ЛЕГКОЕ ПРОГРАММИРОВАНИЕ

ПРОГРАММИРУЕМ С MINECRAFT

СОЗДАЙ СВОЙ МИР С ПОМОЩЬЮ PYTHON

КРЭЙГ РИЧАРДСОН

МИФ ДЕТСТВО

no starch press

Craig Richardson

Learn to program with Minecraft

Transform your world
with the power of Python

no starch
press

San Francisco, 2016

Крэйг Ричардсон

Программируем с Minecraft

Создай свой мир
с помощью Python

Перевод с английского Станислава Ломакина

Москва
«Манн, Иванов и Фербер»
2017

УДК 087.5:004.43
ББК 83.84:32.973-018.1
Р56

Издано с разрешения *No Starch Press*

На русском языке публикуется впервые

Возрастная маркировка в соответствии
с Федеральным законом № 436-ФЗ: 0+

Ричардсон, Крэйг

Р56 Программируем с Minecraft. Создай свой мир с помощью Python / Крэйг Ричардсон ; пер. с англ. Станислава Ломакина ; [науч. ред. Г. Гаджиев]. — М. : Манн, Иванов и Фербер, 2017. — 368 с. : ил.

ISBN 978-5-00100-819-4

Эта книга научит программировать на языке Python. Выполняя пошаговые инструкции, вы познакомитесь с базовыми принципами программирования и создадите программы, которые будут творить в мире Minecraft настоящие чудеса: в мгновение ока возводить постройки, телепортировать игрока, создавать цветные стены, работающий душ, тайные ходы и многое другое.

Для детей от 10 лет и взрослых, желающих освоить Python нескучным способом.

УДК 087.5:004.43
ББК 83.84:32.973-018.1

ISBN 978-5-00100-819-4

ОГЛАВЛЕНИЕ

2. ТЕЛЕПОРТАЦИЯ С ПОМОЩЬЮ ПЕРЕМЕННЫХ

3. МАТЕМАТИКА, МОМЕНТАЛЬНОЕ СТРОИТЕЛЬСТВО И СУПЕРПРЫЖКИ

4. ОБЩАЕМСЯ С ПОМОЩЬЮ СТРОК

5. «ИСТИНА» И «ЛОЖЬ» БУЛЕВЫХ ЗНАЧЕНИЙ

6. КОНСТРУКЦИЯ IF, ДУШ И ПОТАЙНАЯ ДВЕРЬ

7. ЦИКЛ WHILE, ДИСКОТЕКА И ЦВЕТОЧНЫЙ ДОЖДЬ

8. ФУНКЦИИ КАК ИСТОЧНИК БОЛЬШИХ ВОЗМОЖНОСТЕЙ

9. СПИСКИ, СЛОВАРИ И УДАРЫ ПО БЛОКАМ

10. ЦИКЛЫ FOR И ВОЛШЕБСТВО В MINECRAFT

11. КОПИРОВАНИЕ КОНСТРУКЦИЙ С ПОМОЩЬЮ ФАЙЛОВ И МОДУЛЕЙ

12. ОБЪЕКТНО ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ — ЭТО КЛАССНО!

Посвящается всем взрослым и детям, которые читали ранние черновики этой книги, пользовались моими советами и инструкциями и посещали мои семинары. Огромное спасибо за ваш энтузиазм и поддержку!
Эта книга — для вас

ВВЕДЕНИЕ

Эта книга научит вас писать программы на языке программирования Python и управлять с их помощью событиями в мире Minecraft. Освоив основы программирования, вы тут же пустите новые знания в ход: соорудите постройки, создадите мини-игры и превратите обычные предметы в потрясающие артефакты. Навыков, которые вы получите, будет достаточно для воплощения в жизнь самых смелых идей! И не только в мире Minecraft. Если захотите, сможете писать на Python игры, приложения и полезные утилиты. Итак, сейчас вы делаете первый шаг на пути к тому, чтобы стать выдающимся программистом и повелителем мира Minecraft!

Зачем изучать программирование?

Одна из главных причин, по которым люди решают освоить программирование, — тренировка ума для решения сложных задач. Вы научитесь разбивать большие задачи на маленькие, с которыми проще иметь дело, и подключать при этом творческое мышление.

Еще один плюс программирования в том, что оно учит мыслить логически. Логика нужна, чтобы тщательно спланировать структуру программы и сценарий ее работы. Навыки решения задач, творческое и логическое мышление пригодятся вам в самых разных областях жизни, а не только при написании кодов.

Профессия программиста — отличная перспектива: каждый день вы будете решать интересные задачи, требующие нестандартного подхода. А если вы предпочтете другую профессию, программирование

может стать необыкновенно увлекательным хобби. Кстати, я начал писать программы в свободное от работы время, а в итоге стал профессиональным программистом.

И самое главное: программирование может доставить уйму радости! Мало что сравнится с удовольствием, которое испытываешь, глядя, как созданная тобою программа делает нечто прекрасное!

Почему Python?

Все ясно, но зачем вам изучать программирование именно на языке Python? Начнем с того, что этот язык отлично подходит для новичков. Коды на Python легко писать и читать, при этом мощности языка хватает, чтобы создавать на нем серьезные компьютерные программы. Неспроста Python является одним из самых распространенных языков программирования в мире!

Почему Minecraft?

Игра Minecraft очень популярна, ведь она такая увлекательная! Вы можете создавать в игровом мире все, что захотите, — лишь дайте волю воображению! А управляя миром Minecraft с помощью Python, вы еще больше раскроете свой творческий потенциал — сделаете такое, что просто невозможно повторить обычными средствами игры (например, в считаные секунды возведете огромное здание).

Начать программировать бывает непросто, ведь сначала приходится изучать примеры кодов, которые не делают ничего интересного. Однако, совместив Python с Minecraft, вы сможете сразу писать занимательные программы и наблюдать результат их работы прямо в игре.

Что вы найдете в этой книге?

Каждая из 12 глав посвящена определенным возможностям языка Python. Знакомясь с ними, вы будете постепенно наращивать свой багаж знаний. Главы состоят из объяснения устройства языка, рабочих примеров кода и миссий. В ходе миссий вы будете писать программы, взаимодействующие с Minecraft. Основы их кодов я уже подготовил, так что вам потребуется лишь добавить недостающие фрагменты. В результате вы обретете навык решения задач, необходимый каждому программисту.

Рассмотрим вкратце, что ждет вас в каждой из глав.

Глава 1 «Готовимся к приключениям» поможет настроить на вашем компьютере Python и Minecraft, после чего вы сможете приступить к программированию!

Глава 2 «Телепортация с помощью переменных» покажет, как мгновенно переместить игрока. Вы узнаете, что такое переменные и как они помогают хранить данные. Затем углубите полученные навыки, отправив игрока в телепортационное путешествие по игровому миру.

Глава 3 «Математика, моментальное строительство и суперпрыжки» научит использовать математические операции для получения игроком суперспособностей и мгновенной постройки зданий. Хотите создать дом за секунду? Математические операции помогут вам в этом! Хотите подпрыгнуть высоко-высоко над землей? И здесь пригодятся математические операции!

В **главе 4 «Общаемся с помощью строк»** вы узнаете о строках и о том, как с их помощью создать интерактивный чат, а также научитесь писать программы, отправляющие текстовые сообщения, которые смогут прочитать другие пользователи.

Глава 5 «“Истина” и “ложь” булевых значений» покажет, что при помощи булевых значений и логических операций ваши программы для Minecraft могут отвечать на однозначные вопросы, например такие: «Игрок под водой?», «Игрок на дереве?», «Игрок рядом с домом?»

Глава 6 «Конструкция if, душ и потайная дверь» — здесь мы перейдем на новый уровень булевой логики. Вы узнаете, как с помощью конструкции `if` создавать программы, выполняющие разные действия в зависимости от введенных данных. Хотите сделать потайную дверь, которая открывается, если поставить определенный блок в определенное место? С конструкцией if это возможно!

Глава 7 «Цикл while, дискотека и цветочный дождь» расскажет, как научить программу многократно выполнять код с помощью циклов. Вы сможете автоматизировать работу программ и получить фантастические результаты. Например, след из цветов, который будет тянуться за игроком, или волшебный танцпол, переливающийся всеми цветами радуги! Если показать все это друзьям, они будут поражены!

В **главе 8 «Функции как источник больших возможностей»** вы научитесь мгновенно создавать целые леса и поселки при помощи

функций. А еще узнаете, как облегчить себе жизнь, используя части кода повторно.

В **главе 9 «Списки, словари и удары по блокам»** вы создадите мини-игры с помощью списков. Список — важный инструмент программирования, позволяющий хранить множество значений в одном месте. Вам предстоит использовать списки, чтобы программа запоминала, по каким блокам вы ударили мечом. А если добавить в эту программу еще несколько строчек кода, получится забавная мини-игра!

Глава 10 «Циклы for и волшебство в Minecraft» покажет, как возводить постройки (например, пирамиды) с помощью циклов for. Используя их, вы сможете рисовать пиксельные картинки и создавать копии самых разных объектов. Представьте, что вы изваяли великолепную статую. Теперь вы можете скопировать и воссоздать ее многократно, получив целую армию статуй!

В **главе 11 «Копирование построек с помощью файлов и модулей»** вы научитесь создавать и изменять файлы прямо из кода программы, что позволит сохранять постройки и переносить их в другие миры Minecraft. Иными словами, скопировав постройку из игрового мира в файл, вы сможете перенести ее куда пожелаете. Хотите сохранить великолепный особняк, в который вложено столько сил и времени? Без проблем! Просто записываете здание в файл, а потом загружаете эти данные куда угодно.

Глава 12 «Объектно ориентированное программирование — это классно!» расскажет о продвинутых методах программирования — наследовании классов и объектов. Изучив эту главу, вы станете настоящим знатоком Python! Выполняя миссии, вы построите здание, а затем с помощью классов, объектов и наследования будете создавать его копии и вариации (например, поселки и гостиницы) — и все это с помощью нескольких дополнительных строк кода!

«Идентификаторы блоков» — удобная справка по идентификаторам блоков Minecraft, которые вы можете использовать в своих программах.

Интернет-ресурсы

Все тексты программ, приведенные в этой книге, доступны по ссылке *http://mif.to/minecraft/*. Вы можете сверять с ними коды ваших программ,

если они работают некорректно или не запускаются, а также использовать эти коды как основу для создания собственных замечательных программ! На странице книги на сайте издательства также выложены установочные файлы — о процессе загрузки и проверки программного обеспечения я подробно расскажу в главе 1.

Приключение начинается!

Надеюсь, вам уже не терпится начать — так же как и мне. Я получил истинное удовольствие, работая над этой книгой и придумывая миссии, которые помогут вам научиться программировать. Хочется верить, что они вам понравятся. Итак, поехали!

1

ГОТОВИМСЯ К ПРИКЛЮЧЕНИЯМ

Прежде чем приступить к написанию занимательных Python-программ для игры Minecraft, вам нужно установить и настроить на компьютере Minecraft, Python и еще несколько других программ. В этой главе я расскажу, как установить и запустить все необходимое программное обеспечение. Установка программ и настройка компьютера — самое сложное, с чем вы столкнетесь в этой книге, так что не стесняйтесь звать на помощь родителей или других взрослых, которые разбираются в компьютерах. Не спешите и внимательно выполняйте инструкции шаг за шагом, иначе что-нибудь может пойти не так.

На следующих страницах вы найдете три блока инструкций для трех разных операционных систем. Если на вашем компьютере установлена Windows, просто читайте дальше. Если ваша операционная система Mac OS, перейдите к разделу «Установка и настройка программ для Mac OS» на с. 33. Если же вы планируете работать на микрокомпьютере Raspberry Pi, перейдите на с. 44 к разделу «Установка и настройка программ для Raspberry Pi». Если в процессе установки и настройки у вас возникнут проблемы, обратитесь к приложению на с. 343.

! *На странице книги по адресу http://mif.to/minecraft/ можно найти дополнительную информацию и последние правки к этим инструкциям.*

Установка и настройка программ для Windows

Чтобы управлять миром Minecraft с помощью языка Python, вам нужно установить пять программных пакетов:

- Minecraft
- Python 3
- Java
- Minecraft Python API
- Minecraft-сервер Spigot

В этом разделе я расскажу, как установить эти программы на компьютер с операционной системой Windows. Итак, начнем с Minecraft.

Установка Minecraft

Если у вас уже установлена последняя версия Minecraft, переходите к разделу «Установка Python» на с. 21. А если вы в этом не уверены, выполните пошаговые инструкции из этого раздела.

В случае если у вас еще нет Minecraft, игру можно приобрести на официальном сайте *https://minecraft.net/ru-ru/*. При необходимости попросите взрослых вам помочь. Запомните или запишите имя пользователя и пароль, которые вы укажете при покупке Minecraft, — они понадобятся позже, чтобы войти в игру.

После покупки установите игру на компьютер. Для этого выполните следующее:

Download — скачать

1. Перейдите по ссылке *https://minecraft.net/ru-ru/download/*.
2. В разделе «Загрузить Minecraft для Windows» найдите кнопку **Загрузить** и кликните по ней, чтобы скачать файл *MinecraftInstaller.msi*. Если всплывет панель с вопросом, сохранить файл или открыть его, выберите **Сохранить (Save File)**.
3. Дождитесь окончания загрузки файла, а затем откройте его. Если появится диалоговое окно с вопросом, хотите ли вы запустить файл, выберите **Запустить (Run)**. Не волнуйтесь, этот файл безопасен для компьютера!

Next — далее

Install — установить

4. Когда откроется окно установщика Minecraft, кликните **Next**, а затем кликните **Next** еще раз. После этого кликните **Install**.

5. Возможно, программа установки спросит, действительно ли вы хотите установить Minecraft. Разумеется, хотите! Кликните **Да (Yes)**. Дождитесь окончания установки. За это время я успел съесть печенье и выпить стакан воды.

6. Когда установка завершится, нажмите **Finish**.

Finish — завершить

Ну вот, Minecraft установлен. Знаете, чем стоит заняться теперь? Разумеется, поиграть! Для этого нужно выполнить всего несколько действий:

1. Кликните по кнопке **Пуск (Start)** или нажмите на клавишу WINDOWS на клавиатуре, найдите Minecraft в списке программ и кликните по его иконке.

2. Minecraft запустится и, возможно, установит обновления.

3. После этого появится окно входа в игру. Введите имя пользователя и пароль, которые вы использовали при покупке Minecraft, и нажмите **Войти**.

4. Нажмите **Играть**. Перед запуском игры Minecraft загрузит еще несколько обновлений.

5. И наконец, кликните **Одиночная игра (Single Player) → Создать новый мир (Create New World)**. Введите название вашего мира и кликните **Создать новый мир (Create New World)**. После этого программа создаст мир — можете играть, сколько пожелаете.

Повеселитесь! Если прежде вы не играли в Minecraft, играйте, пока в мире Minecraft не стемнеет. Берегитесь монстров! Имейте в виду, что программировать на Python мы будем в многопользовательском режиме игры, который отличается от однопользовательского. Мы поговорим об этом подробнее в разделе «Запуск Spigot и создание профиля игры» на с. 26.

А теперь за работу! Пора установить Python. Находясь в игре, нажмите клавишу ESC, чтобы появился курсор. В открывшемся меню выберите **Сохранить и выйти (Save and Quit to Title) → Выйти из игры (Quit Game)** и закройте Minecraft.

Установка Python

Python — это язык программирования, который вы будете изучать с помощью этой книги. Давайте установим программное обеспечение Python прямо сейчас.

1. Перейдите по ссылке *http://www.python.org/downloads/*.

2. Кликните по кнопке **Download Python 3.5.2** (на момент написания этой книги версия 3.5.2 была последней, однако, если вы обнаружите на сайте более позднюю версию, установите ее).

3. Начнется скачивание Python. Если браузер задаст вам вопрос, сохранить файл или открыть его, выберите **Сохранить файл (Save File)**.

4. Когда загрузка программы-установщика завершится, кликните по ее иконке. Если возникнет диалоговое окно с вопросом, действительно ли вы хотите запустить этот файл, кликните **Запустить (Run)**.

Install now — установить сейчас

5. Когда установщик Python запустится, поставьте галочку в чекбокс Add Python 3.5 to Path, как на рис. 1.1. Затем кликните по **Install Now**.

Рис. 1.1. Убедитесь, что вы поставили галочку в чекбокс Add Python 3.5 to PATH

6. Может появиться окно с вопросом, разрешить ли программе установить программное обеспечение на этом компьютере. Кликните **Да (Yes)** и дождитесь окончания установки Python. За это время я успел подойти к окну, а когда вернулся за компьютер, дело было сделано.

Close — закрыть

7. Кликните **Close**. Python установлен!

Установка Java

Теперь, когда и Minecraft, и Python установлены, нужно сделать так, чтобы они могли друг с другом взаимодействовать. Для этого воспользуемся программой под названием Spigot. Однако для ее работы требуется, чтобы на компьютере была установлена Java. Давайте с этим разберемся.

Возможно, Java уже есть на вашем компьютере — проверьте это так:

1. Кликните по кнопке **Пуск (Start)** или нажмите клавишу WINDOWS на клавиатуре и введите `cmd` в строке поиска. Найдите программу под названием cmd и запустите ее, кликнув по названию.

2. Вы увидите окно с черным фоном и приглашением для ввода команды (на моем компьютере это строка `C:\Users\МИФ>`). В строке приглашения введите команду `java -version` и нажмите ENTER.

3. Если после этого появится сообщение, как на рис. 1.2, и число после первой точки окажется 7 или больше, значит, подходящая версия Java уже установлена. В этом случае переходите к разделу «Установка Minecraft Python API и Spigot» на с. 25.

4. Если же вы увидите сообщение, что команда `java -version` не распознана или версия после точки меньше 7, установите Java, выполнив пошагово инструкции ниже.

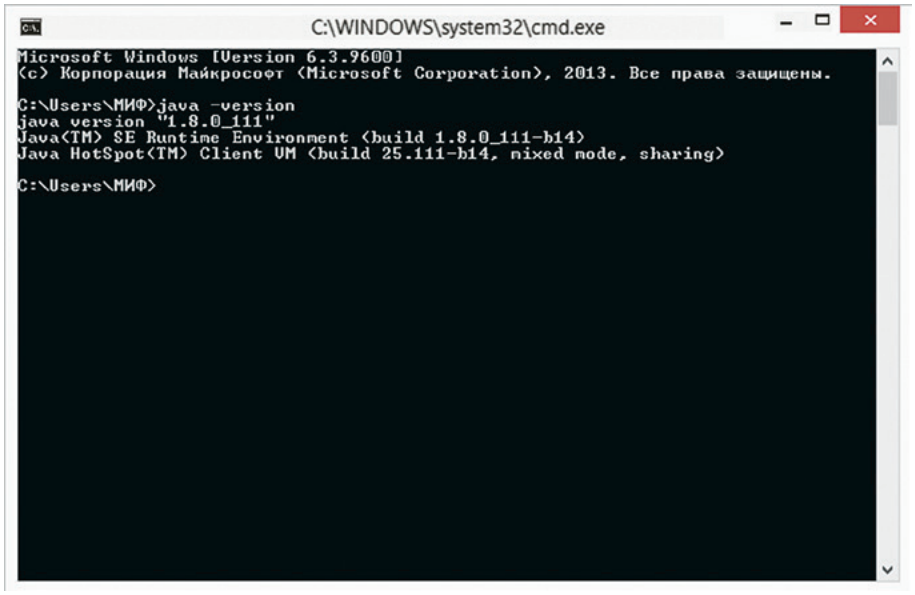

Рис. 1.2. Введя команду `java -version`, я увидел, что Java уже есть на компьютере

Чтобы установить Java, сделайте следующее:

1. Перейдите по ссылке *http://www.java.com/ru/download/*.
2. Кликните по кнопке **Загрузить Java бесплатно**. Затем кликните **Согласиться и начать бесплатную загрузку**.
3. Когда программа установки скачается, кликните по ее иконке. Если возникнет диалоговое окно с вопросом, разрешить ли программе внести изменения на этом компьютере, выберите **Да (Yes)**.
4. Когда появится окно установщика, кликните **Install**.
5. Если откроется страница с вопросом, хотите ли вы установить какую-то еще программу, например приложение Ask Search, панель поиска Yahoo! или что-нибудь в этом роде, снимите соответствующую галочку, чтобы не устанавливать дополнительную программу — она вам не понадобится.
6. Также установщик может спросить, хотите ли вы сделать Yahoo! своей домашней страницей. Вряд ли вам это нужно, поэтому выберите **Do not update browser settings** и кликните **Next**.

Do not update browser settings — не обновлять настройки браузера

7. Дождитесь окончания установки Java. Я успел за это время отправить другу короткое сообщение. Когда установка завершится, кликните **Close**.

Теперь убедитесь, что установка Java прошла успешно:

1. Кликните по кнопке **Пуск (Start)** и введите `cmd` в строке поиска. Запустите программу cmd.
2. В окне cmd введите `java -version` и нажмите ENTER.
3. Если вы увидите сообщение как на рис. 1.2, это означает, что Java успешно установлена. Если же появится сообщение об ошибке «*Не является внешней или внутренней командой, исполняемой программой или пакетным файлом*» (*Java' is not recognized as an internal or external command, operable program or batch file)*, значит, установка по каким-то причинам не произошла. В таком случае попробуйте переустановить Java и снова ввести в окне cmd ту же команду. Если и это не поможет и вы снова увидите сообщение об ошибке, перейдите по ссылке *http://www.java.com/ru/download/help/path.xml* — возможно, описанные там действия помогут решить проблему.

Итак, Java установлена и готова к запуску Minecraft-сервера Spigot. Приступим к его установке!

Установка Minecraft Python API и Spigot

Теперь вам нужно установить на компьютер Minecraft Python API и Minecraft-сервер Spigot.

API — это способ взаимодействия одних программ с другими. В нашем случае Minecraft Python API позволяет игре Minecraft понимать команды, написанные на языке Python. API преобразует ваш код в объекты и действия, чтобы в игре, например, появился новый блок или переместился игрок. Правда, взаимодействовать программы смогут, только когда вы подключитесь к Minecraft-серверу.

Обычно сервер соединяет компьютеры, чтобы несколько онлайн-пользователей могли играть в одном мире Minecraft. В этой же книге речь пойдет о персональном Minecraft-сервере под названием Spigot, который вы установите на свой компьютер.

Итак, теперь вы знаете, для чего нужны Minecraft-сервер и Minecraft Python API, осталось лишь установить их на компьютер. Я уже подготовил все необходимые программы и поместил их в один архив, чтобы вы могли быстро скачать их и установить. Для этого сделайте следующее:

Minecraft Python API — реализация программного интерфейса игры Minecraft на языке Python

API (*сокр.* от Application Programming Interface) — интерфейс программирования приложений

1. Перейдите по ссылке *http://mif.to/minecraft* и скачайте архив для Windows — файл *Minecraft Tools.zip*.

2. Когда файл скачается, кликните по его иконке правой кнопкой мыши и выберите **Извлечь все (Extract All).** Вам будет задан вопрос, куда поместить распакованные файлы. Кликните по кнопке **Обзор (Browse)** и перейдите в папку *Документы*. Кликните по кнопке **Создать новую папку (Make a New Folder)** и назовите папку *Minecraft Python*. Выберите эту папку и кликните **ОК**. Затем кликните **Извлечь (Extract)**, чтобы распаковать файлы.

3. Перейдите в папку *Документы*, а оттуда в папку *Minecraft Python* — вы должны увидеть там распакованные файлы.

4. Откройте папку *Minecraft Tools*. Ее содержимое показано на рис. 1.3.

5. Сделайте двойной клик по файлу *Install_API*. Появится новое окно, и начнется установка программного интерфейса. Если появится предупреждение, кликните **Все равно запустить (Run Anyway).**

6. Когда установка завершится, нажмите любую клавишу.

Minecraft tools — средства для работы с Minecraft

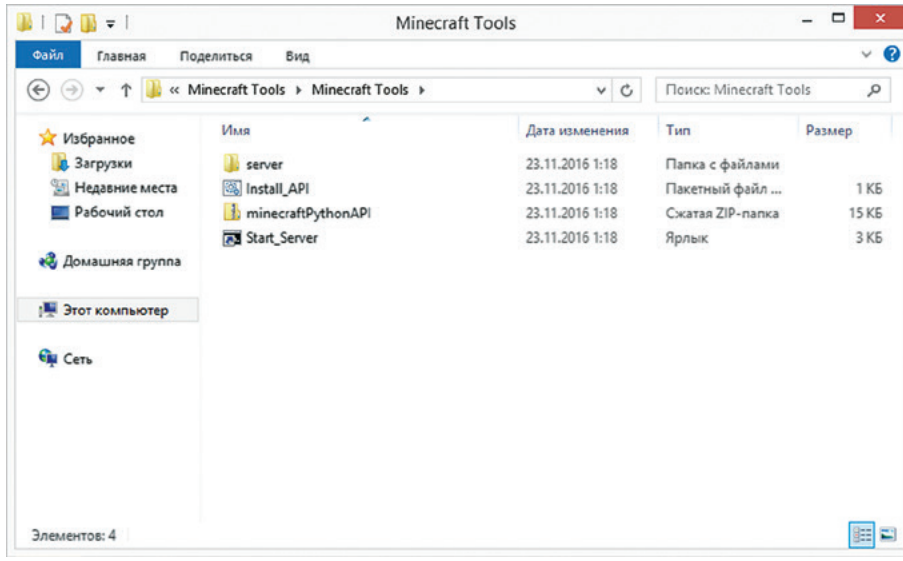

Рис. 1.3. Содержимое папки Minecraft Tools

> ! *Если вы увидите сообщение об ошибке, гласящее, что команда pip не распознана, значит, Python не установлен или установлен некорректно. Вернитесь к разделу «Установка Python» на с. 21 и переустановите программу. При этом обязательно поставьте галочку в чекбоксе Add Python 3.5 to PATH.*

Теперь Minecraft Python API и Minecraft-сервер Spigot установлены. Осталось запустить сервер, чем мы сейчас и займемся.

Запуск Spigot и создание профиля игры

При первом же запуске Spigot создаст мир Minecraft, однако перед тем, как вы начнете играть, нужно настроить профиль игры — для того чтобы проверить, совместимы ли версии Spigot и Minecraft.

Чтобы запустить Spigot, выполните следующие шаги:

1. Перейдите в папку *Minecraft Python*, а оттуда в папку *Minecraft Tools*.

Start server — запустить сервер

2. В папке *Minecraft Tools* сделайте двойной клик по файлу *Start_Server*. Если появится окно с запросом, хотите ли вы открыть этот файл, кликните **Открыть**.

3. Откроется окно, и начнется загрузка сервера. Дождитесь ее завершения и перемотайте текст в окне вверх, к самому началу. Там,

в районе третьей или четвертой строки, должно быть сообщение вида `Starting minecraft server version x.x.x`. Вместо «x» должны стоять конкретные числа — например, на рис. 1.4 это 1.9.4.

Starting minecraft server version x. x. x. — запуск Minecraft-сервера версии x.x.x.

4. Запишите эти числа (номер версии сервера). Окно оставьте открытым.

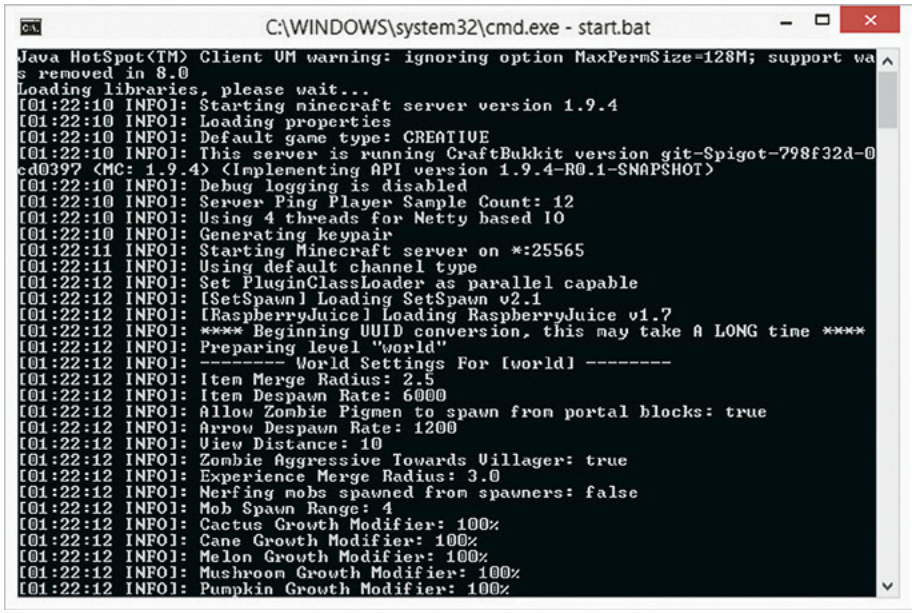

Рис. 1.4. Версия этого Minecraft-сервера — 1.9.4

Выбор подходящей версии Minecraft

Теперь, зная версию сервера, можно настроить профиль игры.

1. Откройте лаунчер Minecraft, кликнув два раза по иконке игры, но пока не нажимайте кнопку **Играть**.

Лаунчер — программа для запуска игры

2. В правом верхнем углу кликните по иконке с тремя горизонтальными линиями. Откроется меню. Выберите пункт «Параметры запуска».
3. В поле «Название» введите: `Программируем с Minecraft`.
4. Кликните по выпадающему списку в поле «Версия» и выберите номер версии вашего сервера. На рис. 1.5 показано, что я выбираю версию 1.9.4.
5. Кликните по кнопке **Сохранить**. Ваш профиль настроен.

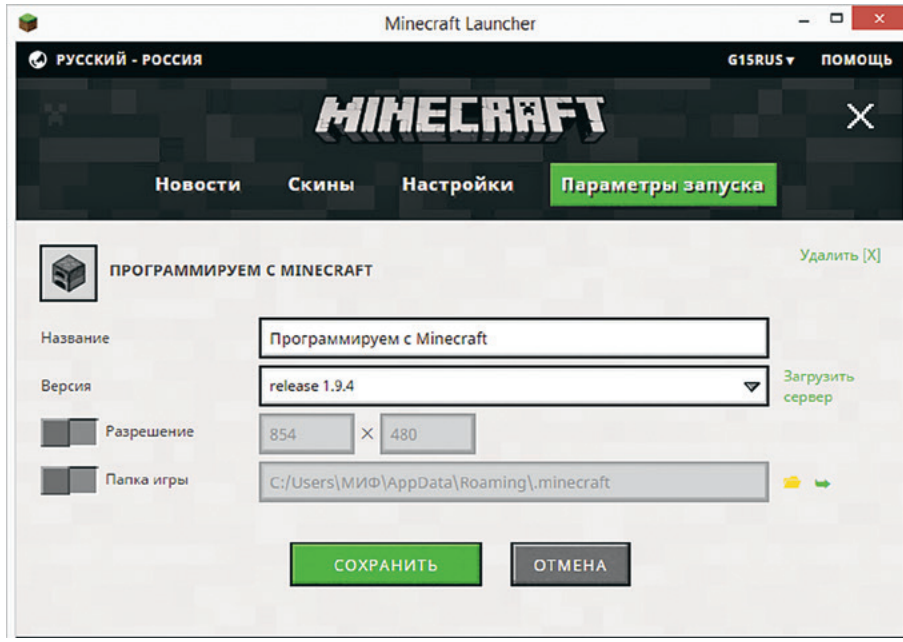

Рис. 1.5. Я создал профиль под названием «Программируем с Minecraft», который использует Minecraft-сервер версии 1.9.4

Теперь каждый раз, когда вам понадобится запустить Minecraft для работы с Python, выбирайте в выпадающем списке лаунчера Minecraft рядом с кнопкой **Играть** профиль **Программируем с Minecraft.** Вы в любой момент сможете переключиться на последнюю версию игры, выбрав в выпадающем меню профиль с последним выпуском.

Создание мира

Профиль настроен, пора войти в новый мир Minecraft.

1. Запустите Minecraft и выберите пункт меню **Сетевая игра (Multiplayer).**
2. Кликните по кнопке **Добавить (Add Server)**.
3. В строке ввода «Название сервера» (Server Name) назовите ваш сервер `Minecraft Python World`, а в строке «Адрес сервера» (Server Address) введите `localhost`, как показано на рис. 1.6. Затем кликните **Готово (Done)**.
4. Сделайте двойной клик по **Minecraft Python World**, и откроется мир, созданный сервером Spigot.

Minecraft Python world — мир Minecraft-Python

Localhost — рабочая станция

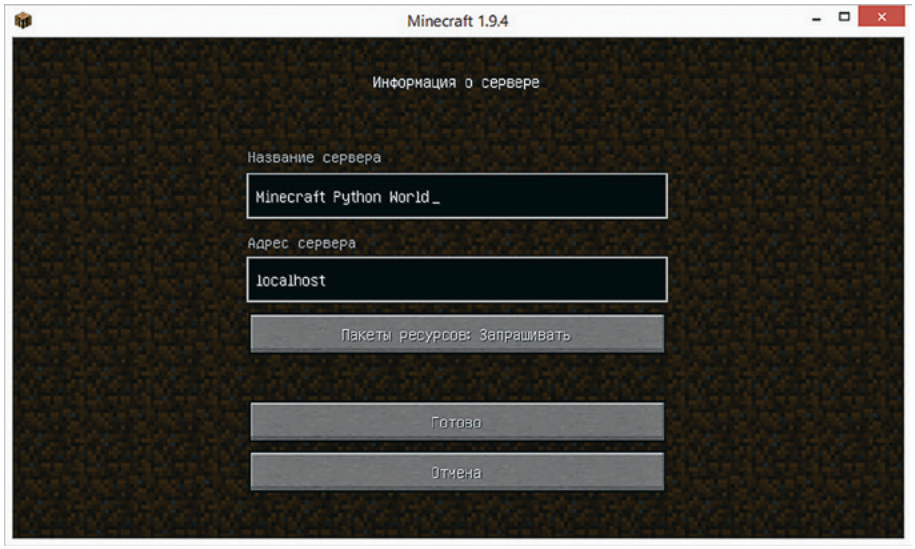

Рис. 1.6. Настройка подключения к серверу

Давайте посмотрим, что за мир появился на сервере Spigot. Этот мир работает в *творческом режиме*, поэтому там ваш игрок может летать. Два раза быстро нажмите на клавишу ПРОБЕЛ, чтобы поднять его в воздух. Нажимая и удерживая ПРОБЕЛ, вы будете поднимать игрока, а нажимая и удерживая SHIFT — опускать. Когда вам наскучат полеты, вновь нажмите ПРОБЕЛ дважды.

Творческий режим по-другому называют режимом креатив. *Прим. науч. ред.*

Создание нового мира

Создание еще одного, нетронутого мира Minecraft на сервере немного отличается от создания нового мира в однопользовательском режиме. Если вы хотите создать другой мир, для начала вам нужно отключиться от сервера.

1. Находясь в игре, нажмите ESC, а затем кликните по кнопке **Отключиться (Disconnect)**, чтобы вернуться к списку серверов.

2. Остановите сервер Spigot, закрыв окно cmd, в котором он запущен.

Чтобы создать новый мир, сделайте копию сервера. Для этого выполните следующее:

1. Перейдите в папку *Minecraft Python*. Кликните правой кнопкой мыши по папке *Minecraft Tools* и выберите **Копировать (Copy)**.

2. Кликните правой кнопкой мыши где угодно внутри папки *Minecraft Python* и выберите **Вставить (Paste).** В результате появится копия папки *Minecraft Tools* с именем *Minecraft Tools — копия*.

3. Кликните правой кнопкой мыши по только что созданной папке-копии и выберите **Переименовать (Rename).** Я назвал новую папку *New World*, но вы у себя на компьютере можете дать ей любое имя, какое пожелаете.

New world — новый мир

4. Откройте папку *New World* (или как вы ее назвали), а затем откройте в ней папку *server*.

Server — сервер

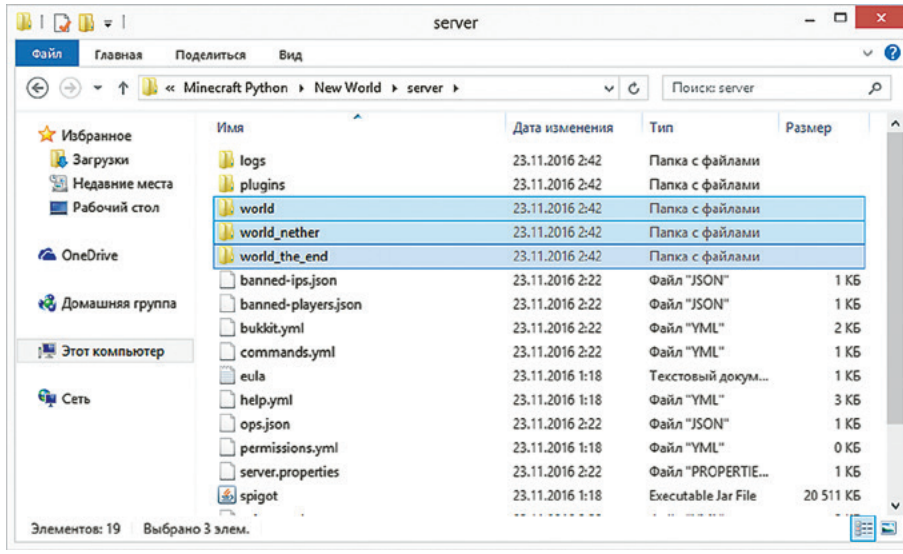

Рис. 1.7. Выделены папки, которые нужно удалить

5. В папке *server* выделите папки *world*, *world_nether* и *world_the_end*, как показано на рис. 1.7. Нажмите клавишу DELETE, чтобы их удалить.

6. Находясь в папке *server*, кликните по файлу *start*. (Нужно кликнуть именно по этому файлу, а не по файлу *Start_Server!*) В результате сервер запустится и создаст новый мир.

7. Теперь, запустив Minecraft и открыв Minecraft Python World, вы увидите только что созданный мир.

Вы можете создать сколько угодно новых миров — просто повторите эти действия. А если захотите открыть предыдущий мир, кликните по файлу *Start_Server* в папке *Minecraft Tools*, а не в папке *New World*.

Чтобы удалить мир, заменив его на новый, просто удалите папки *world*, *world_nether* и *world_the_end* в папке того мира, который вам нужно заменить.

Игра без доступа к интернету

Если у вас нет доступа к интернету, при попытке подключиться к серверу из игры программа выдаст ошибку. Этого можно избежать, изменив настройки сервера. Сначала закройте окно сервера, если он запущен. Затем откройте папку *Minecraft Python*, а в ней папку *Minecraft Tools* и, наконец, папку *server*. Откройте в текстовом редакторе (например, Блокноте) файл *server.properties* и измените значение опции `online-mode` (рис. 1.8) с `true` на `false`. Сохраните изменения, вернитесь обратно в папку *Minecraft Tools* и снова запустите сервер двойным кликом по файлу *Start_Server*. Теперь вы сможете играть в Minecraft без доступа к интернету.

Properties — характеристики

Online-mode — онлайн-режим

True — правда

False — ложь

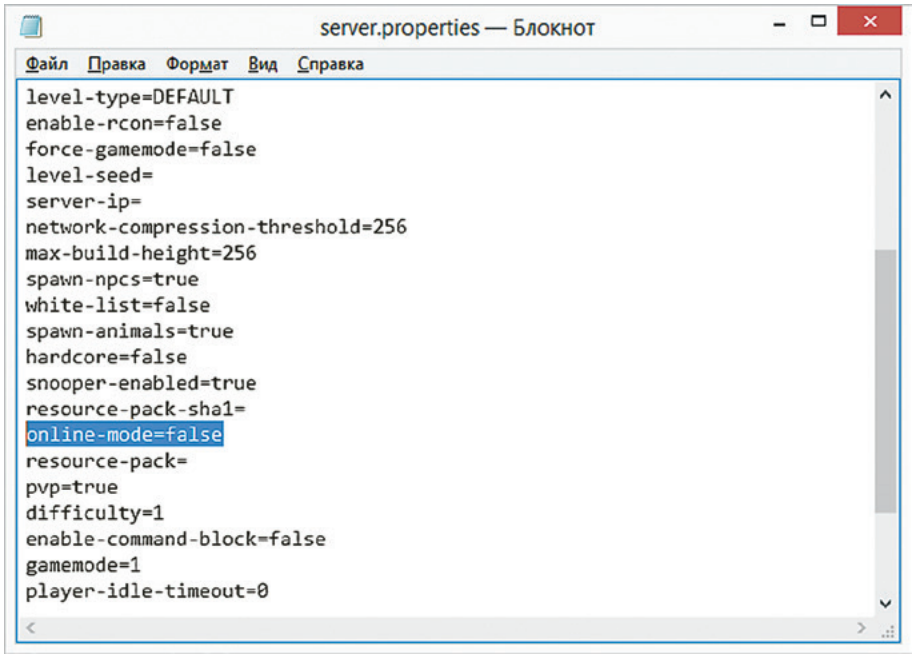

Рис. 1.8. Измените в выделенной строке `true` на `false`

Переключение в режим выживания

Я задал творческий режим в качестве режима сервера по умолчанию. Так проще писать и запускать Python-программы, поскольку ваш игрок не будет терять здоровье и не пострадает от голода и нападений монстров.

Возможно, вы захотите протестировать некоторые программы в *режиме выживания* — просто ради интереса. В этом случае вам нужно будет переключать сервер из творческого режима в режим выживания и обратно. Это довольно просто.

Чтобы переключить сервер из творческого режима в режим выживания, выполните следующие шаги:

1. Откройте папку *Minecraft Tools*, а внутри нее — папку *server*.
2. Найдите файл *server.properties* и откройте его в текстовом редакторе (например, в «Блокноте»).

Gamemode — игровой режим

3. Найдите в этом файле строку `gamemode=1` и измените ее на `gamemode=0`, как показано на рис. 1.9.
4. Сохраните файл и закройте текстовый редактор.
5. Запустите сервер двойным кликом по файлу *Start_Server* в папке *Minecraft Tools*. Теперь, когда вы войдете в Minecraft Python World, игра будет работать в режиме выживания.

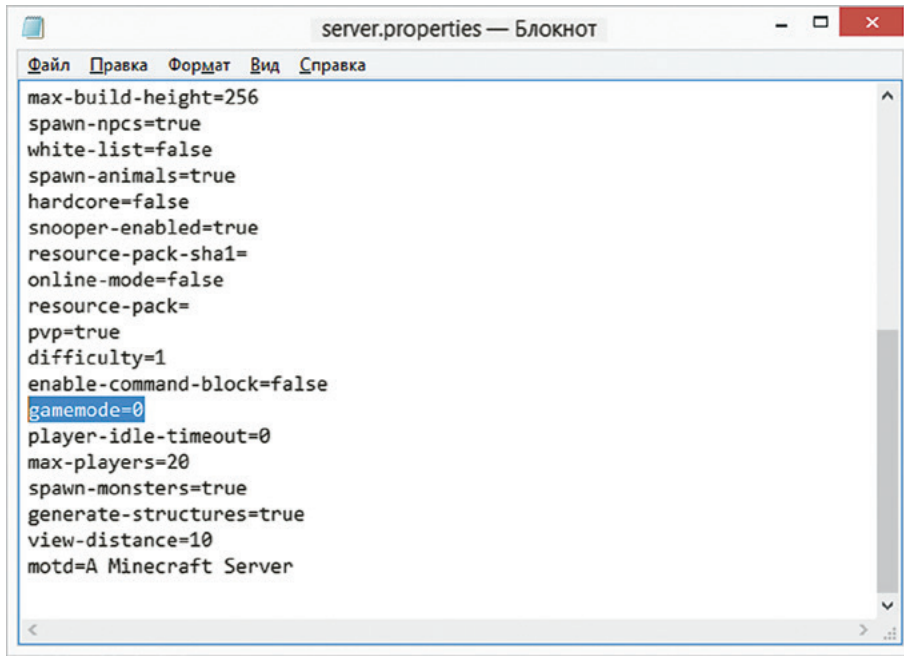

Рис. 1.9. Я переключился в режим выживания, задав `gamemode` значение 0

Чтобы переключиться обратно в творческий режим, достаточно выполнить те же самые шаги, но на шаге 3 заменить в файле *server.properties* `gamemode=0` на `gamemode=1`.

Итак, все необходимые программы настроены! Далее вам предстоит познакомиться со средой разработки IDLE, в которой вы будете писать свои коды. Переходите к разделу «Знакомство с IDLE» на с. 45.

Установка и настройка программ для Mac OS

Чтобы управлять миром Minecraft с помощью Python, на компьютер нужно установить пять программных пакетов:

- Minecraft
- Python 3
- комплект разработчика на языке Java (JDK)
- Minecraft Python API
- Minecraft-сервер Spigot

Итак, начнем с Minecraft.

Установка Minecraft

Если у вас уже установлена последняя версия Minecraft, переходите к разделу «Установка Python» на с. 35. А если вы в этом не уверены, выполните пошаговые инструкции из этого раздела.

В случае если у вас еще нет Minecraft, игру можно приобрести на официальном сайте *https://minecraft.net/ru-ru/*. При необходимости попросите взрослых вам помочь. Запомните или запишите имя пользователя и пароль, которые вы укажете при покупке Minecraft, — они понадобятся позже, чтобы войти в игру.

После покупки установите игру на компьютер. Для этого выполните следующее:

1. Перейдите по ссылке *https://minecraft.net/ru-ru/download*.
2. В разделе «Загрузить Minecraft для macOS» кликните по кнопке **Загрузить**, чтобы скачать файл *Minecraft.dmg*. (Если вы не видите этого раздела, кликните по ссылке *Mac* внизу страницы, под заголовком «Загрузить Minecraft для другого устройства».)
3. Дождитесь окончания загрузки (я на минутку отвернулся к окну, и загрузка уже завершилась), а затем откройте файл. Когда на экране появится окно, перетащите иконку Minecraft в папку *Applications,* как показано на рис. 1.10.

Applications — приложения

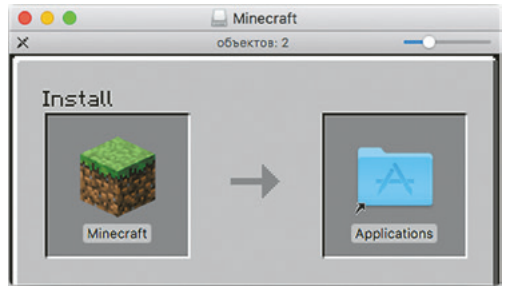

Рис. 1.10. Перетащите иконку Minecraft в папку Applications, чтобы установить игру

Ну вот, теперь Minecraft установлен. Знаете, чем стоит заняться теперь? Разумеется, поиграть! Для этого нужно выполнить лишь несколько действий:

Finder — проводник

Dock — панель быстрого запуска

1. Кликните по иконке файлового менеджера Finder в панели Dock.
2. В боковой панели Finder кликните **Программы (Applications)**.
3. Найдите в папке *Программы* (*Applications*) иконку Minecraft (рис. 1.11). Сделайте по нему двойной клик и выберите **Открыть (Open)**.

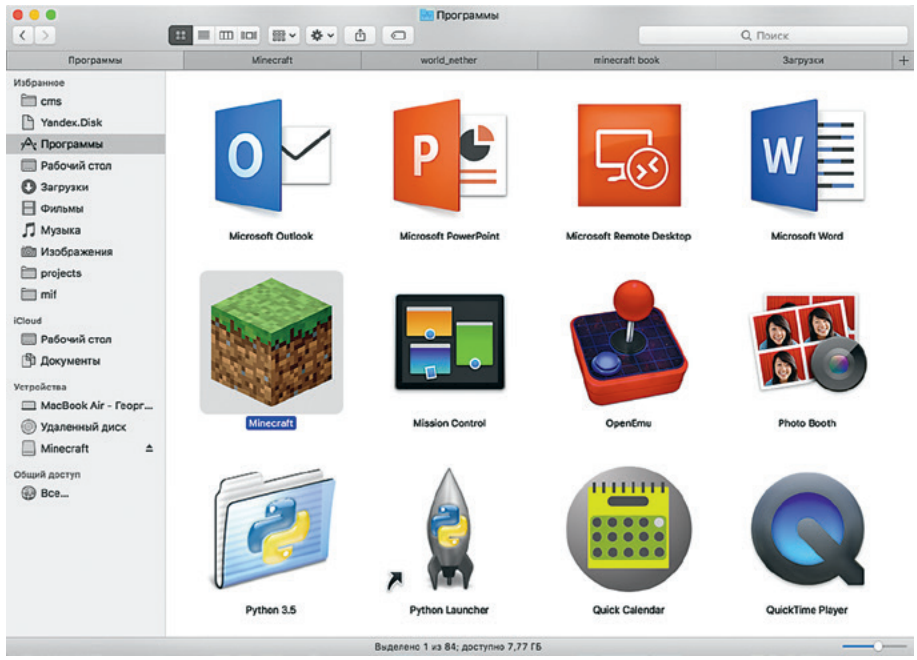

Рис. 1.11. Найдите иконку Minecraft в папке Программы (Applications)

4. Возможно, система задаст вам вопрос, хотите ли вы запустить Minecraft, поскольку этот файл был скачан из интернета. Кликните **Открыть (Open)**.

5. Minecraft запустится и, возможно, установит обновления.

6. Появится окно входа в игру. Введите имя пользователя и пароль, которые вы использовали при покупке Minecraft, и нажмите **Log in**.

Log in — войти

Play — играть

7. Нажмите **Play**. Перед запуском игры Minecraft загрузит еще несколько обновлений.

8. И наконец, кликните **Одиночная игра (Single Player) → Создать новый мир (Create New World)**. Введите название вашего мира и кликните **Создать новый мир (Create New World)**.

Повеселитесь! Если прежде вы не играли в Minecraft, поиграйте, пока в мире Minecraft не стемнеет. Берегитесь монстров! Имейте в виду, что программировать на Python мы будем в многопользовательском режиме игры, который отличается от однопользовательского. Мы поговорим об этом в разделе «Запуск Spigot и создание профиля игры» на с. 39.

А теперь за работу! Пора установить Python. Находясь в игре, нажмите клавишу ESC, чтобы появился курсор. В открывшемся меню кликните **Сохранить и выйти (Save and Quit to Title) → Выйти из игры (Quit Game)** и закройте Minecraft.

Установка Python

Python — это язык программирования, который вы будете изучать с помощью этой книги. Давайте установим программное обеспечение Python прямо сейчас.

1. Перейдите по ссылке *https://www.python.org/downloads/mac-osx/*.

2. Кликните по ссылке *Latest Python 3 Release — Python 3.5.2*. (На момент написания этой книги версия 3.5.2 была последней, однако, если вы обнаружите на сайте более позднюю версию, установите ее.) Начнется загрузка файла.

3. Когда загрузка программы-установщика завершится, кликните по ее иконке.

4. Когда откроется окно установщика, три раза кликните **Продолжить (Continue)**. Установщик спросит, согласны ли вы с лицензионным соглашением. Кликните **Принимаю (Agree)**.

5. Кликните **Установить (Install)** и дождитесь окончания установки Python. Возможно, при этом потребуется ввести пароль администратора. Не бойтесь, установка Python безопасна! За время установки программы я успел посмотреть прогноз погоды.

6. Кликните **Закрыть (Close)**. Python установлен!

Установка Java

Теперь, когда и Minecraft, и Python установлены, нужно сделать так, чтобы они могли друг с другом взаимодействовать. Для этого мы воспользуемся программой под названием Spigot. Однако для ее работы требуется, чтобы на компьютере был установлен комплект разработчика приложений на языке Java — Java Development Kit (JDK). Давайте сделаем это прямо сейчас.

Java download — скачать Java

Accept license agreement — принять лицензионное соглашение

Install — установить

1. Перейдите по ссылке *http://www.oracle.com/technetwork/java/javase/downloads/index.html* и кликните **Java Download**.

2. Выберите **Accept License Agreement**, а затем кликните по файлу для Mac OS X, который указан в списке.

3. После того как установщик скачается, запустите его.

4. Когда откроется окно установщика, сделайте двойной клик по иконке **Install**.

5. Появится запрос вашего пароля — введите его.

6. Дождитесь окончания установки Java, затем кликните **Закрыть (Close)**.

Теперь давайте убедимся, что установка JDK прошла успешно.

1. Откройте **Программы (Applications) → Системное меню → Системные настройки (System Preferences)**.

2. Среди прочих настроек вы должны увидеть иконку Java (рис. 1.12).

Итак, Java установлена и готова к запуску Minecraft-сервера Spigot. Приступим к его установке!

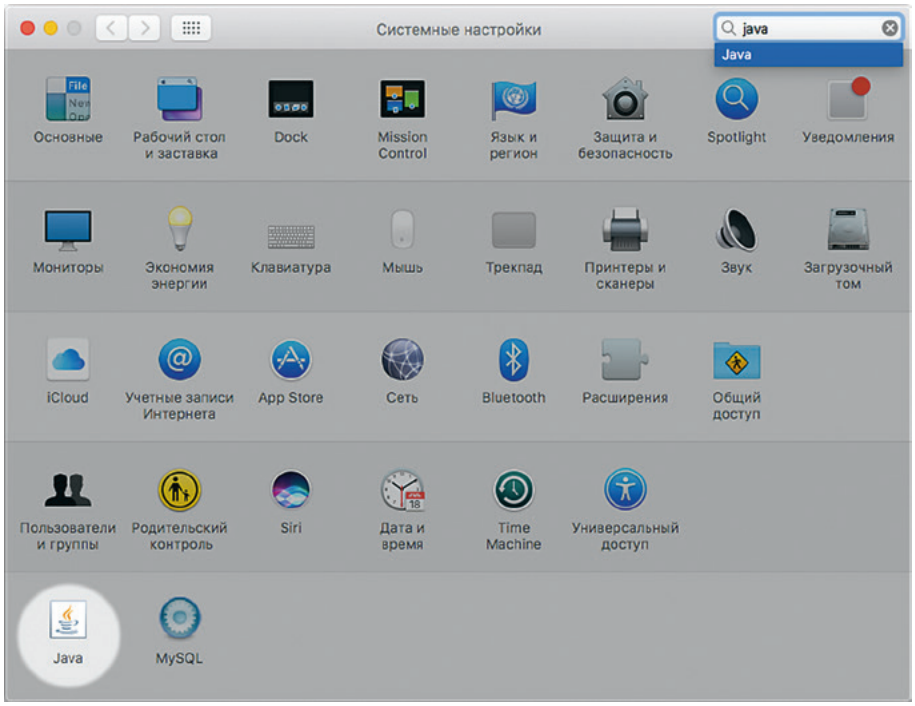

Рис. 1.12. Java установлена

Установка Minecraft Python API и Spigot

Теперь вам нужно установить на компьютер Minecraft Python API и Minecraft-сервер Spigot.

API — это способ взаимодействия одних программ с другими. В нашем случае Minecraft Python API позволяет игре Minecraft понимать команды, написанные на языке Python. API преобразует ваш код в объекты и действия, чтобы в игре, например, появился новый блок или переместился игрок. Правда, взаимодействовать программы смогут, только когда вы подключитесь к Minecraft-серверу.

Обычно сервер соединяет компьютеры, чтобы несколько онлайн-пользователей могли играть в одном мире Minecraft. В этой же книге речь пойдет о персональном Minecraft-сервере под названием Spigot, который вы установите на свой компьютер.

Итак, теперь вы знаете, для чего нужны Minecraft-сервер и API, осталось лишь установить их на компьютер. Я уже подготовил все необходимые программы и поместил их в один архив, чтобы вы могли быстро скачать их и установить. Для этого сделайте следующее:

Minecraft tools — средства для работы с Minecraft

1. Перейдите по ссылке *http://mif.to/minecraft* и скачайте файл *MinecraftToolsMac.zip*.
2. Когда файл скачается, откройте папку *Загрузки* (*Downloads*) и кликните **Открыть в Finder**.
3. В файловом менеджере Finder кликните по скачанному файлу *MinecraftToolsMac.zip*, удерживая CTRL, и выберите **Скопировать (Copy)**.
4. Перейдите в папку *Документы* (*Documents*). Удерживая CTRL, кликните в любом месте внутри папки и выберите **Новая папка (New Folder).** Назовите новую папку *MinecraftPython* (пробелов в названии папки быть не должно).
5. Перейдите в папку *MinecraftPython*. Удерживая CTRL, кликните в любом месте внутри папки и выберите **Вставить объект (Paste Item).** В результате сюда будет скопирован файл *MinecraftToolsMac.zip*.
6. Удерживая CTRL, кликните по zip-файлу и выберите **Открыть в программе (Open With) → Утилита архивирования (Archive Utility).** Появится новая папка *MinecraftTools*.
7. Откройте папку *MinecraftTools*. Ее содержимое показано на рис. 1.13.

