit (interest)
```

Код для проверки:

```
myAccount = InterestAccount(234567, "Степан Ромашкин", 0.11)
print("Имя учетной записи:", myAccount.acct_name)
print("Номер учетной записи:", myAccount.acct_number)
myAccount.displayBalance()
myAccount.deposit(950)
myAccount.addInterest()
```

Глава 15 «Модули»

Проверь свои знания

1. Некоторые преимущества использования модулей:
   - можно написать код один раз и использовать его во многих программах;
   - можно использовать модули других программистов;
   - ваш файл кода становится меньше, в нем легче что-либо найти;
   - можно использовать только те части (модули), которые необходимы в работе.
2. Модуль можно создать, написав код в интерпретаторе Python и сохранив его в файл.
3. Когда ты хочешь использовать модуль, ты прибегаешь к ключевому слову **`import`**.
4. Импортирование модуля – то же, что и импортирование *пространства имен*.
5. Два способа импорта временного модуля с доступом ко всем именам в нем:

```
import time
```

и

```
from time import *
```

Попробуй самостоятельно

1. Чтобы написать модуль, сохрани код твоей функции **`bigname`** в файл – например, в *bigname.py*. Затем импортируй модуль и обратись к функции:

```
import bigname
bigname.printMyNameBig()
```

или так:

```
from bigname import *
printMyNameBig()
```

2. Чтобы поместить **`c_to_f`** в пространство имен основной программы, можно сделать так:

```
from my_module import c_to_f
```

или так:

```
from my_module import *
```

3 Код небольшой программы для вывода пяти случайных целых чисел от 1 до 20 выглядит так:

```
import random
for i in range(5):
    print(random.randint(1, 20))
```

4 Код небольшой программы для вывода случайной десятичной дроби каждые три секунды в течение 30 секунд выглядит так:

```
import random, time
for i in range(10):
    print(random.random())
    time.sleep(3)
```

Глава 16 «Графика»

Проверь свои знания

1 Значение RGB [255, 255, 255] устанавливает белый цвет.
2 Значение RGB [0, 255, 0] устанавливает зеленый цвет.
3 Чтобы нарисовать прямоугольники, используй метод **`pygame.draw.rect()`**.
4 Чтобы нарисовать линии, соединяющие несколько точек вместе, используй метод **`pygame.draw.lines()`**.
5 Термин «*пиксель*» – сокращение от «picture element – элемент изображения», означает одну точку на экране (или бумаге).
6 В окне Pygame координаты [0, 0] – это верхний левый угол.
7 На графике координаты [50, 200] принадлежат букве Б.
8 На графике координаты [300, 50] принадлежат букве Г.
9 Метод **`blit()`** используется для копирования изображений в Pygame.
10 Чтобы переместить или анимировать изображение, действуй в два шага:
   - удали изображение со старого расположения;
   - нарисуй изображение в новой позиции.

Попробуй самостоятельно

1 Ниже приведен код программы, которая рисует несколько разных фигур на экране. Если тебе лень набирать код вручную, ты найдешь его в файле с именем *TIO_CH16_1.py* в папке *ответы* архива с примерами для этой книги.

```
import pygame, sys
pygame.init()
screen=pygame.display.set_mode((640, 480))
```

```
screen.fill((250, 120, 0))
pygame.draw.arc(screen, (255, 255, 0), pygame.rect.Rect(43, 368, 277, 235),
    -6.25, 0, 15)
pygame.draw.rect(screen, (255, 0, 0), pygame.rect.Rect(334, 191, 190, 290))
pygame.draw.rect(screen, (128, 64, 0), pygame.rect.Rect(391, 349, 76, 132))
pygame.draw.line(screen, (0, 255, 0), (268, 259), (438, 84), 25)
pygame.draw.line(screen, (0, 255, 0), (578, 259), (438, 84), 25)
pygame.draw.circle(screen, (0, 0, 0), (452, 409), 11, 2)
pygame.draw.polygon(screen, (0, 0, 255), [(39, 39), (44, 136), (59, 136),
(60, 102), (92, 102), (94, 131), (107, 141), (111, 50), (97, 50), (93, 86),
    (60, 82), (58, 38)], 5)
pygame.draw.rect(screen, (0, 0, 255), pygame.rect.Rect(143, 90, 23, 63), 5)
pygame.draw.circle(screen, (0, 0, 255), (153, 60), 15, 5)
clock = pygame.time.Clock()
pygame.display.flip()
running = True
while running:
    clock.tick(60)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
            running = False
pygame.quit()
```

2 Чтобы заменить изображение мяча другим, измени имя файла в строке

```
my_ball = pygame.image.load('beach_ball.png')
```

другим именем файла или другим изображением.

3 В листинге 16.16 измени код

```
x_speed = 10
y_speed = 10
```

на что-то подобное:

```
x_speed = 20
y_speed = 8
```

4 Чтобы мяч отталкивался от «невидимой» стены, измени в листинге 16.16 строку

```
if x > screen.get_width() - 90 or x < 0:
```

на

```
if x > screen.get_width() - 250 or x < 0:
```

Это заставит мяч менять направление до достижения края окна. Ты можешь сделать то же самое для «пола» с координатами по оси *y*.

5 Ниже представлено, как выглядит листинг 16.16 с перемещением **`display.flip`** в цикл **`while`** и добавлением задержки:

```
import pygame, sys, random
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
for i in range (100):
    width = random.randint(0, 250)
    height = random.randint(0, 100)
    top = random.randint(0, 400)
    left = random.randint(0, 500)
    pygame.draw.rect(screen, [0,0,0], [left, top, width, height], 1)
    pygame.display.flip()
    pygame.time.delay(30)
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Ты будешь видеть каждый прямоугольник отдельно, потому что мы замедлили выполнение программы и теперь обновляем экран после каждого прямоугольника. Если сделать то же самое в программе с синусоидой, можно увидеть каждую точку синусоиды, соединяющуюся с последующей.

Глава 17 «Спрайты и обнаружение столкновений»

Проверь свои знания

1 Обнаружение столкновений означает определение ситуации, когда два графических объекта касаются друг друга или накладываются друг на друга.

2 Обнаружение столкновений на уровне пикселей учитывает контуры графического объекта. Обнаружение столкновений на уровне прямоугольников учитывает границы прямоугольника вокруг объекта. Обнаружение столкновений на уровне пикселей более точное и реалистичное, но требует больше кода, что негативно сказывается на потреблении ресурсов компьютера.

3 Можно отслеживать количество спрайтов с помощью обычного списка Python или группы Pygame для спрайтов.

4 Можно управлять скоростью анимации (частотой кадров) в коде, или добавляя задержки между кадрами, или используя **`pygame.time.Clock`** для

получения особой частоты. Также можно изменить, насколько далеко (на сколько пикселей) объект передвигается в каждом кадре.

5 Метод задержки менее точен, потому что он не принимает в расчет, как долго выполняется сам код для каждого кадра, поэтому нельзя знать наверняка, какую частоту кадров ты получишь.

6 Узнать частоту кадров программы при запуске можно с помощью `pygame.time.Clock.get_fps()`.

Глава 18 «Новый вид ввода – события»

Проверь свои знания

1 Два вида событий, на которые программа может ответить, – это события *мыши* и *клавиатуры*.

2 Фрагмент кода, который работает с событием, называется *обработчиком событий*.

3 Pygame использует событие `KEYDOWN` для определения нажатия клавиш.

4 Атрибут `pos` говорит нам, где находится указатель мыши при событии.

5 Чтобы узнать следующий доступный номер для пользовательского события, используй `pygame.USEREVENT`.

6 Чтобы создать таймер, используй `pygame.time.set_timer()`.

7 Чтобы вывести текст в окне Pygame, используй объект `font`.

8 Ниже представлены три шага использования объекта `font`:

- создание объекта `font`;
- обработка текста, создание поверхности;
- обновление поверхности и ее вывод на поверхность экрана.

Попробуй самостоятельно

1 Почему мяч ведет себя странно, когда ударяется о сторону ракетки, а не о ее вершину? Потому что у нас есть столкновение, поэтому код пытается изменить направление мяча по оси *y* (заставить его двигаться вверх, а не вниз). Но поскольку мяч поступает со стороны, он все еще «сталкивается» с ракеткой даже после смены направления. При следующей итерации цикла (один кадр спустя) он снова меняет направление, то есть идет вниз, и т. д. Простой способ это исправить – всегда настраивать мяч на движение вверх (отрицательная скорость по оси *y*) , когда он сталкивается с ракеткой. Это не идеальный вариант, потому что он означает, что если даже мяч ударится о бок ракетки, он отскочит – не слишком реалистично! Но это решит проблему мяча, прыгающего вокруг ракетки. Если тебе нужно более реалистичное решение, потребуется больше кода. Тебе, возможно, придется добавить что-то при проверке угла ракетки, о который ударился мяч, перед его «отскакиванием».