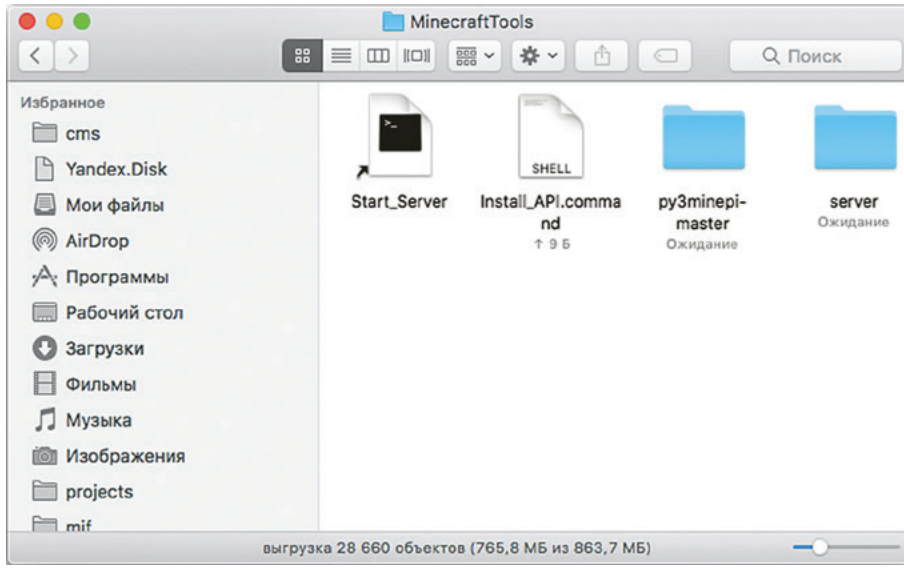

Рис. 1.13. Содержимое папки MinecraftTools

8. Удерживая CTRL, кликните по файлу *Install_API.command* и выберите **Открыть (Open).** Откроется новое окно. Введите в него свой пароль для установки Minecraft Python API.

*Если вы получите сообщение об ошибке, гласящее, что файл Install_API.command нельзя открыть, поскольку его автор неизвестен, кликните **Системные настройки (System Preferences)**, а затем **Защита и безопасность (Security and Privacy)**. Вы увидите сообщение «Программа Install_API.command заблокирована и не может быть открыта, так как автор программы не является установленным разработчиком» (Install_API.command was not opened because it is from an unidentified developer). Кликните **Подтвердить вход (Open Anyway)**. После этого файл должен запуститься.*

9. Когда установка завершится, закройте окно (если оно не закрылось самостоятельно).

Теперь Minecraft Python API и Minecraft-сервер установлены. Осталось запустить сервер, чем мы сейчас и займемся.

Запуск Spigot и создание профиля игры

При первом же запуске Spigot создаст мир Minecraft, однако перед тем, как вы начнете играть, нужно настроить профиль игры — для того чтобы проверить, совместимы ли версии Spigot и Minecraft.

Чтобы запустить Spigot, выполните следующее:

1. Перейдите в папку *MinecraftPython*, а оттуда — в папку *MinecraftTools*.

2. В папке *MinecraftTools* кликните по файлу *Start_Server,* удерживая CTRL, и выберите **Открыть (Open).** Если появится сообщение об ошибке, откройте **Системные настройки (System Preferences),** перейдите в раздел **Защита и безопасность (Security and Privacy)** и кликните **Подтвердить вход (Open Anyway).**

Start server — запустить сервер

3. Когда установка завершится, перемотайте текст в окне вверх, к самому началу. Там, в районе третьей или четвертой строки, должно быть сообщение вида `Starting minecraft server version x.x.x`. Вместо «x» должны стоять конкретные числа — например, на рис. 1.14 это 1.9.

Starting minecraft server version x. x. x. — запуск Minecraft-сервера версии x.x.x.

4. Запишите эти числа (номер версии сервера). Окно оставьте открытым.

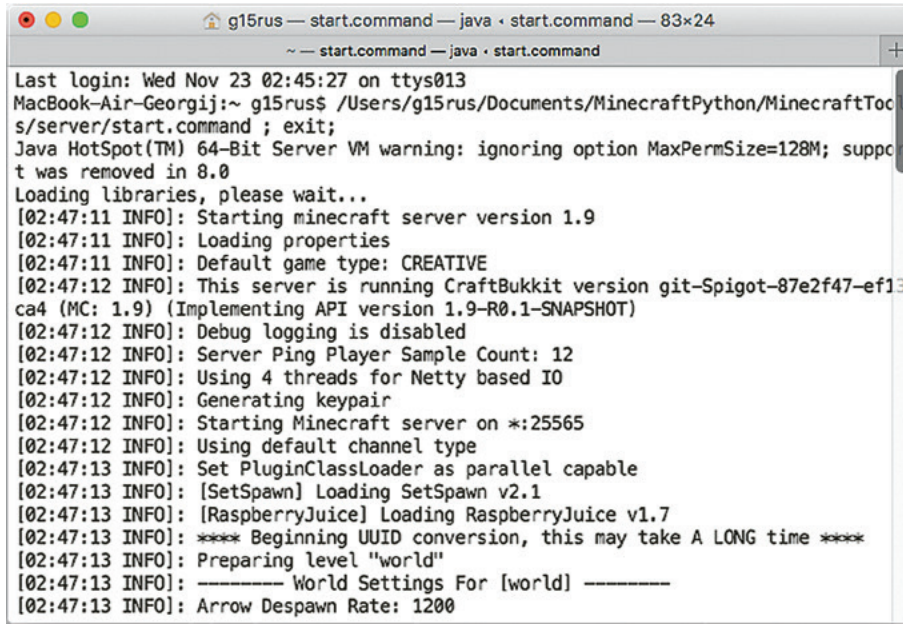

Рис. 1.14. Версия сервера — 1.9

Выбор подходящей версии Minecraft

Теперь, зная версию сервера, можно настроить профиль игры.

Лаунчер — программа для запуска игры. *Прим. науч. ред.*

1. Откройте лаунчер Minecraft, кликнув два раза по иконке игры, но пока не нажимайте кнопку **Играть.**

2. В правом верхнем углу кликните по иконке с тремя горизонтальными линиями. Откроется меню. Выберите пункт «Параметры запуска».

3. В поле «Название» введите `Программируем с Minecraft`.

4. Кликните по выпадающему списку в поле «Версия» и выберите номер версии вашего сервера. На рис. 1.15 показано, что я выбираю версию 1.9.4.

5. Кликните по кнопке **Сохранить**. Ваш профиль настроен!

Теперь каждый раз, когда вам понадобится запустить Minecraft для работы с Python, выбирайте в выпадающем списке лаунчера Minecraft рядом с кнопкой **Играть** профиль **Программируем с Minecraft**. Вы в любой момент сможете переключиться на последнюю версию игры, выбрав в выпадающем меню профиль с последним выпуском.

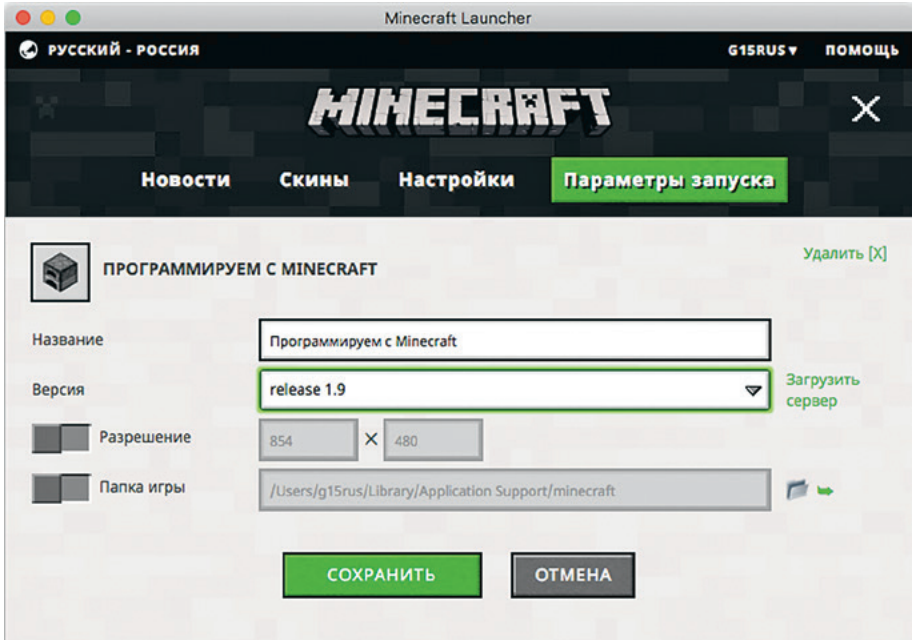

Рис. 1.15. Я создал профиль под названием «Программируем с Minecraft», который использует сервер версии 1.9

Создание мира

Профиль настроен, пора войти в новый мир Minecraft.

1. Запустите Minecraft и кликните **Multiplayer**.
2. Кликните по кнопке **Add Server**.
3. В строке ввода Server Name назовите ваш сервер `Minecraft Python World`, а в строке Server Address введите `localhost`, как показано на рис. 1.16. Затем кликните **Done**.
4. Сделайте двойной клик по **Minecraft Python World** — откроется мир, созданный сервером Spigot.

Multiplayer — сетевая игра

Add server — добавить сервер

Server name — название сервера

Server address — адрес сервера

Localhost — рабочая станция

Done — готово

Давайте посмотрим, что за мир появился на сервере Spigot. Этот мир работает в *творческом режиме*, поэтому там ваш игрок может летать. Два раза быстро нажмите на клавишу ПРОБЕЛ, чтобы поднять его в воздух. Нажимая и удерживая ПРОБЕЛ, вы будете поднимать игрока, а нажимая и удерживая SHIFT — опускать. Когда вам наскучат полеты, вновь нажмите ПРОБЕЛ дважды.

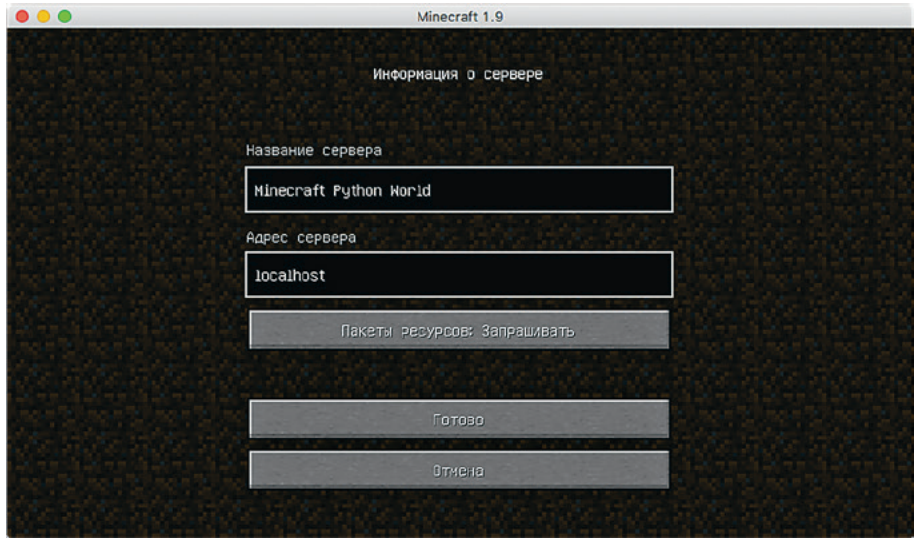

Рис. 1.16. Добавьте сервер, чтобы в будущем к нему было легко подключиться

Создание нового мира

Создание еще одного, нетронутого мира Minecraft на сервере немного отличается от создания нового мира в однопользовательском режиме. Если вы хотите создать другой мир, для начала вам нужно отключиться от сервера.

Disconnect — отсоединиться

1. Находясь в игре, нажмите ESC и затем кликните **Disconnect**, чтобы вернуться к списку серверов.

2. Остановите сервер Spigot, закрыв окно терминала, в котором он запущен.

Чтобы создать новый мир, сделайте копию сервера. Для этого выполните следующее:

Copy — копировать

Paste — вставить

Rename — переименовать

New world — новый мир

1. Перейдите в папку *MinecraftPython*. Удерживая CTRL, кликните по папке *MinecraftTools* и выберите **Copy**.

2. Удерживая CTRL, кликните в любом месте внутри папки *MinecraftPython* и выберите **Paste**. В результате появится копия папки *MinecraftTools* с именем *MinecraftTools copy*.

3. Удерживая CTRL, кликните по только что созданной папке-копии и выберите **Rename**. Я назвал новую папку *New World*, но вы можете дать ей любое имя, какое пожелаете.

4. Откройте папку *New World* (или как вы ее назвали), а затем откройте в ней папку *server*.

5. В папке *server* выделите папки *world*, *world_nether* и *world_the_end*. Нажмите SHIFT + DELETE, чтобы их удалить.

6. Вернитесь в папку *New World* и кликните по файлу *Start_Server*. В результате запустится сервер, который и создаст новый мир.

7. Теперь, запустив Minecraft и открыв Minecraft Python World, вы увидите только что созданный мир.

Вы можете создать сколько угодно новых миров — просто повторите эти действия. А если захотите открыть предыдущий мир, кликните по файлу *Start_Server* в папке *MinecraftTools*, а не в папке *New World*.

Чтобы удалить мир, заменив его на новый, просто удалите папки *world*, *world_nether* и *world_the_end* в папке того мира, который вам нужно заменить.

Игра без доступа к интернету

Если у вас нет доступа к интернету, при попытке подключиться к серверу из игры программа выдаст ошибку. Этого можно избежать, изменив настройки сервера. Сначала закройте окно сервера, если он запущен. Затем откройте папку *MinecraftPython*, а в ней папку *MinecraftTools* и, наконец, папку *server*. Откройте в текстовом редакторе (например, TextEdit) файл *server.properties* и измените значение опции `online-mode` (см. рис. 1.8 на с. 31) с `true` на `false`. Сохраните изменения, вернитесь в папку *MinecraftTools* и снова запустите сервер, кликнув по файлу *Start_Server*. Теперь вы сможете играть в Minecraft без доступа к интернету.

Text edit — текстовый редактор

Properties — характеристики

Переключение в режим выживания

Я задал творческий режим в качестве режима сервера по умолчанию. Так проще писать и запускать Python-программы, поскольку ваш игрок не будет терять здоровье и не пострадает от голода и нападений монстров.

Возможно, вы захотите протестировать некоторые программы в *режиме выживания* — просто ради интереса. В этом случае вам нужно будет переключать сервер из творческого режима в режим выживания и обратно. Это довольно просто.

Чтобы переключить сервер из творческого режима в режим выживания, выполните следующие шаги:

1. Откройте папку *MinecraftTools*, а внутри нее папку *server*.

2. Найдите файл *server.properties* и откройте его в текстовом редакторе (например, в TextEdit).

3. Найдите в этом файле строку `gamemode=1` и измените ее на `gamemode=0` (см. рис. 1.9 на с. 32).

4. Сохраните файл и закройте текстовый редактор.

5. Запустите сервер, кликнув по файлу *Start_Server* в папке *MinecraftTools*. Теперь, когда вы войдете в Minecraft Python World, игра будет работать в режиме выживания.

Чтобы переключиться обратно в творческий режим, достаточно выполнить те же самые шаги, но на шаге 3 заменить в файле *server.properties* `gamemode=0` на `gamemode=1`.

Итак, все необходимые программы настроены! Далее вам предстоит познакомиться со средой разработки IDLE, в которой вы будете писать свои коды. Переходите к разделу «Знакомство с IDLE» на с. 45.

Установка и настройка программ для Raspberry Pi

Start x — запустить графический режим

Включите Raspberry Pi, введите логин и пароль и запустите графическую среду командой `startx`. (Если вы используете последнюю версию операционной системы, эту команду вводить не нужно.)

На вашем Raspberry Pi может быть установлено две или три разные версии Python. Для работы с этой книгой вам понадобится последняя версия — Python 3.

По умолчанию на компьютерах Raspberry Pi уже установлена упрощенная версия Minecraft, которая называется Minecraft: Pi Edition. Все программы, которые нужны для управления миром Minecraft с помощью Python, также уже установлены. Если вы только начинаете знакомство с микрокомпьютером, стоит изучить руководство по основам работы с ним, которое выложено на официальном сайте *http://www.raspberrypi.org*.

Если вы используете устаревший образ системы на SD-карте (созданный раньше августа 2014 года), там, возможно, отсутствует Minecraft. Однако игру несложно установить. В первую очередь нужно подключить ваш Raspberry Pi к интернету. Руководство по настройке подключения можно найти на сайте *http://www.raspberrypi.org*.

Подключившись к интернету, выполните следующие шаги:

1. На рабочем столе сделайте двойной клик по иконке **LXTerminal**.

2. Когда LXTerminal запустится, введите в его окне следующую команду:

```
$ sudo apt-get update
```

3. Когда обновление завершится, введите такую команду:

```
$ sudo apt-get install minecraft-pi
```

4. Дождитесь окончания установки Minecraft.

Версия Minecraft для Raspberry Pi имеет некоторые ограничения по сравнению с версией для настольных компьютеров. Игровой мир там гораздо меньше, отсутствуют многие блоки и другие компоненты игры (например, режим выживания), однако с помощью этой книги вы все равно сможете писать занимательные программы и запускать их на вашем устройстве.

Перед тем как двигаться дальше, давайте создадим папку для хранения ваших будущих Python-программ. Кликните по иконке файлового менеджера на панели задач. Откройте папку *Documents*, кликните правой кнопкой мыши по пустому месту внутри нее и выберите **Create New... → Folder**. Назовите папку *Minecraft Python* и кликните **OK**.

Documents — документы

Create new — создать новую

Folder — папка

Если у вас первая модель Raspberry Pi, вы увидите, что из-за малой мощности устройства некоторые программы, описанные в этой книге, работают медленно. У Raspberry Pi 2 проблем со скоростью меньше.

Чтобы запустить Minecraft, кликните по кнопке меню в левом верхнем углу рабочего стола. (Если у вас старая версия операционной системы, кнопка будет в левом нижнем углу.) Зайдите в раздел **Games** и выберите **Minecraft**. Игра запустится. При первом запуске нужно будет кликнуть **Create World**.

Games — игры

Create world — создать мир

Лучше не менять размер окна Minecraft — из-за этого могут возникнуть проблемы.

Порой другие окна (например, диалог с вопросом, хотите ли вы сохранить Python-программу) будут отображаться позади окна Minecraft. В таких случаях сворачивайте окно Minecraft, чтобы получить доступ к этим окнам. Если после установки игры не все работает гладко, попробуйте перезагрузить Raspberry Pi.

Знакомство с IDLE

Теперь, когда вы установили и настроили все необходимые программы, давайте познакомимся с IDLE — средой разработки для языка Python, в которой вам предстоит писать и запускать программы. IDLE входит в состав дистрибутива Python, поэтому дополнительно ничего устанавливать не нужно. Итак, давайте запустим IDLE прямо сейчас.

IDLE (читается «айдл»; *сокр.* от Integrated DeveLopment Environment) — интегрированная среда разработки

Windows Откройте меню **Пуск (Start)** и введите в строке поиска `IDLE`.

Mac OS Откройте папку *Applications*, а в ней — папку *Python 3.5* и кликните по иконке IDLE.

Raspberry Pi Сделайте двойной клик по иконке IDLE с изображением логотипа Python 3.

Откроется окно IDLE (рис. 1.17). Это окно называется *окном консоли Python*. Окно консоли настолько удобно, что в свое время я, обучаясь программированию на Python, был просто поражен!

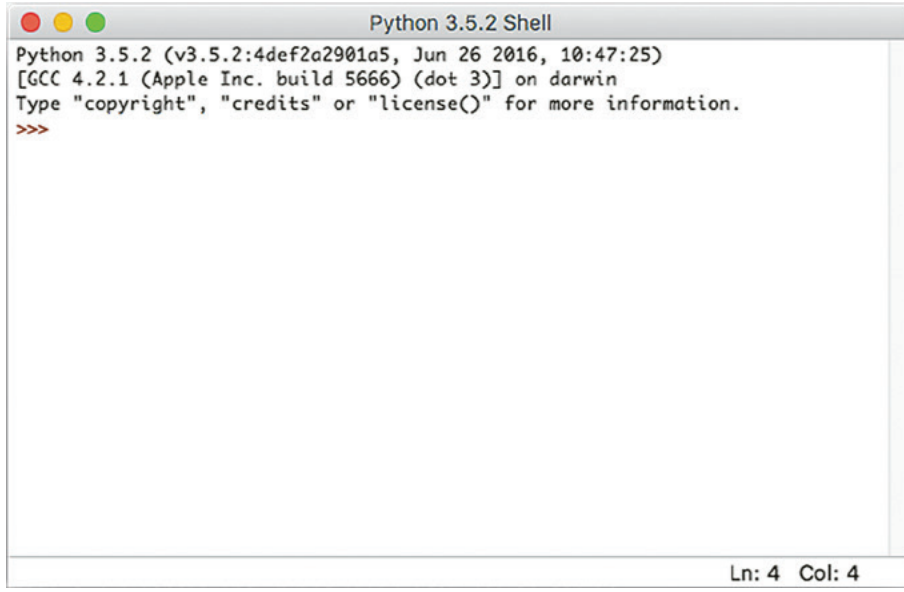

Рис. 1.17. Окно консоли Python служит для создания Python-программ

Знакомство с окном консоли Python

Окно консоли Python позволяет выполнять написанные нами программы построчно. Вы можете написать строку кода, тут же запустить ее, посмотреть на результат, а затем написать следующую строку. Тестировать работу кода таким образом очень удобно.

В окне консоли обратите внимание на три знака «больше» (>>>) в начале строки — это приглашение к вводу команды. Приглашение к вводу означает, что окно консоли готово к считыванию команд. Давайте начнем с элементарного действия — сложения двух чисел.

Кликните мышкой по окну консоли возле приглашения и введите `2 + 2`. Обратите внимание: вам не нужно вводить само приглашение (>>>). Строка при этом должна выглядеть так:

```
>>> 2 + 2
```

После ввода этой команды нажмите ENTER. Окно консоли Python выведет результат. В данном случае это будет 4:

```
>>> 2 + 2
4
```

В окне консоли можно работать и с фрагментами текста. Введите после приглашения следующий код и нажмите ENTER:

```
>>> "Б" + "у" * 5
'Бууууу'
```

Как видите, этот код печатает «Бууууу». Число в конце команды определяет, сколько букв «у» будет в слове. Меняя это число, можно менять длину слова. Попробуйте значение 20 (или любое другое число):

```
>>> "Б" + "у" * 20
'Буууууууууууууууууууу'
```

Буууууууууууууууууууу! С окном консоли Python можно повеселиться на славу!

Обратите внимание, что IDLE раскрашивает код разными цветами. Это называется *подсветкой синтаксиса*. Подсветка синтаксиса помогает отличить один элемент кода от другого. В этой книге показаны те цвета, которые отображаются в IDLE. Когда вы будете вбивать эти примеры у себя на компьютере, цвета команд должны совпасть.

А теперь познакомимся с окном программы IDLE.

Поприветствуйте окно программы

К сожалению, для создания больших программ окно консоли не подойдет. Зато для этого можно использовать *окно программы IDLE*! В отличие от окна консоли, здесь код не выполняется сразу же после ввода. Зато окно программы может по вашей команде запустить программу целиком.

File — файл

New file — новый файл

Находясь в IDLE, кликните по меню **File** и выберите **New File**. Откроется окно, как на рис. 1.18. Это и есть окно программы.

Я так и слышу ваш возглас: «Эй! Это окно программы выглядит точь-в-точь как окно консоли Python!» Что ж, так оно и есть, за исключением

одной принципиальной особенности: в этом окне строки не начинаются с приглашения к вводу команды (>>>).

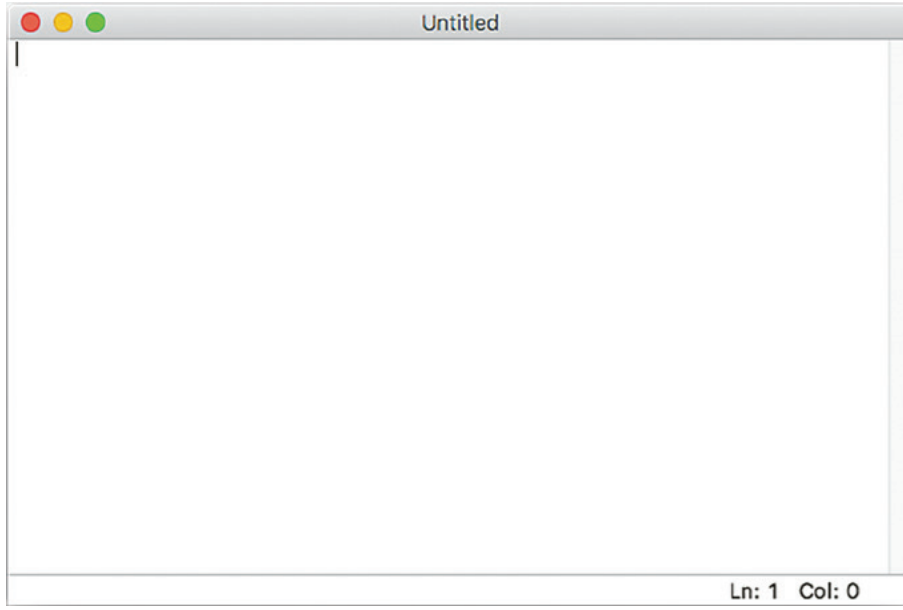

Рис. 1.18. Окно программы IDLE

Разберемся, что это значит. Введите в первой строке окна программы следующий код и нажмите ENTER:

```
print(2 + 2)
```

Ждете, когда что-нибудь произойдет? Как бы не так — в окне программы нажатие ENTER не запускает код, а просто добавляет новую строку. Благодаря этому вы можете ввести сколько угодно строк, перед тем как запустить программу. Давайте добавим еще несколько. Вот как это должно выглядеть на экране:

```
print(2 + 2)
print("Б" + "у" * 20)
print("PYTHON!")
print("<3")
print("Minecraft")
```

Save as — сохранить как

Перед тем как запустить код, введенный в окно программы IDLE, его нужно сохранить — для этого кликните **File** и выберите **Save As**. Создайте

папку внутри папки *Minecraft Python* и назовите ее *Setting Up*. Сохраните код в этой папке, назвав файл *pythonLovesMinecraft.py*.

Теперь давайте запустим этот код. Зайдите в меню **Run** и кликните **Run Module**. Откроется окно консоли, и в нем запустится ваша программа. На рис. 1.19 показан результат ее работы.

В отличие от окна консоли, в окне программы результаты работы команд не выводятся автоматически. Однако их можно получить при помощи команды `print()` — как в нашем примере. Не волнуйтесь, если пока вам не все понятно, — в дальнейшем мы разберем это подробнее.

Setting up — установка

Python loves Minecraft — Python любит Minecraft

Run module — запустить модуль

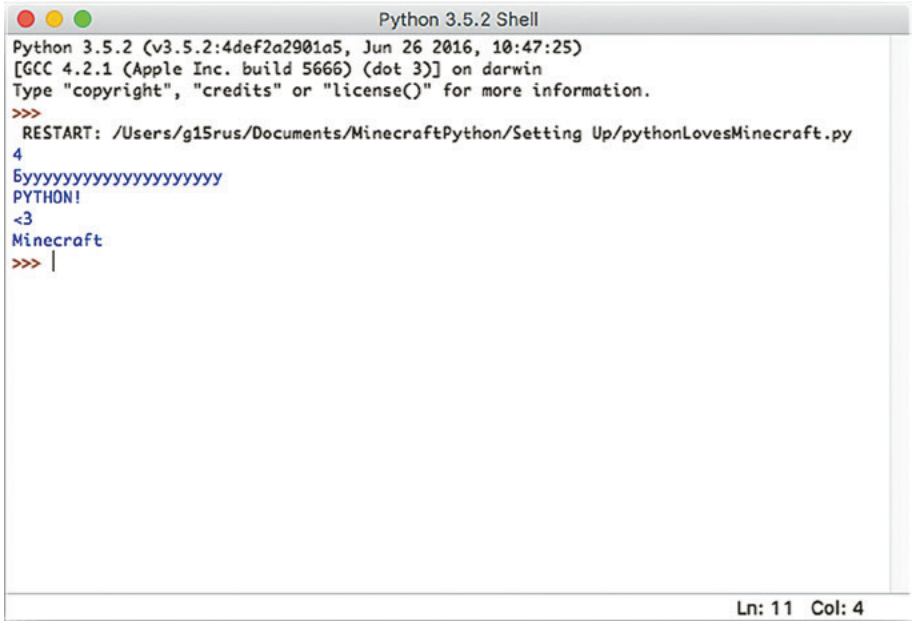

Рис. 1.19. Результат работы Python-программы

Каждый раз при запуске кода из окна программы IDLE будет открываться окно консоли Python. Так происходит всегда, несмотря на то что программа написана в другом окне.

Когда нужно использовать окно консоли, а когда окно программы

Теперь, когда вы знаете, чем отличается окно консоли Python от окна программы IDLE, вам, наверное, любопытно, когда лучше использовать одно, а когда другое. Как правило, я открываю окно консоли, если мне нужно проверить несколько строк кода, но они не потребуются

в дальнейшем. Советую вам тестировать короткие примеры из этой книги именно в окне консоли.

Окно программы я использую для написания программ, состоящих из большого количества строк, или для кода, который собираюсь использовать в других программах. Все задания (миссии), описанные в этой книге, мы будем выполнять в окне программы IDLE, чтобы сохранять состояние игры. Однако вы всегда можете воспользоваться окном консоли, если вам нужно быстренько что-нибудь проверить.

Подсказки

В этой книге перед всеми фрагментами кода, предназначенными для запуска в окне консоли Python, стоит знак приглашения к вводу (>>>):

```
>>> print("Буууууу Minecraft")
```

Советую вводить такой код в IDLE построчно, чтобы лучше освоиться и с кодом, и с IDLE. Результат выполнения команды появится на следующей строке:

```
>>> print("Буууууу Minecraft")
Буууууу Minecraft
```

Код, предназначенный для набора в окне программы, будет показан без значка приглашения, вот так:

```
print("Приключения")
```

Результат выполнения этого кода не появится на экране вашего компьютера автоматически. Чтобы показать, что должно получиться при запуске программы, я буду либо описывать это словами, либо показывать результат в отдельной табличке. Например, так: если запустить код из последнего примера, на экране появится:

```
Приключения
```

Чтобы вам было легче понять работу программы, я отмечаю важные строки кода черными кружками с белыми цифрами. Каждый раз, увидев такую метку, знайте, что ей соответствует объяснение в тексте, и наоборот. Метки выглядят так:

❶ ❷ ❸ ❹ ❺ ❻

Проверяем работу Minecraft и Python

Давайте удостоверимся, что все программы установлены правильно. Для этого воспользуемся очень простой Python-программкой, которая взаимодействует с Minecraft.

Если у вас операционная система Windows или Mac OS X, сделайте следующее:

1. Запустите Spigot. Для этого перейдите в папку *Minecraft Tools* и кликните *Start_Server*.
2. Запустите Minecraft и подключитесь к серверу Spigot, выбрав **Minecraft Python World** из меню многопользовательского режима.
3. Нажмите ESC, чтобы вернуть на экран курсор мыши, и запустите IDLE, чтобы открыть окно консоли Python.

При создании программ, взаимодействующих с Minecraft, эти три приложения должны быть всегда открыты.

Если у вас Raspberry Pi, просто запустите IDLE и Minecraft.

После этого введите в окне консоли следующую строку:

```
>>> from mcpi.minecraft import Minecraft
```

Обратите внимание: заглавные буквы должны оставаться заглавными, строчные — строчными.

Нажмите ENTER и введите следующую строку:

```
>>> mc = Minecraft.create()
```

Если на этом этапе вы увидите сообщение об ошибке примерно как на рис. 1.20, значит, что-то пошло не так.

Поочередно проверьте: запущен ли Minecraft? А Spigot? Находитесь ли вы в многопользовательском режиме? Подходящая ли у вас версия Python (3, а не 2)? Если ошибка возникает сразу после ввода первой строки, значит, Minecraft Python API установлен некорректно. Вернитесь к пошаговой инструкции по его установке и выполните ее снова. Если ошибка появляется после ввода второй строки, возможно, у вас проблемы с установкой Java или Spigot. Попробуйте переустановить эти программные пакеты — сначала один, потом второй.

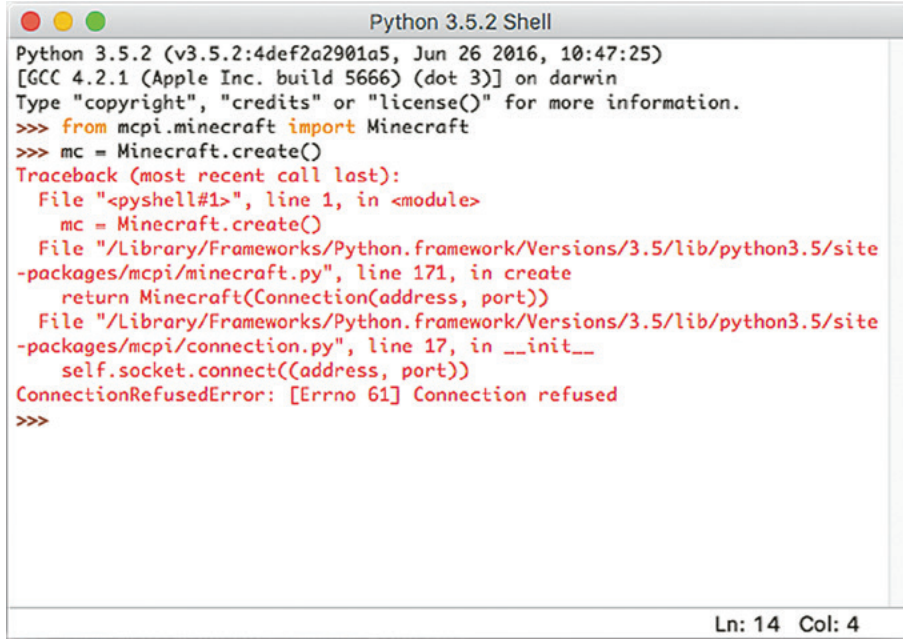

Рис. 1.20. Сообщение об ошибке говорит о том, что я не запустил Spigot

Одна из наиболее частых ошибок — для запуска сервера используется старая версия Java. Убедитесь, что вы установили последнюю версию при помощи команды `java -version`, как описано в разделе «Установка Java» на с. 23 (для Windows) или на с. 36 (для Mac OS X).

Import error: no module named 'mcpi' — ошибка импорта: отсутствует модуль с именем mcpi

Если появится сообщение об ошибке `ImportError: No module named 'mcpi'`, возможно, вы используете старую версию Python. Убедитесь, что на вашем компьютере установлена последняя версия!

Если же сообщения об ошибке нет, введите в окне консоли такую строку:

```
mc.player.setTilePos(0, 120, 0)
```

После этого игрок должен взлететь высоко в небо! Данный код телепортирует игрока в новое место. Подробнее об этом вы узнаете из главы 2. Переворачивайте скорее страницу!

2

ТЕЛЕПОРТАЦИЯ С ПОМОЩЬЮ ПЕРЕМЕННЫХ

Ну что, готовы овладеть силой языка Python, чтобы править миром Minecraft? В этой главе мы пробежимся по основам Python, а затем вы сможете применить свои знания и отправить игрока в телепортационный тур!

Описанные в этой главе понятия и приемы относятся не столько к программированию в игре Minecraft, сколько к программированию на языке Python в целом. С помощью полученных знаний вы сможете создавать любые другие Python-программы!

Что такое программа?

Программа — это набор инструкций, следуя которым компьютер выполняет определенную задачу или несколько задач. Вспомните приложение «Секундомер» на мобильном телефоне. Эта программа содержит инструкции, которые описывают, что должно произойти при нажатии кнопок «старт» и «стоп», а также как будет отображаться на экране счетчик времени. Все эти инструкции появились не сами по себе — их написал программист.

Люди по всему миру каждый день пользуются миллионами разных программ. Эсэмэски с телефона отправляет программа, светофором управляет программа, любая компьютерная игра, например Minecraft, — тоже программа.

Прочитав эту книгу, вы научитесь писать коды (те самые инструкции) на одном из языков программирования (Python) и с их помощью создавать собственный мир Minecraft.

Хранение информации в переменных

Начнем знакомство с языком Python с переменных. *Переменные* нужны для хранения данных, которые позже будут использованы в программе. *Данные* — это любая полезная информация: числа, имена, текст, списки значений и т. д. К примеру, в переменной `pickaxes` может храниться число 12:

Pickaxes — кирки

```
>>> pickaxes = 12
```

Хотя Python и позволяет давать переменным русскоязычные имена, делать так в среде программистов не принято. Используйте в названиях переменных латинские буквы и другие символы. Русскоязычный текст возможен лишь внутри строковых значений переменных (см. гл. 4) и в комментариях. Прим. науч. ред.

В переменных могут храниться даже предложения — такие как «Убирайся, крипер!». Значения переменных можно менять и тем самым проворачивать в Minecraft различные трюки. Уже скоро вы будете сами задавать значения переменным и отправлять игрока куда вам вздумается!

Чтобы создать переменную в Python, нужно ввести в окне консоли имя переменной, поставить знак «равно» (=), а после добавить значение этой переменной. Допустим, вы решили отправить игрока в путешествие по биомам. В этом случае ему понадобятся солидные запасы провизии. Назовем переменную `bread` и зададим ей значение 145:

Биом — природная зона в мире Minecraft

Bread — хлеб

```
>>> bread = 145
```

Имя переменной всегда пишется слева от знака «равно», а значение, которое в ней хранится, — справа (рис. 2.1). Эта строка кода означает, что мы *объявили* переменную `bread` и *присвоили* ей значение 145.

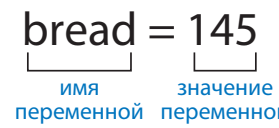

Рис. 2.1. Объявление переменной. Чтобы съесть 145 батонов хлеба, нужно быть очень голодным!

Объявив переменную и присвоив ей значение, вы можете проверить, что в ней хранится. Для этого введите в окне консоли имя переменной и нажмите ENTER:

```
>>> bread
145
```

Вы можете называть переменные как угодно, однако желательно, чтобы имя соответствовало назначению переменной. Так будет проще понять, что происходит в вашей программе. Имена переменных пишутся со строчной буквы. Это не правило языка, а соглашение, принятое Python-программистами. Ему нужно следовать, чтобы другим программистам было удобно читать ваш код.

Значения переменных хранятся во временной памяти компьютера, поэтому при его выключении или завершении программы эти данные исчезают. Попробуйте закрыть IDLE, а затем запустить программу снова. Что произойдет, если вы запросите значение переменной `bread`*?*

Как устроены языки программирования

У каждого языка программирования есть набор правил — *синтаксис*. Эти правила похожи на правила русского языка, которыми мы пользуемся, чтобы составлять предложения. Если вы будете знать синтаксис языка Python, ваши программы будут работать, а если нарушите их, компьютер просто не поймет, что вы от него хотите.

Каждая инструкция в вашем коде похожа на предложение. Только в русском языке конец предложения обозначается точкой, а в Python — переходом на новую строку. Строка с инструкцией называется *командой*.

Предположим, вы решили вести учет кирок (pickaxes), блоков железной руды (iron ore) и булыжников (cobblestone). Для этого можно ввести в окне консоли следующее:

```
>>> pickaxes = 12
>>> iron = 30
>>> cobblestone = 25
```

На рис. 2.2 показано, как это будет выглядеть.

Название блоков железной руды для простоты я сократил до одного слова. Обратите внимание, что каждый тип предметов записан отдельной строкой. Благодаря этому Python понимает, что вы хотите отслеживать количество трех разных предметов. Если вы запишете все команды в одну строку, Python запутается и сообщит о синтаксической ошибке:

```
>>> pickaxes = 12 iron = 30 cobblestone = 25
SyntaxError: invalid syntax
```

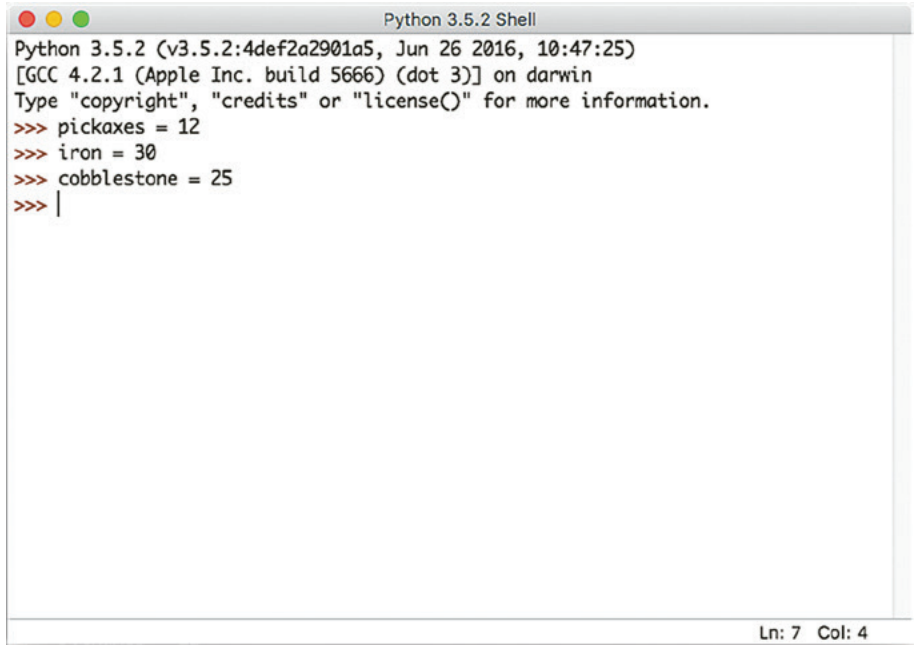

Рис. 2.2. Ввод команд в окне консоли Python

Синтаксическая ошибка (*SyntaxError*) означает, что Python не понимает ваших команд и не может их выполнить. В данном случае программе неясно, где начинается и где заканчивается каждая команда. Такая ошибка называется «неверный синтаксис» (invalid syntax).

Также Python не поймет, что нужно делать, если в начале строки будет стоять пробел:

```
>>>  iron = 30
SyntaxError: unexpected indent
```

Присмотритесь, и вы заметите, что строка начинается с отступа. Синтаксическая ошибка «некорректный отступ» (unexpected indent) означает, что в начале строки есть пробел или пробелы, которых там быть не должно.

Python очень требователен к тому, как вы пишете код. Если при вводе примеров из этой книги вы увидите сообщение о синтаксической ошибке, внимательно его проверьте — скорее всего, в код закралась ошибка.

Синтаксис для переменных

Имена, которые вы будете давать переменным, тоже должны подчиняться синтаксису языка Python. Имейте в виду:

- В имени не должно быть других символов, кроме латинских букв, цифр и нижнего подчеркивания (_).
- Не начинайте имя с цифры. `9bread` не подойдет. Однако в любом другом месте цифра стоять может — например, здесь: `bread9`.
- До и после знака «равно» лучше ставить пробел. Это не обязательно, но так принято делать в среде программистов, чтобы легче читать код.

Переменные очень полезны. Давайте разберемся, как менять их значения, и вы научитесь телепортировать игрока!

Изменение значений переменных

Вы можете в любой момент изменить значение переменной. Например, вы встретили в мире Minecraft пять кошек и хотите сохранить это число. Первым делом объявите переменную `cats` и присвойте ей значение 5. В окне консоли это будет выглядеть так:

Cats — кошки

```
>>> cats = 5
>>> cats
5
```

Вдруг вы встречаете еще пять кошек и хотите обновить содержимое переменной. Что будет, если вы присвоите уже существующей переменной значение 10?

```
>>> cats = 10
>>> cats
10
```

Запросив значение переменной, вы увидите на экране вовсе не цифру 5! И теперь, если вы введете переменную `cats` в какую-то команду, ей будет соответствовать число 10.

В переменных можно хранить данные самых разных типов. *Тип данных* говорит компьютеру о том, как именно нужно обрабатывать каждое конкретное значение. Сначала я расскажу о целых числах — этот тип данных встречается чаще всего. А позже познакомлю с дробными числами.

Целые числа

Целые числа бывают положительными и отрицательными. Числа 10, 32, –6, 194689 и –5 целые, а 3,4 и 6,025 — дробные.

Мы имеем дело с целыми числами каждый день, порой не задумываясь об этом. Встречаются они и в игре Minecraft. Направляясь в шахту, чтобы добыть 5 алмазов, игрок может захватить с собой 2 яблока и повстречать по пути 12 коров. Все эти числа — целые.

Предположим, в вашем Minecraft-мире есть 5 свиней и вы решили написать программу, которая каким-то образом будет использовать это число. Объявите целочисленную переменную `pigs` и присвойте ей значение, которое соответствует количеству свиней:

Pigs — свиньи

```
>>> pigs = 5
```

В переменных можно хранить и отрицательные числа. Например, чтобы обозначить температуру −5 градусов, можно объявить такую переменную:

```
>>> temperature = -5
```

А теперь настало время применить знания о переменных и целых числах, чтобы выполнить первую миссию!

МИССИЯ 1. ТЕЛЕПОРТАЦИЯ ИГРОКА

В ходе этой миссии вы научитесь использовать переменные и работать с целыми числами, создав программу телепортации игрока.

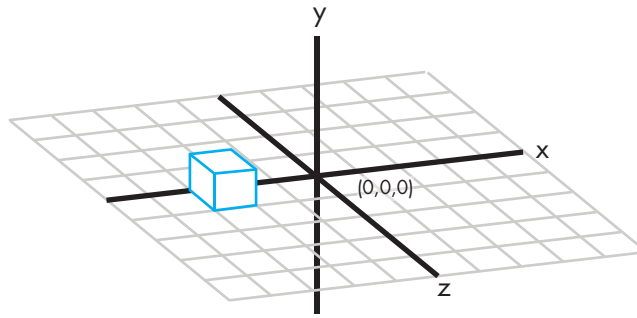

Рис. 2.3. Оси координат в трехмерном пространстве

На рис. 2.3 видно, что позиция игрока в мире Minecraft определяется тремя *координатами*: *x* («икс»), *y* («игрек») и *z* («зэт»). Координата *y* соответствует высоте, *x* и *z* — положению на горизонтальной плоскости.

Если вы используете версию Minecraft для Raspberry Pi, координаты игрока — это три цифры в левом верхнем углу экрана (рис. 2.4). Если же у вас версия Minecraft для Windows или Mac OS, нажмите во время игры клавишу F3, чтобы увидеть координаты — они будут во втором блоке текста слева, после букв XYZ (рис. 2.5).

Рис. 2.4. Координаты игрока в Minecraft для Raspberry Pi

Подвигайте вашего игрока туда-сюда и понаблюдайте, как меняются его координаты. Здорово, правда? Однако перемещения на большие расстояния занимают много времени. Зачем тратить его зря, если игрока можно телепортировать? Просто измените его координаты на любые другие с помощью Python-программы.

Рис. 2.5. Координаты игрока в Minecraft для настольных компьютеров

Выполните следующие шаги:

New window — новое окно

1. Откройте IDLE и кликните **File → New File** (для некоторых версий — **New Window**). Вы увидите пустое окно программы (рис. 2.6). Если у вас Raspberry Pi или на компьютере установлено несколько версий Python, убедитесь, что открылся именно Python 3, а не Python 2.7.

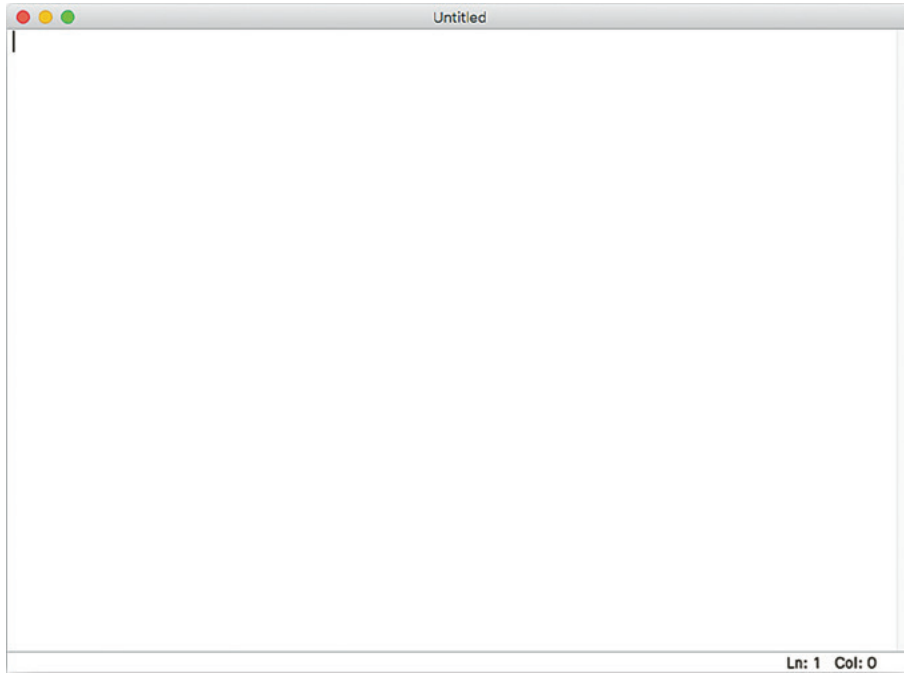

Рис. 2.6. Новое окно программы

2. Кликните **File → Save As**. Откроется диалоговое окно.
3. Зайдите в нем в папку *Minecraft Python*, которую вы создали, выполняя инструкции из главы 1, и добавьте в нее папку под названием *variables*.

Variables — переменные

Teleport — телепортация

4. Откройте папку *variables*, назовите сохраняемый файл *teleport.py* и кликните **Save**.

Теперь переключитесь на окно программы IDLE и введите туда следующие две строчки кода:

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
```

Эти команды нужны для подключения к игре Minecraft, и они должны быть в каждой Python-программе, которая с ней взаимодействует. Затем объявите три целочисленные переменные с именами `x`, `y` и `z`.

```
x = 10
y = 110
z = 12
```

Эти переменные соответствуют координатам, в которые вы собираетесь телепортировать игрока. Пока присвоим им значения 10, 110 и 12.

Далее введите следующую строку кода — она и будет телепортировать игрока:

```
mc.player.setTilePos(x, y, z)
```

Выражение `setTilePos()` — это *функция*, то есть фрагмент ранее созданного программистами кода. Функция `setTilePos(x, y, z)` дает игре команду изменить координаты игрока на значения, указанные в скобках, то есть на значения только что созданных вами переменных. Эти значения в скобках называются *аргументами*. Таким образом, с помощью аргументов вы передаете значения переменных `x`, `y` и `z` в функцию, чтобы после *вызова* она могла ими воспользоваться.

Set tile pos (pos — *сокр.* от position) — задать позицию блока

Если у вас версия игры для Raspberry Pi, не используйте в качестве координат `x` и `z` числа больше 127 и меньше −127. Мир Minecraft Pi невелик, и указание координат, находящихся за его пределами, приведет к сбою игры.

В листинге 2.1 дан полный код для телепортации игрока.

Листинг — текст компьютерной программы. *Прим. науч. ред.*

teleport.py

```
❶ # Подключаемся к Minecraft
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

# Присваиваем переменным x, y и z значения координат
x = 10
y = 110
z = 12

# Меняем позицию игрока
mc.player.setTilePos(x, y, z)
```

Листинг 2.1. Код для телепортации игрока

Чтобы в коде было проще разобраться, я добавил в него комментарии ❶. *Комментарии* — полезная конструкция языка программирования. Они позволяют пояснить, что делает каждая часть кода. Python, выполняя программу, эти пояснения игнорирует. Иными словами, когда вы запустите свой код, Python пропустит эти строки, будто их нет. Однострочный комментарий начинается с символа «решетка» (#).

В комментариях листинга 2.1 я описал, что происходит в каждой части программы *teleport.py*. Советую вам тоже завести привычку писать комментарии к своим программам, чтобы потом было проще вспомнить, как они устроены.

Рис. 2.7. Полный текст кода в окне программы IDLE

На рис. 2.7 показан полный текст программы в окне программы IDLE. Давайте же запустим эту программу! Сделайте следующее:

1. Откройте Minecraft, кликнув по его иконке на рабочем столе.

Start game — начать игру

Create a new world — создать новый мир

2. Если у вас Raspberry Pi, кликните **Start Game**, а затем **Create a New World**. Если же у вас версия Minecraft для настольных компьютеров, откройте игровой мир, как написано в разделе «Запуск Spigot и создание профиля игры» (на с. 26 для Windows и на с. 39 для Mac OS).

3. Когда мир будет создан, нажмите клавишу ESC (или TAB, если у вас Raspberry Pi), чтобы появился курсор мыши. Теперь вы можете выйти из игры, переместив курсор за пределы окна Minecraft, и вернуться в игру, сделав двойной клик по окну Minecraft. На рис. 2.8 показаны окно Minecraft и окно программы IDLE на экране моего компьютера.

Рис. 2.8. Окно Minecraft и окно IDLE на экране моего компьютера

4. Кликните внутри окна программы с кодом программы *teleport.py*.

5. Кликните **Run → Run Module** или нажмите F5. Если вы не сохранили программу, IDLE попросит вас это сделать — кликните в появившемся диалоговом окне **OK.** Если вы кликнете **Cancel**, программа не запустится.

В среде IDLE на Raspberry Pi диалоговое окно с запросом на сохранение программы может открыться позади окна Minecraft, так что его не будет видно. Если вам кажется, что программа IDLE зависла, возможно, проблема в этом. Тогда сверните окно Minecraft, кликните OK в диалоговом окне IDLE, а затем снова разверните окно Minecraft.

Отлично! Теперь программа должна запуститься, и через несколько секунд ваш игрок телепортируется в новое место с координатами (10, 110, 12). У меня он оказался над болотом (рис. 2.9). Поскольку мир Minecraft на вашем компьютере отличается от моего, ваш игрок окажется в другой обстановке.

БОНУСНОЕ ЗАДАНИЕ: ФИГАРО ЗДЕСЬ, ФИГАРО ТАМ

Ну что, разобрались, как работает телепортация? Присвойте переменным `x`, `y` и `z` другие значения и посмотрите, где окажется игрок. Попробуйте ввести отрицательные числа!

Рис. 2.9. Я телепортировал игрока из дома в место с координатами (10, 110, 12), и он оказался над болотом. Караул!

Вещественные числа

Не все числа целые. Существуют и дробные числа. В десятичных дробях дробную часть от целой принято отделять запятой. Например, половину

яблока можно записать так: 0,5. В языке Python дробные числа записываются через точку, а называются *вещественными*. И это еще один тип данных наряду с целыми числами. Кстати, целое число вполне можно записать в виде дробного (например, так: 3.0), но дробное число с цифрами после точки нельзя записать в виде целого числа. Вещественные числа используют вместо целых, когда нужна бóльшая точность.

Возможно, вы уже заметили, что в значениях координат игрока на рис. 2.4 и 2.5 есть цифры после точки, а значит, это вещественные числа!

В Python вы можете присвоить переменной значение в виде вещественного числа — точно так же как вы делали это с целыми числами. Например, чтобы задать переменной `x` значение 1,34, нужно ввести:

```
>>> x = 1.34
```

А чтобы сделать это вещественное число отрицательным, просто поставьте перед ним знак «минус»:

```
>>> x = -1.34
```

В следующей миссии вы обретете полную власть над телепортацией, потому что научитесь перемещать игрока в позицию с точными координатами.

МИССИЯ 2. ПЕРЕМЕЩЕНИЕ В ТОЧНОСТИ ТУДА, КУДА НАДО

Вы уже умеете задавать позицию игрока с помощью целых чисел, однако его координаты можно указать куда более точно, если воспользоваться вещественными числами. В ходе этой миссии вы измените код из миссии 1 и телепортируете игрока в место с вещественными значениями координат. Для этого сделайте следующее:

1. Откройте в IDLE программу *teleport.py* (см. с. 61), кликнув **File → Open** и выбрав соответствующий файл в папке *variables*.
2. Сохраните файл под именем *teleportPrecise.py* (туда же, в папку *variables*).
3. В коде программы *teleportPrecise.py* присвойте переменным `x`, `y` и `z` вещественные значения, введя вместо 10, 110 и 12 числа 57.0, 103.0 и 31.0.
4. Поменяйте последнюю строчку на `mc.player.setPos(x, y, z)`, убрав из команды слово `Tile`.

Teleport precise — точная телепортация

Tile — блок

5. Сохраните код.

6. Откройте Minecraft и запустите программу.

В законченном виде ваш код должен выглядеть так:

teleportPrecise.py

```
# Подключаемся к Minecraft
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

# Присваиваем переменным x, y и z значения координат
x = 57.0
y = 103.0
z = 31.0

# Меняем позицию игрока
mc.player.setPos(x, y, z)
```

Set pos (pos — *сокр.* от position) — задать позицию

Чем отличается команда `mc.player.setPos(x, y, z)` из нового кода от команды `mc.player.setTilePos(x, y, z)` из листинга 2.1? Известно, что пространство Minecraft состоит из блоков. Функция `setTilePos()` с помощью целых чисел указывает игре координаты блока, в который нужно телепортировать игрока. Функция же `setPos()` делает немного иное — при помощи вещественных чисел она задает не только координаты блока, но и точную позицию игрока внутри этого блока. Запустив программу у себя, я переместил игрока на вершину башни (рис. 2.10).

Рис. 2.10. Присвоив переменным точные вещественные значения координат, я телепортировал игрока на вершину башни

БОНУСНОЕ ЗАДАНИЕ: ТОЧНАЯ ТЕЛЕПОРТАЦИЯ

Присвойте переменным `x`, `y` и `z` какие-нибудь положительные и отрицательные вещественные значения и запустите программу. Затем измените цифры после точек и снова запустите. Что произойдет?

Замедление телепортации с помощью модуля time

Python выполняет команды кода почти мгновенно. Однако вы можете замедлить процесс, научив вашу программу делать паузу в несколько секунд.

Time — время

Воспользуйтесь модулем `time`, который содержит готовые функции для работы со временем. Для этого добавьте в самый верх программы следующую строку:

```
import time
```

Sleep — уснуть

Важно помнить, что модуль `time` должен быть импортирован (с помощью команды `import`) раньше всех его функций. Например, функции `sleep()`, которая позволяет приостановить выполнение программы на указанное число секунд. В противном случае Python не сможет найти функцию `sleep()` и, не зная, что делать дальше, остановит вашу программу. Именно поэтому команды импорта всех необходимых вам модулей лучше помещать в начале кода. Я, например, в первых двух строках кода всегда пишу команды для подключения к игре, а третьей строкой импортирую модуль `time`.

Вот пример использования вызова одной из функций модуля `time` — функции `sleep()`:

```
time.sleep(5)
```

Эта команда приостановит выполнение программы на пять секунд. В качестве аргумента функции `sleep()` вы можете использовать любые числа — как целые, так и вещественные. Например:

```
time.sleep(0.2)
```

Когда запущенная программа дойдет до этой строчки, она приостановит свою работу на 0,2 секунды.

Итак, теперь вы можете управлять временем, а значит, настал черед следующей миссии!

МИССИЯ 3. ТЕЛЕПОРТАЦИОННЫЙ ТУР

Прелесть телепортации состоит в том, что вы можете переместить игрока куда угодно! Попробуйте в этой миссии применить все свои знания, полученные ранее, чтобы отправить игрока в телепортационный тур по миру Minecraft!

Вам предстоит изменить код из миссии 1 так, чтобы ваш игрок смог побывать в разных областях игрового мира. Он должен сначала телепортироваться в одно место, а затем, после паузы, — в другое.

1. Откройте в IDLE программу *teleport.py* (с. 61). Для этого кликните **File → Open** и выберите соответствующий файл в папке *variables*.

Tour — путешествие

2. Сохраните программу в той же папке под новым именем *tour.py*.

Import time — импортировать (загрузить) модуль time

3. Сразу после строк кода для подключения к Minecraft добавьте команду `import time`.
4. В конце кода добавьте команду `time.sleep(10)`.
5. Скопируйте строки, в которых объявляются переменные `x`, `y` и `z`, а также строку с функцией `setTilePos()` и вставьте их в конец программы, чтобы они повторялись дважды.
6. Поменяйте все значения `x`, `y` и `z` на любые целые числа. Вы можете узнать координаты любого места в мире Minecraft, перейдя туда и посмотрев нужные значения (ранее в этой главе мы выяснили, как это сделать).
7. Сохраните программу.
8. Откройте Minecraft и запустите программу.

Ваш код должен выглядеть так:

tour.py

```
# Подключаемся к Minecraft
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import time

# Присваиваем переменным x, y и z значения координат
x = # Впишите значение
y = # Впишите значение
z = # Впишите значение

# Меняем позицию игрока
mc.player.setTilePos(x, y, z)
```

```
# Ждем 10 секунд
time.sleep(10)

# Присваиваем переменным x, y и z значения координат
x = # Впишите значение
y = # Впишите значение
z = # Впишите значение

# Меняем позицию игрока
mc.player.setTilePos(x, y, z)
```

Рис. 2.11. Я задал такие координаты, чтобы мой игрок телепортировался сначала в дом, а затем в пустыню

Только вместо комментариев напротив переменных должны стоять конкретные числа.

После запуска программы игрок должен оказаться в месте с первым набором координат, а через 10 секунд — в месте со вторым набором координат (рис. 2.11).

Затем измените код так, чтобы при каждой следующей телепортации менялось значение лишь одной переменной: `x`, `y` или `z`. Необязательно всегда изменять значение всех переменных! А еще попробуйте использовать вместо целых чисел вещественные.

БОНУСНОЕ ЗАДАНИЕ: ЕЩЕ БОЛЬШЕ ТЕЛЕПОРТАЦИИ

Скопируйте и вставьте строчки кода из программы *tour.py* столько раз, сколько раз вы хотите телепортировать игрока. Замените 10 внутри функции `time.sleep(10)` на другое значение. Можете даже использовать разные числа, чтобы игрок проводил в каждом месте разное количество времени.

Отладка

Все мы совершаем ошибки, и зачастую даже лучшие программисты не могут с ходу написать правильный код. Поэтому умение создавать работающие программы — лишь один из навыков, необходимых хорошему разработчику. Другое важное умение — поиск и устранение ошибок в коде. Этот процесс называется *отладкой*. Далее мы рассмотрим приемы, которые помогут вам отлаживать свои программы.

Ошибки в программе могут либо остановить ее работу, либо заставить работать некорректно. Если программа не запускается, Python выдаст *сообщение об ошибке* (рис. 2.12).

На рис. 2.12 видно, что я ввел в окне консоли Python некий код и получил сообщение об ошибке. В этом сообщении содержится немало информации, но если вы обратите внимание на последнюю строчку — `NameError: name 'x' is not defined`, — то заметите, что с переменной `x` что-то не так. Собственно говоря, в коде отсутствует объявление этой переменной. Чтобы исправить эту ошибку, нужно добавить в код соответствующую команду:

Name error: name 'x' is not defined — код ошибки: имя 'x' не определено

```
>>> x = 10
```

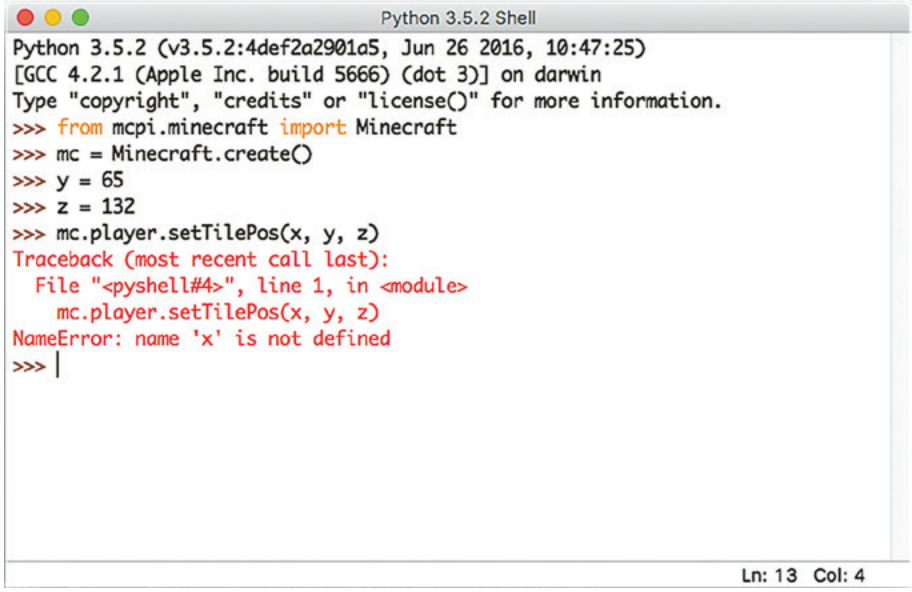

Рис. 2.12. Python показывает, что ошибка произошла из-за нарушения синтаксиса

Она устранит ошибку, и сообщение исчезнет. Но это совсем не значит, что теперь в программе все работает идеально.

Встречаются ошибки, которые не прерывают работу программы и не приводят к выводу сообщений, однако конечный результат говорит о том, что в программе что-то не так. Например, если в коде для телепортации игрока пропустить строку с телепортирующей функцией, такой как `setTilePos()`, программа спокойно запустится и завершится без единого сообщения об ошибке, но положение игрока не изменится. То есть программа получится совершенно бесполезной!

Чаще всего ошибки возникают из-за опечаток. Если ввести команду так, что компьютер не сможет ее понять, программа работать не будет. Пишите код внимательно и помните, что для Python важно, какие буквы вы используете — строчные или прописные!

МИССИЯ 4. ИСПРАВЬТЕ НЕРАБОТАЮЩИЙ ТЕЛЕПОРТАТОР

В ходе этой миссии вам предстоит отладить две программы. Первая программа (листинг 2.2) напоминает программу *teleport.py* со с. 61, однако в этой версии есть несколько ошибок. Откройте в окне программы IDLE новый файл, введите код листинга 2.2 и сохраните под именем *teleportBug1.py*.

Bug — ошибка в программе, баг

teleportBug1.py

```
from mcpi.minceraft import Minecraft
# mc = Minecraft.create()

x = 10
  y = 11
z = 12
```

Листинг 2.2. Первая неработающая программа телепортации

Чтобы отладить программу, выполните следующее:

1. Запустите *teleportBug1.py*.
2. Получив сообщение об ошибке, изучите его последнюю строку — она подскажет, в чем заключается проблема.
3. Исправьте ошибку и снова запустите программу.
4. Продолжайте исправлять ошибки до тех пор, пока программа не телепортирует игрока в место с заданными координатами.

Проверьте, есть ли в вашем коде функция телепортации `setTilePos()`*!*

Теперь отладьте еще одну программу. Код листинга 2.3 запускается, но по какой-то причине игрок перемещается в место, которое не соответствует значениям координат *x*, *y* и *z*. Введите этот код в новый файл IDLE и сохраните как *teleportBug2.py*.

teleportBug2.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

x = 10
y = 110
z = -12

mc.player.setPos(x, z, y)
```

Листинг 2.3. Вторая неработающая программа телепортации

В отличие от кода *teleportBug1.py*, при запуске этой программы вы не получите сообщения об ошибке. Чтобы ее найти и исправить,

внимательно изучайте код, пока не обнаружите неверно записанную команду. Эта программа должна телепортировать игрока в место с координатами 10, 110, –12. Запустив ее, проверьте координаты места, куда на самом деле перенесся игрок, — это поможет найти ошибку.

Отладив обе программы и добившись их правильной работы, добавьте в них комментарии, описывающие, в чем именно заключались ошибки. Это послужит напоминанием и поможет в будущем избежать подобных проблем.

Что вы узнали

Поздравляю! Вы написали свою первую Python-программу, которая перемещает игрока при помощи переменных и функций, узнали о двух типах данных (целые и вещественные числа), попробовали управлять временем и отладили неработающие программы. Также вы познакомились с двумя телепортирующими функциями Minecraft Python API: `setPos()` и `setTilePos()`.

В главе 3 вы научитесь мгновенно возводить постройки при помощи математических операций и функций установки блоков.

3

МАТЕМАТИКА, МОМЕНТАЛЬНОЕ СТРОИТЕЛЬСТВО И СУПЕРПРЫЖКИ

В главе 2 вы узнали, как объявлять переменные и менять их значения, чтобы телепортировать игрока в любую точку мира Minecraft. В этой главе вы научитесь при помощи математических операций мгновенно возводить постройки. А еще ваш герой обретет суперспособность — сможет совершать гигантские прыжки!

Выражения и команды

Чтобы общение имело смысл, собеседник должен вас понимать. Можно сказать, например, «три алмаза» или «за деревом». Однако вас не поймут, потому что сами по себе эти фразы ничего не значат. Они обретают смысл лишь в составе предложения, например такого: «Я нашел за деревом три алмаза».

Выражения и команды в языке Python похожи на фразы и предложения в русском языке.

Объединив числа, переменные и математические операции, вы можете создать небольшой фрагмент кода — *выражение*. Например, такое: `2 + 2`. Из выражений можно составлять команды, о которых мы уже говорили в главе 2. *Команда* — это фрагмент кода, занимающий одну или несколько строк и выполняющий в программе определенные действия. Так, выражение `2 + 2` может быть частью команды `zombies = 2 + 2`.

Несколько строк команда может занимать только в окне программы. *Прим. науч. ред.*

Окно консоли Python и окно программы IDLE обрабатывают выражения по-разному. Если в вашем коде много строк, используйте окно

программы. В отличие от окна консоли, в окне программы необходимо вводить команды целиком. Если ввести выражение `2 + 2` в окне консоли, Python выдаст результат — число `4`:

```
>>> 2 + 2
4
```

Однако, если вы введете `2 + 2` в окне программы, при запуске кода Python не сможет найти значение выражения, поскольку такая запись не является командой. Впрочем, это можно исправить — например, присвоив значение выражения переменной `zombies`:

Zombies — зомби

```
zombies = 2 + 2
```

Теперь выведем содержимое переменной на экран:

```
print(zombies)
```

Запустив эту программу, вы увидите число 4.

Повторю еще раз: при написании кода в окне программы необходимо использовать команды, а не просто выражения.

Операции

В математике операциями называются производимые над числами действия. В программировании *математические операции* позволяют изменять и объединять числа нужным образом.

В Python есть все основные математические операции: сложение, вычитание, умножение и деление, — а также более сложные, такие как возведение в степень. Начнем со сложения.

Сложение

Сложение в Python выполняется так же, как в математике, — при помощи знака плюс (+). Если у игрока есть два цветка (flowers) и он сорвал еще два, произошедшее можно описать командой:

```
>>> flowers = 2 + 2
```

Python вычислит значение выражения справа от знака равно, а затем присвоит его переменной `flowers`, которая стоит слева. В данном

случае результатом будет число 4, а значит, переменная `flowers` примет значение 4.

Ну вот, теперь с помощью операции сложения вы сможете быстро возводить постройки в мире Minecraft. Готовы к новой миссии? Поехали!

МИССИЯ 5. БАШЕНКА ИЗ БЛОКОВ

Set block — установить блок

С помощью функции `setBlock()` в мире Minecraft можно создать блок и поместить его в заданное место. Аналогично `setPos()` и `setTilePos()` функция `setBlock()` принимает значения трех аргументов — координат *x*, *y* и *z*. Но есть у нее и четвертый аргумент: тип блока. Он определяет, какой именно блок нужно добавить в игру.

Каждому типу блоков, будь то трава, лава или арбуз, соответствует определенное целое число. Для стоячей лавы это 11, для воздуха — 0, для воды — 8, а для арбуза — 103. Полный перечень блоков и соответствующих им чисел можно найти в разделе «Идентификаторы блоков» на с. 350.

Давайте вызовем функцию `setBlock()` и присвоим значения ее аргументам `x`, `y` и `z`, а также аргументу, который хранит идентификатор блока, отделив числа запятыми. Допустим, мы хотим поместить блок «арбуз» в место с координатами (6, 5, 28):

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
mc.setBlock(6, 5, 28, 103)
```

После первых двух строк, которые вам уже знакомы, следует функция `setBlock()`. Вы можете объявить переменные и присвоить им нужные значения, результат будет таким же (листинг 3.1).

blockStack.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
x = 6
y = 5
z = 28
blockType = 103
mc.setBlock(x, y, z, blockType)
```

Листинг 3.1. Код для создания блока «арбуз»

Здесь мы сначала объявили переменные, соответствующие координатам блока (`x`, `y` и `z`) и типу блока (`blockType`), а затем передали их в функцию `setBlock()`. А дальше Minecraft Python API стал творить

свое волшебство. Эти переменные можно использовать в программе и дальше, а если вы захотите изменить их значения, достаточно будет сделать это в одном месте.

Создайте внутри папки *Minecraft Python* новую папку *math*. Запустите IDLE, откройте окно программы и сохраните файл в папку *math* под именем *blockStack.py*. Введите в окне программы код листинга 3.1, а затем добавьте туда две строки из листинга 3.2. Так вы создадите еще один блок «арбуз», который появится над предыдущим.

Math — математика

Block stack — башенка из блоков

blockStack.py

```
❶ y = y + 1
❷ mc.setBlock(x, y, z, blockType)
```

Листинг 3.2. Дополнительный код для того, чтобы положить поверх первого арбуза еще один

Увеличив значение `y` ❶ на 1 и вызвав функцию `setBlock()` ❷, вы заставляете второй блок появиться прямо над первым, ведь его позиция на один блок выше по оси *y*.

Теперь попробуйте поставить на эти блоки еще два. Измените код *blockStack.py* так, чтобы при перезапуске программы получилась башенка из четырех блоков — как на рис. 3.1.

Рис. 3.1. Я построил башенку из блоков «арбуз»!

Чтобы поставить второй блок на первый, мы увеличили переменную `y` на 1 и снова вызвали функцию `setBlock()`. Как думаете, что произойдет, если мы добавим эти две команды в конец кода? А если повторим их три раза? Получится ли башенка из пяти блоков?

БОНУСНОЕ ЗАДАНИЕ: РАДУГА

Код программы *blockStack.py* можно изменять различными способами. Например, чтобы создать башню или арку из стоячей лавы. Поэкспериментируйте с типами блоков и посмотрите, что у вас получится!

МИССИЯ 6. СУПЕРПРЫЖОК

В главе 2 вы научились менять позицию игрока. Давайте прокачаем этот навык и с помощью волшебной силы сложения подбросим игрока в небо. Но сначала нам нужно узнать его текущую позицию. Для этого вызовите функцию `getTilePos()`, как в листинге 3.3.

Get tile pos (pos — *сокр.* от position) — получить позицию блока

superJump.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

position = mc.player.getTilePos()
x = position.x
y = position.y
z = position.z
```

Листинг 3.3. Код для определения текущей позиции игрока

Position — позиция

Обратите внимание на точку между переменной `position` и буквами `x`, `y` и `z`. Это так называемая *точечная нотация*. Она используется при работе с некоторыми переменными и функциями — например, со всеми функциями Minecraft Python API, включая `setTilePos()`. В главах 11 и 12 вы познакомитесь с точечной нотацией ближе.

Узнав позицию игрока, нужно присвоить переменным `x`, `y` и `z` его координаты — в коде это переменные `position.x`, `position.y` и `position.z`. Теперь можно телепортировать игрока, сместив его относительно текущей позиции, как показано в листинге 3.4.

superJump.py

```
x = x + 5
mc.player.setTilePos(x, y, z)
```

Листинг 3.4. Код для смещения игрока на пять блоков по оси x

Я сместил игрока на пять блоков вдоль оси *x*. Однако что в этом удивительного? Играя в Minecraft, вы и так можете передвигать игрока по горизонтали. Давайте лучше научим его суперпрыжку!

Ваша задача — сделать так, чтобы игрок поднялся на 10 блоков выше своей текущей позиции. Для этого необходимо немного изменить код. Введите строки из листингов 3.3 и 3.4 в окно программы IDLE, сохраните файл под именем *superJump.py* и вместо `x` таким же образом измените `y`. После запуска программы игрок должен подняться в воздух, как на рис. 3.2.

Super jump — суперпрыжок

Рис. 3.2. Суперпрыжок в действии

Вычитание

Вычитание в Python производится так же, как и сложение. Предположим, в пещере игрока атакует паук, и игрок теряет здоровье:

```
health = 20
health = health - 2
```

В переменной `health` после этого остается значение 18. Как и при сложении, Python вычисляет результат выражения, которое находится справа от знака равно, и присваивает это значение переменной.

Health — здоровье

Не пора ли повеселиться? Давайте посмотрим, что интересного можно сделать в игре Minecraft с помощью вычитания.

МИССИЯ 7. ИЗМЕНИТЕ БЛОК ПОД НОГАМИ ИГРОКА

Признайтесь, вам когда-нибудь хотелось устроить герою Minecraft ловушку? Только представьте, что земля под ногами у ничего не подозревающего героя внезапно превращается в лаву! Python может исполнить это желание. Чтобы разместить под игроком вместо блока «земля» любой другой блок, требуется лишь несколько строк кода и умение вычитать!

В ходе этой миссии вам предстоит поменять блок под ногами игрока на блок «текущая лава», используя функции `getTilePos()` и `setBlock()`. Осторожно: герою грозит опасность — он может упасть в лаву, если вы не успеете передвинуть его на другое место!

Код листинга 3.5 устанавливает блок в текущей позиции игрока. Введите этот код в новом окне программы IDLE и сохраните файл как *blockBelow.py*. Затем измените код, добавив вычитание, чтобы разместить блок лавы прямо под ногами игрока, как показано на рис. 3.3.

Block below — блок ниже

blockBelow.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
blockType = 10
mc.setBlock(x, y, z, blockType)
```

Листинг 3.5. Этот код размещает блок «текущая лава» в позиции игрока

Рис. 3.3. Блок под игроком поменялся, и тот упал в лаву

Обратите внимание, что я дал переменной, хранящей позицию игрока, короткое имя `pos`. Поступил я так потому, что часто обращаюсь к этой переменной, а сокращенное слово вводить проще, чем полное `position`. При этом назначение переменной все равно понятно.

Координата *y* определяет, на какой высоте будет создан блок. Ваша задача — понять, как поменять значение `y`, чтобы блок оказался прямо под игроком.

БОНУСНОЕ ЗАДАНИЕ: БЛОКИ ПОВСЮДУ

Теперь вы знаете, как менять блоки под игроком. А сможете создать их над игроком? Поэкспериментируйте и разместите вокруг игрока сразу несколько блоков. Отсюда уже рукой подать и до постройки зданий!

Попробуйте объединить ваш код с кодом из миссии 6 (с. 78). Сделайте так, чтобы игрок поднялся в воздух, а под ним тут же возник блок, остановив падение. Или вам больше по душе темные делишки? Можете написать код, который поднимет игрока высоко над землей, а тем временем под ним появится озеро лавы.

Математические операции и аргументы

Вызывая функцию, например `setBlock()` или `setTilePos()`, вы присваиваете ее аргументам нужные вам значения.

Теперь, когда вы освоили математические операции сложения и вычитания, можете применить их к аргументам сразу при вызове функции. Давайте вернемся к коду постройки башенки из миссии 5 (с. 76). Если внутри скобок `setBlock()` к аргументу `y` прибавить какое-либо число, значение аргумента будет равно результату сложения. Так что для увеличения `y` не придется использовать отдельную команду.

blockStack1.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

x = 6
y = 5
z = 28
blockType = 103
mc.setBlock(x, y, z, blockType)
❶ mc.setBlock(x, y + 1, z, blockType)
```

Листинг 3.6. Код постройки башенки с операцией сложения в списке аргументов

Листинг 3.6 почти не отличается от первоначальной программы постройки башенки, однако здесь сложение выполняется не отдельной командой, а прямо в списке аргументов функции `setBlock()`. Обратите внимание: в последней строке кода ❶ в качестве аргумента функции выступает выражение `y + 1`. Несмотря на то что в функцию будет передано число `6` (5 + 1), у переменной `y` останется значение `5`. Таким образом, мы передали в функцию увеличенное значение переменной, не изменив ее исходного значения. А это может быть полезным, если дальше в коде нам понадобится первоначальное значение переменной `y`.

Точно так же в качестве аргумента функции можно указывать сумму двух переменных. Листинг 3.7 похож на листинг 3.6, однако в код добавлена еще одна переменная, которая определяет, с каким смещением по оси *y* будет создан новый блок.

blockStack2.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

x = 6
y = 5
z = 28
blockType = 103
up = 1
mc.setBlock(x, y, z, blockType)
❶ mc.setBlock(x, y + up, z, blockType)
```

Листинг 3.7. Другая версия кода постройки башенки с операцией сложения в списке аргументов

Рис. 3.4. Результат работы трех версий кода одинаков

В строке ❶ значения переменных `y` и `up` складываются. Аналогично листингу 3.6 результат сложения (число 6) становится вторым аргументом функции `setBlock()`. Введение дополнительной переменной дает преимущество: если вы захотите поднять новый блок на одну позицию выше по оси *y*, вам нужно будет лишь присвоить переменной `up` значение 2. Результат работы всех трех версий этого кода (листинги 3.1 и 3.2, 3.6 и 3.7) будет выглядеть как на рис. 3.4.

Up — вверх

МИССИЯ 8. БЫСТРОЕ СТРОИТЕЛЬСТВО

Обычно первый день в игре Minecraft уходит на то, чтобы построить убежище. Но с новыми знаниями вы сумеете мгновенно создать небольшой домик, в котором ваш игрок сможет безопасно провести первую игровую ночь! Код, который мы напишем в ходе этой миссии, поможет вам быстро соорудить стены, пол и потолок. Вместо того чтобы тратить уйму времени, устанавливая каждый блок вручную, вы построите убежище, написав лишь несколько строк кода.

Для установки единичного блока мы использовали функцию `setBlock()`, однако есть еще одна функция — `setBlocks()`. Она позволяет создать прямоугольный параллелепипед, который еще называют *кубоидом*, состоящий сразу из нескольких блоков. При этом длина, ширина и высота кубоида могут быть разными.

Set blocks — установить блоки

Функция `setBlocks()` позволяет заполнять блоками большие объемы пространства. Вызвав эту функцию, передайте ей два набора координат и тип блока. Первый набор координат должен соответствовать положению одного угла кубоида, а второй — положению противоположного ему угла. На рис. 3.5 обозначены углы кубоида и показаны их координаты.

Давайте создадим кубоид как на рис. 3.5. В листинге 3.8 в качестве материала для постройки указан булыжник (идентификатор 4), однако вы можете выбрать любой другой тип блока. Конечно, кроме воздуха, лавы и воды — хорошенький получится дом из таких материалов!

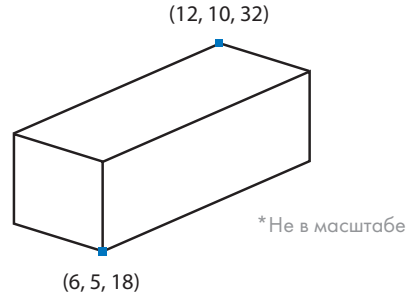

Рис. 3.5. Кубоид и координаты, задающие размеры его сторон

building.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
❶ pos = mc.player.getPos()
```

```
x = pos.x
y = pos.y
z = pos.z
width = 10
height = 5
length = 6
❷ blockType = 4
❸ air = 0
mc.setBlocks(x, y, z, x + width, y + height, z + length, blockType)
```

Листинг 3.8. Код для создания кубоида из блоков

Get pos (pos — *сокр.* от position) — получить позицию

Обратите внимание: в строке ❶ я вызвал функцию `getPos()`, а не `getTilePos()`. Функция `getPos()` работает так же, как `getTilePos()`, однако возвращает координаты в виде трех вещественных, а не целых чисел.

Ширина (width), высота (height) и длина (length) кубоида равны 10, 5 и 6 блокам. Я передаю эти значения соответствующим переменным и, чтобы соорудить кубоид из булыжника, в строке ❷ указываю идентификатор блока 4. Готовая постройка показана на рис. 3.6.

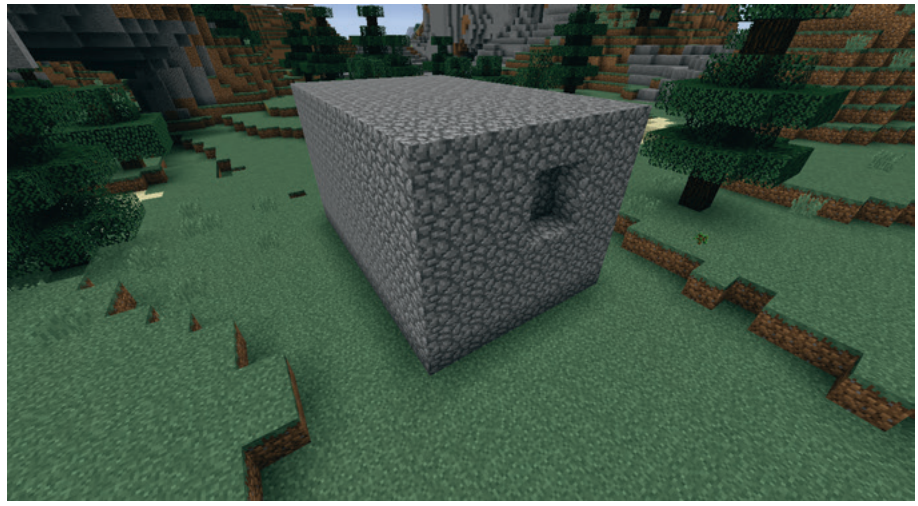

Рис. 3.6. Каменная постройка, созданная программой

Однако в этом «доме» нет ни окон, ни дверей, ни комнат. После завершения работы программы я с помощью героя проделал дыру в одной из стен, чтобы показать: дом сплошь состоит из камня. Впрочем, для начала этот кубоид совсем не плох, осталось лишь создать в нем полость, чтобы игрок смог войти внутрь.

Ваша задача — изменить код так, чтобы каменный кубоид превратился в постройку со стенами, полом и потолком, но остался в текущей позиции игрока. Для этого создайте кубоид из блоков «воздух» внутри сплошного каменного кубоида. В результате у вас получится каменная «коробка», как на рис. 3.7. Я проделал дыру в стене, чтобы показать: внутри пусто.

Рис. 3.7. Ваша программа должна создать полый кубоид. Кубоиды отлично подходят для быстрого строительства!

В листинге 3.8 в строке ❸ уже есть переменная `air`, которая задает тип блока внутреннего кубоида. Введите этот код в окне программы IDLE, сохраните файл как *building.py* и добавьте команды для создания кубоида из блоков «воздух».

Air — воздух

Building — постройка

Для этого вызовите в конце кода функцию `setBlocks()`. Размеры сторон внутреннего кубоида должны быть меньше сторон внешнего кубоида ровно на один блок. Поэтому вам пригодятся операции сложения и вычитания. Будьте настойчивы: если с первого раза не получилось, попробуйте снова.

Чтобы создать воздушный кубоид, отстоящий от стен на один блок, добавьте в код кубоид из блоков «воздух» при помощи функции `setBlocks()` и увеличьте на 1 значения `x`, `y`, `z` из первого набора аргументов. Затем вычтите 1 из значений второго набора аргументов `x + width`, `y + height` и `z + length`.

БОНУСНОЕ ЗАДАНИЕ: СТРОЙТЕ ЧТО УГОДНО

Вы можете переделывать этот код, чтобы строить самые разные здания! Получится ли у вас изменить ширину, высоту и длину готовой постройки? А создать бассейн? Подсказка: для этого нужно поменять тип блока внутреннего кубоида на воду (идентификатор 8 или 9) и убрать верхнюю стенку внешнего кубоида, чтобы игрок мог зайти в бассейн.

Умножение

Математическая операция умножение в Python выглядит несколько иначе, чем в учебниках по математике. Чтобы перемножить два числа, нужно поставить не точку (•) и не крестик (×), а звездочку (*). Но в остальном это все то же старое доброе умножение: 2 * 2 равняется 4, так же как и 2 • 2.

Представьте, что вокруг дома игрока растут четыре дерева (trees). Внезапно число деревьев удваивается. Соответствующие команды в Python будут выглядеть так:

```
trees = 4
trees = trees * 2
```

В итоге переменная `trees` примет значение 8 (результат умножения 4 на 2).

Деление

Деление в Python обозначается не двоеточием (:), а косой чертой (/). Чтобы разделить одно число на другое, поставьте делимое слева от косой черты, а делитель — справа.

Допустим, под стенами вашей крепости было восемь скелетов (skeletons), но потом половина ушли. Чтобы узнать, сколько скелетов осталось, необходимо разделить 8 на 2. Вот как это будет выглядеть в коде:

```
skeletons = 8
skeletons = skeletons / 2
```

Уф-ф, выходит, у стен крепости осталось всего четыре скелета! Теперь давайте попробуем применить эти две математические операции, чтобы создать нечто новое в мире Minecraft.

МИССИЯ 9. ПОТРЯСАЮЩИЕ ШПИЛИ

Переменные хороши тем, что, изменив их значения в одном месте, вы автоматически меняете их во всем коде.

В ходе этой миссии вам предстоит с помощью умножения и деления построить высокую, сужающуюся кверху башню — ее еще называют шпилем (рис. 3.8).

Рис. 3.8. Шпиль из каменных блоков

Чтобы задать высоту шпиля, вам понадобится единственная переменная. Но, чтобы определить высоту каждого из его уровней, пригодятся операции умножения и деления.

В листинге 3.9 используется переменная `height`, которая определяет высоту среднего уровня шпиля.

Height — высота

Введите код листинга 3.9 в новый файл IDLE и сохраните его в папке *math* под именем *spire.py*.

Spire — шпиль

Эта программа может создать шпиль, но его высота не будет зависеть от переменной `height`. Если вы введете другое значение `height` и перезапустите программу, изменится высота только среднего уровня шпиля, а высота основания и верхушки останется прежней.

Необходимо это исправить и добиться, чтобы все уровни шпиля меняли свою высоту при изменении переменной `height`. Для этого нужно присвоить переменным `pointHeight` ❶ и `baseHeight` ❷ другие значения — подставить туда выражения с переменной `height` и операциями умножения или деления.

Point height — высота верхушки

Base height — высота основания

Пусть значение переменной `pointHeight` будет вдвое больше значения `height`, а значение переменной `baseHeight` — вдвое меньше. Если я захочу, чтобы верхушка шпиля была втрое выше его средней части (высота которой равна `height`), я изменю код, указав `pointHeight = height * 3`, при этом высота основания шпиля останется `baseHeight = height / 2`.

spire.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

height = 2
blockType = 1

# Средний уровень: высота должна равняться height
sideHeight = height
mc.setBlocks(x + 1, y, z + 1, x + 3, y + sideHeight - 1, z + 3,
             blockType)

# Верхушка: высота должна быть вдвое больше height
❶ pointHeight = 4
mc.setBlocks(x + 2, y, z + 2, x + 2, y + pointHeight - 1, z + 2,
             blockType)

# Основание: высота должна быть вдвое меньше height
❷ baseHeight = 1
mc.setBlocks(x, y, z, x + 4, y + baseHeight - 1, z + 4, blockType)
```

Листинг 3.9. Код для постройки трехуровневого шпиля

Если внести эти правки, высота всех трех уровней шпиля будет зависеть от переменной `height` и при изменении ее значения тоже изменится. Больше ничего в коде менять не нужно.

Чтобы протестировать работу программы, перезапустите ее, перед этим изменив значение переменной `height`. Если присвоить ей значение 3, шпиль должен выглядеть как на рис. 3.9.

Поскольку высота основания и верхушки шпиля зависит от высоты средней его части, изменить вид шпиля будет легко. Написав код, поэкспериментируйте с ним, присваивая переменной `height` разные значения. Если вносите изменения в программу, не забывайте ее перезапускать!

Рис. 3.9. Вы можете сделать шпиль выше, просто изменив значение `height`

Возведение в степень

Возведение в степень — это математическая операция, которая позволяет умножать число само на себя заданное количество раз. Например, 3^4 (читается «три в четвертой степени») — это то же самое, что 3 • 3 • 3 • 3.

В Python операцию возведения в степень пишут через двойную звездочку (**). Число, которое нужно умножить (основание), записывается слева от оператора, а количество его умножений само на себя (степень) — справа.

Допустим, вы решили создать в мире Minecraft ферму и вырастить на четырех участках земли пшеницу. Вам нужно много пшеницы, поэтому вы решаете, что каждый участок должен быть размером четыре на четыре блока. В математике расчет общей площади земли под пшеницу выглядел бы так: 4 • 4 • 4, или 4^3. А в Python так:

```
wheat = 4 ** 3
```

В итоге переменная `wheat` примет значение 64, потому что 4 • 4 — это 16, а 16 • 4 — это 64. Значит, пшеницей будет засеяно 64 блока.

Wheat — пшеница

Скобки и порядок выполнения операций

Если в одном выражении используется несколько математических операций, обращайте внимание на их порядок, поскольку все они обладают разным *приоритетом*. Операции выполняются слева направо:

сначала — умножение и деление, затем — сложение и вычитание. Давайте посмотрим, как Python вычислит такое выражение:

```
mooshroom = 5 * 2 - 1 + 4 / 2
```

Python начнет расчет с левой части. Умножив 5 на 2, он получит 10. Затем в правой части выражения разделит 4 на 2 и получит 2. То есть выражение превратится в такое: 10 – 1 + 2. Теперь Python снова начинает считать слева: вычитает 1 из 10, а затем прибавляет 2. В результате переменная `mooshroom` принимает значение 11.

Mooshroom — грибная корова

Однако вы можете изменить порядок расчета с помощью скобок. Если в выражении есть операции, заключенные в скобки, они вычисляются прежде всех остальных. Давайте посмотрим, как это работает. Для начала возьмем команду без скобок:

```
zombiePigmen = 6 * 3 - 2
```

Zombie pigmen — зомби-свинолюди

В этом случае значением переменной `zombiePigmen` окажется 16, поскольку 6, умноженное на 3, дает 18, а 18 минус 2 равняется 16. Но результат поменяется, если мы добавим скобки:

```
zombiePigmen = 6 * (3 - 2)
```

На этот раз `zombiePigmen` примет значение 6, потому что Python сначала вычтет 2 из 3, что даст 1, а затем умножит 6 на 1.

Если вам нужно что-то посчитать в определенном порядке, добавьте в выражение круглые скобки. Тогда программа поймет, что эту операцию нужно выполнить в первую очередь. Так ваши возможности программирования на Python станут еще шире!

Полезные математические хитрости

В следующих разделах я покажу вам еще два математических приема. А потом нас ждет миссия, для прохождения которой потребуются все полученные навыки программирования.

Сокращенные операции

Довольно часто бывает необходимо применить к переменной какую-либо математическую операцию, а затем сохранить результат в этой же переменной. Допустим, вы решили добавить в свою отару пять овец:

```
sheep = 6
sheep = sheep + 5
```

Однако со временем команда `sheep = sheep + 5` может вам надоесть. Не волнуйтесь, в языке Python существует и другой, более короткий способ применить к переменной математическую операцию и сохранить ее результат. Это *сокращенные операции*:

Sheep — овца

- сложения (+=)
- вычитания (–=)
- умножения (*=)
- деления (/=)

Пример с овцами можно переписать с помощью сокращенной операции сложения так:

```
sheep = 6
sheep += 5
```

Переменная `sheep`, как и раньше, будет хранить число 11, но запись выражения сократится.

Случайные числа

Случайные числа помогут сделать работу вашей программы более непредсказуемой и интересной. Они используются во многих настольных играх: вспомните игры-бродилки, где нужно бросать кубик, чтобы узнать, сколько шагов по полю должен сделать игрок. Числа, которые выпадают при броске кубика, — классический пример случайных чисел.

Python позволяет с легкостью генерировать случайные числа, так что мы можем написать код для виртуального игрального кубика. Случайное число должно находиться в диапазоне между 1 и 6:

```
❶ import random
❷ diceValue = random.randint(1, 6)
```

Чтобы создать генератор случайных чисел, первым делом импортируйте в начале кода модуль `random` ❶. Затем воспользуйтесь функцией `randint()` ❷ — она выведет в окне консоли случайное целое число. Функция `randint()` содержит два аргумента — наименьшее и наибольшее из возможных значений. В этом примере могут быть сгенерированы числа 1, 2, 3, 4, 5 и 6.

Random — случайный

Rand int (*сокр.* от random integer) — случайное целое число

Функцию `randint()` можно использовать, когда вам нужно прибавить к значению переменной случайное число — даже отрицательное. Посмотрим, как это делается.

```
import random
score = 0
score += random.randint(0, 99)
points = random.randint(-99, 99)
```

Score — счет
Points — очки

Здесь наименьшим значением, которое Python сможет сгенерировать и прибавить к переменной `score`, является 0, а наибольшим — 99. Обратите внимание на переменную `points`, которая из-за отрицательного аргумента функции `randint()` может принимать значения от –99 до 99.

МИССИЯ 10. СУПЕРПРЫЖОК В НЕИЗВЕСТНОСТЬ

Настало время последней (в этой главе) миссии. Ваша задача — сделать так, чтобы игрок переместился, изменив свои координаты *x*, *y* и *z* на случайные. Для этого сначала нужно узнать текущие координаты игрока и прибавить к каждой из них случайное число. Чтобы изменить координаты *x* и *z*, используйте значения от –10 до 10, а для координаты *y* — от 0 до 10.

Введите код листинга 3.10 в новый файл IDLE и сохраните его как *randomJump.py*.

Random jump — случайный прыжок

randomJump.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import random

pos = mc.player.getPos()
x = pos.x
y = pos.y
z = pos.z

❶ x = x + random.randint(-10, 10)
mc.player.setPos(x, y, z)
```

Листинг 3.10. Незаконченный код для случайного перемещения

Здесь отсутствует фрагмент кода для генерации случайных смещений по осям *y* и *z* — добавьте его сами. После этого игрок сможет прыгнуть в неизвестность (рис. 3.10). Позвольте случаю перенести его в новые потрясающие места!

В нашей программе нет сокращенных операций. Попробуйте сами заменить обычную математическую операцию в строке ❶ на сокращенную.

Рис. 3.10. Мой игрок прыгнул в случайном направлении и очутился на верхушке дерева

БОНУСНОЕ ЗАДАНИЕ: СЛУЧАЙНЫЕ ЧИСЛА И ТЕЛЕПОРТАЦИЯ

Давайте сделаем программу *randomJump.py* еще более непредсказуемой! После прыжка в случайное место разместите под ногами игрока блок случайно выбранного типа. А еще вы можете добавить в программу код телепортации из миссии 1 (см. *teleport.py* на с. 61), чтобы каждый раз игрок оказывался в случайном месте. Если вы телепортируете игрока туда, откуда он не сможет выбраться, запустите *teleport.py* еще раз, и герой окажется в более безопасном месте.

Что вы узнали

Из этой главы вы узнали, как в Python-программе работают математические операции сложения, вычитания, умножения, деления и возведения в степень. И теперь сможете использовать их в других кодах из этой книги, а также в собственных программах. Еще вы сгенерировали случайные числа и написали несколько полезных программ для Minecraft: научили игрока совершать суперпрыжки, построили дом и возвели шпиль. Так держать!

В главе 4 вы познакомитесь со строковым типом данных. С помощью строк можно отправлять сообщения в чат игры. Пригодятся они и для других замечательных операций.

4

ОБЩАЕМСЯ С ПОМОЩЬЮ СТРОК

В главах 2 и 3 вы имели дело с числами — целыми и вещественными. В этой главе вам предстоит познакомиться с еще одним типом данных — *строками*. В строках обычно содержатся буквы и знаки, но могут быть и числа. Строки — важная часть Python-программ. С их помощью вы можете вывести на экран *сообщение*, чтобы донести до пользователя нужную информацию.

Строки можно использовать для общения с другими пользователями в чате, если игра запущена в многопользовательском режиме. Имейте в виду, что, в отличие от версий игры для Windows и Mac OS, в Raspberry Pi функция отправки сообщений скрыта. Тем не менее при помощи магии программирования вы все же сможете отправлять сообщения! И делиться в них с друзьями секретной информацией или хвастаться своими сокровищами.

Также в этой главе мы будем использовать разные функции. Некоторые из них вам знакомы. Это `setPos()`, `setTilePos()`, `setBlock()`, `setBlocks()`, `getPos()` и `getTilePos()`. Вы же помните, что одна функция заменяет целый фрагмент кода, чем упрощает его написание? Здорово, правда?

Выполняя миссии из этой главы, вы углубите полученные ранее знания. Вам предстоит с помощью строк отправлять сообщения в чат, запрашивать данные у пользователей и создавать на их основе объекты в мире Minecraft.

Что такое строки?

Строковый тип данных позволяет работать с текстовой информацией разного вида: от буквы и знака, таких как «а» или «:», до целых абзацев. Каждая буква, цифра и знак в составе строки называется *символом*. Если вы хотите использовать в коде буквы, слова или предложения, строки — это именно то, что вам нужно.

Содержимое строки заключается в кавычки-лапки. Вот так:

```
"Берегись! У тебя за спиной зомби!"
```

А вот еще одна строка:

```
'Добро пожаловать на мою секретную базу!'
```

Заметили, чем отличается один пример от другого? Первая фраза заключена в двойные кавычки ("), а вторая — в одинарные ('). В строках можно использовать оба вида кавычек. Однако смешивать их нельзя! Если строка начинается с одиночной кавычки, то и заканчиваться она должна одиночной кавычкой. И наоборот: если строка начинается с двойной кавычки, в конце тоже должна стоять двойная. Оба вида кавычек поддерживаются в Python неспроста. Если в тексте нужно выделить прямую речь, вы без проблем сможете использовать двойные кавычки, заключив строку в одинарные.

Функция print()

Видеть на экране текстовую и другую информацию важно для пользователя: так он сможет понять, что происходит в программе. Данные, которые вы ему показываете, называются *выводом*. В Python для вывода информации используется функция `print()`.

Print — напечатать

Чтобы вывести на экран сообщение, передайте в функцию `print()` строковый аргумент:

```
>>> print("строка")
```

Получив такую команду, Python выведет на экран слово `"строка"`. А если вам захочется напечатать слово `"шоколад"`, введите:

```
>>> print("шоколад")
```

И на экране появится:

```
шоколад
```

Name — имя

Помимо этого, с помощью функции `print()` можно выводить значения переменных. Положим, у вас в программе есть переменная `name`, в которой хранится строка с именем:

```
>>> name = "Глеб Рудокоп"
```

Сохранив имя в переменной, вы можете дать команду `print(name)` и вывести содержимое переменной на экран:

```
>>> print(name)
Глеб Рудокоп
```

Итак, теперь вы знаете, как создавать строки и выводить их значения, а значит, пришло время для миссии, в которой вы поздороваетесь с миром Minecraft!

МИССИЯ 11. ПРИВЕТ, МИР MINECRAFT!

Post to chat — отправить сообщение в чат

Чтобы общаться с другими пользователями (даже в версии Minecraft Pi), вы можете отправлять сообщения в чат с помощью функции `postToChat()`. Эта функция принимает строковый аргумент, отображая его содержимое в окне игрового чата. Код из листинга 4.1 отправляет в чат сообщение `"Привет, мир Minecraft!"`.

message.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
mc.postToChat("Привет, мир Minecraft!")
```

Листинг 4.1. Код, который отправляет в чат Minecraft сообщение с приветствием

Напоминаю, что аргумент — это данные, которые вы передаете в функцию при ее вызове и которые нужны ей для работы. В предыдущей главе мы передавали в функции числовые значения, однако `postToChat()` требуется аргумент-строка.

Функция `postToChat()` похожа на `print()`. Обе выводят строки на экран, и обе могут принимать в качестве аргумента переменную, в которой хранится строка. Отличие их в том, что `print()` выводит строку в окне консоли Python, а `postToChat()` — в чате Minecraft.

Введите в окне программы код листинга 4.1 и сохраните файл как *message.py* в новой папке *strings*. После запуска программы в чате должно появиться сообщение как на рис. 4.1.

Message — сообщение

Strings — строки

Рис. 4.1. Мое сообщение появилось в чате

Попробуйте передать в функцию `postToChat()` другую строку, чтобы на экране появилось новое сообщение.

БОНУСНОЕ ЗАДАНИЕ: ГДЕ ТЫ?

С помощью функции `mc.postToChat()` вы можете отправлять в чат самую разную информацию. Попробуйте вывести на экран текущую *x*-координату игрока или тип блока, на котором он стоит. Напоминаю: текущую позицию игрока можно узнать, вызвав функцию `mc.player.getTilePos()`, а идентификатор блока, который находится в заданных координатах, возвращает функция `mc.getBlock()`.

Get block — получить блок

Функция input()

До сих пор все наши переменные были *жестко запрограммированы (захардкожены)*. Мы присваивали им значения, а затем переписывали код, если нужно было эти значения изменить. Согласитесь, что гораздо

удобнее было бы менять значения переменных прямо во время выполнения программы, иначе говоря, принимать *ввод пользователя*.

Input — ввод

Один из способов это делать — использовать функцию `input()`. Она выводит в окне консоли сообщение, поясняющее, какие данные необходимы программе, а затем ждет, пока пользователь введет эти данные. Откройте окно консоли, наберите следующий код и запустите его:

```
>>> input("Как вас зовут? ")
```

Вы увидите строку — аргумент функции `input()` — и сможете ввести ответ.

```
Как вас зовут?
```

Введите имя сразу после вопроса. У вас получится что-то вроде:

```
Как вас зовут? Артур
'Артур'
```

Отлично! Теперь, чтобы программа могла использовать введенные данные, их нужно сохранить в переменной `name`. При этом результат работы функции `input()` уже не будет автоматически выводиться на экран (а в случае, если код создан в окне программы, результат никогда не выводится автоматически).

```
>>> name = input("Как вас зовут? ")
Как вас зовут? Артур
```

Обратите внимание: на этот раз после того, как вы введете имя и нажмете ENTER, программа ничего не выведет на экран. Чтобы увидеть сохраненное в переменной `name` имя, просто передайте эту переменную в функцию `print()`:

```
>>> print(name)
Артур
```

Замечательно! Вы сохранили введенные данные в переменной, а затем вывели ее значение. Таким образом вы можете запрашивать информацию у пользователя и использовать ее в любом месте вашего кода. Давайте же воспользуемся этим приемом, чтобы отправить сообщения в чат Minecraft!

МИССИЯ 12. ОТПРАВЬТЕ В ЧАТ СООБЩЕНИЕ

Пора добавить чату интерактивности! В ходе миссии 11 вы уже отправляли сообщения из окна консоли Python. Теперь напишем код, в котором ваше сообщение будет храниться в переменной `message`.

Введите код листинга 4.2 в новое окно IDLE и сохраните файл под именем *messageInput.py* в папке *strings*.

Message input — ввод сообщения

messageInput.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
❶ message = "Просто сообщение"
❷ mc.postToChat(message)
```

Листинг 4.2. Отправка сообщения в чат Minecraft

В этом коде текст `"Просто сообщение"` хранится в переменной `message` ❶. Далее программа передает его в функцию `postToChat()` ❷, которая и отправляет сообщение в чат Minecraft.

Как видите, сейчас содержимое переменной захардкожено, а значит, сообщение будет одним и тем же при каждом запуске программы. Но достаточно изменить одну строку кода, и программа будет отправлять в чат любой введенный пользователем текст. Так что вы сможете создать собственную чат-программу!

Чтобы Python принимал ввод пользователя, замените строку `"Просто сообщение"` ❶ на функцию `input()`. Передайте ей подходящий аргумент — например, строку `"Введите сообщение: "`. Не забывайте, что аргумент должен стоять внутри круглых скобок. Изменив код, запустите программу. В окне консоли Python должно появиться приглашение `"Введите сообщение: "`. Напишите сообщение и нажмите ENTER. После этого текст сообщения должен появиться в чате Minecraft, как показано на рис. 4.2.

Теперь программа позволяет вводить сообщение с клавиатуры, а не встраивать его в код. Видите, насколько проще стало писать в чат с помощью функции `input()`?

БОНУСНОЕ ЗАДАНИЕ: ЕЩЕ БОЛЬШЕ СООБЩЕНИЙ

Сейчас программа запрашивает лишь одно сообщение. Сделайте так, чтобы после его ввода программа подождала несколько секунд (воспользуйтесь функцией `sleep()`), а затем запросила еще одно сообщение.

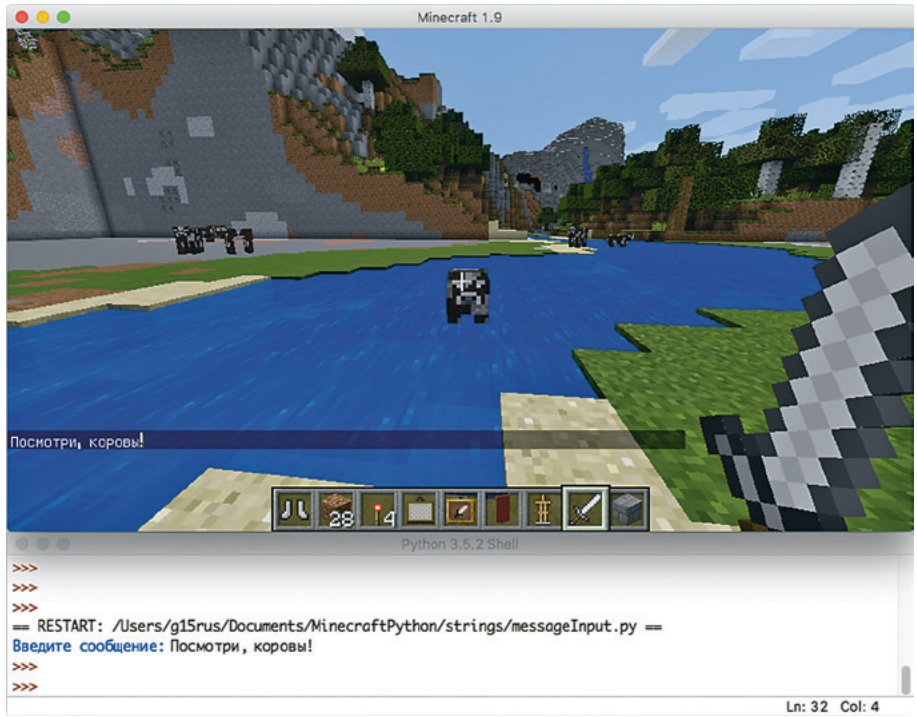

Рис. 4.2. Я ввел сообщение в окне консоли Python, и оно появилось в чате Minecraft

Склейка строк

Зачастую на экран нужно вывести значения нескольких строк, перед этим объединив их. Это называется *склейкой строк*, и в Python сделать ее совсем не сложно.

В главе 3 мы складывали числа при помощи математической операции сложения (+), однако эту операцию можно применить и для склейки строк. Например:

```
firstName = "Маркус"
lastName = "Перссон"
print(firstName + lastName)
```

В этом примере функция `print()` выведет: `"МаркусПерссон"`. Если вам нужно, чтобы между словами был пробел, его можно добавить с помощью той же операции сложения:

```
print(firstName + " " + lastName)
```

В Python часто существует несколько вариантов решения одной задачи. В нашем случае тоже есть второй способ разделить слова пробелом — поставить между строковыми аргументами запятую:

```
print(firstName, lastName)
```

Обе команды напечатают `"Маркус Перссон"`. А еще в Python можно склеивать захардкоженные строки с другими строковыми переменными. Для этого просто запишите их через знак плюс:

```
print("Его зовут " + firstName + " " + lastName)
```

В результате на экране появится: `"Его зовут Маркус Перссон"`.

Склейка фрагментов текста — полезная операция, но иногда нужно склеить строку с данными другого типа, например с целым числом. Сложность здесь в том, что Python не позволит объединить строку и число. В этом случае нужно сначала преобразовать число в строку. Давайте посмотрим, как это делается.

Преобразование числа в строку

В Python часто бывает нужно преобразовать данные одного типа в другой. Представьте, что вы храните количество золотых яблок в целочисленной переменной `myGoldenApples`. Разумеется, вам хочется рассказать друзьям, как много у вас яблок, ведь это редкий фрукт. Вы можете напечатать сообщение вроде `"Мой не совсем секретный запас яблок"` и добавить к нему значение переменной `myGoldenApples`. Однако, чтобы Python позволил вам это сделать, нужно преобразовать хранящееся в переменной целое число в строку.

My golden apples — мои золотые яблоки

Функция `str()` преобразовывает данные нестроковых типов, таких как целые и вещественные числа, в строки. Для этого укажите значение, которое вы хотите преобразовать, внутри скобок функции.

Str (*сокр.* от string) — строка

Вернемся к нашим яблокам. Предположим, вы присвоили переменной `myGoldenApples` значение `2` и теперь хотите, чтобы Python воспринимал это `2` как строку, а не как число. Вот что можно для этого ввести:

```
print("Мой не совсем секретный запас яблок: " + str(myGoldenApples))
```

Эта команда выведет на экран: `"Мой не совсем секретный запас яблок: 2"`.

Вещественные числа тоже можно преобразовывать в строки. Предположим, игрок съел половинку золотого яблока, и теперь в переменной `myGoldenApples` находится число 1.5. Как и в случае с числом 2, `str(myGoldenApples)` преобразует 1.5 в строку, которую вы можете использовать в сообщении.

Давайте-ка повеселимся, превращая числа в строки и склеивая их!

Склейка целых и вещественных чисел

Склеивать можно только строки. Если вы попробуете склеить числа или другие данные, Python просто сложит их, ведь знак плюс означает еще и сложение. Перед склейкой числовые значения необходимо превратить в строковые.

Для этого воспользуемся функцией `str()`:

```
print(str(19) + str(84))
```

Поскольку мы указали, что числа 19 и 84 нужно превратить в строки, а затем склеить, эта команда напечатает 1984, а не 103 (сумму чисел 19 и 84).

В пределах одной команды можно склеивать любое количество строк. Например:

```
print(str(19) + str(84) + " год")
```

Эта команда напечатает: `1984 год`.

Ну что ж, вы научились склеивать строки, теперь пора применить ваши знания при выполнении очередной миссии!

МИССИЯ 13. ДОБАВЬТЕ ПЕРЕД СООБЩЕНИЯМИ ИМЕНА

Если в чат пишут несколько человек, бывает трудно понять, кто какое сообщение написал. Очевидное решение — указывать в начале сообщения имя пользователя. В ходе этой миссии вам предстоит доработать код из миссии 12, добавив перед сообщениями имена.

User chat — чат пользователей

Откройте файл *messageInput.py* в IDLE и сохраните его в папке *strings* под новым именем *userChat.py*. Затем перед запросом самого сообщения добавьте код, запрашивающий имя пользователя. Сообщения в чате Minecraft должны выглядеть примерно так: `"Анка: Мне нужен динамит"`. Чтобы выполнить эту миссию, вам понадобится склейка.

Найдите в коде такую строку:

```
message = input("Введите сообщение: ")
```

Строкой выше нужно добавить еще одну переменную с именем `username`, присвоив ей значение `input("Введите имя пользователя: ")`. После этого отыщите в коде такую строку:

Username — имя пользователя

```
mc.postToChat(message)
```

При помощи склейки объедините строки `username` и `message` внутри функции `postToChat()`. Между этими строками добавьте значение `": "`, чтобы имя пользователя и текст сообщения были разделены двоеточием и пробелом. На рис. 4.3 показано, как должен выглядеть результат работы программы.

Сохраните программу и запустите ее. Сначала в окне консоли Python появится запрос имени пользователя. Введите свое имя.

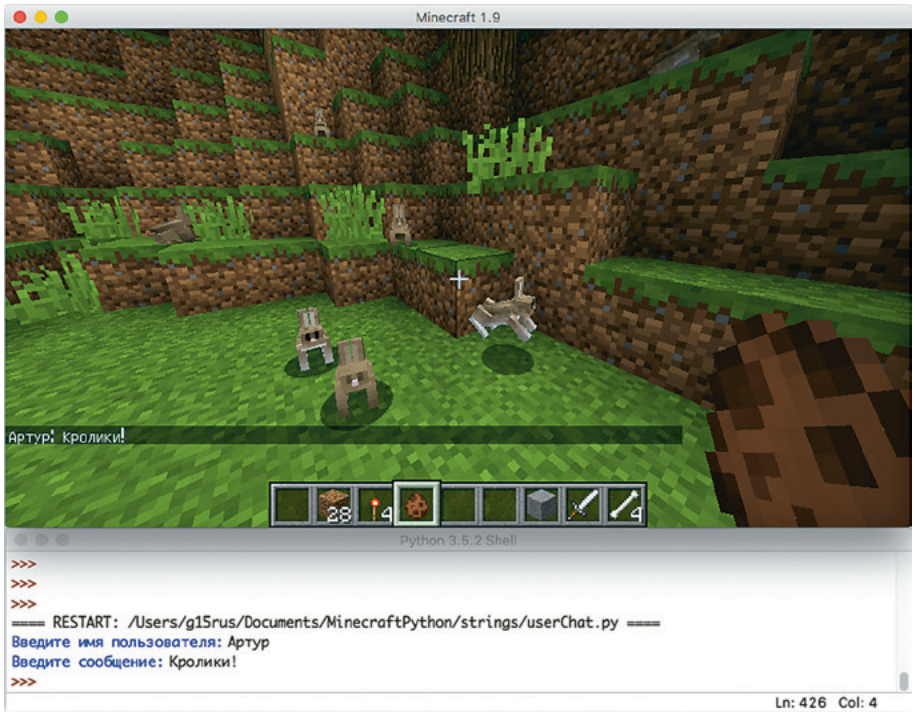

Рис. 4.3. Теперь в сообщениях будет отображаться имя, которое вы ввели

БОНУСНОЕ ЗАДАНИЕ: БЕЗЫМЯННЫЙ ПОЛЬЗОВАТЕЛЬ

Что будет, если в ответ на запрос имени ничего не вводить, а просто нажать ENTER? Как думаете, почему происходит именно это?

Преобразование строки в целое число

Int (*сокр.* от integer) — целое число

Подобно тому как функция `str()` преобразует данные нестроковых типов в строки, функция `int()` преобразует строковые данные в целые числа.

Функцию `int()` удобно использовать вместе с `input()`. Функция `input()` возвращает ввод пользователя в виде строки, а если необходимо работать с введенным значением как с числом, его можно преобразовать в целое число при помощи функции `int()`.

Cans of tuna per cats — банок тунца съела кошка

Работает это следующим образом. Предположим, мы уже присвоили переменной `cansOfTunaPerCat` целочисленное значение, а теперь хотим, чтобы программа спросила, сколько у пользователя кошек, и подсчитала, сколько всего банок тунца было съедено.