Решение представлено в файле *TIO_CH18_2.py*, в папке *ответы* каталога с примерами.

2. Пример кода, который добавляет случайный порядок в программу, ты найдешь в файле *TIO_CH18_2.py* в папке *ответы* каталога с примерами.

Глава 19 «Звуковое сопровождение»

Проверь свои знания

1. Типы файлов для хранения звука: Wave (**.wav**), MP3 (**.mp3**), Ogg Vorbis (**.ogg**) и Windows Media Audio (**.wma**).
2. Модуль `pygame.mixer` используется для воспроизведения музыки.
3. Уровень громкости для звуковых объектов Pygame настраивается с помощью метода `set_volume()` каждого звукового объекта.
4. Уровень громкости фоновой музыки настраивается с помощью функции `pygame.mixer.music.set_volume()`.
5. Чтобы музыка затихала, используй метод `pygame.mixer.music.fadeout()`. Используй количество миллисекунд (тысячных долей секунды) в качестве аргумента для затухания. Например, `pygame.mixer.music.fadeout(2000)` произведет затухание звука через 2 секунды.

Попробуй самостоятельно

Код для программы по угадыванию чисел со звуком ты найдешь в файле *TIO_CH19_1.py* в папке в папке *ответы* каталога с примерами.

Глава 20 «Продолжение работы над графическими интерфейсами»

Проверь свои знания

1. Три названия графических элементов GUI: *элемент управления*, *виджет* и *компонент*.
2. Буква, которую ты нажимаешь (вместе с клавишей **Alt**), чтобы попасть в меню, называется *горячей клавишей*.
3. Файл Qt Designer должен заканчиваться расширением *.ui*.
4. Типы компонентов, которые можно включить в GUI с помощью PyQt, включают в себя кнопки, флажки, индикаторы прогресса, списки, переключатели, поля с изменяемым значением, ползунковые регуляторы, поля ввода (или текстовые поля), изображения, метки и др. На панели **Widget Box** в Qt Designer приведен полный список.
5. Чтобы назначить компоненту действие, потребуется обработчик событий.

6 Символ **&** используется, чтобы определить горячую клавишу в Qt Designer.

7 Содержимое поля **SpinBox** в Qt Designer – всегда целое число.

Попробуй самостоятельно

1 Версию программы по угадыванию чисел с помощью Qt Designer ты найдешь в файлах примеров *TIO_CH20_1. py* и *TIO_CH20_1.ui*.

2 Чтобы исправить проблему поля **SpinBox**, выбери данный виджет в Qt Designer. На панели **Property Editor** измени значения свойств **minimum** и **maximum**. Значение свойства **minimum** должно быть примерно **-1000**, а свойства **maximum** – около **1000000**.

Глава 21 «Форматирование вывода и строк»

Проверь свои знания

1 Если у тебя есть две разные команды **print** и ты хочешь вывести все в одной строке, используй **end=' '** в конце первой команды **print**:

```
print("Как тебя", end=' ')
print(«зовут?»)
```

2 Чтобы добавить дополнительные пустые строки при выводе чего-либо, можно или добавить лишнюю команду **print** без содержимого:

```
print(«Привет,»)
print()
print()
print()
print(«мир»)
```

или указать символ перехода строки, **\n**:

```
print(«Привет,\n\n\nмир»)
```

3 Чтобы выровнять содержимое по колонкам, используй знак табуляции, **\t**.

4 Чтобы вывести число в E-нотации, используй форматирующую строку **%e** или **%E**:

```
>>> number = 12.3456
>>> print('%e' % number)
1.234560e+001
```

Попробуй самостоятельно

1 Программа будет выглядеть так:

```
name = input(«Как тебя зовут? «)
age = int(input(«Сколько тебе лет? «))
color = input(«Какой твой любимый цвет? “)
print(“Тебя зовут”, name, end=’ ‘)
print(“тебе”, age, “лет”, end=’ ‘)
print(«и тебе нравится цвет», color, «цвет.»)
```

2 Код для выравнивания таблицы времени посредством табуляции выглядит так:

```
for looper in range(1, 11):
    print(looper, “\ttimes 8 =\t”, looper * 8)
```

3 Существует два способа вывести дроби числа 8:

```
for looper in range(1, 9):
    fraction = looper / 8
    print(‘%i/8 = %.3f’ % (looper, fraction))
```

Здесь мы используем форматирующие строки как для дроби, так и для десятичного числа.

```
for looper in range(1, 9):
    fraction = looper / 8
    print(str(looper) + ‘/8 = %.3f’ % fraction)
```

Здесь первая часть, **`print str(looper) + ‘/8 =`**, выводит дробь. Последняя часть, **`%.3f’ % fraction`**, выводит десятичное число с тремя знаками после запятой.