```
cansOfTunaPerCat = 4
cats = input("Сколько у вас кошек? ")
cats = int(cats)
dailyTunaEaten = cats * cansOfTunaPerCat
```

Cats — кошки

Преобразовать число кошек, которое пользователь ввел в виде строки, в целое число можно с помощью единственной команды `cats = int(cats)`, указав одну функцию в качестве аргумента другой:

```
cats = int(input("Сколько у вас кошек? "))
dailyTunaEaten = cats * cansOfTunaPerCat
```

Зная, как превращать строки в целые числа, вы можете запрашивать в ваших Minecraft-программах идентификаторы блоков. Например, для того чтобы позволить пользователю выбрать, из какого материала он хочет возвести постройку.

МИССИЯ 14. ПОЗВОЛЬТЕ ПОЛЬЗОВАТЕЛЮ ВЫБРАТЬ ТИП БЛОКА

В Minecraft есть множество разных блоков. Часть из них доступны в творческом режиме игры, но некоторые выбрать нельзя. Однако в Python-программах можно использовать блоки всех типов, создавая их с помощью функций `setBlock()` или `setBlocks()`.

Эти функции вы встречали в главе 3, но тогда тип блока был жестко прописан в коде программы. Пока программа выполнялась, поменять его было невозможно. Теперь же в вашем распоряжении есть функция `input()`, с помощью которой можно выбирать тип блока при каждом запуске программы. Например, при первом запуске вы можете создать блок шерсти, а при следующем — блок железной руды.

В ходе этой миссии вам предстоит написать программу, которая предлагает пользователю выбрать тип создаваемого блока. Введите код листинга 4.3 в новый файл IDLE и сохраните его в папку *strings* под именем *blockInput.py*.

Block input — ввод блока

blockInput.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
❶ blockType = # Добавьте сюда вызов функции input()
pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
mc.setBlock(x, y, z, blockType)
```

Листинг 4.3. Код для установки блока в позиции игрока

Эта программа устанавливает блок в текущей позиции игрока. Измените код так, чтобы значение переменной `blockType` в строке ❶ задавалось с помощью функции `input()`. При этом пользователю должно быть ясно, что программа запрашивает именно идентификатор блока. Для этого передайте в функцию `input()` строку с соответствующим сообщением (иначе Python будет ожидать ввода без каких-либо пояснений).

Как вы уже знаете, `input()` возвращает введенные данные в виде строки, а чтобы получить вместо нее целое число, нужно воспользоваться функцией `int()`. Поэтому команда ввода идентификатора блока должна выглядеть так:

```
blockType = int(input("Введите тип блока: "))
```

Сохраните измененный код, запустите его и введите любой идентификатор блока. На рис. 4.4 показан результат работы программы.

БОНУСНОЕ ЗАДАНИЕ: БОЛЬШЕ ИНТЕРАКТИВНОСТИ

Вы можете запрашивать у пользователя что угодно! Сейчас программа создает блок в текущей позиции игрока. Разберитесь, как с помощью `input()` запросить его координаты. Если же вас тянет на приключения, телепортируйте игрока в позицию с координатами, заданными через `input()`.

Рис. 4.4. Теперь я могу создать блок любого типа!

Обработка исключений

Чтобы программа могла восстановить работу после обнаружения ошибки и продолжить выполнять ее с того же места, в Python существует *обработка исключений*. Это один из способов контроля за пользовательским вводом.

Предположим, ваша программа запрашивает целое число, а пользователь вводит в ответ строку. По умолчанию в таких случаях программа выводит сообщение об ошибке, а затем прекращает работу.

Обработка исключений поможет вам разобраться с возникшей проблемой без перезапуска программы.

Try–except — попытка-исключение

Для этого в Python имеется конструкция `try-except`. С ее помощью можно вывести на экран информационное сообщение вроде `"Введите число"` и дать пользователю возможность повторить ввод, не прекращая работу программы.

Конструкция состоит из двух частей: `try` и `except`. В первую часть — `try` — мы помещаем код, который выполняется в штатных условиях (когда пользователь вводит все данные правильно). Вторая часть конструкции — `except` — запустится, если пользователь допустит ошибку.

Вот фрагмент кода, который спрашивает, сколько у вас солнечных очков (к слову, у меня три пары):

```
try:
❶    noOfSunglasses = int(input("Сколько у вас солнечных очков? "))
except:
❷    print("Неверно. Введите число.")
```

Эта программа запрашивает у пользователя число. Если же вместо цифр вы введете буквы или знаки, она напечатает: `"Неверно. Введите число"`. Ошибка возникает потому, что функция `int()` может обрабатывать только строки, содержащие целые числа ❶. Иначе говоря, если вы введете число, код будет работать как обычно, а если что-то другое, возникнет ошибка.

Кстати, вы заметили тут кое-что необычное? Это первый пример кода, в котором мы используем команды с отступами — то есть ставим перед строками несколько пробелов. Я подробно расскажу об отступах в главах, посвященных конструкции `if` и циклу `for`, а пока просто старайтесь вводить код в точности так, как он напечатан в книге, со всеми пробелами.

Обычно при возникновении ошибки Python отображает маловразумительное сообщение, не объясняющее пользователю, как устранить проблему. Однако с помощью `try-except` вы можете вывести на экран понятную инструкцию. Если при запросе количества солнечных очков пользователь ничего не ввел и нажал ENTER, в обычной ситуации это привело бы к ошибке, однако наш код находится под защитой `try-except`, поэтому на экране появится строка, предлагающая ввести число ❷.

Внутрь конструкции `try-except` можно помещать какой угодно код (даже другие конструкции `try-except`). Этим мы и займемся в ходе следующей миссии!

МИССИЯ 15. ДОПУСКАЮТСЯ ТОЛЬКО ЧИСЛА

Помните код, который вы писали для миссии 14? При вводе пользователем целого числа программа работала как задумано — создавала блок. Однако ввод строки приводил к остановке программы с выводом сообщения об ошибке, как на рис. 4.5.

Такое сообщение об ошибке понятно лишь Python-программистам. Что будет, если человек, никогда не имевший дела с Python, увидит его? Разумеется, он ничего не поймет, поэтому ваша задача — вывести на экран текст, объясняющий суть проблемы.

Block input fix — ввод блока фиксированный

Откройте программу *blockInput.py*, созданную для миссии 14, и сохраните ее в папку *strings* под именем *blockInputFix.py*.

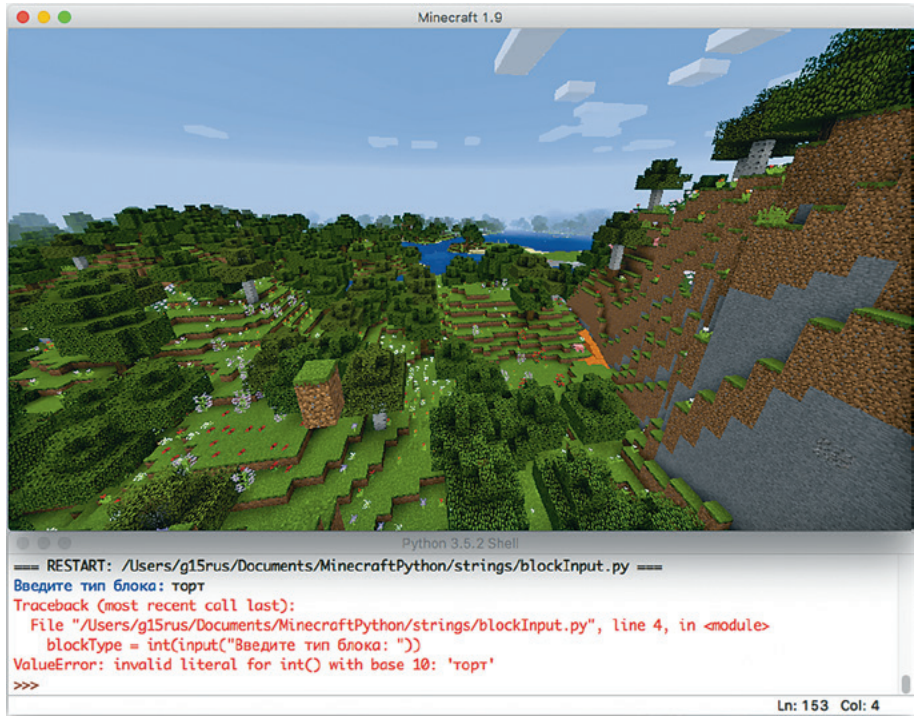

Рис. 4.5. Слово «торт» не число, поэтому программа не создает блок

Измените программу так, чтобы она запрашивала тип блока при помощи конструкции `try-except`. Найдите последнюю строку кода — она выглядит вот так:

```
mc.setBlock(x, y, z, blockType)
```

Добавьте перед ней строку с командой `try`, а перед вызовом функции `mc.setBlock()` поставьте четыре пробела. Затем добавьте выше вызова `setBlock()` еще одну строку кода с запросом ввода: `blockType = int(input("Введите тип блока: "))`.

Строкой ниже вызова `setBlock()` введите команду `except` и добавьте под ней код, отправляющий в чат Minecraft сообщение о том, что тип блока должен быть числом. Например, такое: `"Это не число! В следующий раз введите число"`. Вот как должен выглядеть доработанный код (обратите внимание, что строки ❶ и ❷ начинаются с четырех пробелов, то есть с отступов):

```
try:
❶     blockType = int(input("Введите тип блока: "))
      mc.setBlock(x, y, z, blockType)
except:
❷     mc.postToChat("Это не число! В следующий раз введите число.")
```

Функция `int()` в строке ❶ ожидает значение, которое можно преобразовать в целое число, и при вводе чего-либо, кроме цифр, в ней возникнет ошибка. Но, поскольку мы добавили в код конструкцию `try-except`, вместо обычного сообщения об ошибке в чате появится сообщение с просьбой вводить только числа ❷.

Сохраните программу *blockInputFix.py* и запустите ее, дабы восхититься плодами рук своих. Результат должен выглядеть как на рис. 4.6.

Рис. 4.6. Такое сообщение об ошибке ввода выглядит гораздо понятнее

МИССИЯ 16. ОТЧЕТ О ПЕРЕМЕЩЕНИЯХ

Для прохождения последней миссии из этой главы вам понадобятся все знания о переменных и математических операциях, которые вы получили из глав 2 и 3, а также умение отправлять сообщения в чат. Ваша

задача — отследить, сколько прошел игрок за 10 секунд, и вывести эту информацию в чат.

Напоминаю: чтобы программа приостановила работу на определенное количество секунд, можно использовать такую конструкцию:

```
import time  # Добавьте эту строку в начало программы
time.sleep(30)  # 30-секундная пауза
```

Итак, для начала откройте IDLE и введите код листинга 4.4.

sprint.py

```
import time
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

❶ pos1 = mc.player.getTilePos()
  x1 = pos1.x
  y1 = pos1.y
  z1 = pos1.z

  time.sleep(10)

❷ pos2 = mc.player.getTilePos()
  x2 = pos2.x
  y2 = pos2.y
  z2 = pos2.z

  # Находим разницу между начальным и конечным положением
❸ xDistance = x2 - x1
  yDistance =
  zDistance =

  # Выводим результат в чат
❹ mc.postToChat("")
```

Листинг 4.4. Код для определения по трем координатам дистанции, которую игрок преодолеет за 10 секунд

Y-distance — расстояние по оси *y*

Z-distance — расстояние по оси *z*

Давайте подробно разберем этот код. В строке ❶ программа узнаёт начальную позицию игрока, затем ждет 10 секунд и в строке ❷ узнаёт его конечную позицию. Чтобы завершить эту программу, требуется вычислить разницу между начальной и конечной позициями игрока. Для этого под строкой ❸ присвойте переменным `yDistance` и `zDistance` корректные значения. Чтобы упростить вашу задачу, я уже нашел значение

переменной `xDistance` — это `x2 - x1`. Выражения для `yDistance` и `zDistance` должны выглядеть так же, но вместо `x1` и `x2` там будут стоять другие переменные.

X-distance — расстояние по оси *x*

В самой последней строке кода ❹ нужно вывести сообщение в чат. Оно может иметь такой вид: `"Игрок переместился на x: 45, y: -14 и z: -21"`. Для этого используйте внутри функции склейку строк и значения переменных `xDistance`, `yDistance` и `zDistance`.

Сохраните программу в папку *strings* под именем *sprint.py* и запустите ее. Результат работы должен выглядеть как на рис. 4.7.

Spint — бег на короткую дистанцию

Рис. 4.7. Программа выводит расстояние, на которое переместился игрок

Если программа работает нормально, но вам сложно быстро переключиться из IDLE в окно игры, добавьте в начале кода трехсекундный обратный отсчет. Оставшееся число секунд выведите в чат.

БОНУСНОЕ ЗАДАНИЕ: РАССТОЯНИЕ ОДНИМ ЧИСЛОМ

Сейчас программа отображает расстояние для каждой оси координат. Вместо этого вычислите и выведите на экран расстояние от начальной до конечной точки в виде одного целого числа. Подсказка: для этого вам понадобится знание теоремы Пифагора. Если вы пока не понимаете, как это сделать, не волнуйтесь: подобные расчеты еще встретятся нам в коде миссии 21 (с. 124).

Что вы узнали

Поздравляю, в этой главе вы узнали довольно много! Вы научились создавать строки, выводить их на экран с помощью функции `print()` и склеивать несколько строк в одну. Вы написали программы, принимающие ввод пользователя, узнали, как преобразовывать один тип данных в другой и обрабатывать исключения, а также отправили несколько сообщений в чат Minecraft.

В главе 5 вы научите программу запускать разные команды в зависимости от заданных условий.

5

«ИСТИНА» И «ЛОЖЬ» БУЛЕВЫХ ЗНАЧЕНИЙ

Мы все время задаем себе и другим вопросы, на которые можно ответить только «да» или «нет». На улице дождь? У меня слишком длинные волосы? Получив ответ, мы решаем, что делать дальше: брать с собой зонтик или нет, стричь волосы или нет. Подобный принцип действий лежит и в основе программирования. В этой главе мы выясним, как Python работает с закрытыми вопросами и ответами на них.

В программировании вопросы такого типа чаще всего используют, когда нужно сравнить два значения. Одно значение больше другого или меньше? Или они равны? Такие вопросы называются *условиями*, а для ответа на них вместо «да» и «нет» используются ключевые слова `True` и `False`. Предположим, вы хотите спросить: «У меня больше золотых блоков, чем у друга?» Чтобы превратить такой вопрос в условие, понятное Python, нужно сформулировать его как утверждение, которое может быть «истиной» (True) или «ложью» (False). Например, так: «У меня запас золота больше, чем у друга».

True — истина

False — ложь

Проверка условий на «истину» и «ложь» используется в Python настолько часто, что для этого даже введен специальный тип данных, который содержит всего два значения: `True` и `False`. Вы уже знакомы с целыми числами, вещественными числами и строками. Пришло время добавить к ним новый тип данных — *булевы значения*.

Итак, есть всего два булева значения — `True` и `False`, которые используются в Python при работе с условиями. Если условие «истинно», то его значение `True`, а если «ложно» — `False`.

В этой главе нам предстоит проверить несколько условий. Для этого нам пригодятся булевы значения, операции сравнения и логические операции. После этого мы будем готовы перейти к главе 6, в которой выясним, каким образом проверка условий помогает запускать в различных ситуациях разные фрагменты кода.

Булевы значения: основы

Булевы значения похожи на положения выключателя света: либо свет включен (значение `True`), либо выключен (`False`). Для случая, когда свет включен, можно объявить булеву переменную так:

```
light = True
```

Light — свет

Здесь мы присваиваем переменной `light` значение `True`. А чтобы выключить свет, можно присвоить значение `False`:

```
light = False
```

Всегда пишите ключевые слова `True` и `False` с заглавной буквы, иначе Python не поймет, что это булевы значения, и вместо выполнения команды выбросит исключение!

В ходе следующей миссии булевы значения помогут нам предотвратить разрушение блоков.

МИССИЯ 17. ОТСТАВИТЬ РАЗРУШЕНИЕ БЛОКОВ!

Блоки в мире Minecraft легко разрушаются. Это удобно, если вам нужно добыть ресурсы. Однако представьте себе, что вы потратили уйму времени на постройку великолепной конструкции, а потом ненароком ее задели и разрушили! В этой миссии вашей задачей будет сделать блоки неуязвимыми для ударов.

Setting — настройки

World immutable — неизменяемый мир

С помощью команды `setting("world_immutable", True)` вы можете сделать блоки *неизменяемыми*, так что разрушить их будет невозможно. Здесь `setting()` — это функция наподобие `setTilePos()` и `setPos()`. В листинге 5.1 показано, как ее использовать.

immutableOn.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

mc.setting("world_immutable", True)
```

Листинг 5.1. Код, защищающий блоки от повреждений

У функции `setting()` есть набор опций, которые можно включать, присваивая им значение `True`, и одна из таких опций — `"world_immutable"`. Чтобы ее включить, нужно указать внутри скобок название опции, а после, через запятую, булево значение `True`.

Введите код листинга 5.1 в IDLE и сохраните файл под именем *immutableOn.py* в новой папке *booleans*. После запуска программы игрок не сможет повредить блоки руками (рис. 5.1). Но что делать, если вам все же понадобится что-нибудь разрушить? Скопируйте код в новый файл и измените его, снова разрешив игроку разрушать блоки. (Подсказка — используйте булево значение!) Сохраните программу в папке *booleans* под именем *immutableOff.py*.

Immutable on — защита от разрушений включена

Booleans — булевы значения

Immutable off — защита от разрушений выключена

Рис. 5.1. Как игрок ни старается, блоки остаются целыми!

Склейка строк и булевых значений

Чтобы склеить булево значение со строкой, его так же, как целое или вещественное число, нужно преобразовать в строку. Если вы хотите вывести булево значение с помощью команды `print()`, воспользуйтесь функцией `str()`:

```
>>> agree = True
>>> print("Согласен ли я: " + str(agree))
Согласен ли я: True
```

В переменной `agree` хранится булево значение. Во второй строчке кода оно преобразуется в строку функцией `str(agree)`, склеивается со строкой `"Согласен ли я: "` и выводится на экран.

Agree — согласен

Операции сравнения

Все мы отлично умеем сравнивать два значения. Нам понятно, что 5 больше 2, числа 8 и 8 равны, а 6 и 12 не равны. Компьютер тоже неплохо с этим справляется, однако ему нужно объяснить, о каком сравнении идет речь. А для этого — указать ту или иную *операцию сравнения*. Например, вы можете проверить, больше ли первое значение, чем второе, а можете проверить, меньше ли оно, чем второе.

В Python существует шесть операций сравнения:

- равно (==)
- не равно (!=)
- меньше (<)
- меньше или равно (<=)
- больше (>)
- больше или равно (>=)

Все они возвращают булево значение (`True` или `False`), которое отвечает на вопрос, выполняется данное условие или нет. Давайте познакомимся с операциями сравнения поближе!

«Равно»

Чтобы выяснить, равны ли два значения, используют операцию «равно» (==). Если значения совпадают, операция вернет `True`, а если не совпадают — `False`.

Присвоим значения двум переменным и сравним их с помощью операции «равно»:

```
>>> length = 2
>>> width = 2
>>> length == width
True
```

Length — длина
Width — ширина

Вернется `True`, поскольку значения переменных `length` и `width` совпадают.

Если же они будут разными, операция сравнения вернет `False`:

```
>>> length = 4
>>> width = 1
>>> length == width
False
```

Операцию сравнения «равно» можно применять к данным всех типов: строкам, целым и вещественным числам, булевым значениям.

Обратите внимание, что, сравнивая `length` и `width`, я ввожу ==, а не =, ведь одиночный знак равенства уже использовался для присваивания значений переменным. Чтобы Python мог отличить *сравнение* (проверку значений на равенство) от *присваивания* (объявления переменной), при сравнении используйте двойное равно. Постарайтесь запомнить это различие, чтобы избежать ошибок. Но если они возникнут, не принимайте их близко к сердцу. Порой даже я случайно ввожу = вместо ==!

МИССИЯ 18. ИГРОК В ВОДЕ?

Итак, наша задача — написать программу, которая при помощи операций сравнения определит, находится ли игрок в воде. Результаты своей работы программа должна отправлять в чат.

Чтобы узнать тип стоящего в указанном месте блока, воспользуемся функцией `getBlock()`. Она принимает три аргумента — координаты *x*, *y*, *z* — и возвращает тип блока в виде целого числа.

```
blockType = mc.getBlock(10, 18, 13)
```

С помощью этой команды я сохраняю значение, полученное из функции `mc.getBlock(10, 18, 13)`, в переменной `blockType`. Если в координатах (10, 18, 13) окажется блок «арбуз», в `blockType` попадет число 103.

Block type — тип блока

Давайте испытаем функцию `getBlock()` в деле. Код листинга 5.2 проверяет, находится ли игрок на суше.

swimming.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getPos()
x = pos.x
y = pos.y
z = pos.z

blockType = mc.getBlock(x, y, z)
mc.postToChat(blockType == 0)
```

Листинг 5.2. Этот код проверяет тип блока в позиции игрока

Итак, я определяю все три координаты игрока и передаю их в функцию `getBlock()`. Значение, полученное из `mc.getBlock(x, y, z)`, я сохраняю в переменной `blockType`. Выражение `blockType == 0` проверяет, равен ли тип блока идентификатору воздуха. Если это так (то есть игрок стоит где-то на суше), выражение вернет значение `True`, которое отправится в чат. Если же тип блока не равен идентификатору воздуха, выражение вернет `False`, которое также появится в чате. Это будет означать, что игрок находится в воде или, возможно, в зыбучих песках.

Swimming — плавание

Введите в окне программы код листинга 5.2 и сохраните файл в папке *booleans* под именем *swimming.py*. Измените код так, чтобы он проверял, находится ли игрок в воде (идентификатор блока 9), и запустите программу.

Заведите игрока в воду и снова запустите программу. Теперь в чате должно появиться слово `True`. Если же игрок на суше, в чате появится `False`.

Результат работы программы должен выглядеть как на рис. 5.2.

Рис. 5.2. По-моему, игрок в воде. И Python того же мнения

БОНУСНОЕ ЗАДАНИЕ: ОН ЛЕТИТ!

Немного изменив код, можно проверить, является ли воздухом блок *под* ногами игрока. Так бывает, когда он летит или находится в прыжке. Разберетесь, как это сделать?

«Не равно»

Операция сравнения «не равно» противоположна операции «равно». Вместо того чтобы проверять два значения на равенство, она проверяет их на неравенство. Если значения различаются, операция возвращает `True`, а если совпадают — `False`.

Предположим, вы хотите убедиться, что некий прямоугольник не является квадратом. Если это так, ширина и высота прямоугольника должны различаться. А значит, необходимо составить выражение, которое проверит, действительно ли ширина и высота прямоугольника не равны:

```
>>> width = 3
>>> length = 2
>>> width != length
True
```

Такая проверка заключена в выражении `width != length`. В результате мы получаем ответ `True` — то есть значения переменных `width` и `length` не равны.

Однако, если эти значения будут одинаковыми, то же самое выражение вернет `False`:

```
>>> width = 3
>>> length = 3
>>> width != length
False
```

Операция «не равно», так же как и операция «равно», подходит для сравнения строк, целых и вещественных чисел, а также булевых значений.

МИССИЯ 19. ИГРОК В ВОЗДУХЕ?

Предположим, вам хочется убедиться, что игрок стоит в воде, лаве, гравии или где угодно еще, но не на поверхности земли. В ходе миссии 18 вы проверяли, находится ли в текущей позиции игрока блок воздуха, а затем — находится ли там блок воды.

Конечно, можно сделать несколько копий кода из миссии 18, заменив в каждой из них тип блока «воздух» на лаву, землю, гравий и т. д. Однако это ужасно утомительно. Лучше использовать операцию «не равно». Так вы проверите все варианты сразу!

Not air — не воздух

Откройте программу *swimming.py* и сохраните ее в папке *booleans* под именем *notAir.py*. Затем сотрите последнюю строку и добавьте в этом месте код листинга 5.3.

notAir.py

```
❶ notAir = blockType == 0
mc.postToChat("Игрок не находится в воздухе: " + str(notAir))
```

Листинг 5.3. Добавьте этот фрагмент кода в конец программы swimming

Последняя строка этого кода выводит в чат сообщение, которое начинается словами `"Игрок не находится в воздухе:"`. Результат сравнения ❶ попадает в переменную `notAir`. Если сравнение `blockType == 0` вернет `True`, в `notAir` также окажется `True`, а если `False` — то `False`.

Однако выражение ❶ не совсем верное. Сейчас оно с помощью операции «равно» (==) проверяет, совпадает ли значение переменной `blockType` с идентификатором воздушного блока. Вместо этого нужно убедиться, что идентификатор блока в позиции игрока не соответствует блоку «воздух». Замените операцию «равно» на «не равно». Тогда программа сможет проверить, находится ли в позиции игрока воздушный блок или какой-то другой.

Рис. 5.3. Игрок находится в воде, которая не воздух

Запустите программу и убедитесь, что она работает правильно не только когда игрок находится на суше, но и когда он стоит в воде, лаве, гравии, песке или телепортировался в толщу земли. Сообщение, которое должно появиться в чате, если операция возвращает `True`, показано на рис. 5.3.

«Больше» и «меньше»

Если вам нужно узнать, больше ли одно число другого, используйте операцию сравнения «больше». Она вернет `True`, если значение слева от оператора будет больше, чем значение справа. Если же значение слева будет меньше или равно значению справа, операция вернет `False`.

Предположим, у нас есть вагонетка, в которой можно перевозить не более 99 обсидиановых блоков. Назовем число, превышающее это количество блоков на один, предельной емкостью вагонетки. До тех пор, пока предельная емкость больше, чем количество обсидиановых блоков, эти блоки можно перевезти:

```
>>> limit = 100  # предельная емкость вагонетки
>>> obsidian = 99  # количество блоков «обсидиан»
>>> limit > obsidian
True
```

Превосходно! Вагонетка с предельной емкостью 100 вполне сможет перевезти 99 блоков, а значит, сравнение `limit > obsidian` вернет `True`. Но что произойдет, если добавить к этой горе блоков еще один?

```
>>> limit = 100
>>> obsidian = 100
>>> limit > obsidian
False
```

О нет, мы достигли предельной емкости! Теперь выражение возвращает `False`: 100 не больше, чем 100 (значения равны). Значит, весь этот обсидиан в вагонетке не перевезти.

Операция сравнения «меньше» работает таким же образом.

Чтобы фургон смог проехать под мостом, его высота должна быть меньше высоты моста:

```
>>> vanHeight = 8  # высота фургона
>>> bridgeHeight = 12  # высота моста
>>> vanHeight < bridgeHeight
True
```

В данном случае фургон проедет, ведь его высота (8 блоков) меньше высоты моста (12 блоков). А теперь представьте, что через некоторое время этот фургон очутился перед другим, более низким мостом:

```
>>> vanHeight = 8
>>> bridgeHeight = 7
>>> vanHeight < bridgeHeight
False
```

Число 8 не меньше 7, поэтому сравнение вернет `False`.

«Больше или равно» и «меньше или равно»

Подобно операции «больше», операция «больше или равно» проверяет, больше ли одно значение другого. Однако, в отличие от «больше», она возвращает `True`, даже когда значения равны.

Представьте, что я раздаю наклейки тем, кто пришел на презентацию моей потрясающей Python-программы. Нужно проверить, хватит ли всем наклеек:

```
>>> stickers = 30  # наклейки
>>> people = 30  # люди
>>> stickers >= people
True
```

Да, наклеек хватит: 30 равняется 30, поэтому сравнение `stickers >= people` вернет `True`. Но вдруг появляется опоздавший на презентацию друг, который тоже хочет себе наклейку. Теперь наклейки нужны 31 человеку:

```
>>> stickers = 30
>>> people = 31
>>> stickers >= people
False
```

Не хватает мне наклеек: 30 не больше и не равно 31. Похоже, друга придется огорчить.

Итак, теперь вам по силам почти любое сравнение. Оставаясь в окне консоли Python, самостоятельно опробуйте операцию «меньше или равно» (<=) и посмотрите, как она сравнивает значения.

Операции сравнения «больше», «больше или равно», «меньше» и «меньше или равно» не работают со строками, но годятся для целых и вещественных чисел и булевых значений.

МИССИЯ 20. ИГРОК НАД ЗЕМЛЕЙ?

Как вы знаете, *y*-координата игрока соответствует высоте, на которой он находится. Блоки также привязаны к координатам, что позволяет нам узнавать их тип посредством функции `getBlock()` и устанавливать в конкретном месте с помощью `setBlocks()`.

Значит, мы вполне можем узнать высоту самого верхнего блока, который находится в координатах *x* и *z*. В этом нам поможет функция `getHeight()`. Представьте, что у вас есть башня из блоков и вы знаете ее координаты *x* и *z*. Передав их в функцию `getHeight()` в качестве аргументов, вы сможете узнать высоту всей башни. Эта функция возвращает *y*-координату самого верхнего блока.

Get height — определить высоту

aboveGround.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
pos = mc.player.getTilePos()  # позиция игрока
x = pos.x
y = pos.y
z = pos.z
highestBlockY = mc.getHeight(x, z)  # самый высокий блок
mc.postToChat(highestBlockY)
```

Листинг 5.4. Код, который позволяет узнать y-координату самого высокого блока в x- и z-координатах игрока

Этот код получает текущую позицию игрока, узнает *y*-координату самого высокого блока в его *x*- и *z*-координатах и отправляет это значение в чат.

Добавив в код операцию сравнения «больше или равно», вы сможете узнать, находится ли игрок над землей. Что ж, приступайте!

Введите в окне программы код листинга 5.4 и сохраните его под именем *aboveGround.py*. Измените код так, чтобы он проверял, превышает ли *y*-координата игрока значение переменной `highestBlockY`. Затем добавьте код для вывода результата проверки в чат в виде строки `"Игрок над землей: True/False"`.

Above ground — над землей

Highest block y — самый высокий блок по оси *y*

Напоминаю, что результат сравнения можно сохранить в переменной. Если бы я решил проверить, действительно ли y больше или равно 10, и сохранить результат в переменной `highEnough`, я написал бы так: `highEnough = y >= 10`.

High enough — достаточно высоко

Внесите эти изменения в код и запустите программу. Пример ее работы для случая `False` показан на рис. 5.4.

Рис. 5.4. Игрок сейчас в пещере, так что Python совершенно прав, сообщая, что он не над землей

МИССИЯ 21. ДАЛЕКО ЛИ ИГРОК ОТ ДОМА?

Путешествуя по миру Minecraft, легко заблудиться и забыть, где находится дом. Так можно бродить часами и в итоге обнаружить, что оказался еще дальше от него, чем был в начале.

Как же определить, близко ли дом? С помощью всего нескольких строк кода можно узнать, на каком расстоянии от любой точки мира Minecraft (в нашем случае — от дома) находится игрок. А затем сравнить это расстояние с определенным количеством блоков. Если игрок отошел от дома не дальше чем на 40 блоков, будем считать, что дом близко.

Давайте же создадим такую Python-программу! В этой миссии нужно вычислить, насколько игрок удалился от дома. И основную часть работы возьмет на себя код листинга 5.5.

farFromHome.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import math
❶ homeX = 10
homeZ = 10
pos = mc.player.getTilePos()
x = pos.x
z = pos.z
```

```
❷ distance = math.sqrt((homeX - x) ** 2 + (homeZ - z) ** 2)
❸ mc.postToChat(distance)
```

Листинг 5.5. Код, выводящий в чат расстояние от игрока до дома

Предполагается, что дом игрока имеет координаты $x = 10$ и $z = 10$, которые хранятся в переменных `homeX` и `homeZ` ❶, а y-координата нам в данном случае не нужна. С помощью функции `getTilePos()` я получаю позицию игрока и присваиваю значения переменным `x` и `z`.

Чтобы вычислить расстояние, воспользуемся формулой из *теоремы Пифагора*. Она позволяет узнать длину гипотенузы прямоугольного треугольника по двум его катетам, а нам пригодится, чтобы рассчитать расстояние между двумя точками. Возможно, вы помните эту формулу из уроков математики: $a^2 + b^2 = c^2$, где a и b — это два катета прямоугольного треугольника, а c — его гипотенуза. На рис. 5.5 показано, как рассчитать расстояние от игрока до дома при помощи формулы $c = \sqrt{a^2 + b^2}$. В строке ❷ кода находим длину гипотенузы (на рисунке — c) и присваиваем ее переменной `distance`.

Сохраните код листинга 5.5 под именем *farFromHome.py* в папке *booleans*.

Home x — координата дома x

Home z — координата дома z

Distance — расстояние

Far from home — как далеко от дома

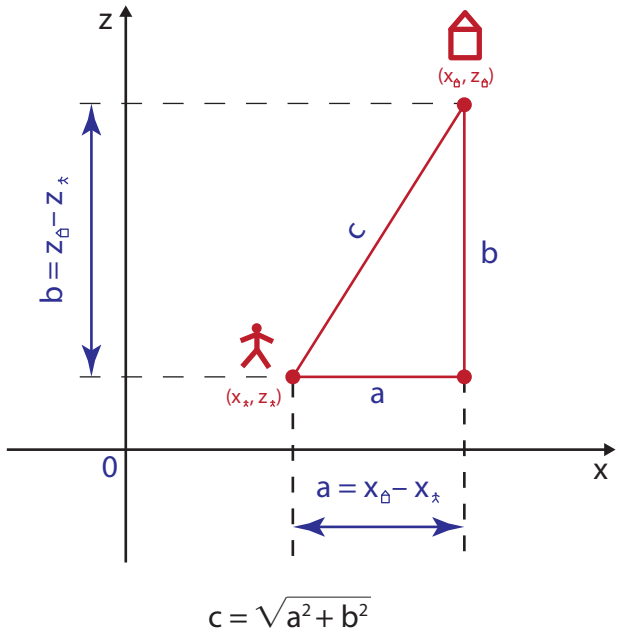

Рис. 5.5. Расчет длины гипотенузы (c) прямоугольного треугольника в системе координат x и z

Код нужно немного доработать. Воспользуйтесь операцией «меньше или равно», чтобы выяснить, действительно ли `distance` меньше или равно 40. Результат отправьте в чат в виде сообщения `"Дом игрока близко: True/False"`. Склейте строку сообщения с результатом проверки и передайте в функцию `postToChat()` ❸.

Запустите программу. Если игрок рядом с домом, она должна вывести в чат `True`, а если дальше чем на 40 блоков — `False`. Результат работы программы показан на рис. 5.6.

Рис. 5.6. Игрок стоит в пределах 40 блоков от дома. Так близко, что даже входную дверь видно!

Логические операции

Зачастую требуется объединить несколько сравнений — например, убедиться, что `True` возвращают сразу два условия (вам может понадобиться машина красного цвета стоимостью менее $10 000).

Для объединения двух и более сравнений в Python используются *логические операции,* которые еще называют *булевыми*. Подобно операциям сравнения, их можно использовать везде, где требуется булево значение. Логических операций три: `and`, `or` и `not`.

And — и

Or — или

Not — не

Логическое «и»

Если вам нужно убедиться, что выполняются два условия, используйте операцию `and`. Она вернет `True`, если оба сравнения возвращают `True`, и `False` во всех остальных случаях.

Предположим, я хочу убедиться, что человеку больше 18 лет *и* у него есть машина. Для этого я могу написать такую программу:

```
>>> age = 21
>>> ownsCar = True
>>> age > 18 ❶ and ownsCar == True ❷
True
```

Здесь я использовал логическую операцию `and`, чтобы объединить сравнения ❶ и ❷. Поскольку в данном случае человеку больше 18 лет (выражение `age > 18` возвращает `True`) и машина у него имеется (`ownsCar == True`), выражение `age > 18 and ownsCar == True` возвращает `True`.

Age — возраст

Owns car — есть машина

Но если хотя бы одно из этих условий вернет `False`, все выражение целиком также выдаст `False`. Допустим, человеку больше 18, но машины у него нет:

```
>>> age = 25
>>> ownsCar = False
>>> age > 18 and ownsCar == True
False
```

Здесь `age > 18` возвращает `True`, а `ownsCar == True` возвращает `False`, из-за чего все выражение возвращает `False`.

В таблице 5.1 показаны результаты всех возможных комбинаций для операции `and`.

Сравнение A	Сравнение B	A and B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Таблица 5.1. Результаты операции `and` при различных комбинациях `True` и `False`

МИССИЯ 22. ИГРОК ПОД ВОДОЙ?

В ходе миссии 18 (с. 117) вы проверяли, находится ли игрок в воде. Программа возвращала `True` или `False` в зависимости от того, соответствовал ли идентификатор блока в позиции игрока водяному блоку. Это означало, что в воде находятся ноги игрока, однако его голова могла быть как в воде, так и на воздухе — это не влияло на результат. Как же нам проверить, находятся ли в воде не только ноги, но и голова?

Для этого достаточно внести в программу *swimming.py* несколько небольших изменений и логическую операцию `and`. Откройте файл *swimming.py* и сохраните его под именем *underwater.py*.

Underwater — под водой

Затем сделайте следующее:

1. Добавьте еще одну переменную, хранящую идентификатор блока, который на 1 выше позиции игрока по оси *y*, а значит, находится на уровне его головы. Назовите эту переменную `blockType2`.

Block type 2 — тип блока 2

2. Проверьте, равны ли значения `blockType` и `blockType2` идентификатору блока «стоячая вода» (9).

3. Отправьте результат сравнения в чат в виде строки: `"Игрок под водой: True/False"`.

При помощи операции `and` убедитесь, что значения обеих переменных `blockType` и `blockType2` соответствуют идентификатору водного блока. Сначала проверьте нижний блок, введя `blockType == 9`. Затем — верхний: `blockType2 == 9`. И наконец, чтобы объединить эти сравнения, поставьте между ними оператор `and`: `blockType == 9 and blockType2 == 9`.

Запустите программу и проверьте, возвращает ли она верный результат для всех трех случаев (игрок на суше, ноги игрока в воде, игрок в воде с головой). Пример работы программы показан на рис. 5.7.

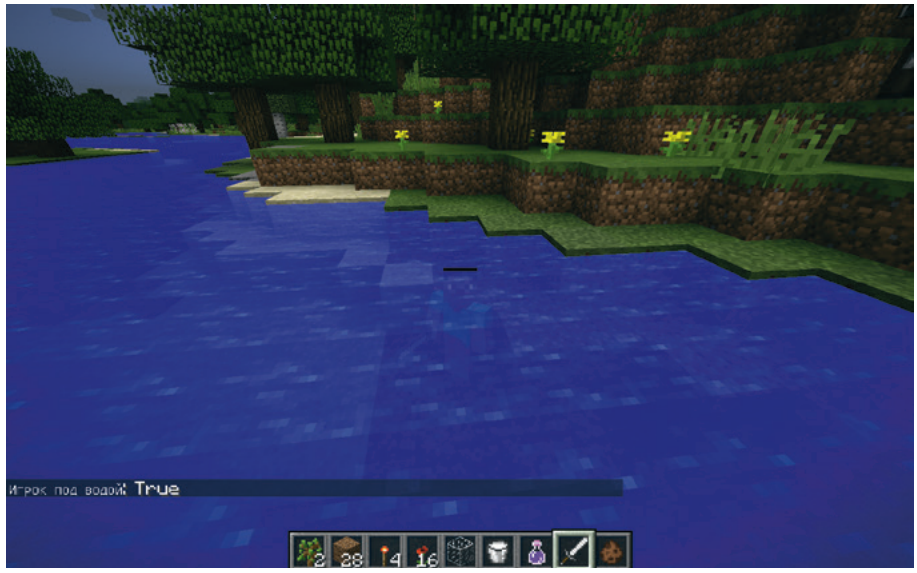

Рис. 5.7. Игрок стоит на дне водоема, находясь полностью под водой

БОНУСНОЕ ЗАДАНИЕ: ИГРОК В ТОННЕЛЕ?

Проделайте в груде камней тоннель, приведите туда игрока и убедитесь, что он находится в тоннеле. Для этого нужно проверить тип блоков под ногами игрока и над его головой.

Логическое «или»

Операция `or` работает не так, как `and`. Если мы объединим с ее помощью два сравнения и хотя бы одно из них вернет `True`, всё выражение тоже вернет `True`. И лишь когда ни одно из сравнений не вернет `True`, выражение с `or` выдаст `False`.

Предположим, я хочу завести кошку черного *или* рыжего цвета. Можно написать код, который сначала запрашивает у пользователя название цвета, а потом сравнивает его со значением `"черного"` или `"рыжего"`:

```
catColor = input("Какого цвета кошка? ")
myCatNow = catColor == "черного" or catColor == "рыжего"
print("Заведу кошку этого цвета: " + str(myCatNow))
```

Если кошка окажется черной или рыжей, я ее возьму. Однако, если у нее любой другой окрас, например серый, переменная `myCatNow` примет значение `False`, а значит, я не заведу эту кошку.

My cat now — у меня сейчас кошка

В таблице 5.2 показаны результаты всех возможных комбинаций для операции `or`.

Сравнение A	Сравнение B	A or B
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Таблица 5.2. Результаты операции `or` при различных комбинациях `True` и `False`

МИССИЯ 23. ИГРОК НА ДЕРЕВЕ?

Предыдущие программы выводили `True` или `False` в зависимости от того, находится ли в позиции игрока блок определенного типа. А как проверить, находится ли игрок на дереве? Деревья состоят из древесины

и листвы, поэтому нам нужно выяснить, действительно ли игрок стоит на древесине или листьях.

In tree — на дереве

Давайте напишем для этого код. Снова откройте файл *swimming.py* и сохраните его под новым именем — *inTree.py*.

Измените код так, чтобы он проверял тип блока на один блок ниже позиции игрока. Вам нужен положительный результат или для листвы (идентификатор 18), или для древесины (идентификатор 17). Поэтому воспользуйтесь операцией `or`. Затем отправьте в чат сообщение с итогами проверки.

Напоминаю: чтобы получить координаты блока под ногами игрока, нужно вычесть 1 из *y*-координаты игрока: `y = y - 1`.

Хоть древесина и листва в Minecraft может быть у разных видов деревьев, их идентификаторы будут одними и теми же и для ели, и для березы. (Из этого правила есть всего два исключения — это древесина и листва акации и темного дуба, им соответствуют собственные идентификаторы. Пока что просто не будем учитывать эти деревья.) Цвет же можно задать отдельной переменной, о чем мы поговорим в главе 8.

Запустите программу. Результат ее работы должен выглядеть примерно как на рис. 5.8.

Рис. 5.8. Игрок на дереве: `True`!

Логическое «не»

Операция `not` заметно отличается от `and` и `or`. Она применяется лишь к одному булеву значению, превращая его в свою противоположность.

Иными словами, операция `not` меняет `True` на `False`, а `False` — на `True`:

```
>>> not True
False
>>> not False
True
```

Операция `not` особенно полезна в сочетании с другими логическими операциями. Давайте присвоим переменной `timeForBed` значение `True`, если игрок *не* голоден (`not hungry`) и хочет спать (`sleepy`).

Time for bed — пора в кровать

```
>>> hungry = False
>>> sleepy = True
>>> timeForBed = not hungry and sleepy
>>> print(timeForBed)
True
```

Оператор `not` относится к булеву значению переменной, которая указана следом за ним. В данном случае `not` меняет значение переменной `hungry` на противоположное (обратите внимание — на переменной `sleepy` это никак не отражается). Поскольку ранее мы присвоили `hungry` значение `False`, выражение `not hungry` вернет `True`. А значение `sleepy` и так равно `True`, так что в итоге переменная `timeForBed` примет значение `True`.

МИССИЯ 24. ЭТО НЕ АРБУЗ?

Ваш игрок проголодался и хочет знать, есть ли в доме еда. Больше всего ему по вкусу арбузы, которые хранятся всегда в одном и том же месте. Но остались ли они в запасе или следует прихватить арбуз по дороге домой?

К счастью, вы изучаете Python и, немного поразмыслив, сможете узнать, есть ли в доме арбузы.

В ходе этой миссии вам предстоит создать программу, которая будет проверять, находится ли в заданных вами координатах блок «арбуз». Перед тем как приступить к написанию кода, подготовьтесь: обследуйте дом, или ферму, или другое место, где вы обычно храните арбузы, и найдите их там. Затем введите код листинга 5.6 в новый файл и сохраните его как *notAMelon.py*.

Not a melon — не арбуз

notAMelon.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

x = 10
y = 11
z = 12
❶ melon = 103
❷ blockType = mc.getBlock(x, y, z)

❸ noMelon =  # Убедитесь, что это не арбуз

❹ mc.postToChat("Нужно раздобыть еду: " + str(noMelon))
```

Листинг 5.6. Основа кода для проверки, есть ли в указанных координатах арбуз

Melon — арбуз

В строке ❶ объявляем переменную `melon` и присваиваем ей идентификатор блока «арбуз» (103). Затем вызываем функцию `getBlock()`, сохраняя результат ее работы (тип блока в координатах *x*, *y*, *z*) в переменной `blockType` ❷. Вам предстоит дописать в строке ❸ код, который вернет `True`, если значение переменной `melon` не равно значению переменной `blockType`. Результат нужно сохранить в переменной `noMelon`, а затем отправить его в чат ❹.

No melon — арбуза нет

Рис. 5.9. У меня на ферме хранится арбуз, так что еду можно не искать

Убедиться, что значения переменных `melon` и `blockType` ❸ не равны, можно двумя способами: воспользоваться операцией сравнения «не равно» либо логической операцией `not`. В обоих случаях результат будет одинаков, но все же постарайтесь применить в этой миссии операцию `not`.

Внесите необходимые изменения и запустите программу. Пример ее работы показан на рис. 5.9.

БОНУСНОЕ ЗАДАНИЕ: ПОРЯДОК В ХОЗЯЙСТВЕ

Измените тип блока, наличие которого проверяет программа. Можете выяснить, растет ли у вас на ферме пшеница или не украл ли кто-нибудь входную дверь.

Порядок выполнения логических операций

В пределах одной команды можно объединять любое количество логических операций. Вот, к примеру, довольно причудливая комбинация из `and`, `or` и `not`:

```
>>> True and not False or False
True
```

Этот код возвращает `True`. Удивлены? Первой здесь выполняется операция `not`, которая возвращает `True`. Так что выражение принимает вид:

```
>>> True and True or False
True
```

Затем выполняется операция `and`: `True and True` возвращает `True`, после чего остается:

```
>>> True or False
True
```

И наконец вычисляется `or`: `True or False` возвращает `True`.

Python выполняет логические операции в определенном порядке. Перепутав что-то местами, вы рискуете получить не тот результат, которого ожидаете. Вот их очередность:

1. `not`

2. `and`

3. `or`

Попрактикуйтесь в составлении выражений с логическими операциями. Старайтесь считать в уме результат каждого выражения и лишь затем проверяйте вашу догадку в окне консоли Python.

Чтобы не путаться, отделяйте пары логических операций скобками. Операции в скобках Python выполняет в первую очередь.

Мое число между двумя другими?

Иногда бывает нужно убедиться, что одно число меньше другого, но больше третьего. Скажем, вы хотите выяснить, на самом ли деле у вас больше 10, но меньше 20 волков (больше 10, поскольку вы любите этих зверей, но меньше 20, потому что на такую стаю не хватит еды). Проверку можно запустить при помощи операции `and`:

```
wolves = input("Введите количество волков: ")
enoughWolves = wolves > 10 and wolves < 20
print("Волков достаточно: " + str(enoughWolves))
```

Wolves — волки

Однако есть и другой способ. Вместо двух выражений и операции `and` поставьте переменную `wolves` между двумя знаками сравнения:

```
wolves = input("Введите количество волков: ")
enoughWolves = 10 < wolves < 20
print("Волков достаточно: " + str(enoughWolves))
```

Enough wolves — волков достаточно

Если запустить любую из этих программ и ввести в окне консоли значение в диапазоне от 10 до 20, но не равное этим числам, переменная `enoughWolves` примет значение `True`. Также можно воспользоваться операциями сравнения «больше или равно» (>=) и «меньше или равно» (<=):

```
wolves = input("Введите количество волков: ")
enoughWolves = 10 <= wolves <= 20
print("Волков достаточно: " + str(enoughWolves))
```

В этом случае при вводе значений 10 и 20 переменная `enoughWolves` также примет значение `True`.

МИССИЯ 25. ИГРОК В ДОМЕ?

С помощью Python можно сделать так, чтобы при появлении игрока в том или ином месте происходило определенное событие. Например, если игрок встанет на блок «кирпичи», откроется потайная дверь. Или игрок попадет в ловушку, если наступит на нее. В ходе этой миссии я покажу, как определить, что в вашем доме есть игрок.

В миссии 8 (с. 83) вы написали программу, которая создает стены, пол и потолок здания, и сохранили ее в папке *math* под именем *building.py*. Найдите эту программу и откройте ее.

Просмотрите код *building.py* и запишите значения переменных `width`, `height` и `length` (у меня это были числа 10, 5 и 6 соответственно). Также запишите координаты места, в котором сейчас находится ваш игрок. Запустите программу, чтобы она построила дом.

Итак, дом у вас есть. Теперь можно создать новый файл IDLE и ввести в него код листинга 5.7. Код необходимо доработать, чтобы он проверял, находится ли игрок в доме.

insideHouse.py

```
from mcpi.minecraft import Minecraft
mc = minecraft.create()

❶ buildX =
  buildY =
  buildZ =
❷ width = 10
  height = 5
  length = 6

  pos = mc.player.getTilePos()
  x = pos.x
  y = pos.y
  z = pos.z

❸ inside = buildX < x < buildX + width and
```

Листинг 5.7. Основа кода, который проверяет, находится ли игрок в доме

Этот фрагмент кода проверяет, находится ли *x*-координата игрока в пределах дома, созданного программой *building.py*. Ваша задача — добавить команды, проверяющие, что *y*- и *z*-координаты игрока также находятся в пределах дома.

Сохраните код листинга 5.7 под именем *insideHouse.py*.

Inside house — внутри дома

Чтобы доделать программу, выполните следующее:

1. Впишите координаты здания (это координаты игрока на момент запуска программы *building.py*) после знака равно в строки, начиная с ❶.

2. Если значения `width`, `height` и `length` в вашей версии *building.py* отличаются от заданных в переменных ❷, исправьте их.
3. Допишите сравнение в переменной `inside` (строка ❸), чтобы узнать, находятся ли координаты игрока в пределах дома. Там уже есть проверка *x*-координаты. Добавьте после `and` сравнения для координат *y* и *z*. Выражения будут примерно такими же, как и для координаты *x* (`buildX < x < buildX + width`).

Inside — внутри

4. Отправьте в чат сообщение со значением переменной `inside`.
5. Закончив вносить изменения, сохраните код и запустите программу. Пример ее работы показан на рис. 5.10.

Рис. 5.10. Игрок находится в спальне, а значит — внутри дома

Что вы узнали

В этой главе вы использовали для проверки условий булевы значения, операции сравнения и логические операции. В главе 6 вам предстоит написать программы, в которых на основе таких проверок будут приниматься различные решения. Это выглядит так: программа проверяет условие и, если оно истинно, запускает один код, а если ложно — другой. Затем, в главе 7, вы создадите программу, которая будет раз за разом выполнять фрагмент кода до тех пор, пока условие возвращает `True`, и выходить из этого цикла, если условие вернет `False`. В этом заключается истинная сила булевых значений и условий — они позволяют выбирать, какой код выполнять и в какой момент это делать.

6

КОНСТРУКЦИЯ IF, ДУШ И ПОТАЙНАЯ ДВЕРЬ

В главе 5 вы узнали, как задавать вопросы на языке Python. А еще — научились использовать в коде операции сравнения («равно», «не равно», «больше», «меньше» и т. д.) и логические операции (`and`, `or`, `not`), чтобы выяснить, выполняются ли те или иные условия или наборы условий — то есть возвращают ли они `True` или `False`. В этой главе мы разберем, как научить программу выбирать тот или иной код на основе результата таких проверок.

Играя в Minecraft, вы часто принимаете решения в зависимости от разных условий. Сейчас ночь? Если да, вы одеваете игрока в алмазные доспехи и даете ему меч, чтобы он сражался с монстрами. Если нет, оставляете снаряжение в тайном штабе. Игрок голоден? Если да, вы кормите его хлебом или яблоком, а если нет, отправляете на поиски приключений, чтобы он нагулял аппетит. Подобно тому как решения принимаете вы, программа тоже может выполнять разные действия в зависимости от *условий*.

Чтобы научить этому вашу программу, воспользуемся специальной конструкцией языка Python `if`. Она определяет, нужно ли выполнять указанный фрагмент кода, при этом программа понимает ее так: «*Если* это условие выполняется, нужно запустить следующий код». Например, программа может проверить, находится ли игрок в запретной комнате, и, если это так, превратить пол в лаву. Или узнать, положил ли игрок в определенное место блок определенного типа, и, если это так, открыть потайную дверь. С помощью конструкции `if` вы сможете создавать в мире Minecraft собственные мини-игры.

If — если

Конструкция `if`

Возможность контролировать выполнение команд трудно переоценить: для серьезного программирования это просто необходимо! Программисты часто называют такой подход «порядок выполнения программы» (control flow). Проще всего контролировать работу программы с помощью конструкции `if`, которая выполняет заданный код, если условие возвращает `True`.

Конструкция `if` состоит из трех частей:

- ключевое слово `if`;
- условие, которое нужно проверить;
- «тело» `if` — фрагмент кода, который нужно выполнить, если условие возвращает `True`.

Давайте посмотрим, как работает конструкция `if`. Следующий код выведет на экран `"Много зомби."`, только если количество зомби, хранящееся в переменной `zombies`, превышает 20:

```
zombies = int(input("Введите количество зомби: "))
if zombies > 20:
    print("Много зомби.")
```

Здесь `zombies > 20` — условие, которое мы проверяем, а `print("Много зомби.")` — команда, которую нужно выполнить, если `zombies > 20` вернет `True`. Двоеточие (`:`) после 20 указывает на то, что со следующей строки начинается тело `if`. Для обозначения команд, которые входят в тело конструкции, используются *отступы* — то есть дополнительные пробелы в начале строк. При программировании на Python принято делать отступы в 4 пробела. Чтобы добавить в тело `if` еще несколько строк кода, достаточно ввести их после `print("Много зомби.")`, соблюдая отступ.

Запустите эту программу несколько раз, вводя разные значения, и посмотрите, что произойдет в каждом случае. В первый раз введите число меньше 20, во второй — число 20, а в третий — число больше 20. Вот что выведет программа, если ввести 22:

```
Введите количество зомби: 22
Много зомби.
```

Что ж, с этим понятно. Давайте еще раз запустим программу и посмотрим, что она выведет, если условие не выполняется:

```
Введите количество зомби: 5
```

Как видите, ничего не происходит. Python просто игнорирует тело `if`, ведь его команды должны выполняться только тогда, когда условие возвращает `True`. После того как программа закончит работать с конструкцией `if`, она перейдет к строке, следующей за ней.

Давайте рассмотрим еще один пример, чтобы лучше в этом разобраться. Следующий код использует конструкцию `if` для проверки пароля:

```
password = "коты"
attempt = input("Введите пароль: ")
if attempt == password:
    print("Пароль верный")
print("Конец программы")
```

После ключевого слова `if` стоит условие: `attempt == password`, а следующая строка с отступом и командой `print` — это тело конструкции `if`.

Attempt — попытка

Password — пароль

Программа напечатает сообщение `"Пароль верный"`, только если значение переменной `attempt` будет равно значению переменной `password`. Последняя же строка кода идет после тела `if` и поэтому в любом случае выведет на экран `"Конец программы"`.

А теперь, узнав все это, давайте что-нибудь разрушим.

МИССИЯ 26. КАК СДЕЛАТЬ КРАТЕР

Вы уже знаете, как телепортировать игрока и поднять его высоко в воздух. Теперь ваша задача — уничтожить блоки, которые находятся рядом с ним.

После запуска программы блоки под и над игроком, а также по сторонам от него должны превратиться в воздух. Будьте осторожны: этот код может повредить или уничтожить ваши постройки или редкие блоки! Чтобы избежать случайных разрушений, программа сначала спросит, действительно ли вы хотите уничтожить блоки, и продолжит работу только при утвердительном ответе.

Код листинга 6.1 создает кратер вокруг игрока, удаляя блоки рядом с ним, а затем отправляет в чат сообщение `"Бабах!"`. Сохраните эту программу под именем *crater.py* в новой папке *ifStatements*.

Crater — кратер

If statements — конструкции if

crater.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

answer = input("Создать кратер? Д/Н ")
```

```
❶ # Добавьте сюда конструкцию if

  pos = mc.player.getPos()
❷ mc.setBlocks(pos.x + 1, pos.y + 1, pos.z + 1,
               pos.x - 1, pos.y - 1, pos.z - 1, 0)
  mc.postToChat("Бабах!")
```

Листинг 6.1. Этот код создает кратер вне зависимости от ответа пользователя

Программа использует функцию `input()`, чтобы задать пользователю вопрос, нужно ли создать кратер. Однако сейчас от ответа пользователя ничего не зависит — кратер будет создан в любом случае.

От вас требуется доработать программу, добавив в нее конструкцию `if`, которая будет проверять, ввел ли пользователь букву «Д» (что соответствует ответу «да»). Поместите эту конструкцию вместо комментария ❶. Обратите внимание, что ответ пользователя попадет в переменную `answer`, так что `if` должна проверять ее значение и выполнять последние три команды, только если игрок введет букву «Д». Не забудьте поставить в начале строк конструкции `if` отступы в четыре пробела.

Answer — ответ

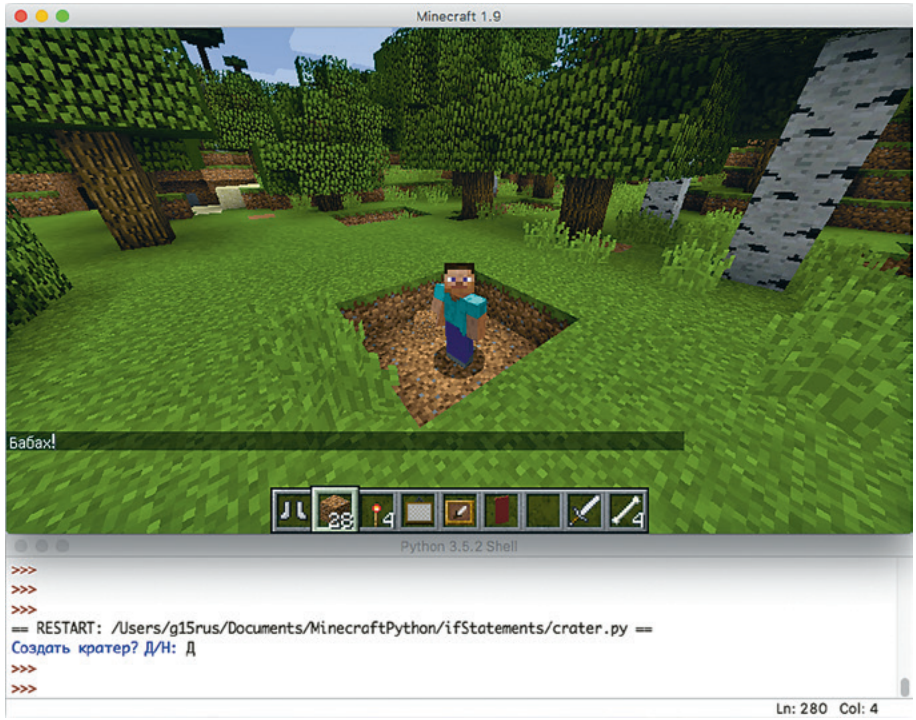

Рис. 6.1. Бабах! Кратер готов!

Последним аргументом функции `setBlocks()` должен быть тип блоков будущего кубоида. В данном случае это 0 — «воздух». Вы создадите воздушный кубоид, но у пользователя возникнет впечатление, будто земля вокруг игрока просто исчезла. Прибавляя и вычитая 1 из значений `pos.x`, `pos.y` и `pos.z`, вы зададите размер кубоида 3 на 3 блока. Это и будет кратер.

Внеся в программу изменения, сохраните ее и запустите. В окне программы IDLE появится запрос: `"Создать кратер? Д/Н"`. Введите заглавную букву «Д» (да) или «Н» (нет).

При вводе «Д» программа должна создать кратер, как на рис. 6.1.

БОНУСНОЕ ЗАДАНИЕ: ПОСТРОЙТЕ ДОМ

Что еще можно сделать с помощью этой программы? Измените ее код так, чтобы вместо кратера в позиции игрока появлялся дом.

Конструкция else

А теперь рассмотрим более сложную конструкцию, которая запускает выполнение другого фрагмента кода, если условие возвращает `False`. Называется эта конструкция `else`.

Else — иначе

Конструкция `else` работает совместно с `if`. Сначала вы пишете `if`, запускающую фрагмент кода, если условие возвращает `True`, а затем добавляете `else`, чтобы программа могла выполнить другой фрагмент кода в случае, если условие вернет `False`. Компьютер понимает это так: «Если условие выполняется, нужно сделать одно, а *если нет* — другое».

Следующий код выведет `"А-а-а! Зомби!"`, если в комнате больше 20 зомби (программа запросит их количество, сохранив его в переменной `zombies`); в противном случае она напечатает: `"Эй, зомби, а вы неплохие ребята"`.

```
zombies = int(input("Введите количество зомби: "))
if zombies > 20:
    print("А-а-а! Зомби!")
else:
    print("Эй, зомби, а вы неплохие ребята.")
```

Так же как и в конструкции `if`, для обозначения тела `else` используются двоеточие и отступы. Однако `else` не может быть самостоятельной командой — эта конструкция всегда должна следовать после `if`.

Кроме того, у `else` нет собственного условия — тело `else` выполняется, если условие, указанное в `if` (в данном случае `zombies > 20`), не возвращает `True`.

Вспомните предыдущий пример с паролем — в его код можно добавить конструкцию `else`, чтобы напечатать сообщение о том, что пароль неправильный. Вот так:

```
password = "коты"
attempt = input("Введите пароль: ")
if attempt == password:
    print("Пароль верный")
else:
    print("Пароль неверный")
```

Если значение переменной `attempt` совпадает со значением переменной `password`, условие `if` возвращает `True` и программа выводит на экран `"Пароль верный"`.

Однако, если значения переменных `attempt` и `password` отличаются, условие вернет `False` — и тогда запустится тело `else`, а на экране появится фраза `"Пароль неверный"`.

А что будет, если добавить в код конструкцию `else` без `if`? Пусть программа состоит лишь из следующих двух строк:

```
else:
    print("Ничего не происходит.")
```

Увы, в таком случае Python не поймет, чего вы хотите, и выведет сообщение об ошибке.

МИССИЯ 27. ПРЕДОТВРАТИТЬ РАЗРУШЕНИЯ ИЛИ НЕТ?

В ходе миссии 17 (с. 114) вы написали код, который благодаря команде `mc.setting("world_immutable", True)` не дает блокам разрушаться, а миру Minecraft — меняться. Он защищает ваши потрясающие постройки от вандалов и случайностей: пожаров, текущей лавы и т. д. Однако, несмотря на очевидные преимущества, пользоваться такой программой довольно неудобно. Чтобы отключить защиту, нужно написать еще один код!

Совместив конструкции `if` и `else` с запросом ввода с клавиатуры, можно создать программу, которая будет спрашивать, хотите ли вы сделать блоки неизменяемыми, и в зависимости от ответа присваивать опции `"world_immutable"` значение `True` либо `False`.

Запустите IDLE и создайте новый файл. Сохраните его под именем *immutableChoice.py* в папке *ifStatements*. Затем напишите код, следуя инструкциям.

Immutable choice — выбор неизменяемости

1. Программа должна задавать пользователю вопрос, хочет ли он сделать блоки неизменяемыми:

```
"Вы хотите защитить блоки от разрушений? Д/Н"
```

   Поместите эту строку в качестве аргумента в функцию `input()`, а значение, которое функция вернет, сохраните в переменной `answer`.

2. Программа должна проверить, находится ли в переменной `answer` значение `"Д"` (оно будет соответствовать ответу «да»). И, если это так, выполнит следующие команды:

```
mc.setting("world_immutable", True)
mc.postToChat("Защита от разрушений включена")
```

   Введите этот код, поместив его в тело конструкции `if`, чтобы он выполнялся, только если значение `answer` равно `"Д"`. Не забывайте про отступы!

3. Если пользователь введет не `"Д"`, а что-либо другое, программа должна выполнить другие команды:

```
mc.setting("world_immutable", False)
mc.postToChat("Защита от разрушений выключена")
```

Введите этот фрагмент кода в тело конструкции `else`, соблюдая отступы.

Сохраните программу и запустите ее. В ответ на запрос, хотите ли вы сделать блоки неизменяемыми, введите `"Д"` либо `"Н"` и нажмите ENTER. Проверьте, что получилось: если вы ввели `"Д"`, блоки должны стать неизменяемыми; в обратном случае они будут разрушаться, как обычно.

На рис. 6.2 показано, как выглядит запрос в окне консоли и сообщение в чате.

Обратите внимание — вы можете ввести как `"Н"`, так и любую чепуху, например `"банан"`, а результат будет одинаковым. Как думаете, почему так происходит?

Рис. 6.2. Я выбрал защиту блоков и теперь не могу ничего разрушить

БОНУСНОЕ ЗАДАНИЕ: УЛУЧШЕННЫЙ ИНТЕРФЕЙС

При помощи логических операций можно сделать программу более дружелюбной. Например, научить ее понимать ответ пользователя, если вместо ответа «Д» он наберет слово «Да», строчными или прописными буквами, или если буква «д» будет строчной. Попробуйте сделать это сами!

Конструкция elif

С помощью конструкций `if` и `else` вы можете запускать один фрагмент кода, если условие возвращает `True`, и другой, если `False`. Но что делать, когда необходимо запустить более двух фрагментов кода?

Elif (*сокр.* от else if) — иначе если

Для этого подойдет конструкция `elif`. Сначала введите `if`, затем — `elif` и в самом конце — `else`. Используя эти конструкции, вы как бы говорите программе: «Если условие возвращает `True`, нужно запустить первый фрагмент кода. Если второе условие возвращает `True` — второй фрагмент кода. И, наконец, если ни одно из условий не выполняется, нужно запустить третий фрагмент кода».

Давайте посмотрим, как это работает. Представьте, что вы выбираете в магазине мороженое. Вы можете рассуждать так: «Если шоколадное мороженое еще осталось, возьму его. А если его нет, но есть клубничное, возьму клубничное. А если нет ни его, ни шоколадного, выберу ванильное».

В коде это решение можно записать так:

```
hasChocolate = False
hasStrawberry = True
if hasChocolate:
    print("Ура! Беру шоколадное.")
elif hasStrawberry:
    print("Выбираю второй вариант — клубничное.")
else:
    print("Ладно, ванильное тоже вкусное.")
```

В первых двух строках задаются начальные условия. Допустим, сегодня в магазине нет шоколадного мороженого, зато есть клубничное. Соответственно, мы присваиваем переменной `hasChocolate` значение `False`, а переменной `hasStrawberry` значение `True`.

Has chocolate — есть шоколадное

Has strawberry — есть клубничное

Далее мы передаем значение переменной `hasChocolate` в конструкцию `if`, и, если оно равно `True`, программа выведет: `"Ура! Беру шоколадное"`. Однако в нашем случае это сообщение не будет напечатано, ведь в `hasChocolate` находится значение `False`. Вместо этого программа перейдет к конструкции `elif` и проверит, равняется ли `True` значение переменной `hasStrawberry`. Так оно и есть, поэтому программа запустит код тела `elif` и выведет на экран: `"Выбираю второй вариант — клубничное"`.

Как видите, тело конструкции `elif` состоит из команды, которая запускается лишь в том случае, если условие `if` возвращает `False`, а условие `elif` — `True`.

И наконец, стоящая после `elif` конструкция `else` сработает только тогда, когда и условие `if`, и условие `elif` вернут `False`. В нашем примере код тела `else` запустился бы и вывел на экран `"Ладно, ванильное тоже вкусное"`, будь и в переменных `hasChocolate` и `hasStrawberry` значение `False`.

Давайте вернемся к программе, выводящей сообщение в зависимости от количества зомби в комнате. Добавим в ее код конструкцию `elif`, чтобы проверить еще одно условие, если условие `if` вернет `False`:

```
zombies = int(input("Введите количество зомби: "))
if zombies > 20:
    print("А-а-а! Зомби!")
elif zombies == 0:
    print("Уф! Здесь нет зомби!")
```

```
else:
    print("Эй, зомби, а вы неплохие ребята.")
```

В `elif` мы проверяем условие `zombies == 0`. Если это так, программа выведет: `"Уф! Здесь нет зомби!"` Если же условие `elif` вернет `False`, программа перейдет к конструкции `else` и напечатает: `"Эй, зомби, а вы неплохие ребята"`.

МИССИЯ 28. ПОДАРОК

Давайте создадим программу, которая определяет, лежит ли в заданном месте подарок, и выводит в чат разные сообщения в зависимости от того, что это за подарок.

Программа будет проверять подарки на соответствие блокам двух типов: «алмазный блок» и «саженец» (ведь не у всех дарителей в запасе достаточно алмазных блоков).

Код листинга 6.2 должен проверять, находится ли в позиции (10, 11, 12) алмазный блок либо саженец, или там нет ни того, ни другого. Однако код не закончен.

gift.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
x = 10
y = 11
z = 12
gift = mc.getBlock(x, y, z)

# Если это алмазный блок
❶ if

# Иначе, если это саженец
❷ elif

else:
    mc.postToChat("Положите подарок сюда: " + str(x) + ", " + str(y)
                  + ", " + str(z))
```

Листинг 6.2. Основа кода для проверки наличия подарка в заданном месте

Gift — подарок

Создайте в IDLE новый файл и сохраните его под именем *gift.py* в папке *ifStatements*. Добавьте в него код листинга 6.2. Команда для определения типа блока там уже есть. Результат ее работы хранится в переменной

`gift`. Конструкция `else` должна сработать и вывести в чат сообщение с просьбой положить подарок в место с указанными координатами, только если там нет ни алмазного блока, ни саженца. Вы можете присвоить переменным `x`, `y` и `z` любые другие значения, какие только захотите.

Чтобы создать полноценный код, выполните следующее:

1. Дополните конструкцию `if` ❶ фрагментом кода, который будет проверять, соответствует ли значение переменной `gift` идентификатору алмазного блока (57). Если это так, пусть программа отправит в чат сообщение: `"Спасибо за алмаз"`.

2. Дополните конструкцию `elif` ❷ фрагментом кода, который будет проверять, соответствует ли значение переменной `gift` идентификатору блока «саженец» (6). Если соответствует, пусть программа отправит в чат сообщение: `"Эх, наверно, саженец не хуже алмаза"`.

Внеся эти изменения, сохраните программу, запустите ее, предварительно положив в место с указанными координатами алмазный блок, и посмотрите, что будет. Затем проделайте то же самое с саженцем. Кроме того, узнайте, что выведет в чат программа, если подарка не окажется. Не забудьте, что саженец нужно сажать в блок «земля» или «дерн»! Выводит ли программа (которую постоянно нужно запускать заново) правильные сообщения для каждого случая?

Результат работы моей программы показан на рис. 6.3.

Рис. 6.3. Мой подарок — саженец

БОНУСНОЕ ЗАДАНИЕ: ЖЕРТВА АРБУЗНОМУ БОГУ

Вы можете изменить код, чтобы проверить подарок на соответствие любым другим типам блоков. Сделайте так, чтобы программа узнавала, находится ли в заданной позиции «золотой блок» или «арбуз». Попробуйте добавить код, уничтожающий подарок.

Цепочки конструкций `elif`

После `if` можно ставить любое количество конструкций `elif` — хоть одну, хоть сто. Python обработает их все по очереди.

Приведу пример на основе программы про зомби:

```
zombies = int(input("Введите количество зомби: "))
if zombies > 20:
    print("A-a-a! Зомби!")
❶ elif zombies > 10:
    print("Еще столько же зомби, и случится зомби-апокалипсис.")
elif zombies == 0:
    print("Уф, здесь нет зомби!")
else:
    print("Эй, зомби, а вы неплохие ребята.")
```

Здесь мы добавили сразу после `if` еще одну конструкцию `elif` ❶, проверяющую, больше ли в комнате 10 зомби? Если так и есть, код тела `elif` выведет: `"Еще столько же зомби, и случится зомби-апокалипсис"`. Если нет — перейдет к следующей конструкции `elif`.

При этом очередность конструкций `if` и `elif` очень важна. Если вы введете их в неправильном порядке, некоторые проверки могут стать недоступными и программа будет работать не так, как вы ожидаете.

Например, если поменять местами тела конструкции `if` и первого `elif`, возникнут проблемы:

```
zombies = int(input("Введите количество зомби: "))
if zombies > 10:
    print("Еще столько же зомби, и случится зомби-апокалипсис.")
elif zombies > 20:
    print("A-a-a! Зомби!")
elif zombies == 0:
    print("Уф, здесь нет зомби!")
else:
    print("Эй, зомби, а вы неплохие ребята.")
```

Почему же так делать нельзя? Давайте посмотрим, что произойдет, если в комнате окажется 22 зомби. Поскольку 22 больше, чем 10, условие `zombies > 10` вернет `True` и программа выполнит код тела `if`. После этого оставшиеся конструкции `elif` и `else` будут пропущены. То есть программа никогда не дойдет до строки `elif zombies > 20`. Это ошибка в коде.

Если вы обнаружите, что конструкция `if` выдает странные результаты, обязательно проверьте, стоят ли блоки `if` и `elif` в правильном порядке.

МИССИЯ 29. ТЕЛЕПОРТАЦИЯ В НУЖНОЕ МЕСТО

Если очередность конструкций `if` и `elif` нарушена, код, который должен сработать, выполняться не будет. Вместо этого запустится код, который выполняться не должен. Из-за этого в программе могут возникать странные ошибки. Чтобы исправить код, вам придется заменить очередность проверок на верную. Давайте попробуем это сделать.

Код листинга 6.3 работает неправильно. Он должен телепортировать игрока в различные точки мира Minecraft в зависимости от количества очков (points), которое введет пользователь.

В строках, отвечающих за выбор места для телепортации в зависимости от количества очков, ошибки нет. Однако, судя по всему, очередность конструкций `if` и `elif` нарушена.

Чем больше у игрока очков, тем лучше место, в которое он отправится. Посмотрите на следующий код — координаты для телепортации являются аргументами функции `setPos()`.

teleportScore.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

points = int(input("Введите количество очков: "))
if points > 2:
    mc.player.setPos(112, 10, 112)
elif points <= 2:
    mc.player.setPos(0, 12, 20)
elif points > 4:
    mc.player.setPos(60, 20, 32)
elif points > 6:
    mc.player.setPos(32, 18, -38)
```

Листинг 6.3. Этот код должен телепортировать игрока в разные точки мира Minecraft в зависимости от количества очков

В коде отдельно проверяется, больше ли количество очков, чем 2, 4, 6, а также меньше ли оно 2.

Teleport score — телепортация по очкам

Создайте в IDLE новый файл и сохраните его под именем *teleportScore.py* в папке *ifStatements*. Введите код листинга 6.3 и измените его так, чтобы все точки телепортации стали доступны. Протестируйте программу, вводя разное количество очков, — убедитесь, что конструкция `if` и каждая из конструкций `elif` могут запуститься. На рис. 6.4 показан результат работы неправильного кода.

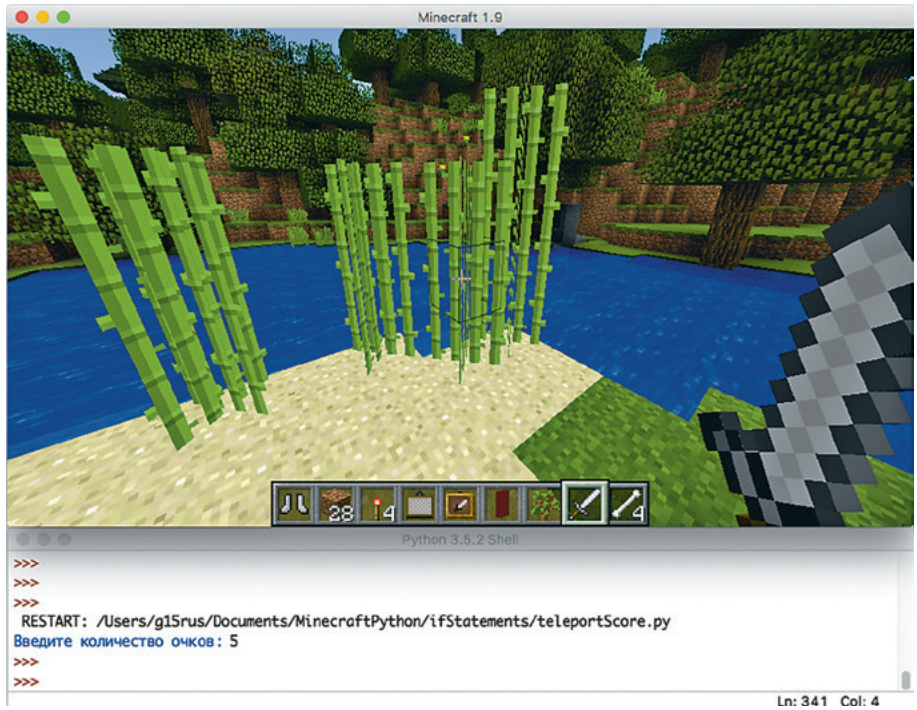

Рис. 6.4. Я не ожидал, что игрок окажется здесь!

Поскольку код листинга 6.3 работает неправильно, после ввода числа 5 игрок перенесся в место, заданное для количества очков более 2, хотя должен был оказаться в месте, заданном для более чем 4 очков.

«Телепортируй меня, Скотти!» — фраза из сериала «Звездный путь», ставшая интернет-мемом. *Прим. перев.*

БОНУСНОЕ ЗАДАНИЕ: ТЕЛЕПОРТИРУЙ МЕНЯ, СКОТТИ!

Создайте программу, которая позволит указать место телепортации в виде строки — например, `"замок"`. Для выбора места телепортации для каждого из вариантов используйте конструкции `if` и `elif`.

Вложенные конструкции `if`

Предположим, у вас есть конструкция `if` и, если ее условие возвращает `True`, программе необходимо проверить еще одно условие (и выполнить некий код в случае, если второе условие также вернет `True`). Например, вам нужно замаскировать вход в тайное убежище. Вы можете написать код, проверяющий, стоит ли игрок на блоке, который является переключателем. Если стоит, другая команда должна проверить, есть ли у него с собой предмет, открывающий дверь. Но как это сделать?

Вы можете поместить одну конструкцию `if` в тело другой конструкции `if`. Тогда она будет называться *вложенной конструкцией `if`*.

В листинге 6.4 показан пример вложенных конструкций: простейший банкомат проверяет, достаточно ли на вашем счету денег, чтобы снять нужную сумму. Если достаточно, запрашивает подтверждение и, если ответ положительный, выдает деньги.

```
withdraw = int(input("Сколько денег хотите снять? "))
balance = 1000

❶ if balance >= withdraw:
      confirm = input("Вы уверены? ")
❷     if confirm == "Да":
          print("Вот ваши деньги.")
else:
      print("Извините, на вашем счету недостаточно средств.")
```

Листинг 6.4. Программа для воображаемого банкомата

Обратите внимание, что вторая конструкция `if` дана внутри первой конструкции с отступом. Если условие внешнего `if` ❶ возвращает `True`, значит у вас на счету достаточно денег и программа запрашивает подтверждение: `"Вы уверены?"` Затем, если условие внутреннего `if` ❷ возвращает `True`, программа выводит: `"Вот ваши деньги"`.

МИССИЯ 30. ПОТАЙНАЯ ДВЕРЬ

Задача этой миссии связана с одним из предыдущих примеров. Вам предстоит создать дом с потайным входом, который открывается, если поставить на пьедестал алмазный блок. А если поставить туда блок другого типа, пол превратится в лаву!

Сначала постройте дом. Чтобы не тратить на это время, найдите в папке *math* программу *building.py* (с. 83) и запустите ее. Не делайте в доме дверь. Поставьте снаружи одиночный блок, который будет играть роль пьедестала. Когда вы поставите на него алмазный блок, в стене откроется проход. Основа этого кода дана в листинге 6.5.

secretDoor.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

x = 10
y = 11
z = 12
gift = mc.getBlock(x, y, z)
if gift != 0:
    # Добавьте сюда ваш код
else:
    mc.postToChat("Поставьте подарок на пьедестал.")
```

Листинг 6.5. Основа кода, который открывает потайную дверь, если поставить на пьедестал алмазный блок

Secret door — потайная дверь

Создайте в IDLE новый файл и сохраните его под именем *secretDoor.py* в папке *ifStatements*. Введя код листинга 6.5, измените значения `x`, `y` и `z` так, чтобы они соответствовали координатам места на пьедестале, куда нужно поместить ключ-алмаз.

Теперь запрограммируйте следующие действия:

- Если на пьедестале находится алмазный блок (57), необходимо открыть потайную дверь (подсказка: чтобы создать проход, замените в стене несколько блоков, из которых построен дом, на воздушные).
- Если на пьедестале находится что-либо другое, нужно превратить участок поверхности под игроком в лаву (10).

Чтобы дописать код, вам понадобятся вложенные конструкции `if`.

Эта программа сложнее предыдущих, поэтому пишите и тестируйте ее поэтапно. Запрограммировав часть действий, запустите программу, убедитесь, что она работает как надо, и лишь затем добавляйте следующие команды. Тестировать и дорабатывать короткие фрагменты кода проще, чем исправлять длинные, если обнаружится, что программа работает неверно или вовсе не работает. На рис. 6.5 показан открывшийся тайный проход.

БОНУСНОЕ ЗАДАНИЕ: ЭСКАЛАТОР

Как еще можно усовершенствовать программу *secretDoor.py*? Попробуйте создать автоматическую дверь, которая открывается, если встать возле нее. Или эскалатор, переносящий игрока вверх по лестнице, если оказаться на ее нижней ступени.

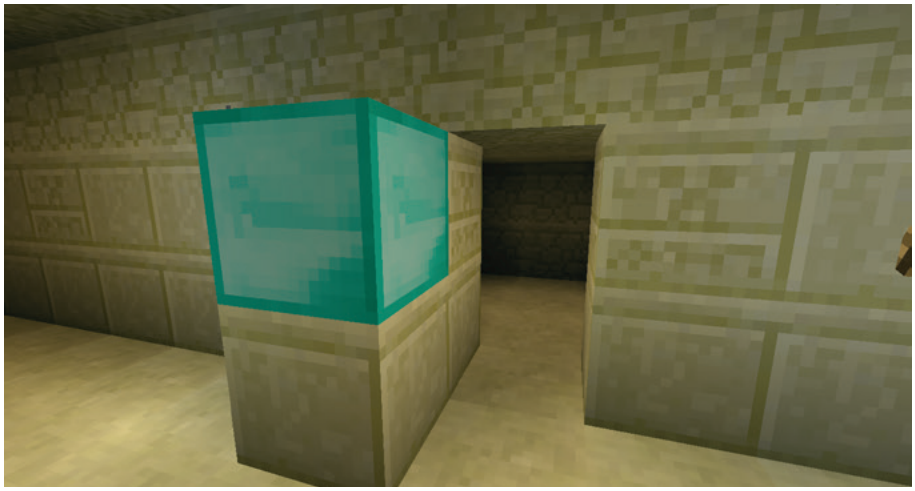

Рис. 6.5. Алмазный блок открыл потайную дверь

Проверка диапазона значений с помощью if

Как вы знаете из главы 5, с помощью Python можно определить, попадает ли некое значение в диапазон между двумя другими значениями. Поскольку такая проверка возвращает `True` или `False`, вы можете использовать ее как условие для конструкции `if`, подобно простым проверкам на больше-меньше или равно — не равно. В сущности, любое выражение, возвращающее `True` или `False`, может быть условием конструкции `if`.

Предположим, вы потратили весь день в мире Minecraft, чтобы собрать ингредиенты для вкуснейших тортов, испекли 30 тортов и теперь хотите их продать. Ваш покупатель может взять любое количество, от 1 до 29, но не больше — вы не хотите, чтобы он один забрал себе всё!

В коде ситуацию с тортами можно выразить так:

```
cakes = int(input("Сколько тортов вы хотите купить? "))
❶ if 0 < cakes < 30:
    print("Вот, возьмите " + str(cakes) + " тортов.")
❷ elif cakes == 0:
    print("Неужели вы не хотите тортика?")
❸ else:
    print("Это слишком много! Не будьте эгоистом!")
```

Cakes — торты

Если в переменной `cakes` находится число больше 0 и меньше 30, например 15, программа напечатает `"Вот, возьмите 15 тортов"` ❶. Если 0, программа спросит: `"Неужели вы не хотите тортика?"` ❷, а если больше 30 — `"Это слишком много! Не будьте эгоистом!"` ❸.

При помощи логических операций можно проверять и более сложные выражения. Если я, такой чудак, не хочу продавать хлеб, когда покупатель просит от 20 до 30 батонов, то отследить количество батонов, попадающее в этот диапазон, мне поможет логическая операция `not`:

```
bread = int(input("Сколько батонов вам нужно? "))
if not 20 <= bread <= 30:
    print("Вот ваши " + bread + " батонов.")
else:
    print("У меня есть причины не продавать вам столько хлеба.")
```

В этом примере я использую `not` вместе с операциями сравнения «меньше или равно». Сначала я проверяю, входит ли значение переменной `bread` в диапазон от 20 до 30, а затем применяю `not`, в результате чего `True` превращается в `False`, а `False` — в `True`. Если значение `bread` от 20 до 30, условие `if` возвращает `False`, а значит, будет выполнен код конструкции `else`. Если же количество батонов не входит в диапазон от 20 до 30 — например, это число 40, — условие вернет `True` и программа выведет: `"Вот ваши 40 батонов"`.

Bread — хлеб

Если покупатель решит взять 23 батона, программа этого не допустит. Однако 17 или 32 батона продаст без проблем.

МИССИЯ 31. ОГРАНИЧЬТЕ ОБЛАСТЬ ТЕЛЕПОРТАЦИИ

Помните программу для телепортации игрока, которую вы написали в главе 2? Она называлась *teleport.py*. В ходе этой миссии вам предстоит ограничить область телепортации при помощи конструкции `if`, которая будет проверять, попадают ли введенные пользователем координаты в заданный диапазон. В Minecraft для Raspberry Pi есть места, куда вы не сможете переместить игрока, поскольку они находятся за пределами игрового мира. Однако программа все равно попытается выполнить телепортацию, и это приведет к ошибке. Если же у вас версия Minecraft для настольных компьютеров, доступный вам игровой мир гораздо больше, однако программа с ограничением телепортации все равно может пригодиться — например, для того чтобы во время игры в прятки ограничить область, где можно прятаться.

Код листинга 6.6 запрашивает у пользователя *x*-, *y*- и *z*-координаты игрока и телепортирует его в это место. Однако код не закончен.

teleportLimit.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
```

```
valid = True

x = int(input("Введите x: "))
y = int(input("Введите y: "))
z = int(input("Введите z: "))

if not -127 < x < 127:
    valid = False

# Убеждаемся, что y не находится в диапазоне от -63 до 63

# Убеждаемся, что z не находится в диапазоне от -127 до 127

if valid:
    mc.player.setPos(x, y, z)
else:
    mc.postToChat("Пожалуйста, введите допустимые координаты.")
```

Листинг 6.6. Код, ограничивающий область телепортации

Чтобы ограничить область телепортации, в программу добавлена переменная `valid`. Она может принимать значения `True` или `False`, которые будут означать, допустимо ли перемещение игрока в место с указанными координатами. Мы запрашиваем у пользователя координаты *x*, *y* и *z*, а затем в условии `if` проверяем, находится ли значение `x` в диапазоне от –127 до 127, и меняем результат проверки на обратный с помощью `not`. Таким образом, если значение `x` не входит в диапазон, условие `if` вернет `True`, и тогда мы присвоим переменной `valid` значение `False` — телепортация недопустима.

Valid — допустимо

Телепортирующая функция `setPos()` будет вызвана, если переменная `valid` примет значение `True`. А это произойдет, если проверки подтвердят, что все три координаты игрока находятся в заданном диапазоне. В противном случае вместо телепортации программа выведет в чат сообщение с просьбой указать допустимые координаты.

Создайте в IDLE новый файл и введите туда код листинга 6.6. Сохраните программу под именем *teleportLimit.py* в папке *ifStatements*.

Teleport limit — ограничение телепортации

Доработайте программу, чтобы она проверяла не только координату *x*, но и координаты *y* и *z*, присваивая переменной `valid` значение `False`, если координаты окажутся недопустимыми.

Закончив, запустите программу. Она должна телепортировать игрока, если введенные координаты *x* и *z* окажутся в диапазоне от –127 до 127, а координата `y` — от –63 до 63. На рис. 6.6 показано, что произойдет, если хоть одно из введенных значений окажется за пределами своего диапазона.

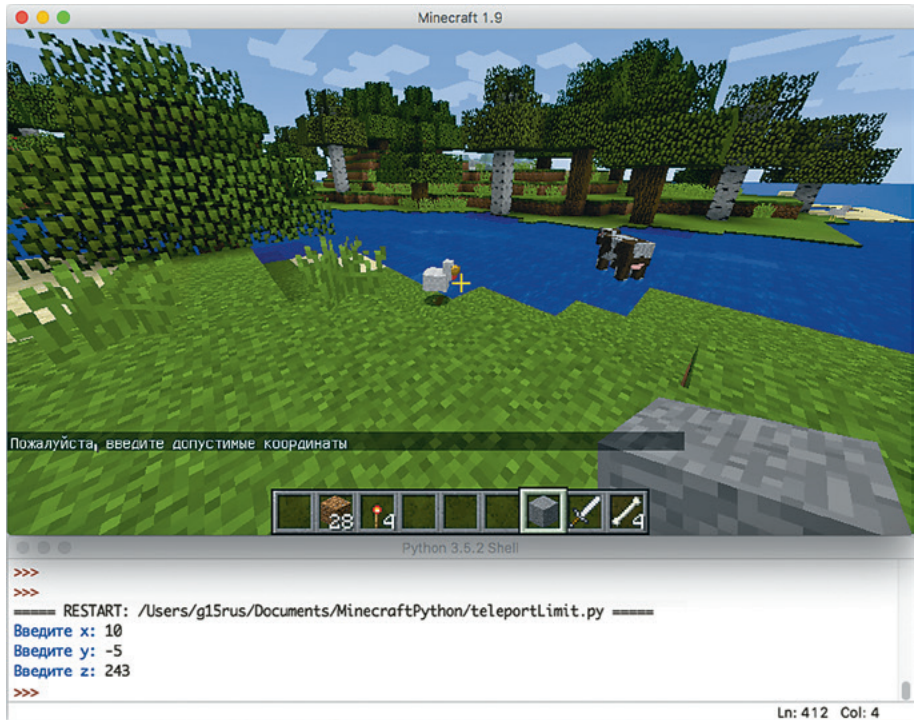

Рис. 6.6. Значение z слишком велико, поэтому игрок не телепортировался

БОНУСНОЕ ЗАДАНИЕ:
ОТСТАВИТЬ ТЕЛЕПОРТАЦИЮ ПОД ЗЕМЛЮ!

У нашей программы есть один недостаток: она может телепортировать игрока под землю, откуда тот не сумеет выбраться. Измените код так, чтобы предотвратить такую возможность. Для этого сравнивайте введенную y-координату со значением, которое возвращает функция `getHeight()`, и запрещайте телепортацию, если точка назначения окажется под землей.

Логические операции и конструкция if

В коде последней миссии вы применили в условиях `if` логическую операцию `not`. Операции `and` и `or` тоже можно использовать с конструкцией `if`, при этом работать конструкция будет так же, как с простыми условиями: если выражение в целом возвращает `True`, запускается тело `if`. Вот программа, которая спрашивает, есть ли у пользователя торт и готов ли он им поделиться:

```
hasCake = input("У вас есть торт? Д/Н")
wouldShare = input("Поделитесь тортиком? Д/Н")

if hasCake == "Д" and wouldShare == "Д":
    print("Ура!")
else:
    print("Эх!")
```

Поскольку здесь используется логическая операция `and`, Python выведет `"Ура!"`, только если у пользователя есть торт (выражение `hasCake == "Д"` вернет `True`) и он готов им поделиться (`wouldShare == "Д"` тоже вернет `True`). Если хоть одно из этих сравнений вернет `False`, запустится тело конструкции `else` и программа выведет: `"Эх!"`

Has cake — торт есть

Would share — может поделиться

Можно заменить операцию `and` на `or` — тогда программа напечатает `"Ура!"`, если выполняется хотя бы одно из двух условий: у пользователя есть торт или он желает им поделиться, то есть хотя бы одно из сравнений `hasCake == "Д"` или `wouldShare == "Д"` возвращает `True`.

```
hasCake = input("У вас есть торт? Д/Н")
wouldShare = input("Поделитесь тортиком? Д/Н")

if hasCake == "Д" or wouldShare == "Д":
    print("Ура!")
else:
    print("Эх!")
```

Сообщение `"Эх!"` мы увидим лишь в том случае, если оба условия вернут `False`, то есть если у пользователя нет тортов и делиться ими он в любом случае не намерен.

А теперь попробуем совместить конструкцию `if` с операцией `not`:

```
wearingShoes = input("Игрок сейчас в ботинках? Д/Н")
if not wearingShoes == "Д":
    print("Игрок сейчас без ботинок.")
```

Этот код предлагает пользователю ответить `"Д"`, если его игрок в ботинках, и `"Н"` в обратном случае и сохранить введенное значение в переменной `wearingShoes`. Далее идет условие `if`, в котором значение `wearingShoes` проверяется на равенство `"Д"`, причем к результату этой проверки применяется операция `not`, которая меняет значение на обратное — `True` превращает в `False`, а `False` — в `True`. Поэтому, если пользователь введет `"Д"`, сравнение вернет `True`, а `not` поменяет результат на `False`, так что значением условия в целом станет `False` и программа ничего не выведет. Если же введенное значение не равно `"Д"`, сравнение

Wearing shoes — в ботинках

вернет `False`, а `not` превратит его в `True`. Значит, условие вернет `True`, и программа выведет в чат: `"Игрок сейчас без ботинок"`.

МИССИЯ 32. ДУШ

Лучшие дома в мире Minecraft обустроены с большим вниманием к деталям. Многие пользователи, заботясь об уюте, делают в доме деревянный пол, очаг, украшают стены картинами. Вы же превзойдете их всех, создав работающий душ.

Для этого вам понадобятся проверки диапазонов и логические операции. Вы создадите душевую кабину, в которой будет включаться вода, когда внутри окажется игрок. Иными словами, если игрок находится в заданном диапазоне координат, кабина заполнится водяными блоками.

Основа кода показана в листинге 6.7. Ваша задача — его дописать.

shower.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

❶ shwrX =
  shwrY =
  shwrZ =

❷ width = 5
  height = 5
  length = 5

  pos = mc.player.getTilePos()
  x = pos.x
  y = pos.y
  z = pos.z

❸ if shwrX <= x < shwrX + width and
❹     mc.setBlocks(shwrX, shwrY + height, shwrZ,
                   shwrX + width, shwrY + height, shwrZ + length, 8)
  else:
      mc.setBlocks(shwrX, shwrY + height, shwrZ,
                   shwrX + width, shwrY + height, shwrZ + length, 0)
```

Листинг 6.7. Основа кода для работающего душа

Shower — душ

Введите код листинга 6.7 в окно программы IDLE, сохранив файл в папке *ifStatements* под именем *shower.py*.

Первым делом присвойте значения переменным `shwrX`, `shwrY` и `shwrZ` — они должны соответствовать координатам вашей душевой кабины ❶. Затем укажите размеры кабины, добавив значения переменных `width`, `height` и `length` ❷, — я уже присвоил им значение 5, однако вы можете выбрать любые размеры на свой вкус.

Допишите условие `if`. В нем необходимо проверить, действительно ли переменные `y` и `z` находятся в пределах душевой кабины ❸. Чтобы вам было проще, я уже добавил проверку координаты *x* (`shwrX` < `x` < `shwrX + width`). Выражения для `y` и `z` будут аналогичными. Подсказка: эти три проверки нужно объединить с помощью операции `and`.

Функция `setBlocks()` нужна для включения и выключения душа ❹. При включенном душе она будет заполнять кабину блоками «текущая вода» (идентификатор 8), а при выключенном — блоками «воздух» (идентификатор 0).

Вызовы функции `setBlocks()` внутри конструкций `if` и `else` разбиты на две строки. Для длинного списка аргументов допускается такая запись. В окне программы можно было записать эти вызовы одной строкой, но в книге это сделать невозможно. Поэтому я разделил каждую строку на две — к тому же так программу удобнее читать.

На рис. 6.7 показан душ, который работает в доме моего игрока.

Рис. 6.7. А вот и игрок, принимающий душ

Если вы запустите программу, когда игрок будет находиться в душевой кабине, она создаст вокруг него водяные блоки. Вода будет течь до тех пор, пока герой не выйдет из душа, а вы не запустите программу снова. Приятного купания!

БОНУСНОЕ ЗАДАНИЕ: ЭКОНОМЬТЕ ВОДУ

Добавьте в программу лимит времени, после которого вода должна отключаться автоматически.

Что вы узнали

Из этой главы вы узнали, как запускать проверку условий при помощи конструкций `if`, `else` и `elif`, и теперь ваши программы будут решать сами, какой код при каких условиях выполнять.

В главе 7 вы познакомитесь с циклом `while`. Подобно конструкциям `if`, циклы `while` помогают программам понять, что и когда им делать. Однако, в отличие от `if` и `else`, запускающих один код при выполнении условия и другой в обратном случае, цикл `while` выполняет указанный код снова и снова, пока условие не вернет `False`.

7

ЦИКЛ WHILE, ДИСКОТЕКА И ЦВЕТОЧНЫЙ ДОЖДЬ

Циклы позволяют программам выполнять один и тот же код снова и снова. Вместо того чтобы копировать фрагмент кода и многократно вставлять его в программу, вы можете представить его в виде цикла и запускать любое количество раз подряд. В главе 7 вы познакомитесь с одним из циклов языка Python — циклом `while`. Он позволяет повторять одинаковые команды без перезапуска программы.

Простейший цикл while

Цикл `while` используют для многократного повторения фрагментов кода. Аналогично конструкции `if`, он выполняет содержащийся в теле цикла код, если условие возвращает `True`.

Отличие в том, что тело `if` выполняется только один раз, тогда как тело `while` может выполняться повторно. Программисты называют такие повторы кода *итерациями*.

В этой программе цикл `while` используется для вывода чисел от 1 до 5:

```
count = 1
while count <= 5:
    print(count)
```

```
    count += 1
print("Конец цикла")
```

Count — счетчик

Переменная `count` используется для подсчета количества итераций. Сначала мы присваиваем ей значение 1. Затем условие цикла `while` проверяет, действительно ли значение `count` меньше или равно 5.

Как вы знаете из главы 3, `+=` — это сокращенная операция. Вместо нее можно использовать более привычную запись `count = count + 1`. Результат будет точно таким же.

При первом входе в цикл значение `count` равно 1. Это число меньше 5 — условие цикла возвращает `True`, поэтому запускается тело цикла, которое выводит значение `count` на экран, а затем увеличивает его на 1. Далее цикл идет на новый виток, опять проверяя условие и выполняя команды тела раз за разом, пока значение `count` не станет больше 5.

Следом за циклом располагается единственная команда, выводящая на экран сообщение `"Конец цикла"`.

Сохранив программу и запустив ее, вы увидите вот что:

```
1
2
3
4
5
Конец цикла
```

Поэкспериментируйте с этим кодом: попробуйте изменить условие, чтобы программа выводила на экран больше пяти чисел, или поменять величину, на которую увеличивается значение `count`. Вот памятка по логике работы цикла `while`.

1. Проверка — возвращает ли условие `True`?
2. Если условие возвращает `True`:
   а) выполнить тело цикла;
   б) перейти к шагу 1.
3. Если условие возвращает `False`:
   а) игнорировать тело цикла.
4. Перейти к строке программы, следующей за телом цикла.

Давайте испытаем цикл `while` в игре Minecraft. Воспользуемся им для телепортации игрока во множество новых мест!

МИССИЯ 33. ТЕЛЕПОРТАЦИЯ В СЛУЧАЙНЫЕ МЕСТА

В ходе миссии 3 (с. 68) вы телепортировали игрока в различные места игрового мира. Давайте перепишем ту программу, добавив в нее цикл `while`, чтобы выполнять телепортацию снова и снова.

Поместив код телепортации внутрь цикла, вы можете сделать программу мощнее и при этом гораздо более легкой для чтения. Круто, правда?

Следующий код присваивает переменным `x`, `y` и `z` случайные значения и телепортирует игрока в место с этими координатами.

```
import random
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

❶ # Добавьте сюда переменную count
❷ # Начните здесь цикл while
❸ x = random.randint(-127, 127) # Добавьте отступы в этой строке и далее
  y = random.randint(0, 64)
  z = random.randint(-127, 127)

  mc.player.setTilePos(x, y, z)
❹ # Увеличьте значение переменной count на 1
```

Сейчас этот код телепортирует игрока только один раз. Здорово, ничего не скажешь! Однако вы можете сделать программу по-настоящему потрясающей. Добавьте в нее цикл, чтобы код выполнялся пять раз, и пусть игрока закружит вихрь приключений!

Чтобы использовать в программе цикл, выполните следующие шаги.

1. Объявите переменную `count`, которая будет считать повторы цикла ❶.
2. Добавьте цикл `while` с условием, проверяющим значение `count` ❷.
3. Поставьте отступы в начале строк, составляющих тело `while` ❸.
4. Увеличивайте значение `count` на каждом повторе цикла ❹.

Смысл переменной `count` и ее приращения в том, чтобы вести счет повторам цикла. Я еще коснусь этой темы чуть позже, а сейчас просто имейте в виду, что `count` позволяет контролировать число итераций.

В листинге 7.1 показан код с внесенными изменениями.

randomTeleport.py

```
import random
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

count = 0
while count < 5:
    x = random.randint(-127, 127)
    y = random.randint(0, 64)
    z = random.randint(-127, 127)
    mc.player.setTilePos(x, y, z)
    count += 1
```

Листинг 7.1. Код для случайной телепортации игрока в мире Minecraft

While loops — циклы while

Random teleport — случайная телепортация

Создайте новый файл, введите в него код листинга 7.1 и сохраните в новой папке *whileLoops* под именем *randomTeleport.py*. Запустите программу — вы увидите, как игрок молниеносно перемещается по игровому миру; всё его путешествие займет меньше секунды! Давайте это исправим.

Чтобы замедлить телепортацию, нам понадобится модуль `time`. Выполните следующие шаги:

1. Добавьте в начало кода команду `import time`. Она загрузит модуль `time`, идущий в комплекте с Python, в котором содержатся полезные функции для работы со временем (и не только).

2. Добавьте в конец тела `while` строку с функцией `time.sleep(10)` — эта команда приостанавливает выполнение программы на 10 секунд. Не забудьте начать эту строку с отступа, чтобы она попала в тело цикла!

Сохраните программу и запустите ее. Теперь игрок должен телепортироваться из одного случайного места в другое каждые 10 секунд.

На рис. 7.1 показан результат работы программы.

БОНУСНОЕ ЗАДАНИЕ: ПАУЗА В НАЧАЛЕ

Сейчас программа выдерживает 10-секундную паузу в конце каждой итерации цикла. А что будет, если перенести команду `time.sleep(10)` из конца цикла в его начало?

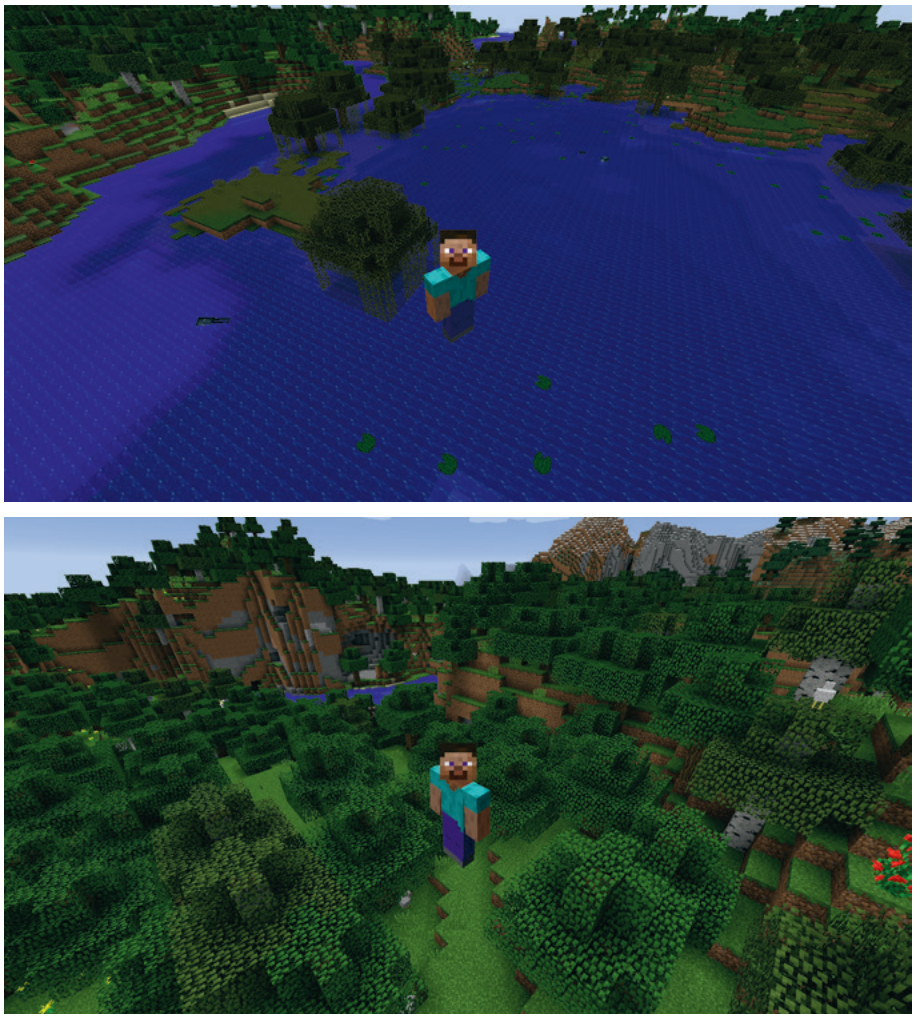

Рис. 7.1. Каждые 10 секунд программа телепортирует игрока в новое место

Управление циклами с помощью переменной count

Переменная `count` (так называемый *счетчик цикла*) часто используется в кодах для учета повторов цикла. Вы уже видели примеры ее работы. Вот еще один:

```
count = 0
while count < 5:
      print(count)
      count += 1
```

Условие цикла проверяет, действительно ли значение переменной `count` меньше 5. В теле цикла я изменяю значение `count` так, чтобы оно соответствовало числу сделанных итераций. Такое приращение называют *инкрементированием переменной*.

Последняя строка этого кода увеличивает `count` на 1. Перед каждым своим повтором цикл будет проверять значение `count`, чтобы узнать: оно все еще меньше 5? Если значение `count` будет равно числу 5 или превысит его, цикл закончит свою работу.

Если вы забудете, что `count` нужно увеличивать на 1, у вас получится бесконечный цикл, как в этом примере:

```
count = 0
while count < 5:
      print(count)
```

В данном случае `count` всегда будет содержать 0, ведь мы нигде не меняем значение этой переменной. Следовательно, условие цикла будет возвращать только `True`, а тело цикла — повторяться бесконечно. Если не верите, запустите код сами!