Глава 22 «Ввод и вывод файла»

Проверь свои знания

1 Тип объекта в Python, который используется для работы с файлами, называется *объектом файла*.

2 Объект файла создается с помощью функции **`open()`**, которая является одной из встроенных функций Python.

3 *Имя файла* – имя, используемое для хранения файла на диске (или другом носителе, например Flash-накопителе). *Объект файла* часто используется для работы с файлами в Python. Имя объекта файла может не совпадать с именем файла на диске.

4 Когда программа заканчивает чтение или запись файла, его нужно закрыть.

5 Если ты открыл файл в режиме добавления информации и записал в него что-то, эта информация будет добавлена в конец файла.

6 Если ты открыл файл в режиме записи и записал в него что-то, все, что было в файле, будет потеряно и заменено новыми данными.

7 Чтобы сбросить точку чтения файла обратно на начало, используй метод **`seek()`** с аргументом **`0`**:

```
myFile.seek(0)
```

8 Чтобы сохранить объект Python в файл с помощью модуля **`pickle`**, используется метод **`pickle.dump()`** с объектом, который ты хочешь сохранить, и именем файла в качестве аргумента:

```
pickle.dump(myObject, "my_pickle_file.pkl")
```

9 Чтобы получить объект из сохраненного файла, используй метод **`pickle.load()`** с сохраненным файлом в качестве аргумента:

```
myObject = pickle.load("my_pickle_file.pkl")
```

Напомним, что файлы в модуле `pickle` должны быть открыты в двоичном режиме ('`wb`' или '`rb`').

Попробуй самостоятельно

1 Ниже представлен код простой программы для создания глупых предложений:

```
import random
noun_file = open("nouns.txt", 'r')
nouns = noun_file.readline()
noun_list = nouns.split(',')
noun_file.close()
adj_file = open("adjectives.txt", 'r')
adjectives = adj_file.readline()
adj_list = adjectives.split(',')
adj_file.close()
verb_file = open("verbs.txt", 'r')
verbs = verb_file.readline()
verb_list = verbs.split(',')
verb_file.close()
adverb_file = open("adverbs.txt", 'r')
adverbs = adverb_file.readline()
adverb_list = adverbs.split(',')
adverb_file.close()
noun = random.choice(noun_list)
```

```
adj = random.choice(adj_list)
verb = random.choice(verb_list)
adverb = random.choice(adverb_list)
print("The", adj, noun, verb, adverb + '.')
```

Файлы со словами должны быть списками слов, разделенных запятыми.

2 Ниже представлен код программы, которая сохраняет данные в текстовый файл:

```
name = input(«Введи свое имя: «)
age = input(«Введи свой возраст: «)
color = input(«Введи свой любимый цвет: «)
food = input(«Введи свое любимое блюдо: «)
my_data = open("my_data_file.txt", 'w')
my_data.write(name + "\n")
my_data.write(age + "\n")
my_data.write(color + "\n")
my_data.write(food)
my_data.close()
```

3 Программа, которая сохраняет некоторые данные с помощью модуля **`pickle`**:

```
import pickle
name = input("Введи свое имя: ")
age = input(«Введи свой возраст: «)
color = input(«Введи свой любимый цвет: «)
food = input(«Введи свое любимое блюдо: «)
my_list = [name, age, color, food]
pickle_file = open("my_pickle_file.pkl", 'wb')
pickle.dump(my_list, pickle_file)
pickle_file.close()
```

Открываем в двоичном режиме

Глава 23 «Случайный порядок»

Проверь свои знания

1 *Случайное событие* – это то, что происходит (событие), и ты не знаешь заранее, каков будет его результат. Два примера: бросание монеты (ты не знаешь, выпадет орел или решка) и бросание игральных костей (ты не знаешь, какие числа выпадут).