```
0
0
0
0
0
--и так далее--
```

Чтобы остановить программу, нажмите CTRL + C. Для ее исправления необходимо просто добавить в тело цикла команду `count += 1`, и бесконечные повторы вам больше не грозят. Какое облегчение!

Переменную `count` не обязательно увеличивать именно на 1. Порой бывает нужно прибавлять к счетчику цикла другое значение. В следующем примере значение `count` на каждой итерации возрастает на 2, и в результате программа печатает четные числа от 0 до 98.

```
count = 0
while count < 100:
      print(count)
      count += 2
```

Также можно использовать обратный отсчет, уменьшая, или *декрементируя*, значение `count`. Следующий код выводит числа в обратном порядке, от 100 до 1:

```
count = 100
while count > 0:
      print(count)
      count -= 1
```

Главное отличие этого примера от предыдущих — условие цикла. В данном случае я использую оператор сравнения «больше» (>). Цикл будет выполняться, пока значение `count` больше 0. Когда же оно достигнет 0, цикл закончит свою работу.

Необязательно давать переменной, которую вы будете использовать в качестве счетчика цикла, имя `count`. Можете назвать ее `repeat` или как угодно еще. Заглянув в коды других программистов, вы обнаружите великое разнообразие названий счетчиков цикла.

Repeat — повтор

МИССИЯ 34. ВОДЯНОЕ ПРОКЛЯТИЕ

Давайте-ка займемся недобрым делом и создадим кратковременное проклятие. В видеоиграх проклятия тем или иным образом ослабляют игрока: например, замедляют его движение или увеличивают наносимый урон — обычно на короткий период времени.

Мы создадим программу-проклятие, которая раз в секунду будет помещать в позицию игрока блок «текущая вода», и так на протяжении 30 секунд. Игроку станет сложно передвигаться, поскольку поток воды будет сносить его в сторону.

Вот этот код.

waterCurse.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getPos()
mc.setBlock(pos.x, pos.y, pos.z, 8)
```

Листинг 7.2. Код, который смещает игрока потоком воды

Однако такая программа помещает блок с текущей водой в позицию игрока лишь один раз. Ваша задача — добавить в нее повторы. Код должен повторяться 30 раз так, чтобы каждая итерация цикла длилась 1 секунду.

Сохраните программу под именем *waterCurse.py* в папке *whileLoops* и убедитесь, что она работает. После запуска в позиции игрока должен появиться одиночный блок «текущая вода».

Water curse — водяное проклятие

Давайте выясним, что необходимо добавить в программу, чтобы проклятие работало как нужно. Вспомните все, что вы узнали о цикле `while` и счетчике цикла, и сделайте вот что.

1. Добавьте в код переменную `count`.
2. Добавьте цикл, повторяющий две последние строки кода (определение позиции игрока и установку водяного блока). Цикл должен повторяться 30 раз.
3. Прирастите переменную `count` в конце тела цикла.
4. Загрузите модуль `time` (введите команду импорта первой строкой программы), а затем добавьте в конец тела `while` односекундную паузу.

Сохраните программу и проверьте, как она работает. Программа должна каждую секунду создавать блок воды и делать это в течение 30 секунд. Если вы запутаетесь, выполняя задание, вернитесь к описанию миссии 33 (с. 163) и перечитайте список необходимых шагов.

На рис. 7.2 показано водяное проклятие в действии.

Рис. 7.2. О нет! Моего игрока преследует маленькое наводнение!

БОНУСНОЕ ЗАДАНИЕ: НАВОДНЕНИЕ ПОСЕРЬЕЗНЕЙ

Что нужно сделать, чтобы тело цикла выполнялось вдвое быстрее (раз в полсекунды), но проклятие по-прежнему длилось 30 секунд?

Бесконечный цикл while

В большинстве случаев необходимо, чтобы условие цикла `while` рано или поздно вернуло `False` — иначе цикл будет повторяться без конца и ваш компьютер может зависнуть.

Однако порой программе требуется именно *бесконечный цикл*. Например, в видеоиграх бесконечные циклы часто используют для проверки пользовательского ввода и управления передвижениями игрока. Разумеется, в таких играх есть и кнопка «выход», чтобы вы могли приостановить или завершить цикл, когда захотите передохнуть.

Простой способ сделать цикл бесконечным — указать в качестве его условия значение `True`. Вот так:

```
while True:
      print("Привет")
```

Этот цикл будет повторяться без конца, выводя сообщение `"Привет"` снова и снова. Создали ли вы бесконечный цикл намеренно, или это произошло по недосмотру, его всегда можно остановить, нажав CTRL + C в окне консоли Python. Кроме того, для остановки цикла из IDLE можно выбрать в меню **Shell → Restart Shell**.

Обратите внимание: какой бы код ни находился *после* бесконечного цикла, программа до него никогда не дойдет. В следующем примере последняя команда кода не сработает, поскольку перед ней стоит бесконечный цикл:

```
while True:
    print("Привет")
print("Эта строка никогда не запустится")
```

Хоть бесконечные циклы и способны доставить уйму проблем, с их помощью можно сделать немало интересных вещей. Этим мы сейчас и займемся!

МИССИЯ 35. ЦВЕТОЧНЫЙ СЛЕД

Программа, которую вам предстоит создать в ходе этой миссии, напоминает программу из миссии 34. Однако теперь ваша задача — не создавать водяные блоки, а сделать так, чтобы за игроком оставался след из цветов, ведь цветущий луг гораздо симпатичнее, чем наводнение!

Найдите в папке *whileLoops* файл *waterCurse.py*, откройте его и сохраните под новым именем *flowerTrail.py*.

Flower trail — цветочный след

Чтобы за игроком всегда тянулся след из цветов, внесите в программу следующие изменения.

1. Укажите в качестве условия цикла `while` значение `True`.
2. Удалите из кода переменную `count`.
3. Измените аргумент функции `setBlock()`, который отвечает за тип блока, указав вместо 8 число 38.
4. Уменьшите значение, передаваемое функции `sleep()`, до `0.2`, чтобы за секунду программа создавала пять маков.
5. Сохраните программу и запустите ее. На рис. 7.3 показано, что должно получиться.

Рис. 7.3. Смотрите, какие милые цветочки!

БОНУСНОЕ ЗАДАНИЕ: СМЕРТЕЛЬНЫЙ СЛЕД

Код программы *flowerTrail.py* годится для многого. Попробуйте изменить тип блоков, которые создает программа. Много радости может вам доставить блок «динамит» (`setBlock(x, y, z, 46, 1)`). Обратите внимание на дополнительный аргумент 1, который следует за идентификатором блока «динамит». Значение 1 придает взрывчатому веществу свойство детонировать от простого прикосновения, без помощи огнива. Наведите прицел на блок «динамит», нажмите левую кнопку мыши, и произойдет взрыв!

Замысловатые условия

Поскольку условием цикла `while` может быть все что угодно, лишь бы оно возвращало булево значение, в нем можно использовать все известные нам операции сравнения, а также логические операции. Вы уже видели, что операторы «больше» и «меньше» работают с этим циклом так же, как с условиями.

Однако есть и другие способы управлять циклами с помощью сравнений и логических операций. Давайте посмотрим, какие именно!

Для начала сделаем условие более интерактивным. В следующем примере переменная `continueAnswer` создается до входа в цикл и затем фигурирует в условии, где ее значение сравнивается со строкой `"Д"`. (Внимание: просто слово `continue` в качестве имени переменной использовать нельзя, поскольку в Python оно является зарезервированным ключевым словом.)

Continue answer — вопрос о продолжении

```
continueAnswer = "Д"
coins = 0
while continueAnswer == "Д":
    coins = coins + 1
    continueAnswer = input("Продолжать? Д/Н ")
print("У вас есть " + str(coins) + " монет")
```

В последней строке тела `while` программа запрашивает ввод пользователя. Если тот введет что-нибудь, кроме `"Д"`, цикл закончит работу. Пользователь может нажимать `"Д"` снова и снова, и каждый раз после этого значение переменной `coins` будет увеличиваться на 1.

Обратите внимание, что переменная `continueAnswer`, значение которой проверяется в условии `while`, объявлена до начала цикла. Если бы мы этого не сделали, программа выдала бы ошибку. А все потому, что в условии цикла нельзя использовать необъявленную переменную.

МИССИЯ 36. СОСТЯЗАНИЕ НЫРЯЛЬЩИКОВ

Давайте немного повеселимся! В ходе этой миссии вам предстоит с помощью цикла `while` и операции «равно» (==) написать код мини-игры, в которой игроку нужно как можно дольше продержаться под водой. Программа запомнит, сколько секунд он там провел, и в конце отобразит счет. Кроме того, если игрок продержится дольше 6 секунд, программа осыплет его цветами.

Вот основа кода:

divingContest.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import time

score = 0
pos = mc.player.getPos()
❶ blockAbove = mc.getBlock(pos.x, pos.y + 2, pos.z)

❷ # Добавьте сюда цикл while
time.sleep(1)
pos = mc.player.getPos()
❸ blockAbove = mc.getBlock(pos.x, pos.y + 2, pos.z)
❹ score = score + 1
mc.postToChat("Текущий счет: " + str(score))

mc.postToChat("Окончательный счет: " + str(score))

❺ if score > 6:
    finalPos = mc.player.getTilePos()
    mc.setBlocks(finalPos.x - 5, finalPos.y + 10, finalPos.z - 5,
                 finalPos.x + 5, finalPos.y + 10, finalPos.z + 5, 38)
```

Листинг 7.3. Код мини-игры, в которой игроку нужно как можно дольше продержаться под водой

Diving contest — состязание ныряльщиков

Score — счет

Block above — блок сверху

Сохраните этот код под именем *divingContest.py* в папке *whileLoops*. Переменная `score` предназначена для учета времени, которое игрок проведет под водой.

Запустите программу и посмотрите, что произойдет. Код еще недописан: он просто засчитывает игроку одно очко и завершает работу.

Перед тем как дорабатывать программу, давайте разберемся, что в ней происходит. В переменной `blockAbove` хранится тип блока в позиции головы игрока ❶. Если голова игрока находится под водой, в эту переменную попадает число 8 (идентификатор блока «текущая вода»). Спустя несколько строк кода опять проверяем тип блока в позиции головы ❸.

Когда вы добавите в код цикл `while`, значение переменной `blockAbove` будет обновляться на каждой его итерации. В строке ❹ программа добавляет к счету игрока одно очко — это должно происходить каждую секунду, которую тот проведет под водой. И наконец, в строке ❺ программа использует конструкцию `if` для проверки, превысил ли счет значение 6, и, если это так, осыпает игрока цветами.

От вас требуется создать в строке ❷ цикл с переменной `blockAbove` в условии. Пусть он проверяет, находится ли игрок под водой, то есть соответствует ли значение переменной `blockAbove` идентификатору блока «текущая вода» (8) или «стоячая вода» (9). Используйте для этого выражение `blockAbove == 8 or blockAbove == 9`. Это и есть условие, которое должно проверяться на каждой итерации цикла.

Чтобы убедиться в работоспособности программы, найдите водоем глубиной не менее трех блоков и заведите в него игрока. Программа запустится, только если игрок уже находится под водой. Она должна начать отсчитывать секунды и отображать их в чате. Подождите еще немного и выведите игрока на берег. После этого программа должна показать окончательный счет и осыпать игрока цветами, если тот провел под водой более 6 секунд.

На рис. 7.4 показан игрок под водой и счет, который отображает программа, а на рис. 7.5 — цветы, символизирующие победу.

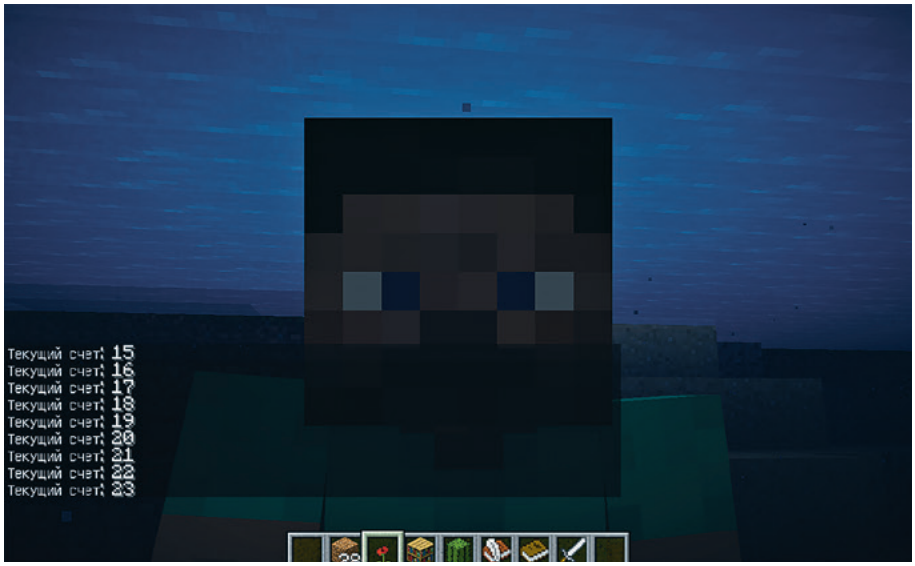

Рис. 7.4. Игрок сидит под водой, задержав дыхание, а на экране идет отсчет секунд

Рис. 7.5. Он заслужил этот цветочный дождь!

БОНУСНОЕ ЗАДАНИЕ: ДА ЗДРАВСТВУЕТ ПОБЕДИТЕЛЬ!

Добавьте дополнительные призы, изменив конструкцию `if` в конце программы. Если игрок наберет много очков, можете вручить ему золотой блок. Попробуйте добавить несколько уровней сложности со своими призами за прохождение каждого.

Логические операции и цикл while

Вместе с циклом `while` можно использовать логические операции `and`, `or` и `not`. Они пригодятся, если потребуется проверить несколько условий. Например, следующий цикл будет выполняться, пока пользователь не введет правильный пароль с первой, второй или третьей попытки:

```
password = "коты"
passwordInput = input("Введите пароль: ")
attempts = 0

❶ while password != passwordInput and attempts < 3:
❷     attempts += 1
❸     passwordInput = input("Неправильно. Введите пароль: ")

❹ if password == passwordInput:
      print("Пароль верный.")
```

Attempts — неудачные попытки

Условие цикла ❶ выполняет две задачи: во-первых, проверяет, отличается ли верный пароль от введенного пользователем (`password != passwordInput`), а во-вторых — меньше ли трех попыток сделал пользователь (`attempts < 3`)? Объединить эти два условия в одно условие цикла позволяет операция `and`. Если условие цикла вернет `False`, в его теле произойдет приращение переменной `attempts` ❷, и программа выдаст запрос на повторный ввод пароля ❸. Цикл завершится, если пользователь введет верный пароль либо использует все три попытки. После выхода из цикла конструкция `if` ❹ проверит, был ли введен правильный пароль, и, если это так, выведет сообщение `"Пароль верный"`.

Проверка диапазона значений в условии while

Также в цикле `while` можно узнать, попадает ли значение в указанный диапазон. Следующий код проверяет, входит ли введенное пользователем число в диапазон от 0 до 10. Если не входит, цикл завершает свою работу.

```
position = 0
❶ while 0 <= position <= 10:
    position = int(input("Введите позицию игрока от 0 до 10: "))
    print(position)
```

Если в переменной `position` окажется число больше 10, цикл завершится ❶. То же самое произойдет, если в `position` попадет число меньше 0. В Minecraft это может пригодиться для проверки соответствия координат игрока заданной области игрового мира. Этим мы и займемся в ходе следующей миссии.

Position — позиция

МИССИЯ 37. ПОСТРОЙТЕ ТАНЦПОЛ

Пришло время танцев! А чтобы от души поплясать, нужен танцпол. Программа, которую вы сейчас напишете, сгенерирует танцпол, меняющий свой цвет каждые полсекунды, если на нем в это время находится игрок.

Код создает танцпол в текущей позиции игрока, используя конструкцию `if`. Как обычно, от вас требуется этот код дописать.

danceFloor.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import time

pos = mc.player.getTilePos()
floorX = pos.x - 2
floorY = pos.y - 1
floorZ = pos.z - 2
width = 5
length = 5
block = 41
❶ mc.setBlocks(floorX, floorY, floorZ,
               floorX + width, floorY, floorZ + length, block)

❷ while floorX <= pos.x <= floorX + width and  # z в пределах танцпола?
❸     if block == 41:
          block = 57
      else:
          block = 41
      mc.setBlocks(floorX, floorY, floorZ,
                   floorX + width, floorY, floorZ + length, block)
      pos = mc.player.getTilePos()
      time.sleep(0.5)
```

Листинг 7.4. Основа кода для танцпола, который мигает разными огнями, если на него встает игрок

Dance floor — танцпол

Floor — пол

Откройте IDLE, создайте новый файл и сохраните код из листинга 7.4 под именем *danceFloor.py* в папке *whileLoops*. Программа строит в позиции игрока танцпол ❶, размеры которого содержатся в переменных `width` и `length`, а координаты хранятся в переменных `floorX`, `floorY` и `floorZ`. Внутри цикла `while` находится конструкция `if`, которая меняет тип блоков ❸, благодаря чему и создается эффект мигания.

Чтобы программа работала правильно, вам нужно дописать условие цикла `while`, добавив проверку нахождения *z*-координаты игрока в пределах танцпола ❷. Иными словами, убедитесь, что значение `pos.z` больше или равно `floorZ` и меньше или равно `floorZ + length`. В качестве примера посмотрите, как я с помощью выражения (`floorX <= pos.x <= floorX + width`) проверяю, находится ли значение `pos.x` в пределах танцпола. На рис. 7.6 мой танцпол во всей красе!

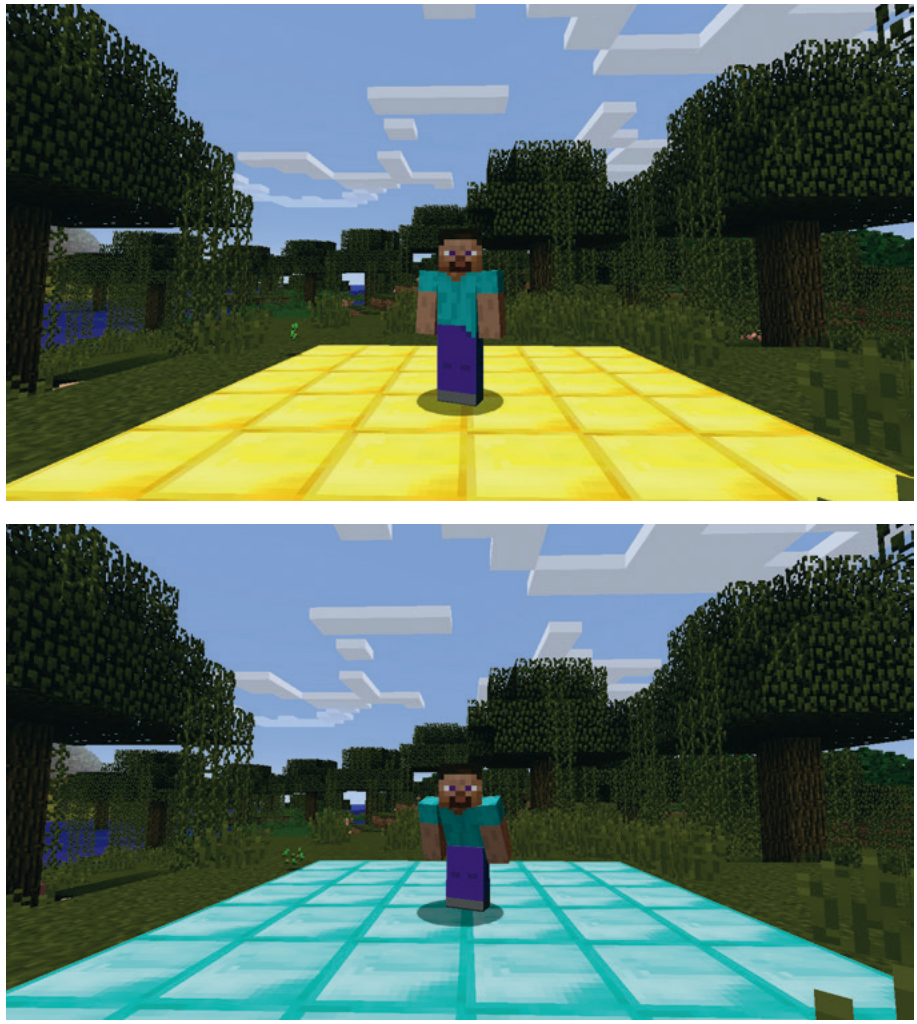

Рис. 7.6. Игрок отплясывает на танцполе

Дописав программу, сохраните ее и запустите. Под ногами игрока должен возникнуть подиум, меняющий цвет каждые полсекунды. Не отказывайте себе в удовольствии — потанцуйте! Когда закончите, покиньте танцпол и убедитесь, что он перестал мигать. Мигание больше не включится, если только вы не запустите программу еще раз и не создадите новый танцпол.

БОНУСНОЕ ЗАДАНИЕ: КОНЕЦ ВЕЧЕРИНКИ

Измените программу так, чтобы танцпол исчезал, когда игрок сходит с него. Для этого после выхода из цикла замените блоки танцпола на блоки «воздух».

Вложенные конструкции if и циклы while

Конструкции `if` можно размещать как внутри циклов `while`, так и снаружи — это позволяет писать более гибкие и мощные программы. Наверное, вы уже заметили в коде миссии 37 (с. 175) конструкцию `if`, вложенную в тело цикла.

В следующем примере вложенная конструкция `if` проверяет значение последнего напечатанного слова, определяя, какое слово выводить следующим — `Mine` или `craft`. Всего этот цикл сделает 50 итераций.

```
word = "Mine"
count = 0
while count < 50:
    print(word)
    if word == "Mine":
        word = "craft"
    else:
        word = "Mine"
    count += 1
```

Word — слово

После запуска программы переменной `word` присваивается первое слово, которое выводится на экран. Находящаяся внутри цикла конструкция `if` проверяет, равно ли текущее значение `word` строке `"Mine"`, и, если это так, заменяет его, чтобы при следующей итерации вывести `"craft"`. Если же в переменной `word` находится не `"Mine"`, в теле `else` ей будет присвоено значение `"Mine"`.

В цикл `while` также можно вкладывать конструкции `elif` и другие циклы `while`.

Следующая программа спрашивает пользователя, выводить ли ей на экран числа от единицы до миллиона:

```
userAnswer = input("Напечатать числа от 1 до 1000000? (да/нет): ")

❶ if userAnswer = "да":
      count = 1
❷     while count <= 1000000:
          print(count)
          count += 1
```

Конструкция `if` проверяет, ввел ли пользователь слово `"да"` ❶. Если это так, запустится вложенный в тело `if` цикл `while` ❷. Если же пользователь ввел что-то другое, цикл работать не будет и программа завершится.

МИССИЯ 38. ПРИКОСНОВЕНИЕ МИДАСА

Мида́с — мифический царь, который превращал в золото все, к чему прикасался. В ходе этой миссии вам предстоит написать программу, превращающую блоки, на которые наступит игрок, в золотые. Очевидное исключение составляют лишь блоки стоячей воды и воздуха, которые не позволят вашему игроку сдвинуться с места. Напоминаю: золотым блокам соответствует идентификатор 41, стоячей воде — 9, воздуху — 0.

midas.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

air = 0
water = 9

❶ # Добавьте сюда бесконечный цикл while
      pos = mc.player.getTilePos()
      blockBelow = mc.getBlock(pos.x, pos.y - 1, pos.z)

❷     # Добавьте сюда конструкцию if
          mc.setBlock(pos.x, pos.y - 1, pos.z, 41)
```

Листинг 7.5. Основа кода для превращения блоков, на которые наступит игрок, в золото

Откройте IDLE, создайте новый файл и введите в него код листинга 7.5. Сохраните программу под именем *midas.py* в папке *whileLoops*. Чтобы программа работала как полагается, вам придется ее доработать. Первым делом создайте бесконечный цикл `while` ❶. Напоминаю: чтобы цикл был бесконечным, его условие должно всегда возвращать `True`. Затем добавьте конструкцию `if`, проверяющую, не является ли блок под ногами игрока воздушным или водным ❷. Идентификатор блока под игроком записывается в переменную `blockBelow`, а типы блоков для воздуха и воды хранятся в переменных `air` и `water` соответственно.

Block below — блок ниже

Закончив программу, сохраните ее и запустите. Если код работает правильно, за игроком появится золотой след, однако, если он прыгнет в воду или взлетит в воздух, блоки не должны измениться.

На рис. 7.7 показан результат работы программы.

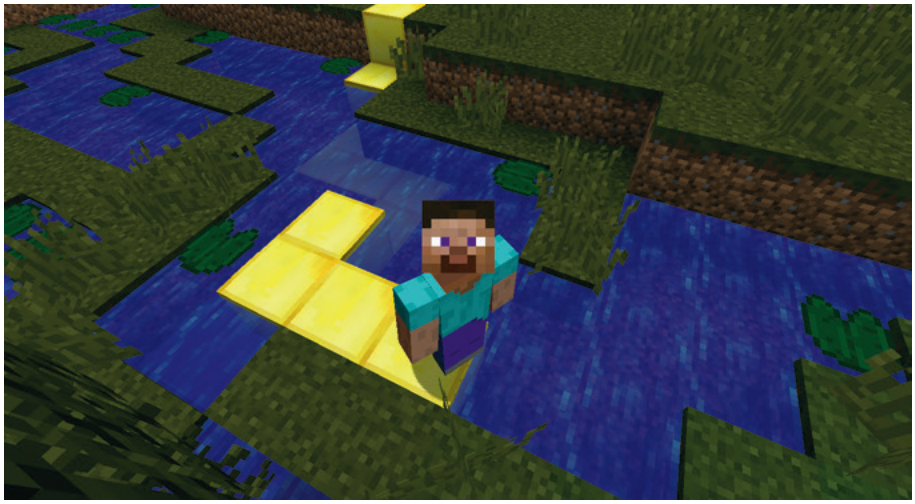

Рис. 7.7. Каждый блок, на который ступает игрок, обращается в золото

Чтобы завершить бесконечный цикл, переключитесь в IDLE и выберите **Shell → Restart Shell** либо кликните по окну консоли и нажмите CTRL + C.

БОНУСНОЕ ЗАДАНИЕ: ДОРОГУ ПАХАРЮ!

Если немного изменить программу *midas.py*, ее можно будет использовать и для других целей. Как думаете, что нужно поменять в коде, чтобы блоки земли, по которым пройдет игрок, превращались в обработанную почву? А как насчет превращения земляных блоков в траву?

Выход из цикла while с помощью break

Программист свободен сам выбирать, как и когда завершить цикл `while`. До сих пор вы использовали для этого лишь условие цикла, однако в вашем распоряжении есть команда `break`, которая позволяет выйти из цикла в любой момент. Давайте посмотрим, как это работает.

Break — отмена

Один из способов использовать команду `break` — поместить ее внутрь конструкции `if`, находящейся в теле цикла. В этом случае цикл завершит свою работу, когда условие `if` вернет `True`. Следующая

программа будет раз за разом запрашивать команду, пока вы не введете слово "выход":

```
❶ while True:
❷     userInput = input("Введите команду: ")
❸     if userInput == "выход":
❹         break
      print(userInput)
❺ print("Цикл завершен")
```

Этот цикл бесконечный, ведь в качестве условия у него указано `True` ❶. На каждой его итерации запрашивается ввод команды ❷. Затем с помощью конструкции `if` программа проверяет, ввел ли пользователь слово `"выход"` ❸. Если так и есть, выполняется команда `break`, которая останавливает цикл ❹. После этого программа продолжает работать со строки, идущей сразу после тела `while`, и выводит сообщение `"Цикл завершен"` ❺.

МИССИЯ 39. ПОСТОЯННЫЙ ЧАТ НА ОСНОВЕ ЦИКЛА

В ходе миссии 13 (с. 102) вы усовершенствовали программу для отправки сообщений в чат Minecraft. Хоть она и приносит пользу, возможности программы сильно ограниченны, ведь для отправки каждого сообщения ее приходится запускать заново.

Вам предстоит доработать код из миссии 13, добавив в него цикл `while`. Это позволит отправлять в чат любое количество сообщений, не перезапуская программу.

Chat loop — цикл в чате

Откройте файл *userChat.py* из папки *strings* и сохраните его в папку *whileLoops* под новым именем *chatLoop.py*.

Чтобы получить возможность отправлять новые сообщения без перезапуска программы, внесите в код следующие изменения.

1. Добавьте бесконечный цикл `while`.
2. Добавьте внутрь цикла конструкцию `if`, проверяющую, ввел ли пользователь слово `"выход"`. При вводе `"выход"` следует завершить цикл командой `break`.
3. Убедитесь, что переменная `username` объявлена до входа в цикл.

Внеся эти доработки, сохраните программу и запустите ее. В окне консоли Python должен появиться запрос имени пользователя. Введите имя и нажмите ENTER. Теперь программа предложит ввести сообщение — введите его и снова нажмите ENTER. Далее программа должна раз за разом предлагать вам ввести сообщение,

пока вы не напечатаете "выход". На рис. 7.8 показана программа в действии.

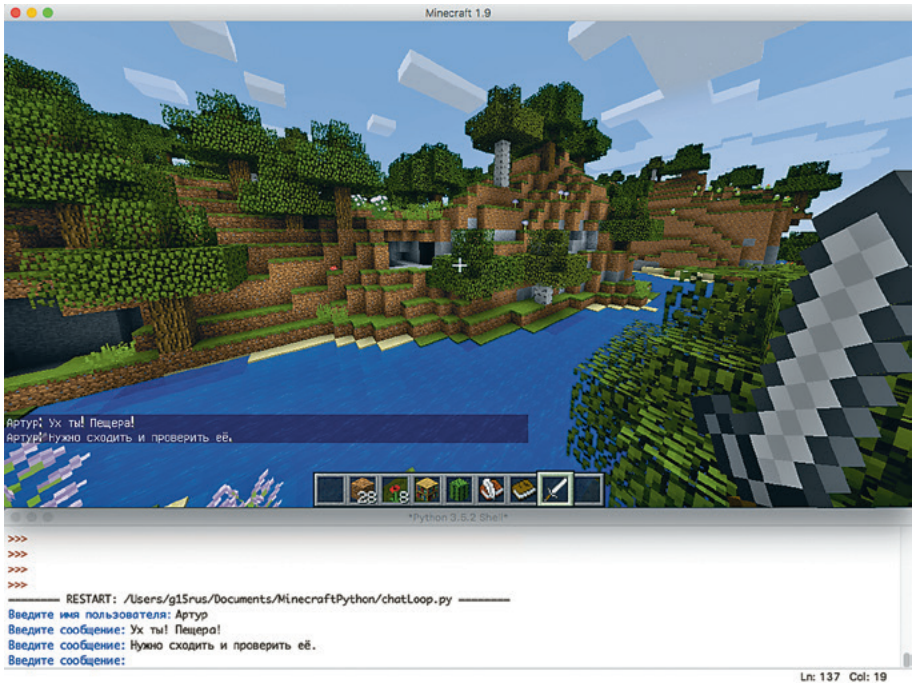

Рис. 7.8. Пользователь разговаривает сам с собой

БОНУСНОЕ ЗАДАНИЕ: ЧАТ И БЛОКИ

Добавьте в программу возможность создавать блоки через сообщения в чате. Пусть в программе появится, например, блок шерсти, если пользователь введет "шерсть". Сделать это можно, добавив `elif` в конструкцию `if`, проверяющую ввод пользователя.

Конструкция while-else

Подобно конструкции `if`, цикл `while` позволяет обрабатывать с помощью конструкции `else` случаи, когда условие цикла не выполняется.

Тело конструкции `else` запускается, если условие `while` возвращает `False`. Причем, в отличие от тела цикла, тело `else` выполняется только один раз:

```
message = input("Введите сообщение.")

while message != "выход":
    print(message)
    message = input("Введите сообщение.")
else:
    print("Пользователь вышел из чата.")
```

Этот цикл будет повторяться до тех пор, пока введенное сообщение не совпадет со строкой `"выход"`. В противном случае цикл закончит работу и запустится код из тела `else`, который выведет на экран: `"Пользователь вышел из чата"`.

Однако, если завершить цикл с помощью `break`, конструкция `else` не сработает. Следующий пример очень похож на предыдущий, но здесь в тело цикла добавлена конструкция `if` с командой `break`. Если пользователь введет не `"выход"`, а `"прервать"`, цикл завершит работу без вывода сообщения `"Пользователь вышел из чата"`:

```
message = input("Введите сообщение.")

while message != "выход":
    print(message)
    message = input("Введите сообщение.")
    if message == "прервать":
        break
else:
    print("Пользователь вышел из чата.")
```

Вложенная конструкция `if` проверяет, является ли введенное сообщение строкой `"прервать"`. Если это так, будет выполнена команда `break` и цикл завершит работу. Поскольку выход из цикла произойдет через `break`, тело `else` выполняться не будет и мы не увидим сообщения `"Пользователь вышел из чата"`.

МИССИЯ 40. «ГОРЯЧО ИЛИ ХОЛОДНО»

В ходе этой миссии мы создадим для Minecraft игру «Горячо или холодно». Если вам не доводилось в нее играть, коротко расскажу, в чем смысл. Ваш друг прячет какой-то предмет, а вы должны его найти. При этом друг дает подсказки в зависимости от того, насколько далеко от нужного места вы ищете. Если близко, он говорит «горячо», а если далеко — «холодно». Когда вы совсем рядом с предметом, друг кричит «обожжешься», а если очень далеко — «замерзнешь».

Цель нашей игры будет заключаться в том, чтобы отыскать алмазный блок, созданный в случайном месте игрового мира, и встать на него. В этой версии игры подсказывать, насколько ваш игрок далек от цели, будет программа. После того как он встанет на алмазный блок, игра завершится.

Код листинга 7.6 создает блок в месте со случайными координатами.

blockHunter.py

```
from mcpi.minecraft import Minecraft
import math
import time
import random
mc = Minecraft.create()

destX = random.randint(-127, 127)
destZ = random.randint(-127, 127)
❶ destY = mc.getHeight(destX, destZ)

block = 57
❷ mc.setBlock(destX, destY, destZ, block)
mc.postToChat("Блок создан")

while True:
    pos = mc.player.getPos()
❸     distance = math.sqrt((pos.x - destX) ** 2 + (pos.z - destZ) ** 2)

❹     if distance > 100:
        mc.postToChat("Замерзнешь")
    elif distance > 50:
        mc.postToChat("Холодно")
    elif distance > 25:
        mc.postToChat("Тепло")
    elif distance > 12:
        mc.postToChat("Горячо")
    elif distance > 6:
        mc.postToChat("Обожжешься!")
    elif distance == 0:
❺         mc.postToChat("Блок найден")
```

Листинг 7.6. Основа кода для игры «Горячо или холодно»

Прежде чем создать блок, программа выбирает для него место так, чтобы блок не оказался под землей. Для этого в коде используется функция `getHeight()` ❶ — она случайным образом выбирает координаты *x* и *z* и находит соответствующую *y*-координату плотного блока, который расположен выше остальных, а значит, на поверхности земли. Затем программа создает в этом месте алмазный блок ❷.

Sqrt (*сокр.* от square root) — квадратный корень

Код в строке ❸ находит расстояние от игрока до алмазного блока, используя при этом функцию `sqrt()` из модуля `math`, поэтому в начале программы стоит команда `import math`. Функция `sqrt()` вычисляет квадратный корень из переданного ей числа.

В коде листинга 7.6 используется формула из теоремы Пифагора, которая позволяет по двум известным сторонам прямоугольного треугольника найти длину третьей стороны. Мы уже применяли эту формулу в миссии 21. Чтобы освежить знания, обратитесь к рис. 5.5 на с. 125. В данном случае я беру расстояния от игрока до спрятанного блока по осям x и z и нахожу расстояние по прямой.

Сообщение-подсказка, которое выводит программа, зависит от удаленности игрока от искомого блока. Для выбора одного из вариантов подсказки используется конструкция `if`, которая проверяет значение переменной `distance` ❹. Если игрок очень далеко от алмаза, программа выведет `"Замерзнешь"`, а если совсем близко — `"Обожжешься"`.

Distance — расстояние

Block hunter — охотник за блоками

Введите код листинга 7.6 в новый файл и сохраните его под именем *blockHunter.py* в папке *whileLoops*.

Хотя эта программа работает, она не завершается, если блок найден. Чтобы исправить это, вам следует добавить в код команду `break`, которая должна выполняться, если расстояние от игрока до искомого блока станет нулевым ❺.

Дописав код, сохраните его, а затем запустите. В случайном месте игрового мира появится алмазный блок, который вам нужно найти. Когда игрок встанет на него, программа должна завершить работу. На рис. 7.9 показано, где обнаружил блок я.

БОНУСНОЕ ЗАДАНИЕ: ВРЕМЯ НЕ ЖДЕТ

Программа *blockHunter.py* позволяет искать блок сколь угодно долго. Разберитесь, что нужно сделать, чтобы узнать время, которое было потрачено на поиски, или даже ограничить его.

Что вы узнали

Отлично! Вы многое узнали о циклах `while` и теперь умеете создавать бесконечные циклы и циклы с условиями, используя при этом логические операции и операции сравнения. Теперь, когда в вашем арсенале есть циклы, вы можете писать программы с фрагментами кода, которые

будут выполняться многократно. Это сбережет время, позволив сосредоточиться на освоении мира Minecraft.

В главе 8 вы узнаете еще один способ повторно использовать код — создание функций.

Рис. 7.9. Я отыскал блок, осталось лишь встать на него

8

ФУНКЦИИ КАК ИСТОЧНИК БОЛЬШИХ ВОЗМОЖНОСТЕЙ

Функции — это предназначенные для решения определенных задач фрагменты кода, к которым можно обратиться из любого места программы. Предположим, вы пишете программу, создающую в Minecraft дерево. Если вам понадобится несколько деревьев, можно ввести код для создания одного дерева несколько раз (при помощи копирования и вставки). Однако это крайне неэффективно, особенно если потом вы захотите изменить код.

Вместо копирования и вставки можно воспользоваться функцией. В предыдущих главах вы уже имели дело с некоторыми встроенными функциями языка Python, например `str()`, `input()`, `int()`. Также вам доводилось работать с функциями из программного интерфейса Minecraft, такими как `getBlock()` и `setPos()`. В этой главе вы узнаете, как создавать собственные функции.

Для создания и использования своих функций есть веские причины.

Экономия места и времени. Больше не нужно вводить один и тот же код многократно, поэтому писать программы становится проще и быстрее.

Удобство отладки. Когда код сгруппирован в функции, проще искать ошибки и исправлять их.

Мóдульность. Функции можно создавать самим, а затем делиться ими с другими программистами. А еще разные функции, объединенные общей темой, можно группировать в модули, о которых вы узнаете в главе 11.

Масштабируемость. С помощью функций проще обрабатывать большие объемы данных и увеличивать возможности программы.

Создание собственных функций

Давайте разберемся, как создавать собственные функции. В следующем примере я создаю функцию `greeting()`, которая выводит на экран две строки:

Greeting — приветствие

```
def greeting():
    print("Привет")
    print("Приятно познакомиться")
```

Ключевое слово `def` указывает на то, что мы создаем функцию. Следом за `def` нужно написать имя функции (в данном случае это `greeting`). После имени должны стоять скобки и двоеточие — не забывайте об этом! Последующие строки кода составляют *тело функции*, которое будет запускаться при каждом ее вызове.

Def (*сокр.* от define) — определить

Следите за отступами в коде: всегда начинайте строки, составляющие тело функции, с четырех пробелов.

Функция может содержать любое количество команд, а также конструкции `if`, циклы, переменные, условия, математические операции и так далее. Чтобы Python понял, какие строки относятся к функции, а какие — к другим частям вашей программы, не забывайте делать отступы в теле функции.

В коде одной программы может быть сколько угодно функций при условии, что их имена не совпадают.

Вызов функции

Чтобы использовать, или *вызвать,* функцию, введите ее имя, а следом в скобках укажите необходимые ей аргументы. Если функция не принимает аргументов, просто поставьте после имени пару скобок. Вызвать функцию `greeting()` можно так:

```
greeting()
```

Обращаться к одной и той же функции можно сколько угодно раз. Давайте вызовем `greeting()` три раза подряд:

```
greeting()
greeting()
greeting()
```

Когда вы запустите эту программу, строки, которые выводит функция, должны повториться трижды:

```
Привет
Приятно познакомиться
Привет
Приятно познакомиться
Привет
Приятно познакомиться
```

Чтобы функция выполнилась, ее нужно вызвать в коде программы. Если после запуска программы ничего не происходит, возможно, вы просто забыли добавить в код вызов функции.

Также функции — как встроенные в Python, так и ваши собственные — можно вызывать из тела других ваших функций.

Совсем скоро вы увидите все это в действии.

Функции принимают аргументы

В скобках после имени функции указываются ее аргументы, то есть значения, которые функция использует. Этим значениям соответствуют специальные переменные. Впрочем, не все функции принимают значения аргументов — скажем, для функции `greeting()` из предыдущего примера аргументы не нужны.

Но что делать, если вы хотите поприветствовать кого-то по имени? Давайте создадим функцию, которую можно вызывать много раз для приветствия разных людей:

```
def fancyGreeting(personName):
    print("Привет, " + personName)

fancyGreeting("Тимур")
fancyGreeting("Артур")
```

В этом примере функция вызывается дважды — сначала с аргументом `Тимур`, затем с аргументом `Артур`. После запуска программа выведет:

```
Привет, Тимур
Привет, Артур
```

Если вы забудете указать необходимый функции аргумент, при ее вызове программа выдаст ошибку. То же самое произойдет при вызове функции, содержащей несколько аргументов, если вы не укажете в скобках хотя бы один из них. Попробуем вызвать функцию `fancyGreeting()`, не передав ей аргументы:

Fancy greeting — необычное знакомство

```
fancyGreeting()
```

В результате Python выдаст сообщение об ошибке:

```
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    fancyGreeting()
❶ TypeError: fancyGreeting() missing 1 required positional argument:
'personName'
```

Причина ошибки указана в последней строке этого сообщения: *«Тип ошибки: fancyGreeting() требуется 1 аргумент (передано 0)»* ❶. Функция `fancyGreeting()` принимает один аргумент, но была вызвана без его указания, что и привело к ошибке.

Можно создать функцию, принимающую несколько аргументов. Например, в следующей программе я создал функцию, которая здоровается с человеком по имени, делает паузу в заданное количество секунд, а затем прощается. Эта функция принимает имя человека в качестве первого аргумента, а количество секунд ожидания — в качестве второго:

```
import time

❶ def helloAndGoodbye(personName, secsToWait):
    print("Привет, " + personName)
    time.sleep(secsToWait)
    print("Пока, " + personName)

❷ helloAndGoodbye("Тимур", 10)
helloAndGoodbye("Артур", 23)
```

При создании функции аргументы указываются через запятую ❶. Затем, при вызове этой функции ❷, они выводятся на экран в том же порядке.

Говоря о функциях, программисты используют понятия «аргумент» и «параметр» почти как равнозначные. В зависимости от параметров функция принимает те или иные типы аргументов — значения, которые вы передаете в функцию при ее вызове. Чтобы избежать путаницы, в этой книге мы в обоих случаях будем использовать термин «аргумент».

МИССИЯ 41. ПОСАДИТЕ ЛЕС

Итак, теперь ваша миссия — посадить лес в мире Minecraft. Поскольку лес — это множество деревьев, мы напишем функцию, которая создает одно дерево, и будем вызывать ее до тех пор, пока не вырастим лес.

Основа кода для этой миссии показана в листинге 8.1.

forest.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

❶ def growTree(x, y, z):
    # Создаем дерево в заданных координатах
    # Добавьте сюда свой код

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

❷ growTree(x + 1, y, z)
```

Листинг 8.1. Основа кода для создания леса из отдельных деревьев

Grow tree — выращиваем дерево

Forest — лес

Functions — функции

Созданная в этом коде функция `growTree()` ❶ принимает в качестве аргументов координаты места, в котором должно появиться дерево. Ваша задача — добавить в тело функции код, создающий дерево в этих координатах. Для этого вам понадобятся функции `setBlock()` и `setBlocks()`.

Введите код листинга 8.1 в новый файл и сохраните его как *forest.py* в новой папке *functions*.

Когда вы добавите код, создающий нечто похожее на дерево, и убедитесь, что после запуска программы дерево появляется в игре, переходите к следующему шагу. Добавьте после первого вызова функции ❷ другие вызовы с различными значениями аргументов, чтобы деревья «вырастали» в разных местах. Постарайтесь сделать так, чтобы после запуска программы перед игроком возникло не менее девяти деревьев. На рис. 8.1 показаны деревья, созданные моей программой.

Рис. 8.1. У меня получился ровный ряд деревьев

БОНУСНОЕ ЗАДАНИЕ: СЛУЧАЙНЫЙ ЛЕС

Воспользуйтесь функцией `randint()` из модуля `random`, чтобы создать лес со случайным расстоянием между деревьями.

Randint (*сокр.* от random int) — случайное целое число

Рефакторинг кода

В Python-программах, которые вы напишете, часто один и тот же фрагмент кода будет повторяться несколько раз. Изменять такой код, внося одинаковые правки в нескольких местах, неудобно. Возможно, вы уже не раз попадали в подобную ситуацию, однако решение этой проблемы есть.

Сделайте код своих программ более удобным для чтения. Для этого нужно вынести повторяющиеся фрагменты кода в отдельные функции, которые можно вызывать сколько угодно раз. Тогда вместо нескольких правок одного и того же кода будет достаточно отредактировать код одной функции. Кроме того, размер программы существенно уменьшится, и дорабатывать ее станет проще. Подобное упрощение кода называется *рефáкторинг*.

Следующая программа здоровается с тремя людьми, поочередно спрашивая имя каждого и выводя персональное приветствие:

```
name1 = input("Привет! Как тебя зовут? ")
print("Приятно познакомиться, " + name1)
name2 = input("Привет! Как тебя зовут? ")
print("Приятно познакомиться, " + name2)
name3 = input("Привет! Как тебя зовут? ")
print("Приятно познакомиться, " + name3)
```

Одинаковый код повторяется здесь трижды. Но что если вам понадобится изменить текст вопроса или приветствия? Пока речь идет о трех повторах, это не проблема, однако представьте, что нужно поприветствовать 100 человек!

Вместо того чтобы копировать код, можно вынести повторяющийся фрагмент в функцию и вызвать ее три раза. Вот код программы после рефакторинга:

```
def helloFriend():
    name = input("Привет! Как тебя зовут? ")
    print("Приятно познакомиться, " + name)

helloFriend()
helloFriend()
helloFriend()
```

Как и прежде, программа задает вопрос и выводит приветствие, делая это три раза:

```
Привет! Как тебя зовут? Артур
Приятно познакомиться, Артур
Привет! Как тебя зовут? Все еще Артур
Приятно познакомиться, Все еще Артур
Привет! Как тебя зовут? Конечно, Артур!
Приятно познакомиться, Конечно, Артур!
```

Новая версия программы работает так же, как и прежняя, однако теперь код легче читать и изменять.

МИССИЯ 42. ДА ЗДРАВСТВУЕТ РЕФАКТОРИНГ!

Бывает так, что, уже написав программу, вы понимаете: нужно было использовать функции (со мной это случается постоянно). Рефакторинг кода с помощью функций — крайне полезный навык.

В ходе этой миссии вам предстоит поупражняться в рефакторинге, заменив повторяющийся несколько раз код вызовами функции.

Код листинга 8.2 три раза, с паузой в 10 секунд, помещает под ноги игрока арбузный блок. Сейчас для этого используются повторяющиеся фрагменты кода. Результат работы программы показан на рис. 8.2.

melonFunction.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

pos = mc.player.getPos()
x = pos.x
y = pos.y
z = pos.z
mc.setBlock(x, y - 1, z, 103)
time.sleep(10)

pos = mc.player.getPos()
x = pos.x
y = pos.y - 1
z = pos.z
mc.setBlock(x, y, z, 103)
time.sleep(10)

pos = mc.player.getPos()
x = pos.x
y = pos.y - 1
z = pos.z
mc.setBlock(x, y, z, 103)
time.sleep(10)
```

Листинг 8.2. Код, который нуждается в рефакторинге

Не очень-то симпатичный код, верно? Многие команды в нем повторяются, и это верный знак того, что нужно провести рефакторинг, заменив их на функции.

Определите, какие фрагменты кода повторяются, чтобы понять, что должна делать функция.

Измените код так, чтобы программа создавала шесть арбузов с помощью шести вызовов вашей функции. Введите код листинга 8.2 в новый файл IDLE и сохраните его под именем *melonFunction.py* в папке *functions*. Затем проведите рефакторинг: скопируйте повторяющийся фрагмент кода и перенесите его в функцию, а повторы удалите. Назовите функцию `makeMelon()`.

Melon function — функция Арбуз

Make melon — создать арбуз

Рис. 8.2. Три аппетитных подземных арбуза

БОНУСНОЕ ЗАДАНИЕ: ПОДНОЖНЫЕ БЛОКИ

Добавьте в функцию `makeMelon()` аргументы, определяющие тип блока, продолжительность паузы, а также глубину, на которой должен создаваться блок.

Комментирование с помощью строк документации

Комментарии в Python-программах поясняют, как работает код. При выполнении кода комментарии никак не влияют на работу программы, поскольку она их попросту не замечает. Основное предназначение комментариев — объяснить логику работы программы людям, которые изучают ваш код или хотят использовать его в своих целях. Помимо этого, комментарии напомнят вам, как устроена программа, когда вы вернетесь к ней в будущем.

Поскольку функции предназначены для повторного использования, в коде желательно указывать их назначение. Для этого мы будем использовать развернутые пояснения, которые называются *строками документации* (по-английски — *docstrings)*. По сути, строки документации работают как многострочный комментарий. Он располагается в начале функции и описывает ее работу.

Duplicate word — повтор слова

В следующем примере функция `duplicateWord()` содержит docstring, который поясняет ее назначение:

```
def duplicateString(stringToDbl):
❶    """Печатает содержимое строки дважды.
    Аргумент stringToDbl должен быть строковым"""
    print(stringToDbl * 2)
```

Располагаться docstring должен в самом начале тела функции ❶, а начинаться и заканчиваться — тремя кавычками подряд (""" """). При этом он может занимать любое количество строк.

Переносы строк в списке аргументов

Чтобы длинные списки аргументов было легче воспринимать, Python позволяет разделить их на несколько строк. В этой программе аргументы функции `setBlocks()` расположены на двух строчках, что делает код более удобным для чтения:

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getPos()
width = 10
height = 12
length = 13
block = 103
mc.setBlocks(pos.x, pos.y, pos.z,
             pos.x + width, pos.y + height, pos.z + length, block)
```

Возвращаемое значение функции

Функции бывают двух видов: *не возвращающие значение* и *возвращающие*. Все функции, которые мы создавали до сих пор, ничего не возвращали. Теперь разберемся, что представляет собой второй вид функций.

Умение функции возвращать значение — очень полезная способность, ведь это позволяет обрабатывать данные в теле функции и помещать результат в основной код.

Допустим, вы продаете булочки, а чтобы продажа одной булочки принесла хорошую прибыль, рассчитываете ее отпускную цену так: прибавляете к сумме, затраченной на изготовление булочки (себестоимости), две золотые монеты и умножаете результат на 10. Если вы передадите эти расчеты в функцию, которая возвращает значение (отпускную цену), то сможете использовать ее в программе сколько угодно раз.

Функция возвращает значение с помощью ключевого слова `return`. Так что функция для расчета цены булочки будет выглядеть так:

Return — возврат

```
def calculateCookiePrice(cost):
    price = cost + 2
    price = price * 10
    return price
```

Итак, чтобы вернуть значение из функции, достаточно использовать оператор `return`, указав после него нужную переменную — в данном случае отпускную цену. Затем функцию можно вызывать в любом месте кода, в котором понадобится это значение. Присвоим переменной `priceOfCookie` результат функции `calculateCookiePrice()`, которая содержит в качестве аргумента себестоимость булочки (6 монет):

Price of cookie — цена булочки

Calculate cookie price — посчитать цену булочки

```
priceOfCookie = calculateCookiePrice(6)  # Вернет число 80
```

Значение, которое возвращает функция, можно сохранить в переменной (это удобно, если обращаться к нему придется не один раз) или передать в качестве аргумента в другую функцию. Можно даже использовать вызов одной функции в качестве аргумента другой функции.

Если же функция ничего не возвращает, ее нельзя использовать в качестве значения переменной. Давайте посмотрим, в чем разница.

Number of chickens — количество цыплят

```
def numberOfChickens():
    return 5

coop = numberOfChickens()
print(numberOfChickens())
```

Запустите этот код и посмотрите, что получится. Результат, который возвращает функция, можно расценивать как обычное значение (число или строку) и даже производить с ним математические действия. В следующем примере я прибавляю к значению, которое вернула функция, число 4 и сохраняю результат в переменной `extraChickens`.

Extra chickens — еще цыплята

```
extraChickens = 4 + numberOfChickens()  # Получится 9
```

Обратите внимание: в теле следующей функции нет команды `return`. Эта функция ничего не возвращает, а значит, ее нельзя использовать там, где может понадобиться ее значение. Функцию можно только вызвать:

```
def chickenNoise():
    print("Кудах-тах-тах")

chickenNoise()
```

Если запустить этот код, на экране появится слово `"Кудах-тах-тах"`, но в составе других команд вызов функции `chickenNoise()` использовать не получится, ведь она ничего не возвращает. Попытаемся склеить вызов этой функции со строкой:

Chicken noise — кудахтанье

```
multipleNoises = chickenNoise() + ", ко-ко-ко"
```

Запустив эту программу, вы получите сообщение об ошибке:

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    multipleNoises = chickenNoise + ", ко-ко-ко"
TypeError: unsupported operand type(s) for +: 'function' and 'str'
```

В сообщении говорится, что склеить вызов этой функции со строкой невозможно, поскольку объединять тут нечего — ведь функция не возвращает значение.

Я могу изменить код, чтобы функция стала его возвращать, а не просто выводила на экран:

```
def chickenNoise():
    return "Кудах-тах-тах"

multipleNoises = chickenNoise() + ", ко-ко-ко"
print(multipleNoises)
```

Тогда программа запустится без ошибок и выведет на экран:

```
Кудах-тах-тах, ко-ко-ко
```

Запомните это различие. Используйте команду `return`, если вам нужно, чтобы функция вернула значение, и не используйте ее, если ничего возвращать не надо. Чем больше вы создадите функций, тем проще вам будет определить, нужно ли возвращать значение.

МИССИЯ 43. НАПОМИНАЛКА ТИПОВ БЛОКОВ

В Minecraft так много разных блоков, что держать в памяти идентификаторы всех типов сложно. Я хорошо помню идентификаторы блоков «арбуз» (103) и «воздух» (0), но все остальные забываю. В итоге мне приходится строить дома из арбузов!

Поэтому я хочу, чтобы вы создали для меня программу, которая будет напоминать идентификаторы блоков. В этой программе окажется много

функций. Имя каждой из них должно соответствовать типу блока, идентификатор которого функция возвращает.

Melon — арбуз

В листинге 8.3 создана функция с именем `melon()`, которая возвращает идентификатор блока «арбуз» (103).

blockIds.py

```
def melon():
    """Возвращает идентификатор блока 'арбуз'"""
    return 103
```

Листинг 8.3. Начало кода, который напоминает пользователю идентификаторы разных типов блоков

Block IDs — идентификаторы блоков

Создайте в IDLE новый файл *blockIds.py* и сохраните его в папке *functions*. Добавьте в файл код листинга 8.3 и создайте функции, возвращающие идентификаторы следующих блоков (см. раздел «Идентификаторы блоков» на с. 350):

- вода
- книжная полка
- стоячая лава
- динамит
- мак
- алмазный блок

После создания этих функций проверьте их работу, установив соответствующие блоки. Поскольку каждая функция возвращает идентификатор блока, ее значение можно присвоить переменной, а затем передать эту переменную в функцию `setBlock()`. Начните код так:

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

# Здесь должны быть функции

block = melon()
pos = mc.player.getTilePos()
mc.setBlock(pos.x, pos.y, pos.z, block)
```

На рис. 8.3 показан результат работы такой программы: функция `melon()` вернула идентификатор блока «арбуз», и программа создала этот блок в текущей позиции игрока.

Рис. 8.3. Благодаря созданным функциям мне больше не нужно вспоминать идентификаторы блоков

Чтобы установить в игровом мире алмазный блок, динамит или блок иного типа, создайте функцию, которая возвращает его идентификатор. Затем вызовите эту функцию в программе — так же как я в этом примере вызываю функцию `melon()`.

БОНУСНОЕ ЗАДАНИЕ: ЕЩЕ БОЛЬШЕ БЛОКОВ

Добавьте функции и для других типов блоков, каких захотите.

If и while внутри функций

Из глав 6 и 7 вы узнали, что конструкции `if` можно помещать внутрь других конструкций `if`, а циклы `while` — внутрь других циклов `while`. Более того, можно помещать `if` в `while`, и наоборот! В этом разделе мы разберемся, как использовать `if` и `while` в теле функций. Это сделает ваши функции более гибкими, поскольку в них можно будет проверять условия и повторять фрагменты кода.

Конструкция if

Синтаксис для создания конструкции `if` в теле функции ничем не отличается от синтаксиса обычной `if`. Нужно лишь не забыть увеличить отступы еще на четыре пробела, чтобы Python понял, что это часть тела функции.

Следующая функция принимает строку с названием числа и возвращает соответствующее целое число. Приняв строку `"четыре"`, она вернет `4`:

Word to number — число в виде слова

Num to convert (num — *сокр.* от number) — число для преобразования

```
def wordToNumber(numToConvert):
    """Преобразует строку с названием числа в целое число"""
    if numToConvert == "один":
        numAsInt = 1
    elif numToConvert == "два":
        numAsInt = 2
    elif numToConvert == "три":
        numAsInt = 3
    elif numToConvert == "четыре":
        numAsInt = 4
    elif numToConver == "пять":
        numAsInt = 5

    return numAsInt
```

Давайте рассмотрим еще один пример. Следующая функция проверяет, встречались ли вы с человеком прежде, и в зависимости от этого выводит разные приветствия:

```
❶ def chooseGreeting(metBefore):
    """Выбирает приветствие в зависимости от того, виделись ли вы
    прежде. Аргумент metBefore должен быть булевым значением"""
    if metBefore:
❷       print("Давно не виделись")
    else:
❸       print("Приятно познакомиться")

chooseGreeting(True)
chooseGreeting(False)
```

Choose greeting — выбрать приветствие

Meet before — встречались прежде

Функция `chooseGreeting()` принимает один аргумент булева типа `metBefore` ❶. Внутри функции находится конструкция `if`, которая в зависимости от значения этого аргумента выбирает одно из двух сообщений. Если значение аргумента `True`, на экране появится сообщение `"Давно не виделись"` ❷, а если `False`, то `"Приятно познакомиться"` ❸.

МИССИЯ 44. ЦВЕТ ШЕРСТИ

Вы уже знаете, что аргументами функций `setBlock()` и `setBlocks()` являются координаты блока и его идентификатор. Однако у этих функций есть еще один, необязательный аргумент — *состояние блока*.

Все блоки в Minecraft находятся в одном из шестнадцати состояний, которые обозначаются числами от 0 до 15. Для шерстяных блоков каждому состоянию соответствует свой цвет. С блоком «динамит» (идентификатор 46) ничего не случится, если вы ударите по нему в его обычном состоянии (состояние 0), однако в состоянии 1 он взорвется. При этом, хотя у блока любого типа есть все 16 состояний, они не обязательно различаются между собой.

Чтобы задать состояние блока, функциям `setBlock()` или `setBlocks()` нужно передать дополнительный аргумент.

Следующий код создает блок шерсти розового цвета:

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

block = 35
state = 6
# Создаем блок розовой шерсти
mc.setBlock(10, 3, -4, block, state)

# Создаем кубоид из розовой шерсти
mc.setBlocks(11, 3, -4, 20, 6, -8, block, state)
```

Шерсть (идентификатор 35) — очень полезный материал, в том числе и потому, что может быть разных цветов. Однако запомнить, какой цвет соответствует какому состоянию, непросто. К счастью, это делать не обязательно, ведь за вас состояния может помнить программа.

Ваша задача — написать код, который хранит состояния для шерстяных блоков. В нем должна быть функция, принимающая в качестве аргумента строку с названием цвета и возвращающая целое число — состояние шерстяного блока. Основная часть кода будет располагаться в теле функции, однако понадобится еще несколько команд, чтобы запросить у пользователя значение — название цвета — и создать блок. Чтобы задать цвет блока, программа будет обращаться к вашей новой функции.

Вам понадобятся состояния блока для разных цветов шерсти — их можно найти в разделе «Идентификаторы блоков» на с. 350. Ниже привожу основу кода, чтобы вам было с чего начать (розовому цвету соответствует состояние 6).

woolColors.py

```
def getWoolState(color):
    """Принимает строку с названием цвета (color), возвращает
    состояние блока шерсти для этого цвета"""
❶   if color == "розовый":
        blockState = 6
    elif  # Добавьте в конструкцию elif команды для других цветов
        # Верните значение blockState (номер состояния)

❷ colorString = input("Введите цвет блока: ")
state = getWoolState(colorString)

❸ pos = mc.player.getTilePos()
mc.setBlock(pos.x, pos.y, pos.z, 35, state)
```

Get wool state — принять состояние шерсти

Я уже начал создавать функцию `getWoolState()`. Сейчас она содержит лишь одну конструкцию `if` для розового цвета ❶. Также в программе есть код, который запрашивает название цвета ❷, и код для создания блока шерсти в позиции игрока ❸.

Добавьте в тело функции `getWoolState()` конструкции `elif` для остальных цветов шерсти. Функция должна принимать строку с названием цвета и возвращать целое число — состояние блока. Например, для значения `"розовый"` функция вернет число 6. Дополнительные подсказки смотрите в комментариях.

Wool colors — цвета шерсти

Сохраните файл с кодом как *woolColors.py* в папке *functions*.

Вы можете сделать программу более дружелюбной для пользователя, выводя в чат сообщение, если он указал неверное название цвета. На рис. 8.4 показан созданный в игре блок шерсти и ввод пользователя в окне консоли Python.

Цикл while

Циклы `while` тоже можно использовать в коде функции — синтаксис будет таким же, как у обычных циклов. Нужно лишь сделать отступ в начале каждой строки в четыре пробела, чтобы Python понял, что цикл находится внутри функции.

To print — напечатать

В следующем примере значение аргумента `toPrint` выводится на экран в цикле `while`, вложенном в функцию. Количество итераций цикла определяет значение аргумента `repeats`:

```
def printMultiple(toPrint, repeats):
    """Выводит строку столько раз, сколько указано в аргументе
    repeats"""
    count = 0
    while count < repeats:
```

```
        print(toPrint)
        count += 1
```

Рис. 8.4. Теперь я могу создать блок шерсти, указав название цвета

Внутри одной функции вы также можете совместить цикл `while` и оператор `return`. В большинстве случаев `return` должен находиться вне цикла (если `return` окажется внутри, цикл тут же завершится и функция вернет значение). Например:

```
def doubleUntilHundred(numberToDbl):
    """Удваивает число, пока оно не достигнет 100.
    Возвращает количество удвоений, которые для этого потребовались"""
    count = 0
    while numToDbl < 100:
        numberToDbl = numberToDbl * 2
        count += 1
❶   return count

print(doubleUntilHundred(2))
```

Эта программа удваивает число, пока оно не станет больше или равно 100. Затем возвращает количество сделанных удвоений ❶.

Также внутрь циклов можно помещать вызовы функций (как мы это делали в предыдущих главах).

МИССИЯ 45. БЛОКИ ПОВСЮДУ

Вы уже знаете, что внутрь функций можно помещать циклы, конструкции и другие функции и использовать в них значения передаваемых в функцию аргументов. Например, для цикла можно брать из аргумента количество итераций, а для функции `setBlock()` — тип блока и другие значения.

В ходе этой миссии вам предстоит создать функцию, которая помещает блоки в случайные места мира Minecraft. Количество этих блоков и их тип должны определяться аргументами функции.

Код из этой миссии может произвести значительные разрушения. Чтобы не лишиться своих драгоценных построек, опробуйте его в новом мире Minecraft.

Код листинга 8.4 создает в случайном месте арбуз.

blocksEverywhere.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import random

def randomBlockLocations(blockType, repeats):
❶    count = 0
❷    # Создайте здесь цикл
    x = random.randint(-127, 127)
    z = random.randint(-127, 127)
❸    y = mc.getHeight(x, z)
    mc.setBlock(x, y, z, blockType)
    count += 1
```

Листинг 8.4. Эта функция помещает арбузный блок в случайное место

Blocks everywhere — блоки везде

Random block locations — случайные позиции блоков

Введите код листинга 8.4 в новый файл IDLE и сохраните его под именем *blocksEverywhere.py* в папке *functions*. Внутри функции `randomBlockLocations()` создайте цикл `while`, в котором будут повторяться две команды, следующие ниже строки ❷. Переменная `count` ❶ поможет вам следить за количеством повторов цикла. Сравните аргумент `repeats` (необходимое количество повторов) с переменной `count` в условии цикла, чтобы ограничить количество итераций. Добавьте в начало строк, которые идут после ❷, дополнительный отступ, чтобы они

оказались внутри цикла. Функция `getHeight()` нужна здесь для того, чтобы создавать блоки над землей ❸.

Добавьте три вызова функции `randomBlockLocations()`. Пусть первый вызов создает 10 блоков, второй — 37, а третий — 102. Выберите любые типы блоков, какие захотите.

Сохраните программу и запустите ее. Она должна создавать блоки в случайных местах игрового мира. Пример работы программы показан на рис. 8.5.

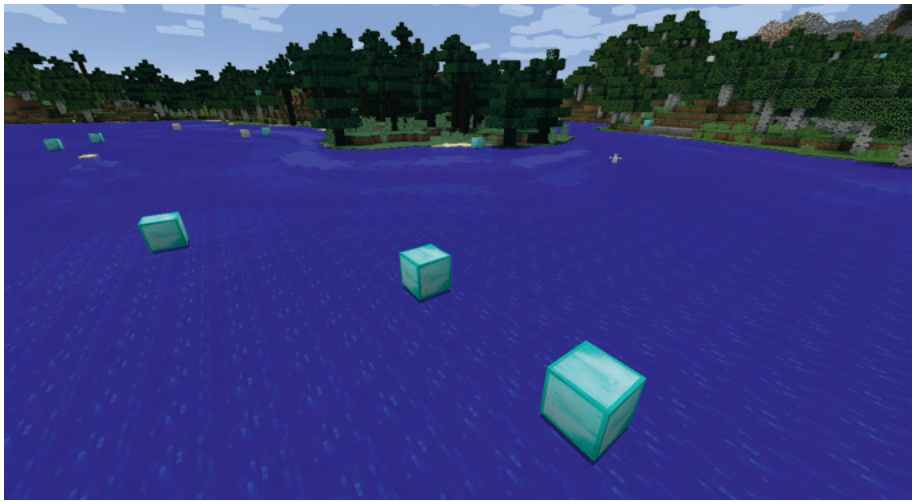

Рис. 8.5. Здесь видны некоторые блоки, которые появились в случайных местах. Чтобы программа не повредила мои постройки, перед ее запуском я создал новый игровой мир

Глобальные и локальные переменные

С созданием функций связана одна особенность, а именно *область видимости переменных*. Область видимости определяет, каким образом программа может обращаться к содержимому этой переменной. Давайте разберемся на примере. Следующий код должен увеличивать количество яиц, припасенных для пикника:

```
❶ eggs = 12

  def increaseEggs():
❷     eggs += 1
      print(eggs)

  increaseEggs()
```

Здесь используются две переменные с именем `eggs` — одна внутри функции ❷, а другая вне ее ❶. Выглядит этот код вполне невинно, однако при попытке запустить программу Python выдаст ошибку. Вот фрагмент сообщения об ошибке:

```
UnboundLocalError: local variable 'eggs' referenced before assignment
```

Eggs — яйца

Проблема в том, что переменная `eggs` находится вне функции и, когда мы пытаемся обратиться к ней из функции, Python эту переменную не видит. С точки зрения Python, переменная в коде функции отличается от переменной в основном коде, даже если их имена совпадают. Так задумано специально, чтобы переменные, которые находятся внутри различных функций, не стали причиной неожиданных ошибок в работе программы, если их имена случайно совпадут.

В Python есть два способа объявить переменную: вы можете создать *глобальную* переменную — ее можно будет использовать во всей программе или во всем файле, либо создать переменную *локальную* — тогда ее значение будет доступно лишь внутри определенной функции или цикла. Иными словами, вы можете либо использовать одну и ту же переменную и внутри, и снаружи функции, либо создать две переменные, которые будут относиться к разным частям кода (даже если их имена совпадают).

Глобальная переменная постоянна: если изменить ее значение внутри функции, изменится и значение вне функции, и наоборот. Для того чтобы объявить глобальную переменную, используйте ключевое слово `global` ❶:

Global — глобальная

```
eggs = 12

def increaseEggs():
❶     global eggs
    eggs += 1
    print(eggs)

increaseEggs()
```

После запуска этого кода на экране появится новое значение переменной `eggs` — число 13.

Переменные, объявленные без ключевого слова `global`, являются локальными. В этом случае переменные с одинаковым именем снаружи и внутри функции будут совершенно разными. Если изменить переменную внутри функции, на переменную снаружи это никак не повлияет, и наоборот. Вы можете сделать переменную `eggs` локальной ❶, изменив код примера следующим образом:

```
eggs = 12

def increaseEggs():
❶     eggs = 0
      eggs += 1
❷     print(eggs)

increaseEggs()
❸ print(eggs)
```

Когда функция выполнится, произойдет вывод значения `eggs` на экран ❷. Мы увидим число 1, ведь значение `eggs` вне функции не влияет на одноименную локальную переменную в теле функции. Получается, что внутри функции `increaseEggs()` переменная `eggs` примет значение 1, тогда как переменная `eggs` снаружи функции будет по-прежнему хранить число 12 ❸.

Increase eggs — увеличение количества яиц

МИССИЯ 46. САМОДВИЖУЩИЙСЯ БЛОК

Однажды я подумал: было бы здорово сделать особенный блок, который путешествует по миру Minecraft сам по себе. Каждую секунду он движется вперед, а столкнувшись с деревом, стеной или еще чем-нибудь высоким, поворачивает в другую сторону. Ну а если блок упадет в какую-нибудь яму, выбраться оттуда он уже не сможет.

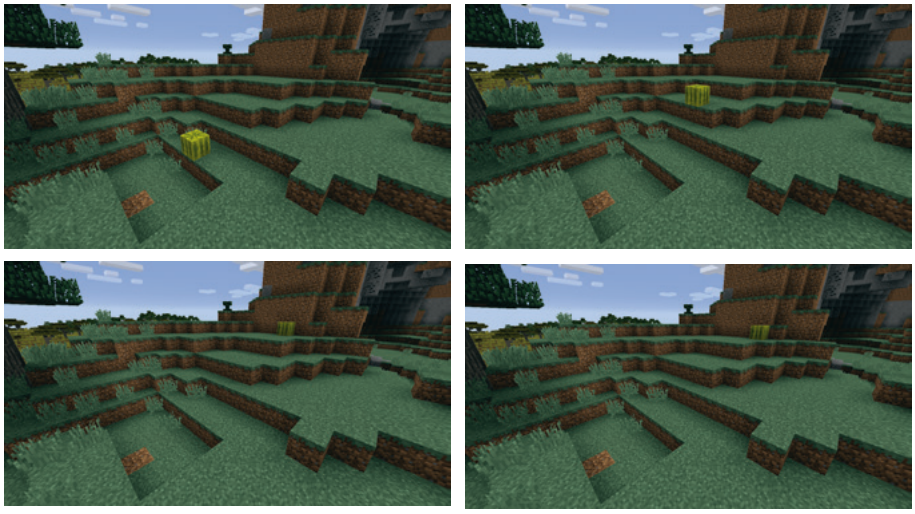

Рис. 8.6. Арбузный блок преодолел ступеньки, добрался до стены и двигается вдоль нее

В листинге 8.5 показана основа кода, с помощью которого можно создать волшебный самодвижущийся блок.

movingBlock.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

def calculateMove():
    """Меняет x- и z-координаты самодвижущегося блока. Если перед ним
    окажется препятствие высотой не более 2 блоков, блок продолжит
    двигаться вперед. В противном случае он попробует повернуть налево,
    затем назад и, наконец, направо."""
❶   # Объявите здесь глобальные переменные

    currentHeight = mc.getHeight(x, z) - 1

    forwardHeight = mc.getHeight(x + 1, z)
    rightHeight = mc.getHeight(x, z + 1)
    backwardHeight = mc.getHeight(x - 1, z)
    leftHeight = mc.getHeight(x, z - 1)
    if forwardHeight - currentHeight < 3:
        x += 1
    elif rightHeight - currentHeight < 3:
        z += 1
    elif leftHeight - currentHeight < 3:
        z -= 1
    elif backwardHeight - currentHeight < 3:
        x -= 1

    y = mc.getHeight(x, z)

pos = mc.player.getTilePos()
x = pos.x
z = pos.z
y = mc.getHeight(x, z)

while True:
    # Рассчитываем траекторию перемещения блока
    calculateMove()

    # Устанавливаем блок
    mc.setBlock(x, y, z, 103)

    # Пауза
    time.sleep(1)

    # Уничтожаем блок
    mc.setBlock(x, y, z, 0)
```

Листинг 8.5. Код для создания самодвижущегося блока

Однако сейчас этот код не работает, поскольку переменные в функции `calculateMove()` не являются глобальными.

Ваша задача — дописать код листинга 8.5. Введите его в окне программы IDLE и сохраните под именем *movingBlock.py* в папке *functions*. Добавьте в начало создаваемой функции `calculateMove()` строки, объявляющие глобальные переменные `x`, `y` и `z` ❶.

Запустите программу — вы должны увидеть движущийся блок. На рис. 8.6 показано, как блок преодолевает низкие препятствия, приближается к стене и начинает ее огибать.

Calculate move — вычисляем перемещение

Moving block — движущийся блок

БОНУСНОЕ ЗАДАНИЕ: УМНЫЙ АРБУЗНЫЙ БЛОК

Запустив программу *movingBlock.py*, вы можете заметить, что блок чаще всего перемещается вдоль оси x, иногда попадая из-за этого в ловушку между двумя блоками. Дело в том, что код не учитывает направление, в котором блок двигался на предыдущем шаге, и всегда первым делом старается переместить его вдоль оси x. Постарайтесь разобраться, как можно сохранить последнее направление движения блока, и изменить конструкцию `if` так, чтобы программа сначала пыталась переместить блок в этом сохраненном направлении.

Что вы узнали

Ура! В этой главе вы научились создавать функции. Теперь вы можете добавлять в их коды циклы и конструкции `if`, а также возвращать из функций значения.

В главе 9 вы познакомитесь со списками, которые позволяют хранить несколько значений в единственной переменной.

9

СПИСКИ, СЛОВАРИ И УДАРЫ ПО БЛОКАМ

В повседневной жизни мы составляем списки покупок, вещей в дорогу или перечисляем по пунктам порядок каких-либо действий. В Python *списки* устроены похожим образом: они позволяют хранить целые наборы данных в определенной последовательности. Данными могут быть строки, числа, булевы значения и даже другие списки.

Обычная переменная содержит единственное значение. Списки же позволяют помещать в одну переменную множество значений, например числа от 1 до 100 или имена друзей. В других языках программирования списки часто называются *массивами*.

При создании программ для Minecraft в списках можно хранить идентификаторы блоков, координаты игрока и много чего еще. Всё это добавляет вашим программам гибкости.

В этой главе вы узнаете, как с помощью списков сыграть в мини-игру, ведущую учет высот, как создать столбик-секундомер и написать программу, которая заставляет игрока скользить по игровому миру.

Работа со списками

Создать список несложно — для этого заключите набор перечисленных через запятую значений в квадратные скобки. Впрочем, в квадратных скобках может и не быть значений — тогда получится *пустой список*.

Список ингредиентов для вермишелевого супа может выглядеть так:

```
>>> noodleSoup = ["вода", "соевый соус", "зеленый лук", "вермишель",
                  "говядина"]
```

Список `noodleSoup` содержит несколько значений, и все они являются строками. Пустой список выглядит так:

Noodle soup — вермишелевый суп

```
>>> emptyList = []
```

Он ничего не содержит, однако вы можете добавить в него значения позже.

В Python можно не только хранить в списках любые значения, но и объединять значения разных типов в один список. Например, в этом списке хранятся две строки и одно целое число:

```
>>> wackyList = ["кофта", 33, "воздушный шар"]
```

Списки бывают очень длинными и потому сложными для восприятия. Чтобы код было удобнее читать, длинный список можно разбить на несколько строк — на работе программы это никак не скажется. Следующий список ингредиентов для вермишелевого супа полностью повторяет список `noodleSoup` из примера в начале раздела:

```
>>> noodleSoup = ["вода",
                  "соевый соус",
                  "зеленый лук",
                  "вермишель",
                  "говядина"]
```

А теперь давайте научимся получать доступ к данным списка и изменять их.

Доступ к элементам списка

Чтобы получить значение из списка (эти значения называются *элементами*), нужно указать его позицию — так называемый *индекс*. Узнать первый элемент списка ингредиентов для вермишелевого супа можно так:

```
>>> print(noodleSoup[0])
вода
```

Обратите внимание: первому элементу соответствует индекс 0. Соответственно, у второго элемента будет индекс 1, у третьего — 2 и так далее. Дело в том, что, работая со списками, компьютер ведет отсчет элементов с нуля.

Отсчет с нуля может показаться странным, однако для этого есть причины: первые компьютеры были очень медленными и обладали малым объемом памяти. Отсчет с нуля позволял повысить быстродействие и эффективность программ. И хотя с тех пор компьютеры стали гораздо мощнее, они по-прежнему считают с нуля.

Попытавшись обратиться к элементу, которого нет в списке, вы получите сообщение об ошибке. Попробуем получить элемент с индексом 5 (хотя всего в списке 5 элементов, а значит, индекс последнего из них — 4):

```
>>> print(noodleSoup[5])
```

Вот выдержка из сообщения об ошибке:

```
IndexError: list index out of range
```

Index error: list index out of range — ошибка в индексе: индекс находится за пределами допустимых значений

Сообщение `IndexError: list index out of range` означает, что по указанному индексу в списке ничего нет, поскольку позиция 5 находится за пределами списка. Python не может вернуть значение, которого нет!

Изменение элементов списка

Отдельные значения, хранящиеся в списке, можно менять — точно так же как можно менять значения переменных. Это допустимо, потому что списки являются *изменяемыми*. Чтобы присвоить элементу новое значение, нужно обратиться к нему по индексу и задать значение так же, как мы присваивали его переменной (при помощи знака «равно»).

Давайте заменим ингредиент `говядина` в списке `noodleSoup` на `курица`. Это пятый элемент списка, а значит, ему соответствует индекс 4 (вспомните: мы ведем отсчет от 0). Присвоить элементу 4 новое значение можно так:

```
>>> noodleSoup[4] = "курица"
```

А теперь воспользуемся списками, чтобы сделать кое-что интересное в мире Minecraft.

МИССИЯ 47. ВЫСОКО И НИЗКО

Я люблю вспоминать свои путешествия по игровому миру. Мне нравится исследовать разные территории — от высоких гор до глубоких пещер. Порой мы с друзьями соревнуемся, кто заберется выше других или спустится ниже всех. Чтобы исключить жульничество, я написал программу, которая запоминает *y*-координаты самой низкой и самой высокой точек среди тех, которые игрок посетил в течение 60 секунд.

Основа программы показана в листинге 9.1. Введите этот код в новый файл IDLE и сохраните его под именем *highAndLow.py* в новой папке *lists*.

High and low — высоко и низко

Lists — списки

highAndLow.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

❶ heights = [100, 0]
  count = 0

  while count < 60:
      pos = mc.player.getTilePos()

      if pos.y < heights[0]:
❷         # Присвойте наименьшей высоте значение y-координаты игрока
      elif pos.y > heights[1]:
❸         # Присвойте наибольшей высоте значение y-координаты игрока

      count += 1
      time.sleep(1)

❹ mc.postToChat("Низшая позиция: ")  # Получите из списка наименьшую высоту
❺ mc.postToChat("Высшая позиция: ")  # Получите из списка наибольшую высоту
```

Листинг 9.1. Основа программы, которая запоминает и выводит в чат наименьшее и наибольшее значение y-координаты игрока

Heights — высóты

Эта программа должна сохранять в списке `heights` ❶ наименьшую и наибольшую из *y*-координат игрока за все время его перемещений. Первый элемент списка (с индексом 0) содержит наименьшее значение, а второй (с индексом 1) — наибольшее. При этом «наименьший» элемент должен содержать достаточно большое число, а «наибольший» — маленькое, ведь программа выведет в чат лучшие показатели игрока, только если они превзойдут начальные значения `heights`. Иначе — напечатает именно их. Я взял для «наименьшего» элемента значение 100, а для «наибольшего» — 0.

Цикл `while` работает в течение 60 секунд, выполняя по одной итерации в секунду и постоянно обновляя значения в списке `heights`. Условие конструкции `if` проверяет, меньше ли текущая *y*-координата игрока, чем значение «наименьшей» высоты в списке. Затем конструкция `elif` проверяет, больше ли *y*-координата игрока, чем «наибольшее» значение высоты.

Чтобы программа работала, вам нужно сохранять низшую позицию игрока `pos.y` в элементе с индексом 0 — `heights[0]` ❷. Элементу списка значение присваивается так же, как и обычной переменной: `height[0] = pos.y`. Помимо этого, вам нужно сохранять высшую позицию игрока `pos.y` в элементе с индексом 1 — `heights[1]` ❸.

И наконец, две последние строки кода нужны для того, чтобы вывести в чат наименьшее ❹ и наибольшее ❺ значение *y*-координаты игрока. Возьмите эти значения из списка `heights` (еще раз напоминаю — элементу с наименьшей высотой соответствует индекс 0, а элементу с наибольшей высотой — индекс 1).

Запустите программу — и вперед! Посмотрим, как низко ваш игрок сможет опуститься и как высоко забраться. Через 60 секунд цикл прекратит работу и программа покажет низшую и высшую позицию игрока. Запустите программу еще несколько раз и постарайтесь побить собственный рекорд!

На рис. 9.1 показана одна из моих попыток.

Рис. 9.1. Низшая точка, на которой побывал мой игрок, — 15 блоков, а высшая — 102 блока

БОНУСНОЕ ЗАДАНИЕ: ИСПРАВЬТЕ ОШИБКУ

Сейчас начальные значения для наименьшей и наибольшей высоты в программе *highAndLow.py* равны 100 и 0. Пока во время работы программы игрок опускается ниже 100 и поднимается выше 0, проблем нет. Однако, если он не окажется ниже 100 и не поднимется выше 0, в списке останутся начальные значения и программа выдаст неверный результат. Постарайтесь разобраться, как это исправить.

Изменение структуры списка

У списков есть встроенные функции для выполнения таких операций, как добавление, вставка и удаление элемента.

Добавление элемента

Append — добавить

С помощью функции `append()` можно *добавить* новый элемент в конец списка — просто передайте этой функции аргумент со значением элемента.

Вермишелевый суп без овощей — это не дело. Давайте воспользуемся функцией `append()` и добавим в суп овощи:

```
>>> noodleSoup.append("овощи")
```

В конце списка `noodleSoup` появится еще один элемент — строка `"овощи"`.

Функция добавления элемента особенно полезна, если вы создали список пустым. Вот как можно добавить в пустой список первый элемент:

```
>>> food = []
>>> food.append("торт")
```

Вставка элемента

Insert — вставить

Элемент можно не только добавить в конец списка, но и *вставить* в середину. Функция `insert()` помещает новый элемент между двумя существующими и меняет индексы элементов, идущих после вставленного.

`Insert()` принимает два аргумента: индекс позиции, на которую вы хотите поместить элемент, и значение этого элемента.

Вот наш список `noodleSoup`:

```
>>> noodleSoup = ["вода", "соевый соус", "зеленый лук", "вермишель",
                  "говядина", "овощи"]
```

Давайте добавим в него элемент `"перец"`, поместив его в позицию с индексом 3:

```
>>> noodleSoup.insert(3, "перец")
```

После вставки список будет содержать следующие элементы:

```
["вода", "соевый соус", "зеленый лук", "перец", "вермишель", "говядина",
 "овощи"]
```

Если вы захотите вставить элемент, указав индекс, выходящий за границы списка, элемент будет добавлен в самый конец. Допустим, вы пытаетесь вставить в список из семи элементов новое значение с индексом 10:

```
>>> noodleSoup.insert(10, "соль")
```

После запуска программы `"соль"` станет последним элементом списка:

```
["вода", "соевый соус", "зеленый лук", "перец", "вермишель", "говядина",
 "овощи", "соль"]
```

Обратите внимание: новому элементу здесь соответствует не индекс 10, а индекс 7.

Удаление элемента

Del (*сокр.* от delete) — удалить

Порой бывает нужно избавиться от какого-нибудь элемента списка. Для этого в Python есть специальная команда. После ключевого слова `del` нужно ввести имя списка, а затем в квадратных скобках указать индекс удаляемого элемента.

Попробуем *удалить* из списка `noodleSoup` элемент `"говядина"`, которому сейчас соответствует индекс 5:

```
>>> del noodleSoup[5]
```

Команду `del` можно использовать вместе с функцией `index()`, которая позволяет узнать индекс элемента:

```
>>> beefPosition = noodleSoup.index("говядина")
>>> del noodleSoup[beefPosition]
```

После удаления элемента индексы следующих за ним элементов изменятся. Вот как будет выглядеть наш список после удаления говядины:

```
["вода", "соевый соус", "зеленый лук", "перец", "вермишель", "овощи",
 "соль"]
```

Индекс элемента `"овощи"` поменялся с 6 на 5, а элемента `"соль"` — с 7 на 6. Обратите внимание: индексы элементов, стоящих перед удаленным, остались прежними. Помните об этой особенности!

МИССИЯ 48. СТОЛБИК-СЕКУНДОМЕР

Давайте воспользуемся некоторыми функциями списков, чтобы создать в игре Minecraft индикатор состояния в виде столбика — наподобие того, который показывает уровень игрока в компьютерных ролевых играх.

В нашей программе индикатор будет отсчитывать 10 секунд. Сначала создадим столбик из 10 стеклянных блоков. Пусть каждую секунду нижний блок меняется на блок лазурита. На рис. 9.2 показан столбик-секундомер через 5 секунд после запуска программы.

Рис. 9.2. Столбик-секундомер окрашен на 50% (5 из 10 блоков — лазурит)

Progress bar — индикатор состояния

Откройте IDLE и создайте новый файл, сохранив его под именем *progressBar.py* в папке *lists*. Незаконченная версия программы показана в листинге 9.2. Введите этот код в окне программы.

progressBar.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

pos = mc.player.getTilePos()
x = pos.x + 1
y = pos.y
z = pos.z

# Добавьте в пустой список 10 блоков "стекло" (идентификатор 20)
❶ blocks = []
barBlock = 22  # Блок лазурита

count = 0
while count <= len(blocks):

    mc.setBlock(x, y, z, blocks[0])
    mc.setBlock(x, y + 1, z, blocks[1])
    mc.setBlock(x, y + 2, z, blocks[2])
❷   # Добавьте вызовы setBlock() для остальных блоков в списке

    count += 1

❸   # Удалите последний блок в списке

❹   # Добавьте блок лазурита в самое начало списка

    time.sleep(2)
```

Листинг 9.2. Незаконченный код для создания столбика-секундомера

Чтобы дописать программу, нужно сделать следующее:

1. Добавить 10 блоков «стекло» (идентификатор 20) в пустой список ❶.
2. С помощью функции `setBlock()` создать в игровом мире столбик из 10 блоков, указанных в списке ❷. Команды для первых трех блоков в программе уже есть.
3. Добавить команду, удаляющую последний блок (с индексом 9) из списка ❸. Напоминаю: для удаления элементов служит ключевое слово `del`.

4. Вставить в начало списка блок лазурита ❹. Воспользуйтесь для этого функцией `insert()` и переменной `barBlock`, в которой хранится идентификатор 22, чтобы поместить блок в позицию с индексом 0.

В коде есть комментарии, которые помогут найти места для вставки описанных выше команд.

БОНУСНОЕ ЗАДАНИЕ: СВЕРХУ ВНИЗ

Сейчас столбик постепенно окрашивается снизу вверх, и на этом все заканчивается. Попробуйте изменить программу так, чтобы, дойдя до верха, индикатор начинал отсчет в обратном направлении — сверху вниз.

Работа со строками как со списками

Строка — это тоже список, поскольку она содержит последовательность элементов: букв, цифр и других символов. Мы даже можем получить доступ к отдельному символу по его индексу, однако изменять их состав и последовательность с помощью функций `insert()` или `append()` не можем, поскольку строки в Python являются *неизменяемыми*.

Следующий код выведет на экран второй символ строки `"Виноград"`:

```
>>> flavor = "Виноград"
>>> print(flavor[1])
и
```

Итак, можно получить доступ к отдельным символам строки — так же как мы получаем доступ к отдельным элементам списка. Допустим, вы хотите узнать первую букву имени (firstName) и фамилии (lastName), чтобы напечатать инициалы:

```
>>> firstName = "Маркус"
>>> lastName = "Перссон"
>>> initials = firstName[0] + " " + lastName[0]
>>> print(initials)
М П
```

Новая строка `М П` составлена из символов, которые были получены по индексам. Обратите внимание: индексы внутри строк тоже отсчитываются с нуля!

Кортежи

Кортеж — это неизменяемая разновидность списка. Так же как обычный список, он может содержать значения разных типов. При создании кортежей используются круглые скобки, внутри которых элементы перечисляются через запятую.

Представьте себе спортсмена, совершившего серию прыжков в длину. Результаты этих прыжков (в метрах) можно поместить в кортеж:

```
>>> distance = (5.17, 5.20, 4.56, 5.64, 6.58, 6.41, 2.20)
```

Если спортсмен прыгнет один раз, можно создать кортеж с единственным значением, однако после него все равно следует поставить запятую:

```
>>> distance = (5.17,)
```

Давать скобки при создании кортежа необязательно, можно просто перечислить значения, разделив их запятыми:

```
>>> distance = 5.17, 5.20, 4.56, 5.64, 6.58, 6.41, 2.20
```

Чтобы получить доступ к элементам кортежа, используйте квадратные скобки — так же как при работе с обычным списком. Давайте присвоим переменной `jump` значение элемента с индексом 1:

Jump — прыжок

Distance — дальность

```
>>> jump = distance[1]
>>> print(jump)
5.20
```

Главное отличие кортежей от списков в том, что кортежи неизменяемы — их содержимое всегда остается одним и тем же. Вы не можете добавить, вставить, удалить элементы или присвоить им новые значения. Кортежи следует использовать вместо списков, если вы уверены, что изменять состав и значения элементов не понадобится.

Присвоение значений переменным с помощью кортежей

У кортежей есть полезное свойство — с их помощью можно присвоить значения сразу нескольким переменным. Это уменьшает размер кода и позволяет группировать переменные, связанные общим контекстом.

Обычно с кортежами работают так же, как со списками, — через единственную переменную:

```
measurements = 6, 30
```

Но предположим, что мы хотим хранить значения в двух переменных, а не в одной. Синтаксис для этого несложен: нужно ввести имена переменных через запятую, поставить знак «равно» и справа от «равно» поместить кортеж. Значение каждого элемента кортежа будет присвоено переменной, стоящей на соответствующем месте.

В команде ниже двум переменным, `width` и `height`, присваиваются два значения — 6 и 30 соответственно:

Width — ширина

Height — высота

```
width, height = 6, 30
```

В итоге мы имеем две переменные. В одной, под названием `width`, находится значение 6, а в другой, `height`, — 30. И добились мы этого благодаря всего лишь одной строчке кода!

МИССИЯ 49. СКОЛЬЖЕНИЕ

Присваивать значения переменным с помощью кортежей — простой и быстрый способ повысить компактность программ. Также это помогает группировать переменные, близкие по значению. До сих пор мы использовали для присвоения значений переменным `x`, `y` и `z` такой код:

```
x = 10
y = 11
z = 12
```

Вместо этого можно при помощи кортежа записать всё одной строкой:

```
x, y, z = 10, 11, 12
```

А теперь пора применить новые знания на практике! Ваша задача в этой миссии — создать программу, которая случайным образом понемногу сдвигает игрока в сторону. Выглядит это так, будто он скользит по льду. Я уже подготовил основу кода и поместил его в листинг 9.3, однако некоторых команд не хватает, и добавить их предстоит вам.

sliding.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import random
import time

❶ # Получите координаты игрока

❷ # С помощью кортежа сохраните эти координаты в переменных x, y и z

while True:
❸     x += random.uniform(-0.2, 0.2)
      # Измените значение переменной z на случайное вещественное число
❹     z +=
      y = mc.getHeight(x, z)

      mc.player.setPos(x, y, z)
      time.sleep(0.1)
```

Листинг 9.3. Основа кода для скольжения игрока

Sliding — скольжение

Uniform — равномерно

Введите код листинга 9.3 в новый файл IDLE и сохраните его под именем *sliding.py* в папке *lists*. Чтобы доделать программу, вам нужно добавить в нее код для получения начальных координат игрока ❶, а затем — код, который поместит их в переменные `x`, `y` и `z` ❷. Сделайте это с помощью кортежа. Функция `uniform()` ❸ похожа на функцию `randint()` (см. «Случайные числа» на с. 91), однако вместо целого числа она возвращает вещественное число. Воспользуйтесь этой функцией для того, чтобы изменить значение переменной `z` ❹ внутри цикла. Для переменной `x` это уже сделано ❸.

На рис. 9.3 показано, как мой игрок скользит после запуска программы.

БОНУСНОЕ ЗАДАНИЕ: СКОЛЬЗЯЩИЕ БЛОКИ

Sliding.py заставляет игрока скользить по игровому миру в случайном направлении. Постарайтесь разобраться, как можно изменить программу, чтобы она создавала скользящий блок.

Рис. 9.3. Игрок медленно скользит по своему саду

Функции, возвращающие кортеж

Некоторые из встроенных функций Python возвращают кортежи. Вы в своих функциях тоже можете так делать — для этого укажите кортеж после команды `return`.

Давайте создадим функцию, которая преобразует дату, записанную в виде строки, в числовую запись. Дату-строку сделаем аргументом, а возвращать функция будет кортеж с последовательностью чисел: день (day), месяц (month), год (year).

```
def getDateTuple(dateString):
    day = int(dateString[0:2])
    month = int(dateString[3:5])
    year = int(dateString[6:10])
    return day, month, year
```

Если передать в функцию дату-строку, функция вернет кортеж с тремя числами, стоящими в таком порядке: день, месяц, год:

```
>>> getDateTuple("27.09.2017")
(27, 9, 2017)
```

Получив из функции кортеж, мы можем поместить его содержимое куда угодно. Следующий код сохраняет значение каждого элемента кортежа в отдельной переменной:

```
day, month, year = getDateTuple("27.09.2017")
```

Теперь мы умеем быстро разделять кортеж с датой на отдельные составляющие. Я как программист использую этот прием постоянно.

Другие полезные свойства списков

Со списками можно выполнять немало разных действий. В этом разделе мы выясним, как узнать длину списка, как выбрать случайный элемент и как с помощью конструкции `if` проверить, есть ли в списке нужное значение.

Длина списка

Len (*сокр.* от length) — длина

Функция `len()` позволяет узнать длину списка. Она принимает список в качестве аргумента и возвращает количество элементов в нем. Например:

```
>>> noodleSoup = ["вода", "соевый соус", "зеленый лук", "вермишель",
                  "говядина", "овощи"]
>>> print(len(noodleSoup))
6
```

Хотя индексы в Python начинаются с нуля, подсчет элементов списка производится обычным образом. Самый большой индекс в этом примере равен 5, однако Python знает, что в списке 6 элементов!

МИССИЯ 50. УДАРЫ ПО БЛОКАМ

В Minecraft Python API есть функция, которая возвращает список нанесенных мечом ударов. Из его элементов можно извлечь координаты блоков, по которым игрок нанес эти удары. В этой и следующей главах вы убедитесь, насколько такая возможность полезна.

В ходе этой миссии ваша цель — создать небольшую игру, где будет вестись подсчет ударов по блокам, сделанных игроком за минуту. Устройте с другом соревнование: кто наберет больше очков! Программу можно будет усовершенствовать — к примеру, сделать так, чтобы она запоминала лучший результат.

На рис. 9.4 показана программа в действии.

Чтобы создать эту игру, понадобится не так уж много строчек кода. Вот что должна делать программа.

1. Подключаться к игре Minecraft.
2. Ждать 60 секунд.
3. Получать список ударов по блокам.
4. Выводить длину этого списка в чат.

Рис. 9.4. За 60 секунд мой игрок нанес 19 ударов

Единственным новым для вас кодом будет фрагмент, который предназначен для получения списка ударов:

```
blockHits = mc.events.pollBlockHits()
```

Poll block hits — список ударов по блокам

Block hits — удары по блокам

Функция `pollBlockHits()` возвращает список ударов по блокам. Мы сохраняем его в переменной `blockHits`. С этой переменной можно работать так же, как с любым другим списком, — например, получить его длину или обратиться к элементам по индексам.

В этой игре вам придется в течение минуты нажимать на правую кнопку мыши, поскольку функция `pollBlockHits()` ведет учет только тех блоков, по которым ударили мечом при помощи именно этой кнопки. В версии Minecraft для настольных компьютеров такой удар больше напоминает защиту (на рис. 9.5 показано, как это выглядит). Запомните: удары мечом с помощью левой кнопки мыши учитываться не будут, равно как и удары с помощью правой кнопки, но не мечом, а чем-нибудь еще! Меч при этом может быть любым, в том числе железным, золотым или алмазным.

Выведя на экран содержимое списка, вы увидите примерно следующее (значения будут другими, в зависимости от того, по каким блокам вы ударяли):

```
[BlockEvent(BlockEvent.HIT, 76, -2, 144, 1, 452),
 BlockEvent(BlockEvent.HIT, 79, -2, 145, 1, 452),
 BlockEvent(BlockEvent.HIT, 80, -3, 147, 1, 452),
 BlockEvent(BlockEvent.HIT, 76, -3, 149, 1, 452)]
```

В этом списке содержатся сведения о четырех ударах, включающие координаты каждого блока, по которым они были нанесены. О том, как получить эти координаты, вы узнаете в ходе миссии 55 (с. 243).

Чтобы облегчить вашу задачу, я подготовил основу кода — она показана в листинге 9.4.

swordHits.py

```
# Подключаемся к Minecraft
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

# Ждем 60 секунд
time.sleep(60)
```

```
# Получаем список ударов
❶ blockHits =

# Выводим длину списка ударов в чат
❷ blockHitsLength =
mc.postToChat("Ваш счет " + str(blockHitsLength))
```

Листинг 9.4. Основа кода для игры, ведущей подсчет количества ударов мечом по блокам

Рис. 9.5. При нажатии правой кнопки мыши игрок держит меч вот так

Запустите IDLE, создайте новый файл и введите код листинга 9.4. Сохраните программу под именем *swordHits.py* в папке *lists*. Присвойте результат функции `pollBlockHits()`, которая возвращает список ударов

Sword hits — удары мечом

по блокам, переменной `blockHits` ❶, а затем сохраните длину списка, находящегося в `blockHits`, в переменной `blockHitsLength` ❷.

Block hits length — длина списка ударов по блокам

Выбор случайного элемента

Наверное, вы уже заметили, что я часто использую в коде случайные значения. Это делает работу программы при каждом запуске непредсказуемой.

Иногда бывает нужно получить случайный элемент из списка — например, выбрать случайный блок из списка блоков. Для этого хорошо подходит функция `choice()` из модуля `random`. Эта функция принимает список в качестве аргумента и возвращает случайный элемент.

Choice — выбрать

Colors — цвета

В коде листинга 9.5 список `colors` содержит названия разных цветов. Программа выбирает одно из них с помощью `choice()` и выводит его на экран:

```
import random
colors = ["красный", "зеленый", "синий", "желтый", "оранжевый",
          "пурпурный"]
print(random.choice(colors))
```

Листинг 9.5. Программа, которая выбирает случайный цвет из списка и выводит его на экран

После запуска этой программы на экране должно появиться случайное название цвета.

МИССИЯ 51. СЛУЧАЙНЫЙ БЛОК

Выбор случайного идентификатора блока из всего диапазона значений осложняется тем, что некоторым значениям не соответствует какой-либо тип блока. Один из способов решить эту проблему — создать список с подходящими идентификаторами и выбирать из него случайное значение с помощью `choice()`.

В этой миссии ваша задача — составить список идентификаторов блоков, выбрать из него случайное значение и создать соответствующий блок в позиции игрока. В качестве отправной точки используйте код листинга 9.5.

Первым делом создайте список идентификаторов. Затем получите случайный элемент, вызвав функцию `random.choice()`. Далее используйте функцию `setBlock()`, чтобы создать в игровом мире блок выбранного типа.

Random block — случайный блок

Сохраните программу под именем *randomBlock.py* в папке *lists*.

Включите в список столько идентификаторов, сколько захотите. Мой список состоит из пяти типов блоков, включая «арбуз», «алмазный блок» и «золотой блок». На рис. 9.6 показан результат работы этой программы.

Рис. 9.6. Программа выбрала случайный блок, и он оказался золотым

Копирование списков

В большинстве языков программирования при копировании списков возникают сложности. Дело в том, что переменная, в которой хранится список, не содержит его элементов. Вместо этого в ней находится ссылка на адрес в памяти компьютера, где хранятся другие ссылки, указывающие на значения элементов. Хотя компьютер и обслуживает эту структуру данных автоматически, вам стоит иметь о ней представление, чтобы лучше разбираться в программировании! Получить адрес списка в памяти можно с помощью функции `id()`:

ID — идентификатор

```
>>> cake = ["Яйца",
            "Масло",
            "Сахар",
            "Молоко",
            "Мука"]
>>> print(id(cake))
```

На моем компьютере эта программа напечатала число `3067456428` — адрес, где хранится список `cake`. Запустив программу у себя, вы, скорее всего, увидите другое значение, поскольку список будет располагаться в другом участке памяти вашего компьютера.

Не обязательно понимать в деталях, что при этом происходит, но следует знать, какие последствия имеет копирование списка в другую переменную. Если вы присвоите список новой переменной, в нее будет скопирован адрес в памяти компьютера, а вовсе не элементы списка, как этого

можно было бы ожидать. Следовательно, изменив содержимое списка в переменной-копии, вы измените список-оригинал, и наоборот.

Chocolate cake — шоколадный торт

В следующем коде сначала создается список `cake`, а затем значение `cake` присваивается переменной `chocolateCake`. После этого мы добавляем в список `chocolateCake` элемент `"Шоколад"`:

```
>>> cake = ["Яйца",
            "Масло",
            "Сахар",
            "Молоко",
            "Мука"]

>>> # Сохраняем список в другой переменной
>>> chocolateCake = cake
>>> chocolateCake.append("Шоколад")
```

Увы, элемент `"Шоколад"` появится и в оригинальном списке `cake`, хотя такой команды мы не давали. В этом можно убедиться, выведя на экран содержимое списков:

```
>>> print(cake)
['Яйца', 'Масло', 'Сахар', 'Молоко', 'Мука', 'Шоколад']
>>> print(chocolateCake)
['Яйца', 'Масло', 'Сахар', 'Молоко', 'Мука', 'Шоколад']
```

Проблема в том, что переменные содержат всего лишь адрес списка, а не его содержимое.

Простой способ избежать этой проблемы — использовать *срез списка*. Представьте, что вы разрезаете батон на куски. Срез списка в Python — похожая операция: делая срез, вы получаете часть списка, сохраняя при этом основной список. Так можно извлекать определенные элементы, но сейчас мы воспользуемся срезом, чтобы скопировать наш список вместе с его содержимым в переменную `chocolateCake`:

```
>>> chocolateCake = cake[:]
```

Теперь список в переменной `chocolateCake` содержит все элементы списка `cake`, но полностью независим от него, поскольку располагается в другом участке памяти компьютера.

Итак, с помощью среза мы добились того, что ингредиенты шоколадного торта можно исправить, не меняя первоначальный набор:

```
>>> cake = ["Яйца",
            "Масло",
```

```
            "Сахар",
            "Молоко",
            "Мука"]

>>> # Сохраняем список в переменной
❶ >>> chocolateCake = cake[:]
>>> chocolateCake.append("Шоколад")
```

Как видите, элементы списка `cake` копируются в `chocolateCake` с помощью значка `[:]` ❶.

Вот что теперь содержится в этих двух списках:

```
>>> print(cake)
['Яйца', 'Масло', 'Сахар', 'Молоко', 'Мука']
>>> print(chocolateCake)
['Яйца', 'Масло', 'Сахар', 'Молоко', 'Мука', 'Шоколад']
```

Списки полностью независимы, поэтому значение `"Шоколад"` появилось только в `chocolateCake`.

Проверка элементов и конструкция if

Чтобы узнать, есть ли в списке некое значение, можно воспользоваться оператором `in`, введя его после искомого значения, а следом указав имя списка. Если значение в списке есть, выражение вернет `True`, если нет — `False`.

In — в

Вот как можно проверить, есть ли в списке `cake` значение `"Яйца"`:

```
>>> cake = ["Яйца", "Масло", "Сахар", "Молоко", "Мука"]
>>> print("Яйца" in cake)
```

Этот код выведет `True`, поскольку `Яйца` — один из элементов списка `cake`.

Оператор `in` можно поместить в условие конструкции `if`. Следующий код делает работу программы более совершенной за счет использования конструкции `if`, которая вместо булева значения выводит развернутый ответ. Программа проверяет, есть ли в списке `cake` значение `Ветчина`, и, если это так, выводит на экран фразу: `"Отвратительный торт"`. В противном случае программа выведет: `"Отлично. Ветчине в торте не место"`:

```
>>> cake = ["Яйца", "Масло", "Сахар", "Молоко", "Мука"]
>>> if "Ветчина" in cake:
>>>     print("Отвратительный торт.")
```

```
>>> else:
>>>     print("Отлично. Ветчине в торте не место.")
```

Можно также совместить операторы `not` и `in`, чтобы добиться обратного эффекта: если значение в списке есть, условие будет возвращать не `True`, а `False`, и наоборот. Вот какой вид примет в этом случае код из последнего примера (обратите внимание, что тела конструкций `if` и `else` поменялись местами):

```
>>> cake = ["Яйца", "Масло", "Сахар", "Молоко", "Мука"]
>>> if "Ветчина" not in cake:
>>>     print("Отлично. Ветчине в торте не место.")
>>> else:
>>>     print("Отвратительный торт.")
```

В своих программах вы можете использовать оба варианта — просто выберите способ, который больше всего подходит в конкретной ситуации.

МИССИЯ 52. МЕЧ НОЧНОГО ВИДЕНИЯ

Случалось ли так, что, отправляясь исследовать пещеры в мире Minecraft, вы забывали прихватить с собой достаточно факелов? Со мной это происходит постоянно: факелы кончаются, а до выхода из пещеры слишком далеко. В итоге мой игрок неуклюже шатается во мраке, не понимая, есть ли рядом что-нибудь полезное. Однако знание языка Python поможет вам искать сокровища в темноте при свете меча!

Давайте напишем программу, которая с помощью функции `pollBlockHits()` проверяет, являются ли блоки, по которым ударяет игрок, алмазной рудой. Это пригодится при исследовании неосвещенных пещер или для игры «Найди алмазную руду в темноте».

Night vision sword — меч ночного видения

Введите код листинга 9.6 в новый файл и сохраните его под именем *nightVisionSword.py* в папке *lists*.

nightVisionSword.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

blocks = []

while True:
    hits = mc.events.pollBlockHits()
```

```
    if len(hits) > 0:
        hit = hits[0]
❶       hitX, hitY, hitZ = hit.pos.x, hit.pos.y, hit.pos.z
        block = mc.getBlock(hitX, hitY, hitZ)
        blocks.append(block)

❷   # Добавьте сюда конструкцию if

    time.sleep(0.2)
```

Листинг 9.6. Эта программа поможет найти алмазную руду в темноте

Программа проверяет каждый блок, по которому игрок наносит удар, и помещает его координаты в кортеж. Эти координаты можно узнать, использовав точечную нотацию (я упоминал о ней в миссии 6 на с. 78). В данном случае список с информацией об ударах хранится в переменной `hit`, а координаты блоков можно получить, введя `hit.pos.x`, `hit.pos.y` и `hit.pos.z` ❶.

Hit — удар

Код дан практически полностью, осталось лишь определить, нашли вы алмазную руду или нет. Добавьте конструкцию `if` с проверкой ❷, находится ли идентификатор алмазной руды (56) в списке `blocks`, и если это так, выведите в чат сообщение: `"Вы нашли алмазную руду!"` Затем добавьте в тело `if` команду `break`, чтобы цикл завершился, если руда найдена.

На рис. 9.7 показан результат работы программы.

Рис. 9.7. Здесь темно, но я нашел алмазную руду. Ура!

Если вы не столь забывчивы, как я, и путешествуете по пещерам с запасом факелов, можете использовать этот код для игры. Сделайте подземную комнату без света и поместите в одну из стен блок алмазной руды. Запустите программу и посмотрите, за какое время у вас получится отыскать руду в темноте. Не забывайте, что ударять по блокам нужно мечом, нажимая на правую кнопку мыши! Только в этом случае функция `pollBlockHits()` вернет информацию о блоках, по которым вы ударяете.

БОНУСНОЕ ЗАДАНИЕ: ИГРА «НАЙДИ АЛМАЗ»

Программу *nightVisionSword.py* можно превратить в полноценную мини-игру! Попробуйте автоматически создать темную комнату, случайным образом поместить в одну из стен алмазный блок, а затем телепортировать туда игрока и засечь время, которое он потратит на поиски.

Словари

Словарь — это еще одна разновидность списка. Только для доступа к его элементам используются не индексы, а набор *ключей*, которые задает сам программист.

Race times — результаты забега

В следующем коде словарь `raceTimes` содержит имена спортсменов, которые участвовали в забеге на время, и результаты каждого из них:

```
raceTimes = {'Екатерина': 26,
             'Оксана': 30,
             'Георгий': 19}
```

Каждому ключу в словаре соответствует определенное значение. В этом коде ключом является имя участника. Так, ключу `Катя` соответствует значение `26`.

Как и списки, словари являются изменяемыми.

Создание словаря

Чтобы создать словарь, укажите в фигурных скобках его содержимое в виде пар «ключ — значение». Если вам нужно сохранить в словаре информацию о человеке, создайте ключи наподобие `имя` или `любимоеЖивотное` и свяжите их с определенными значениями, вот так:

```
person = {'имя': 'Вася',
          'возраст': 42,
          'любимоеЖивотное': 'Змея',
          'любимоеМесто': 'Картонная коробка'}
```

Как видите, после ключа ставится двоеточие, а далее указывается его значение. Элементы словаря разделяются запятыми. В данном примере все ключи — строки, а значения — строки и целые числа. Так, ключу `возраст` соответствует значение `42`.

Словари делают код понятнее, поскольку сразу видно, что хранится в каждом элементе. При взгляде на последний пример нам ясно: ключу `имя` соответствует имя человека, а не число или какая-нибудь другая информация.

Ключами могут быть целые и вещественные числа. Это бывает полезно, когда значения нужно сопоставить числам, а порядок их следования не важен. В следующем примере показан словарь с расписанием поездов, где время отправления (вещественное число) является ключом, а пункт прибытия — значением:

```
trainTimes = {13.00: 'Самара',
              14.30: 'Новокуйбышевск',
              15.15: 'Чапаевск',
              15.45: 'Октябрьск',
              15.55: 'Сызрань'}
```

Поскольку в словарях хранятся пары «ключ — значение», вещественные числа хорошо подходят для таких случаев. Используй я вместо словаря с расписанием обычный список, я не смог бы сопоставить время и пункт прибытия, ведь индексы списка — это числа 0, 1, 2, 3, 4 и так далее, которые не подходят для обозначения времени.

Доступ к элементам словаря

Как и при работе со списками, чтобы получить доступ к элементу словаря, нужно использовать квадратные скобки, только внутрь поместить не индекс, а ключ. Обычно ключ — это строка или целое число. Создавая словарь с ключами-строками, не забудьте, что их нужно вводить в кавычках.

Так, чтобы получить из словаря `person` значение, соответствующее ключу `имя`, используйте следующий синтаксис:

Person — персональные данные

```
person = {'имя': 'Вася',
          'возраст': 42,
          'любимое животное': 'Змея',
```

```
                'любимое место': 'Картонная коробка'}

agentName = person['имя']
```

Agent name — имя агента

Переменная `agentName` примет значение `Вася`, поскольку оно соответствует указанному ключу `имя`. А если вы захотите узнать возраст агента, используйте ключ `возраст`:

```
agentAge = person['возраст']
```

Agent age — возраст агента

После этого переменная `agentAge` примет значение `42`.

В примере с расписанием поездов название пункта прибытия можно получить по его ключу — времени, которое является вещественным числом:

```
trainTimes = {13.00: 'Самара',
              14.30: 'Новокуйбышевск',
              15.15: 'Чапаевск',
              15.45: 'Октябрьск',
              15.55: 'Сызрань'}

myTrain = trainTimes[15.15]
```

My train — мой поезд

В результате переменная `myTrain` примет значение `Чапаевск`, которому и соответствует указанный ключ `15.15`.

МИССИЯ 53. ПУТЕВОДИТЕЛЬ

В словаре можно хранить данные любых типов, даже списки и кортежи. Например, кортеж с координатами *x*, *y* и *z*.