2 Бросание 11-гранной кости отличается от бросания двух 6-гранных костей, потому что при 11-гранной кости все числа от 2 до 12 имеют одинаковые шансы. При двух 6-гранных костях некоторые числа (суммы двух костей) имеют больше шансов, чем другие.

3 Два способа моделировать бросание кости в Python:

```
import random
sides = [1, 2, 3, 4, 5, 6]
die_1 = random.choice(sides)
```

и

```
import random
die_1 = random.randint(1, 6)
```

4 Чтобы представить отдельную карту, мы использовали объект.

5 Чтобы представить колоду карт, мы использовали список. Каждый элемент списка был одной картой (объектом).

6 Чтобы удалить карту из колоды или руки, мы использовали метод **`remove()`** для списков, например **`deck.remove()`** или **`hand.remove()`**.

Попробуй самостоятельно

Просто попробуй – и увидишь, что произойдет.

Глава 24 «Компьютерное моделирование»

Проверь свои знания

1 Компьютерное моделирование используется по ряду причин:

- чтобы сэкономить деньги (проводить эксперименты, которые могут быть слишком дорогими в реальной жизни);
- чтобы защитить людей и оборудование (проведение опасных экспериментов);
- чтобы попробовать невозможные в реальном мире вещи (столкновение астероида с Луной);
- чтобы ускорить время (заставить эксперимент идти быстрее, чем в реальном мире). Хорошо при изучении длительных процессов, например таяния ледников;
- чтобы замедлить время (заставить эксперимент идти медленнее). Хорошо для изучения быстрых событий, например электронов, проникающих в проволоку.

2 Ты можешь составить список любых примеров компьютерного моделирования. Это могут быть игры, математические или научные программы, прогноз погоды (которые создаются с помощью компьютерного моделирования).

3 Объект `timedelta` используется для хранения разницы между двумя точками времени или датами.

Попробуй самостоятельно

Все программы этой главы слишком длинные для книги. Ты можешь найти их в папке *ответы* каталога с примерами:

1 *TIO_CH24_1.py* – «Луноход» с проверкой выхода за орбиту;
2 *TIO_CH24_2.py* – «Луноход» с возможностью новой игры;
3 *TIO_CH24_3.py* – «Тамагочи» с кнопкой «Пауза».

Глава 26 «Создание сетевых соединений с помощью сокетов»

Проверь свои знания

1 Сервер – это программа, которая принимает сетевые подключения. Многие люди используют слово «сервер» для обозначения большого специализированного компьютера, запускающего серверное программное обеспечение.
2 Тебе нужно либо закодировать строку с помощью команды `.encode('utf-8')`, либо поставить букву `b` перед строкой, чтобы превратить ее в объект `bytes`.
3 Если ты и твой друг находитесь в разных локальных сетях, локальный IP-адрес компьютера твоего друга не будет работать в твоей сети.
4 Команда `cd` – сокращение от Change Directory (Изменить каталог).

Попробуй самостоятельно

1 Открой окно оболочки, введи команду `telnet helloworldbook3.com 80`, нажми клавишу **Enter**, введи `Host: helloworldbook3.com` и дважды нажми клавишу **Enter**. Веб-сервер должен отправить тебе ответ! (Если ты используешь операционную систему Windows, то можешь не увидеть, что вводимые вами данные отображаются в окне оболочки. Не волнуйся, просто продолжайте печатать. Ты должен получить ответ.)
2 Версия сервера чата с командой `/me` находится на веб-сайте; она называется *TIO_CH26_2.py*.
3 Версия сервера чата, которая оповещает о новых пользователях, также находится на веб-сайте; она называется *TIO_CH26_3.py*.
4 Версия клиентской программы, которая заменяет слово `:pizza:` на изображение куска пиццы, также есть на сайте; она называется *TIO_CH26_4.py* (Изображение куска пиццы под названием *pizza.png* ты также можешь найти среди файлов примеров.)


pygame window
Welcome to my server!
Whats your name:
Ты: Картер
Hi, Картер! Type a message and press Enter to send it.
[Папка]: Привет, Картер! Что ты хочешь на обед?
Ты:
[Папка]: И много ты съешь?
Ты:

Предметный указатель

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Уоррен Сэнд, Картер Сэнд

Hello World!

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Райтман М. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 39,49. Тираж 200 экз.

Отпечатано в ПАО «Т8 Издательские Технологии»
109316, Москва, Волгоградский проспект, д. 42, корпус 5

Веб-сайт издательства: **www.dmkpress.com**