```
places = {'Гостиная': (76, 1, -61), 'Спальня': (61, 9, -61)}
```

Places — места

Словарь `places` содержит два элемента, в которых ключами являются названия комнат (гостиная и спальня), а значениями — кортежи с координатами этих мест. Если мне понадобятся координаты гостиной, я могу получить их, введя следующий код:

```
location = places['Гостиная']
x, y, z = location[0], location[1], location[2]
```

В этой миссии вам необходимо создать программу, которая будет хранить в словаре координаты различных мест мира Minecraft, чтобы можно было телепортировать игрока, указав название одного из них. Добавьте в словарь столько мест, сколько считаете нужным. Для телепортации вам понадобится получить из словаря кортеж с координатами, а затем присвоить переменным `x`, `y` и `z` значения, хранящиеся в кортеже.

Введите код листинга 9.7 в окне программы IDLE и сохраните файл в папке *lists* под именем *sightseeingGuide.py*. Комментарии подскажут, куда нужно добавить строчки кода.

Sightseeing guide — путеводитель

sightseeingGuide.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

# Добавьте названия и координаты мест в словарь
places = {}

choice = ""
while choice != "выход":
❶     choice = input("Введите название места (или 'выход', чтобы
                     завершить программу): ")
❷     if choice in places:
          # Получите из словаря кортеж по ключу (значению choice)
          location =
          # Сохраните значения кортежа в переменных x, y и z
          x, y, z =
          mc.player.setTilePos(x, y, z)
```

Листинг 9.7. Удобный код для телепортации игрока в разные места мира Minecraft

Я добавил в код команду, которая запрашивает название места для телепортации игрока. Введенная строка попадает в переменную `choice` ❶. Конструкция `if` нужна для проверки, есть ли в словаре значение, связанное с ключом, который хранится в `choice` ❷. В последней строке функция `setTilePos()` телепортирует игрока в место с координатами *x*, *y* и *z*, полученными из словаря.

Choice — выбор

Запустив программу, введите название места, в которое вы хотите отправить игрока. На рис. 9.8 показано, как моя версия программы телепортирует его в разные точки игрового мира.

Изменение и добавление элементов словаря

Изменить значение элемента в словаре совсем не сложно. Для этого введите имя словаря, затем в квадратных скобках укажите ключ и присвойте ему значение, как обычной переменной (при помощи знака «равно»). Этим же способом можно добавить в словарь новое значение.

Рис. 9.8. Я телепортировал игрока в гостиную (вверху) и спальню (внизу)

Давайте поменяем значение элемента `возраст` в словаре `person` с `42` на `43`:

```
person['возраст'] = 43
```

А теперь добавим в словарь новый элемент с ключом `местонахождение` и значением `Дискавери`:

```
person['местонахождение'] = 'Дискавери'
```

«Дискавери» — космический корабль из сериала «Звездный путь». *Прим. перев.*

После запуска программы в словаре появится новый элемент со значением `Дискавери`.

Удаление элементов словаря

Порой возникает необходимость удалить из словаря элемент. Для этого служит та же команда `del`, которой мы удаляли элементы из списка. Чтобы удалить из словаря `person` элемент с ключом `любимоеЖивотное`, введите:

```
del person['любимоеЖивотное']
```

Как видите, запись выглядит точно так же, как и при работе с обычным списком.

МИССИЯ 54. УДАРЫ ПО БЛОКАМ И ТАБЛИЦА РЕЗУЛЬТАТОВ

В ходе миссии 50 вы написали программу, которая показывает, сколько раз игрок ударил по блокам мечом за 60 секунд. Эта игра хороша, но она станет еще лучше, если будет запоминать результат каждого игрока.

Чтобы добавить в игру таблицу результатов, вам понадобится словарь, в котором ключам с именами игроков будут соответствовать значения с набранными ими очками. Эти очки потом можно будет отобразить.

Sword hits score — очки за удары по блокам

Откройте *swordHits.py* и сохраните ее под именем *swordHitsScore.py* в папке *lists*. Отредактируйте код так, чтобы он соответствовал листингу 9.8. Там показаны изменения, которые я внес в программу, чтобы при запуске код запрашивал имя игрока и выводил таблицу результатов (помимо этого я добавил команды, которых не хватало для нормальной работы *swordHits.py*). Строки, оставшиеся без изменений, показаны серым цветом. Не забудьте добавить отступы перед командами в теле цикла!

swordHitsScore.py

```
# Подключаемся к Minecraft
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

name = ""
scoreboard = {}

while True:
    # Запрашиваем имя пользователя
```

```
name = input("Как вас зовут? ")
# Завершаем цикл, если введено "выход"
if name == "выход":
    break
mc.postToChat("Поехали!")

# Ждем 60 секунд
time.sleep(60)

# Получаем список ударов
blockHits = mc.events.pollBlockHits()

# Выводим длину списка ударов в чат
blockHitsLength = len(blockHits)
mc.postToChat("Ваш счет " + str(blockHitsLength))

❶ # Добавляем имя игрока в таблицу результатов

# Выводим на экран таблицу результатов
print(scoreboard)
```

Листинг 9.8. Основа кода для игры, ведущей подсчет количества ударов мечом, с таблицей результатов

Scoreboard — таблица

Необходимо дописать программу так, чтобы она сохраняла имя и счет каждого игрока. Для этого добавьте в словарь элементы с именами и результатами ❶. Словарь называется `scoreboard`, а имена игроков хранятся в переменной `name`.

На рис. 9.9 показана моя таблица результатов.

БОНУСНОЕ ЗАДАНИЕ: ЛУЧШИЙ РЕЗУЛЬТАТ

Если кто-нибудь сыграет в игру *swordHitsScore.py* несколько раз (постоянно вводя одно и то же имя), программа занесет в таблицу только последний результат. Постарайтесь разобраться, как с помощью конструкции `if` проверить, есть ли в таблице имя этого игрока, и заставить программу сохранять новый результат, только если он больше предыдущего. Начните со строчки кода, которая проверяет, есть ли в словаре `scoreboard` элемент с ключом `name`:

```
if name in scoreboard:
```

Наверное, вы заметили, что такую таблицу результатов не очень-то удобно читать. В ходе миссии 59 (с. 254) вы узнаете, как это исправить.

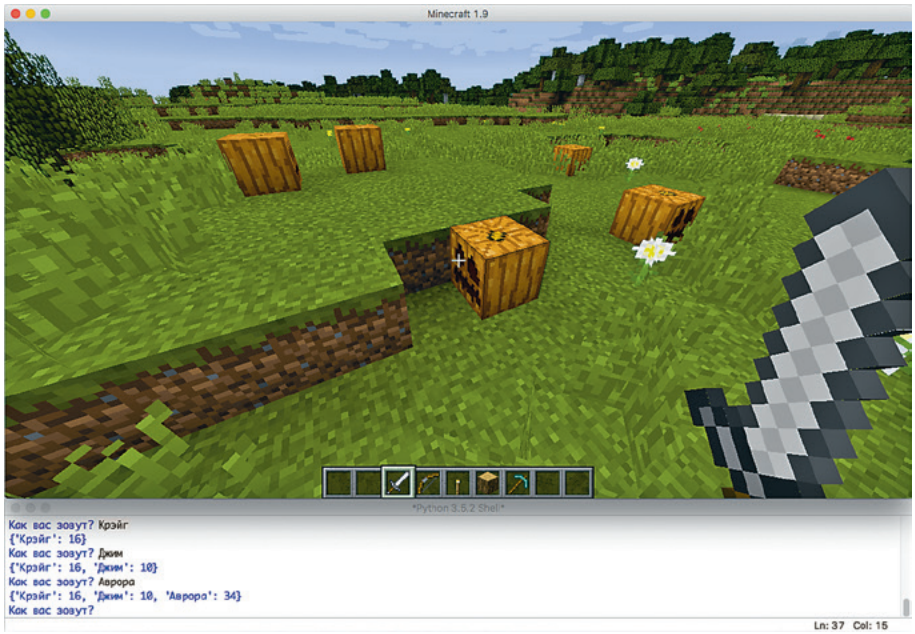

Рис. 9.9. Мы сыграли с друзьями, и со счетом 34 удара победила Аврора

Что вы узнали

Превосходно! В этой главе вы научились работать со списками, кортежами и словарями — типами данных, которые позволяют хранить в одной переменной сразу несколько значений. Благодаря спискам данные в ваших программах стало легко приводить в порядок.

Выполнив миссии из главы 9, вы написали несколько увлекательных программ. С помощью списков создали столбик-секундомер из стекла и лазурита. Благодаря кортежам научились одной строкой присваивать значения переменным `x`, `y` и `z`. А словари позволили вам сохранять координаты важных мест в мире Minecraft и телепортировать туда игрока, указав название места.

В главе 10 вы углубите свои знания о списках, освоив цикл `for`, и создадите несколько по-настоящему крутых программ, одна из которых позволит вам делать копии своих построек.

10

ЦИКЛЫ FOR И ВОЛШЕБСТВО В MINECRAFT

Пришла пора познакомиться с циклами `for`. Эти циклы необычайно полезны, поскольку с их помощью можно перебирать элементы списков — тех самых, о которых шла речь в главе 9.

В миссиях этой главы вы будете применять циклы `for` для создания лестниц, колонн, пирамид и обветшалых стен. С помощью вложенных циклов `for` и списков вы сможете мгновенно создавать пиксельные рисунки и уникальные сооружения. В общем, эти циклы — необычайно мощный инструмент для строительства в мире Minecraft!

Простой цикл for

Цикл `for` не использует условие, как цикл `while` или конструкция `if`. Вместо этого он повторяет фрагмент кода для каждого элемента списка, пока список не закончится.

Список, с которым работает цикл `for`, может состоять из любого количества элементов какого угодно типа. Этот цикл перебирает все элементы по очереди, то есть по индексу.

Чтобы вывести на экран все ингредиенты вермишелевого супа, можно использовать такой код:

```
noodleSoup = ["вода", "соевый соус", "зеленый лук", "перец", "вермишель",
              "говядина", "овощи"]
```

```
for ingredient in noodleSoup:
    print(ingredient)
```

Сначала мы указываем ключевое слово `for`, затем — переменную `ingredient`. Она соответствует элементу списка, с которым цикл работает в настоящий момент. Значение этой переменной будет меняться на каждой итерации цикла, пока программа не переберет весь список. На первой итерации `ingredient` примет значение элемента с индексом 0 (`вода`), на второй итерации — с индексом 1 (`соевый соус`), на третьей — с индексом 2 (`зеленый лук`) и т. д.

Ingredient — ингредиент

Оператор `in` и имя списка в конце строки показывают, к какому списку следует обращаться. В данном случае это список `noodleSoup`.

Тело цикла выполняется по одному разу для каждого элемента, пока список не закончится. Вот что выведет программа:

```
вода
соевый соус
зеленый лук
перец
вермишель
говядина
овощи
```

Перед нами весь список! А теперь самое время придумать в игровом мире что-нибудь увлекательное с циклами `for`.

МИССИЯ 55. ВОЛШЕБНАЯ ПАЛОЧКА

Каждый инструмент в мире Minecraft имеет свое назначение. Лопатой можно копать землю, киркой — разбивать каменные блоки, топором — рубить деревья, а мечом — крушить врагов. Обычно назначение инструмента от вас не зависит — остается лишь принять тот факт, что меч годится только для сражений. Однако с помощью Python многое можно исправить. В этой миссии мы создадим программу, которая превратит меч в волшебную палочку.

В главе 9 мы использовали функцию `pollBlockHits()` — она возвращает список координат тех блоков, по которым игрок ударил мечом. С помощью цикла `for` можно перебрать все наборы координат в этом списке и, ни много ни мало, превратить эти блоки в арбузы. Результат такого превращения показан на рис. 10.1.

Magic wand — волшебная палочка

For loops — циклы for

В листинге 10.1 показана основа этого кода. Сохраните текст программы в файле с именем *magicWand.py* в новой папке *forLoops*.

Рис. 10.1. Абракадабра! Все блоки, по которым я ударил, превратились в арбузы!

magicWand.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

time.sleep(60)

❶ hits = mc.events.pollBlockHits()
  block = 103

❷ for
❸     x, y, z = hit.pos.x, hit.pos.y, hit.pos.z
❹     # Установите блок «арбуз» в заданных координатах
```

Листинг 10.1. Основа кода для превращения меча в волшебную палочку

Чтобы получить список ударов по блокам, мы вызываем функцию `pollBlockHits()` и сохраняем результат в переменной `hits` ❶.

В программе есть команда для получения координат любого блока, по которому ударил игрок, и сохранения их в переменных `x`, `y` и `z` ❸. Чтобы присвоить значения трем переменным одной строкой, используется кортеж (см. раздел «Кортежи» на с. 220).

Сейчас эта строка кода не работает, поскольку в программе нет переменной `hit`. Создайте цикл `for` ❷ и объявите его переменную `hit`. Цикл должен перебирать элементы списка `hits`. Для этого введите:

```
for hit in hits:
```

Не забудьте сделать отступ в четыре пробела в начале строки, где переменным `x`, `y` и `z` присваиваются значения ❸, чтобы команда попала в тело цикла. Добавьте в цикл еще одну строку с функцией `setBlock()` для установки блока «арбуз» в месте с координатами *x*, *y* и *z* ❹.

После запуска программы у игрока будет 60 секунд, чтобы пробежаться, ударяя по блокам мечом. Спустя минуту все блоки, на которые вы успели нажать правой кнопкой мыши, превратятся в арбузы.

БОНУСНОЕ ЗАДАНИЕ: ВЫ — ВОЛШЕБНИК

Измените программу *magicWand.py* так, чтобы она телепортировала игрока следующим образом: первый удар мечом определяет координаты места, а второй — телепортирует туда игрока.

Функция range()

Функция `range()` создает список целочисленных значений. Она отлично подходит для того, чтобы быстро сгенерировать последовательность чисел для цикла `for`. Вызовем функцию `range()`, передав ей в качестве аргументов значения `0` и `5`:

Range — диапазон

```
aRange = range(0, 5)
```

Согласитесь, создавать список чисел таким образом удобнее, чем если бы мы указывали значения всех элементов по отдельности, вот так:

```
aRange = [0, 1, 2, 3, 4]
```

Обратите внимание, что значение второго элемента функции — число 5, но последним элементом списка стало число 4. Дело в том, что функция `range()` создает значения, которые меньше второго аргумента, но не равны ему.

Чтобы создать с помощью функции `range()` цикл и вывести на экран числа от 1 до 15, можно написать такой код:

```
for item in range(1, 16):
    print(item)
```

А вывести удвоенные значения всех элементов можно так:

```
for item in range(1, 16):
    print(item * 2)
```

В сущности, то же самое можно проделать при помощи цикла `while`, который мы рассмотрели в главе 7. Следующий код выводит числа от 1 до 15:

```
count = 1
while count < 16:
    print(count)
    count += 1
```

Обратите внимание: фрагмент кода с циклом `for` компактнее и понятнее. При создании больших и сложных программ часто удобнее использовать циклы `for`, чем `while` с переменной-счетчиком `count`.

МИССИЯ 56. ВОЛШЕБНАЯ ЛЕСТНИЦА

Одно из главных преимуществ программирования на Python в игре Minecraft состоит в том, что вы можете быстро создавать объекты, написав лишь несколько строк кода. Вместо того чтобы часами возводить стены, вы запускаете программу — и дело сделано. Кроме того, код можно использовать сколько угодно раз, экономя время и усилия.

Постройка лестницы — дело нелегкое. К счастью, вы можете запросто ее возвести с помощью короткой программы. В ходе этой миссии вам предстоит создать в игровом мире лестницу, воспользовавшись циклом `for`.

Stairs — лестница

Сохраните код листинга 10.2 в файл с именем *stairs.py* в папке *forLoops*.

stairs.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getTilePos()
x, y, z = pos.x, pos.y, pos.z

stairBlock = 53

step = 0
while step < 10:
    mc.setBlock(x + step, y + step, z, stairBlock)
    step += 1
```

Листинг 10.2. Код, который создает лестницу при помощи цикла while

Хотя цикл `while` неплохо делает свое дело, цикл `for` больше подойдет для этой задачи. В отличие от `while`, он не потребует отдельной переменной вроде `count` или `step` — вы можете задать количество итераций цикла с помощью функции `range()`.

Усовершенствуйте этот код, заменив цикл `while` на цикл `for`.

Результат работы программы показан на рис. 10.2.

Рис. 10.2. Куда приведет волшебная лестница?

БОНУСНОЕ ЗАДАНИЕ: ЛЕСТНИЦА В НЕБО

Сейчас программа *stairs.py* строит лестницу лишь в одну сторону. Постарайтесь разобраться, как построить ее в других направлениях. Подсказка: вам понадобится прибавлять или вычитать значения из переменных `x` и `z`, а также использовать необязательный аргумент функции `setBlock()` — состояние.

Эксперименты с функцией range()

Вы уже немного знакомы с функцией `range()` и знаете, что произойдет, если передать ей два аргумента. А если аргумент будет только один? Введите этот код в окне консоли IDLE:

```
>>> aRange = range(5)
>>> list(aRange)
[0, 1, 2, 3, 4]
```

Если передать функции `range()` один аргумент, она начнет отсчитывать значения с 0, увеличивая каждое следующее на 1, пока не дойдет до числа, которое на 1 меньше переданного аргумента. Иными словами, будет действовать так, будто вы передали ей первым аргументом 0, а вторым 5.

List — список

A range — ряд

Функция `list()` в этом примере показывает значения, созданные функцией `range()` (иначе вы их не увидите!). Вызов `list(aRange)` возвращает список из пяти чисел: `[0, 1, 2, 3, 4]`. Теперь мы знаем быстрый способ создавать последовательности чисел от 0 и выше!

```
>>> aRange = range(2, 5)
>>> list(aRange)
[2, 3, 4]
```

В этом примере создается список с числами `[2, 3, 4]`.

Можно передать функции `range()` три аргумента — в этом случае третий аргумент определяет *шаг* между значениями. Обычно каждое следующее значение списка, создаваемого `range()`, на единицу больше предыдущего. Задавая шаг, вы меняете разницу между значениями. Если указать шаг 2, каждое следующее значение будет на 2 больше предыдущего. Шаг 3 сделает следующее значение больше предыдущего на 3 и т. д.:

```
>>> aRange = range(3, 10, 2)
>>> list(aRange)
[3, 5, 7, 9]
```

В этом примере каждое следующее значение на 2 больше предыдущего (5 — это 3 + 2, 7 — это 5 + 2, 9 — это 7 + 2).

Функции `range()` можно передать даже отрицательное значение шага:

```
>>> newRange = range(100, 0, -2)
>>> list(newRange)
[100, 98, 96, 94, 92, 90, 88, 86, 84, 82, 80, 78, 76, 74, 72, 70, 68,
66, 64, 62, 60, 58, 56, 54, 52, 50, 48, 46, 44, 42, 40, 38, 36, 34, 32,
30, 28, 26, 24, 22, 20, 18, 16, 14, 12, 10, 8, 6, 4, 2]
```

Поскольку третий элемент функции `-2`, каждое следующее значение списка меньше предыдущего на 2.

Другие функции для работы со списками

Раз уж мы работаем со списками, давайте изучим еще несколько списковых функций.

Reversed — в обратном порядке

Функция `reversed()` принимает в качестве аргумента один список и возвращает другой, в котором все значения из первого списка стоят в обратном порядке. Давайте перевернем содержимое списка с уже знакомой нам последовательностью `[3, 5, 7, 9]`:

```
>>> backwardsList = reversed(aRange)
>>> list(backwardsList)
[9, 7, 5, 3]
```

Порядок элементов поменялся на обратный, как мы и хотели. Вы также можете перевернуть содержимое списка при объявлении цикла `for`, не сохраняя новый список в отдельной переменной.

Следующий код генерирует последовательность чисел от 1 до 100 с помощью функции `range()`. Затем содержимое этого списка переворачивается, и значения элементов выводятся на экран в цикле `for` — получается обратный отсчет от 100 до 1:

```
countDown = range(1, 101)
countDown = reversed(countDown)
for item in countDown:
    print(item)
```

Запустите этот код и посмотрите, что получится!

```
100
99
98
97
96
--пропущено--
3
2
1
```

Содержимое списка можно перевернуть прямо при создании цикла `for`:

```
for item in reversed(range(0, 101)):
    print(item)
```

Этот фрагмент кода делает то же самое, однако он более компактный и помогает сберечь время для постройки сооружений!

МИССИЯ 57. КОЛОННЫ

Было бы здорово построить в игре дворец, верно? Поскольку дворцам положено быть величественными, нам понадобится ряд высоких, стройных колонн. Разумеется, возводить их вручную мы не будем, ведь гораздо проще использовать для этого цикл `for`.

Мы создадим функцию для постройки колонны, а затем будем вызывать ее столько раз, сколько нам нужно. Программа из листинга 10.3 содержит код этой функции. Сохраните его под именем *pillars.py* в папке *forLoops*.

Pillars — колонны

pillars.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

def setPillar(x, y, z, height):
    """Создает колонну. Аргументы задают ее позицию и высоту"""
    stairBlock = 156
    block = 155

    # Вершина колонны
    mc.setBlocks(x - 1, y + height, z - 1, x + 1, y + height, z + 1,
                 block, 1)
    mc.setBlock(x - 1, y + height - 1, z, stairBlock, 12)
    mc.setBlock(x + 1, y + height - 1, z, stairBlock, 13)
    mc.setBlock(x, y + height - 1, z + 1, stairBlock, 15)
    mc.setBlock(x, y + height - 1, z - 1, stairBlock, 14)

    # Основание колонны
    mc.setBlocks(x - 1, y, z - 1, x + 1, y, z + 1, block, 1)
    mc.setBlock(x - 1, y + 1, z, stairBlock, 0)
    mc.setBlock(x + 1, y + 1, z, stairBlock, 1)
    mc.setBlock(x, y + 1, z + 1, stairBlock, 3)
    mc.setBlock(x, y + 1, z - 1, stairBlock, 2)

    # Ствол колонны
    mc.setBlocks(x, y, z, x, y + height, z, block, 2)

pos = mc.player.getTilePos()
x, y, z = pos.x + 2, pos.y, pos.z
❶ # Добавьте сюда цикл for
❷ # Вызовите здесь функцию setPillar()
```

Листинг 10.3. Код для постройки колонны

Set pillar — установить колонну

Функция `setPillar()`, с помощью которой мы создаем колонну, принимает четыре аргумента: координаты *x*, *y* и *z*, а также высоту колонны.

Чтобы доделать программу, добавьте в нее цикл `for` ❶ и вызовите в его теле функцию `setPillar()` ❷. Нам нужно создать ряд из 20 колонн,

отстоящих друг от друга на 5 блоков. Используйте функцию `range()` с тремя аргументами, чтобы задать количество колонн и расстояние между ними. Складывая при вызове функции `setPillar()` значение переменной `x` или `z` со значением переменной цикла `for`, вы можете расположить все колонны в ряд на одинаковом расстоянии друг от друга.

На рис. 10.3 показаны некоторые из созданных программой колонн.

Рис. 10.3. Великолепный ряд колонн

МИССИЯ 58. ПИРАМИДА

Продолжая тему постройки восхитительных сооружений с помощью цикла `for`, давайте возведем пирамиду. Пирамида состоит из нескольких квадратных плит, из которых нижняя — самая большая. Каждая следующая плита нашей пирамиды будет на два блока у́же предыдущей. Если сторона нижней плиты состоит из семи блоков, сторона следующей плиты будет состоять из пяти блоков, плиты выше — из трех блоков и, наконец, верхней — из одного блока.

Программа из листинга 10.4 строит пирамиду. Введите код в новый файл и сохраните под именем *pyramid.py* в папке *forLoops*.

Pyramid — пирамида

pyramid.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

block = 24  # песчаник
❶ height = 10
❷ levels = range(height)
```

```
pos = mc.player.getTilePos()
❸ x, y, z = pos.x + height, pos.y, pos.z

❹ for level in levels:
❺     mc.setBlocks(x - level, y, z - level, x + level, y, z + level,
                   block)
      y += 1
```

Листинг 10.4. Код, создающий перевернутую пирамиду

Код листинга 10.4 создает пирамиду, однако в нем есть небольшая ошибка, которую вам нужно исправить. Высота пирамиды задается переменной `height` ❶. Вы можете изменить значение этой переменной на любое другое. Переменная `levels` принимает полученный из функции `range()` список, в котором каждый элемент — номер уровня пирамиды ❷. Присваивая значения переменным `x`, `y` и `z`, мы берем координаты игрока и прибавляем к его *x*-координате значение переменной `height` ❸. Если этого не сделать, после запуска программы игрок окажется в центре пирамиды.

Levels — уровни

Цикл `for` делает по одной итерации для каждого элемента списка `levels` ❹. Строка кода, создающая уровни пирамиды, использует переменную `level` при вычислении длины стороны каждой следующей плиты ❺, которая всегда будет вдвое больше значения `level`.

Помните, я говорил про ошибку? Запустите программу и посмотрите на результат. Пирамида перевернута вверх ногами!

Ваша задача — исправить этот недочет и установить пирамиду как подобает. Используйте функцию `reversed()` с переменной `levels`, чтобы значения списка уменьшались от начала к концу. Впрочем, вместо этого вы можете схитрить, присвоив функции `range()` отрицательное значение аргумента и вызвав ее.

На рис. 10.4 показан результат работы исправленной команды — пирамида в правильном положении.

Перебор элементов словаря в цикле

Также цикл `for` можно использовать для перебирания элементов словаря. Синтаксис при этом останется таким же, как для списка, однако цикл будет перебирать только ключи.

Следующий код выводит на экран значение переменной цикла на каждой итерации. В результате на экране появятся все ключи:

```
inventory = {'драгоценные камни': 5, 'зелья': 2, 'коробки': 1}

for key in inventory:
    print(key)
```

Рис. 10.4. Величественная пирамида

Вот что выведет программа:

```
драгоценные камни
зелья
коробки
```

Выводить на экран значения словаря тоже можно — используйте запись вида `словарь[ключ]`. Чтобы программа печатала и ключи, и их значения. Для этого изменим команду следующим образом:

```
inventory = {'драгоценные камни': 5, 'зелья': 2, 'коробки': 1}

for key in inventory:
    print(key + " " + str(inventory[key]))
```

Теперь программа выведет:

```
драгоценные камни 5
зелья 2
коробки 1
```

Результат работы этой программы еще более информативный, чем предыдущей. Используя для вывода словаря цикл, вы контролируете отображение его элементов на экране.

МИССИЯ 59. ТАБЛИЦА РЕЗУЛЬТАТОВ

Вспомните игру *swordHitsScore.py* из миссии 54 (с. 239). Она запоминала количество ударов, которые участники нанесли по блокам в течение минуты, и сохраняла результат каждого игрока в словаре рядом с его именем. Хотя сама игра работала как положено, таблица результатов в ней отображалась не самым удобным образом — программа просто выводила словарь в чат.

В ходе этой миссии вам предстоит усовершенствовать программу *swordHitsScore.py*, чтобы она выводила содержимое словаря `scoreboard` в удобном для чтения виде. Для этого понадобится цикл `for`.

Score board — таблица результатов

Откройте файл *swordHitsScore.py* (ищите его в папке *lists*) и сохраните под новым именем *scoreBoard.py* в папке *forLoops*. Найдите в коде следующую строку:

```
print(scoreboard)
```

Замените ее на цикл `for`, который будет выводить на экран имена игроков вместе с набранными ими очками. Эта информация хранится в словаре `scoreboard`: имя каждого игрока является ключом, а набранные очки — значением.

На рис. 10.5 показана усовершенствованная таблица результатов.

Рис. 10.5. Теперь результат работы программы выглядит гораздо понятнее

Конструкция else и цикл for

Вместе с циклом `for` можно использовать конструкцию `else`. При этом тело `else` будет выполнено после того, как цикл `for` переберет все значения списка. А если цикл не дойдет до конца списка, тело `else` выполняться не будет.

Вот код, выводящий на экран с помощью конструкции `else` ингредиенты для сэндвича:

```
sandwich = ["Кусок хлеба", "Масло", "Тунец", "Лист салата", "Майонез",
            "Кусок хлеба"]

for ingredient in sandwich:
    print(ingredient)
else:
    print("Вот и весь сэндвич.")
```

Если запустить эту программу, она выведет:

```
Кусок хлеба
Масло
Тунец
Лист салата
Майонез
Кусок хлеба
Вот и весь сэндвич.
```

Вам может показаться, что этот код ничем не отличается от такого:

```
for ingredient in sandwich:
    print(ingredient)
print("Вот и весь сэндвич.")
```

Что ж, вы правы: оба варианта делают одно и то же. Так зачем же мы использовали `else` вместе с `for`? Дело в том, что, если завершить цикл командой `break`, конструкция `else` поведет себя иначе. Давайте с этим разберемся.

Выход из цикла for-else с помощью break

Выход из цикла `for` с помощью `break` — единственный способ не допустить выполнения тела конструкции `else`.

В следующем примере внутри конструкции `if` используется команда `break`. Цикл прекратит работу, если текущим элементом окажется строка `"Майонез"`:

```
sandwich = ["Кусок хлеба", "Масло", "Тунец", "Лист салата", "Майонез",
            "Кусок хлеба"]

for ingredient in sandwich:
    if ingredient == "Майонез":
        print("Не люблю сэндвичи с майонезом.")
        break
    else:
        print(ingredient)
else:
    print("Вот и весь сэндвич.")
```

Как считаете, что выведет этот код? Подумайте об этом прежде, чем запустите программу и всё увидите.

МИССИЯ 60. АЛМАЗОИСКАТЕЛЬ

Порой, когда я играю в Minecraft с друзьями, они мне запрещают мне создавать блоки алмазной руды при помощи Python-программ. Но что делать, если алмазы необходимы мне для доспехов, инструментов и постройки алмазных дворцов? Можно копать наугад, но ведь найти алмазную руду непросто.

Чтобы сберечь время, я написал код, который определяет, есть ли в земле под игроком алмазная руда. Программа получает текущие координаты игрока и с помощью цикла `for` по одному перебирает блоки под ногами, проверяя, не алмазные ли они. Если найдется блок «алмазная руда», в чат будет отправлена информация, на какой глубине он находится; если же руды не обнаружится, программа сообщит, что под игроком таких блоков нет.

Создайте в IDLE новый файл и сохраните его под именем *diamondSurvey.py* в папке *forLoops*.

Diamond survey — поиск алмазов

Уменьшайте значение переменной `y` на единицу при каждом повторе цикла `for`. В общей сложности цикл должен сделать 50 итераций, перебрав до 50 блоков в глубину. Конструкция `if` на каждой итерации будет проверять, является ли блок в этой позиции алмазной рудой (идентификатор 56), и, если это так, отправлять в чат сообщение с информацией, на какой глубине относительно игрока находится блок. Завершите цикл командой `break`. Если программа не обнаружит алмазной руды, то перейдет к конструкции `else` после цикла `for` и выведет сообщение, что алмазной руды под игроком не найдено.

На рис. 10.6 показан результат работы программы.

Рис. 10.6. На расстоянии четырех блоков под моим игроком залегает алмазная руда. Пора копать!

БОНУСНОЕ ЗАДАНИЕ: ЗОЛОТАЯ ЛИХОРАДКА

Измените программу *diamondSurvey.py*, чтобы она искала и другие виды блоков, например железную и золотую руду.

Вложенные циклы for и многомерные списки

Списки можно вкладывать внутрь других списков — такие структуры данных называются *многомерными списками*. В этом разделе мы изучим двухмерные (2D) и трехмерные (3D) списки, которые помогут нам создавать постройки в мире Minecraft.

Думаем в двух измерениях

Вы уже знаете, как создавать списки, точнее — *одномерные* списки. Одномерными они называются потому, что каждая позиция такого списка содержит лишь один элемент.

Посмотрите на список `oneDimensionalRainbowList`. В коде он показан немного непривычным образом, чтобы подчеркнуть: каждая его позиция содержит единственный элемент. В остальном же он ничем не отличается от прочих списков, с которыми вы имели дело:

One dimensional rainbow list — одномерный радужный список

```
oneDimensionalRainbowList = [0,
                             1,
                             2,
                             3,
                             4,
                             5]
```

Список содержит шесть элементов — это числа от 0 до 5. Каждый элемент хранит единственное значение, следовательно, этот список одномерный.

В коде листинга 10.5 данная последовательность чисел используется для создания столбика из разноцветных блоков шерсти. Эту программу, *rainbowStack1.py*, можно найти на странице книги по ссылке *http://mif.to/minecraft/*. Скачайте ее или введите код листинга в новый файл IDLE самостоятельно.

Rainbow stack 1 — радужный столбик № 1

rainbowStack1.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

❶ oneDimensionalRainbowList = [0, 1, 2, 3, 4, 5]

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

❷ for color in oneDimensionalRainbowList:
    mc.setBlock(x, y, z, 35, color)
    y += 1
```

Листинг 10.5. Код, который создает радужный столбик из блоков шерсти

Эта программа использует список с цветами шерстяных блоков ❶, перебирая его в цикле `for`, чтобы создать радужный столбик ❷.

Запустив эту программу, вы увидите одинокий столбик из шерстяных блоков, как на рис. 10.7. Обратите внимание, что его высота — шесть блоков, а ширина — один блок. Мы уже не раз использовали переменные с координатами *x*, *y* и *z*. Каждую из этих переменных можно назвать *измерением*. Так вот, данная программа создает столбик из шестяных блоков в измерении *y*. Заменив `y` в последней строке кода на `x`, вы можете создать «столбик» в измерении `x` — он показан на рис. 10.8.

Поскольку наш список одномерный, вы можете менять значение только одной переменной, которая соответствует одному из измерений: ширине, высоте или длине. Иначе говоря, вы можете присвоить новое

значение либо переменной `y`, либо переменной `x`, либо переменной `z`, но не всем этим переменным сразу.

Рис. 10.7. Радужный столбик из блоков, который создала программа rainbowStack1.py

Рис. 10.8. Если заменить `y` на `x` в последней строке кода, программа создаст «горизонтальный столбик»

Ну а теперь начнем думать в двух измерениях! Одномерный список позволяет хранить в каждой позиции по одному элементу, однако в двухмерном списке элементов может быть несколько. Давайте поместим в каждую позицию первоначального списка еще по одному списку, вот так:

```
❶ twoDimensionalRainbowList = [[0, 0, 0],
❷                              [1, 1, 1],
❸                              [2, 2, 2],
                               [3, 3, 3],
                               [4, 4, 4],
❹                              [5, 5, 5]]
```

Внимательно присмотревшись, вы обнаружите в первой строке этого кода открывающую квадратную скобку, после которой идет список с нулями, а затем запятая ❶. Это не что иное, как список внутри списка! Мы можем назвать основной список *внешним* и сказать, что в нем находятся *вложенные списки*.

По индексу 1 располагается список с тремя единицами ❷. По индексу 2 — с тремя двойками ❸. Подобные списки мы видим и в следующих строках. В последней строке внешнего списка находится список с тремя пятерками, а за ним стоит закрывающая квадратная скобка ❹. Итак, в этом коде показан список с шестью элементами, каждый из которых также является списком. Вот он какой, двухмерный список!

Вы лучше поймете, что такое двухмерные списки, начав использовать их в игре Minecraft. Давайте откроем наш код *rainbowStack1.py* и сохраним его как *rainbowRows.py*. Для начала заменим в нем одномерный список двухмерным.

Rainbow rows — радужные ряды

rainbowRows.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

twoDimensionalRainbowList = [[0, 0, 0],
                             [1, 1, 1],
                             [2, 2, 2],
                             [3, 3, 3],
                             [4, 4, 4],
                             [5, 5, 5]]

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

❶ startingX = x

❷ for row in twoDimensionalRainbowList:
❸     for color in row:
❹         mc.setBlock(x, y, z, 35, color)
❺         x += 1
```

```
❻     y += 1
❼     x = startingX
```

Прежде чем я объясню, что происходит в коде, посмотрите на рис. 10.9, где показан результат его работы. Это стена, высота которой равна шести блокам (в измерении *y*), а ширина — трем блокам (в измерении *x*).

Рис. 10.9. Радужная стена, которую я создал с помощью двухмерного списка

Поскольку мы работаем в двух измерениях, для обработки значений из списка `twoDimensionalRainbowList` нам понадобятся два цикла `for`. Первый перебирает элементы внешнего списка ❷, а второй, вложенный, — элементы каждого из вложенных списков ❶.

Two dimensional rainbow list — двухмерный радужный список

Происходит это так: на первой итерации внешнего цикла из списка `twoDimensionalRainbowList` программа берет элемент с индексом 0 и сохраняет его значение в переменной `row` ❷. Теперь в `row` находится список `[0, 0, 0]`, ведь это и есть первый элемент внешнего списка.

Row — ряд

Затем второй, вложенный цикл перебирает элементы из списка `row`, поочередно сохраняя каждый из них в переменной `color` ❸. В данном случае все эти элементы равны 0. Во вложенном цикле программа создает блоки, используя переменную `color` для задания цвета каждому из них ❹. Разместив три блока, вложенный цикл завершает свою работу, и программа возвращается во внешний цикл. Далее она переходит к элементу внешнего цикла с индексом 1 и сохраняет его в переменной `row` — теперь там находится список `[1, 1, 1]`. Опять запускается вложенный цикл, который устанавливает очередные три блока, и все эти действия повторяются снова и снова, пока не будет обработан каждый элемент списка `twoDimensionalRainbowList`.

Color — цвет

Работая в двух измерениях, вы можете одновременно менять значения двух координатных переменных. В этом примере мы инкрементируем (приращиваем) `y` в предпоследней строке внешнего цикла ❻, чтобы каждый ряд из трех блоков располагался выше предыдущего. Кроме того, во вложенном цикле мы увеличиваем значение `x` ❺, благодаря чему блоки выстраиваются в ряд. После этого на каждой итерации внешнего цикла ❼ мы сбрасываем значение `x` до первоначального (оно хранится в переменной `startingX` ❶). Это нужно для того, чтобы первый блок каждого следующего ряда располагался точно над первым блоком предыдущего, второй — над вторым и т. д. В итоге стена получается ровной.

Starting x — первоначальное значение *x*

Доступ к элементам 2D-списка

Чтобы получить или изменить элемент одномерного списка, индекс заключают в квадратные скобки. В следующем коде вы видите список `scores`, предназначенный для хранения очков, набранных игроком. Второй строкой элементу с индексом 2 присваивается новое значение — `7` вместо прежнего `6`:

```
scores = [1, 5, 6, 1]
scores[2] = 7
```

Получение или изменение элементов двухмерного списка ненамного отличается от этой схемы. Вам по-прежнему будут нужны квадратные скобки и индексы, только теперь в удвоенном виде, поскольку обращаетесь вы сразу к двум спискам. Давайте попробуем!

Вот список из предыдущего примера:

```
twoDimensionalRainbowList = [[0, 0, 0],
                             [1, 1, 1},
                             [2, 2, 2],
                             [3, 3, 3],
                             [4, 4, 4],
                             [5, 5, 5]]
```

Чтобы присвоить второму элементу (индекс 1) из первого вложенного списка (индекс 0) значение 7, нужно написать такой код:

```
twoDimensionalRainbowList[0][1] = 7
```

Индекс 0 внутри первой пары квадратных скобок говорит о том, что программе нужно обратиться к первому элементу из списка

`twoDimensionalRainbowList` — ему соответствует вложенный список `[0, 0, 0]`. Индекс 1 внутри второй пары квадратных скобок отсылает нас ко второму элементу этого вложенного списка. С помощью знака «равно» программа присваивает найденному элементу значение `7`.

Я добавил этот код в программу *rainbowRows.py* (с. 260) и перезапустил ее. Результат работы показан на рис. 10.10. Обратите внимание, что цвет второго блока в первом ряду поменялся — это произошло потому, что я присвоил второму элементу первого вложенного списка значение `7`.

Рис. 10.10. Цвет центрального блока в нижнем ряду поменялся благодаря замене одного из элементов вложенного списка

Получить хранящееся в двухмерном списке значение можно при помощи все тех же двух пар квадратных скобок. Если вам нужно вывести цвет блока, стоящего в первой позиции (индекс 0) верхнего ряда (индекс 5), используйте такой код:

```
print(twoDimensionalRainbowList[5][0])
```

Этот код напечатает `5`.

МИССИЯ 61. ПИКСЕЛЬ-АРТ

Пиксели — это маленькие разноцветные квадратики, из которых состоят компьютерные изображения. Комбинируя множество пикселей, ваш компьютер может отображать текст, фотографии, видеоролики

и все остальное, что вы видите на экране монитора. Все изображения на вашем компьютере состоят из отдельных пикселей.

Пиксельное рисование пользуется в игре Minecraft немалой популярностью. Игроки составляют из разноцветных блоков оригинальные картинки — *пиксель-арты*. Часто среди них встречаются изображения персонажей из 2D-игр. Вы тоже можете создавать пиксель-арты вручную, но есть и другой способ — генерировать картинки с помощью Python-программы.

В ходе этой миссии вам предстоит создать пиксель-арт, использовав в коде двухмерный список и вложенные циклы. Основа кода показана в листинге 10.6. Введите его в новый файл IDLE и сохраните как *pixelArt.py* в папке *forLoops*.

Pixel art — пиксель-арт

pixelArt.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getTilePos()
x, y, z = pos.x, pos.y, pos.z

❶ blocks = [[35, 35, 35, 35, 35, 35, 35, 35],
            [35, 35, 35, 35, 35, 35, 35, 35],
            [35, 35, 35, 35, 35, 35, 35, 35],
            [35, 35, 35, 35, 35, 35, 35, 35]]

❷ for row in reversed(blocks):
      for block in row:
          mc.setBlock(x, y, z, block)
          x += 1
      y += 1
      x = pos.x
```

Листинг 10.6. Рисуем смайлик с помощью двухмерного списка

Создаем двухмерный список `blocks`, где хранятся идентификаторы блоков ❶. Затем при помощи двух циклов программа устанавливает эти блоки в игровом мире ❷. Чтобы блоки, указанные в первом вложенном списке, оказались наверху, а в последнем — внизу, во внешнем цикле используется функция `reversed()` ❷ — иначе изображение окажется перевернутым.

Сейчас в двухмерном списке `blocks` находятся только блоки белой шерсти — картинку из них не составишь. Ваша задача — изменить содержимое списка так, чтобы получился смайлик, как на рис. 10.11.

Вам нужно будет подставить вместо идентификаторов блоков шерсти (35) идентификаторы блоков лазурита (22). Первую строку можно заменить на такую:

```
blocks = [[35, 35, 22, 22, 22, 22, 35, 35],
```

Помимо этого, в двухмерный список `blocks` нужно добавить несколько новых списков, чтобы высота изображения была такой же, как на рис. 10.11.

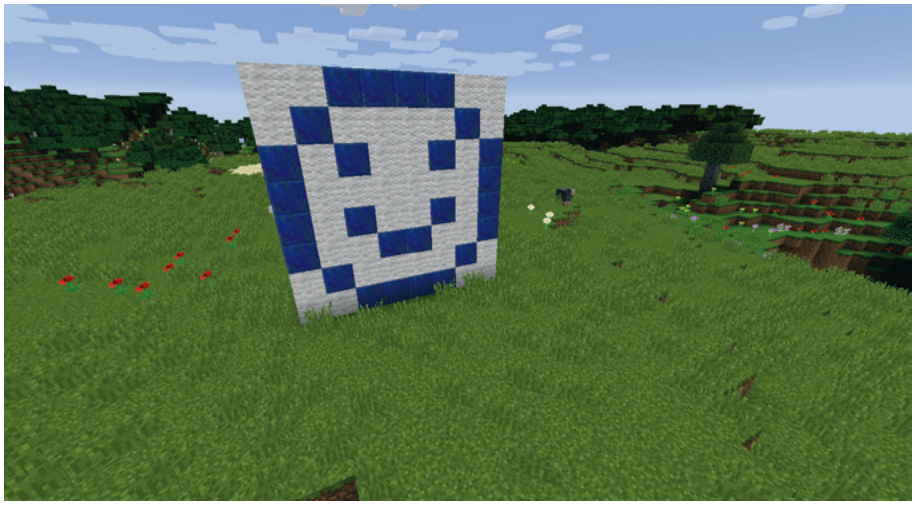

Рис. 10.11. Смайлик из блоков белой шерсти и блоков лазурита

БОНУСНОЕ ЗАДАНИЕ: РИСУЙТЕ САМИ

Измените содержимое двухмерного списка так, чтобы вместо смайлика программа создавала придуманный вами рисунок. Размеры списков тоже можно менять. Сначала сделайте набросок на листе в клеточку, затем преобразуйте его в набор значений для двухмерного списка и, наконец, запустите программу, чтобы она создала ваш пиксель-арт в мире Minecraft!

Генерация 2D-списка с помощью циклов

Программы, в которых используются случайные значения, особенно интересны, ведь каждый их запуск приводит к новому неожиданному результату. В своей жизни я написал немало программ, заполнявших двухмерные списки случайными значениями, чтобы получить картинки. Каждое случайное число может представлять собой цвет или, в случае с Minecraft, идентификатор блока.

Вот основа программы, которая генерирует случайные числа и заполняет ими двухмерный список:

```
import random
❶ randomNumbers = []

for outer in range(10):
❷     randomNumbers.append([])
      for inner in range(10):
❸         number = random.randint(1, 4)
          randomNumbers[outer].append(number)
print(randomNumbers)
```

Random numbers — случайные числа

В самом начале программы создается пустой список `randomNumbers` ❶. На каждой итерации внешнего цикла `for` в этот список добавляется новый пустой список ❷. Затем во внутреннем цикле программа генерирует случайное число от 1 до 4, добавляя его во вложенный список ❸. Так внутренний цикл делает 10 итераций, заполняя вложенный список десятью значениями.

Если добавить переносы строк, чтобы читать программу было удобнее, результат ее работы будет выглядеть примерно так (обратите внимание — в 10 вложенных списках находится по 10 значений):

```
[[3, 1, 4, 1, 4, 1, 2, 3, 2, 2],
 [1, 3, 4, 2, 4, 3, 4, 1, 3, 2],
 [4, 2, 4, 1, 4, 3, 2, 3, 4, 4],
 [1, 4, 3, 4, 3, 4, 3, 3, 4, 4],
 [3, 1, 4, 2, 3, 3, 3, 1, 4, 2],
 [4, 1, 4, 2, 3, 2, 4, 3, 3, 1],
 [2, 4, 2, 1, 2, 1, 4, 2, 4, 3],
 [3, 1, 3, 4, 1, 4, 2, 2, 4, 1],
 [4, 3, 1, 2, 4, 2, 2, 3, 1, 2],
 [3, 1, 3, 3, 1, 3, 1, 4, 1, 2]]
```

Добавив в код своих 2D-творений случайные числа, вы сможете получить очень интересные эффекты, воспроизвести которые вручную будет нелегко!

МИССИЯ 62. ОБВЕТШАЛАЯ СТЕНА

Сооружая в игре стены, я использую разные типы блоков. Заменив часть булыжников замшелыми булыжниками, я могу превратить обыкновенную стену в обветшалую. Но как бы ни интересно было возводить стены вручную, я никогда не мог добиться того, чтобы блоки, которые я добавлял в игровой мир, выглядели естественно. Наверное, вы уже

догадываетесь, что средством, которое поможет стенам обрести естественный вид, будет программа на языке Python.

Чтобы сгенерировать обветшалую стену, нужно выполнить две основные задачи.

1. Создать двухмерный список и заполнить его идентификаторами блоков «булыжник», «замшелый булыжник» и «булыжные ступеньки».

2. Установить блоки из этого списка в игровом мире.

В листинге 10.7 уже есть команды для выбора случайного блока, создания списка и получения координат игрока. Введите этот код в новый файл IDLE и сохраните его под именем *brokenWall.py* в папке *forLoops*.

Broken wall — неровная стена

brokenWall.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import random

❶ def brokenBlock():
    brokenBlocks = [48, 67, 4, 4, 4, 4]
    block = random.choice(brokenBlocks)
    return block

pos = mc.player.getTilePos()
x, y, z = pos.x, pos.y, pos.z
brokenWall = []
height, width = 5, 10

# Заполните двухмерный список блоками

# Установите блоки
```

Листинг 10.7. Основа кода для создания неровной стены

Broken block — неровный блок

Функция `brokenBlock()` возвращает случайный идентификатор блока для постройки стены ❶. Переменные `width` и `height` определяют ширину и высоту стены.

Чтобы доделать программу, вам нужно сгенерировать двухмерный список с идентификаторами блоков и затем использовать эти данные для сооружения стены.

Создайте внешний цикл `for` и вложенный в него цикл `for`. Во вложенном цикле сгенерируйте случайные идентификаторы блоков, вызывая для этого функцию `brokenBlock()`. Сохраняйте полученные

значения в списках, добавляя затем эти списки в список `brokenWall`. После этого используйте еще одну пару циклов (внешний и вложенный), чтобы установить блоки в игре.

Дописав код, отправляйтесь в то место мира Minecraft, где вы хотите возвести стену, и запускайте программу. Ее можно использовать для постройки замка или создания таинственных руин посреди леса — пробуйте разные места и смотрите, что вам больше по душе!

На рис. 10.12 показан пример сгенерированной обветшалой стены.

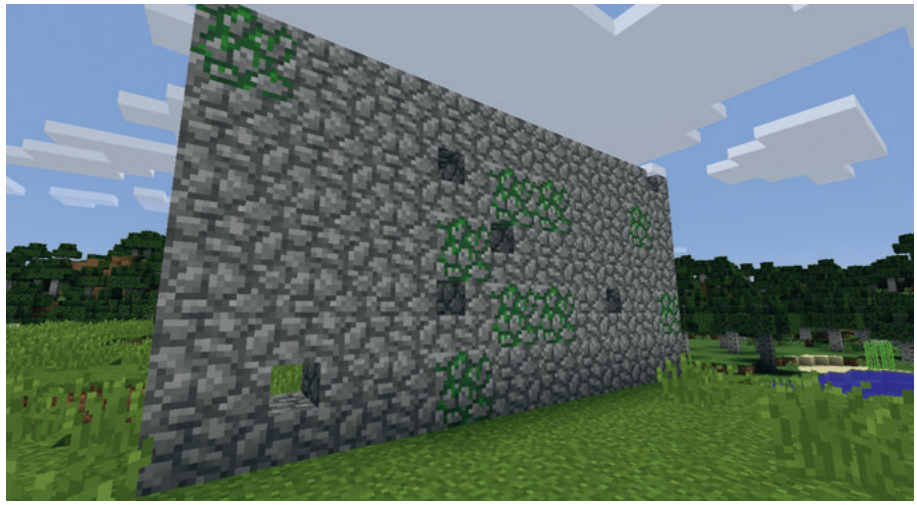

Рис. 10.12. Стена со случайно выбранными блоками. Пожалуй, за ней могут водиться привидения!

БОНУСНОЕ ЗАДАНИЕ: СОЗДАЙТЕ РАЗНОЦВЕТНУЮ СТЕНУ

Переделайте код *brokenWall.py*, изменив идентификаторы блоков в списке `brokenBlocks` внутри функции `brokenBlock()`, чтобы программа создавала стены не только из камня. Попробуйте использовать для постройки разноцветные шерстяные блоки!

Думаем в трех измерениях

Как известно, в мире Minecraft используется три измерения, и вы на протяжении этой книги тоже их использовали — каждое значение переменных `x`, `y` и `z` представляет собой измерение.

Вы знаете, что, поместив внутрь списка набор других списков, можно получить двухмерный список и использовать его для создания классного

пиксель-арта или постройки обветшалой стены. Если же внутрь двухмерного списка поместить еще один набор списков, получится трехмерный список, который позволит поднять ваши навыки строительства на новый уровень!

Трехмерные списки очень ценны для Minecraft, поскольку в них можно хранить копии трехмерных сооружений: домов, скульптур и многих других.

Трехмерный список из листинга 10.8 содержит четыре вложенных списка. И по каждому индексу в этих вложенных списках хранится еще один список! В сущности, каждый элемент внешнего списка является двухмерным списком. Для наглядности я разделил эти 2D-списки пустыми комментариями.

```
cube = [[[57, 57, 57, 57],
         [57, 0, 0, 57],
         [57, 0, 0, 57],
         [57, 57, 57, 57]],
         #
        [[57, 0, 0, 57],
         [0, 0, 0, 0],
         [0, 0, 0, 0],
         [57, 0, 0, 57]],
         #
        [[57, 0, 0, 57],
         [0, 0, 0, 0],
         [0, 0, 0, 0],
         [57, 0, 0, 57]],
         #
        [[57, 57, 57, 57],
         [57, 0, 0, 57],
         [57, 0, 0, 57],
         [57, 57, 57, 57]]]
```

Листинг 10.8. Трехмерный список с вложенными двухмерными списками

Этот код можно использовать для постройки классной кубической конструкции! Далее мы разберем программу, которая как раз это и делает.

Отображение 3D-списков

Списки с тремя измерениями отлично подходят для хранения информации о трехмерных объектах — таких как ваши изумительные постройки. Хранить трехмерные объекты важно, но не менее важно правильно отображать их в игровом мире. Поскольку трехмерный список — это список внутри списка внутри списка, для доступа к хранящимся там данным нужно использовать цикл `for` внутри другого цикла `for`, который находится внутри еще одного цикла `for`. Иными словами, вам помогут три цикла `for`, вложенные друг в друга.

Cube — куб

Я скопировал в листинг 10.9 трехмерный список из листинга 10.8, добавил три вложенных цикла `for` и другие команды и назвал программу *cube.py*. Она использует три вложенных цикла `for` для постройки трехмерной конструкции.

cube.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
cube = [[[57, 57, 57, 57], [57, 0, 0, 57], [57, 0, 0, 57], [57, 57, 57, 57]],
        [[57, 0, 0, 57], [0, 0, 0, 0], [0, 0, 0, 0], [57, 0, 0, 57]],
        [[57, 0, 0, 57], [0, 0, 0, 0], [0, 0, 0, 0], [57, 0, 0, 57]],
        [[57, 57, 57, 57], [57, 0, 0, 57], [57, 0, 0, 57], [57, 57, 57, 57]]]

startingX = x
❶ startingY = y

❷ for depth in cube:
    for height in reversed(depth):
        for block in height:
            mc.setBlock(x, y, z, block)
            x += 1
        y += 1
        x = startingX
❸   z += 1
❹   y = startingY
```

Листинг 10.9. Код, который создает куб из алмазных блоков

Результат работы программы показан на рис. 10.13.

Программа *cube.py* очень похожа на *rainbowRows.py* (с. 260), которая строит двухмерную разноцветную стену. Главное отличие ее в том, что *cube.py* использует не два, а три цикла `for`. Это нужно для работы с трехмерным списком. Дополнительный цикл добавляет конструкции еще одно измерение — глубину ❷. Соответственно, теперь наша конструкция обладает шириной, высотой и глубиной.

На каждой итерации внешнего цикла ❷ обрабатывается двухмерный список, для чего служат команды `for height in reversed(depth)` и `for block in height`. Коды вложенных циклов аналогичны паре циклов программы *rainbowRows.py*, то есть эти два цикла создают нечто вроде стены.

Давайте посмотрим на результат работы каждой итерации внешнего цикла, чтобы понять, как шаг за шагом появляется наш куб. На первой итерации программа создает блоки, хранящиеся в списке `cube` по индексу 0. Это список такого вида:

```
[[57, 57, 57, 57],
 [57, 0, 0, 57],
 [57, 0, 0, 57],
 [57, 57, 57, 57]]
```

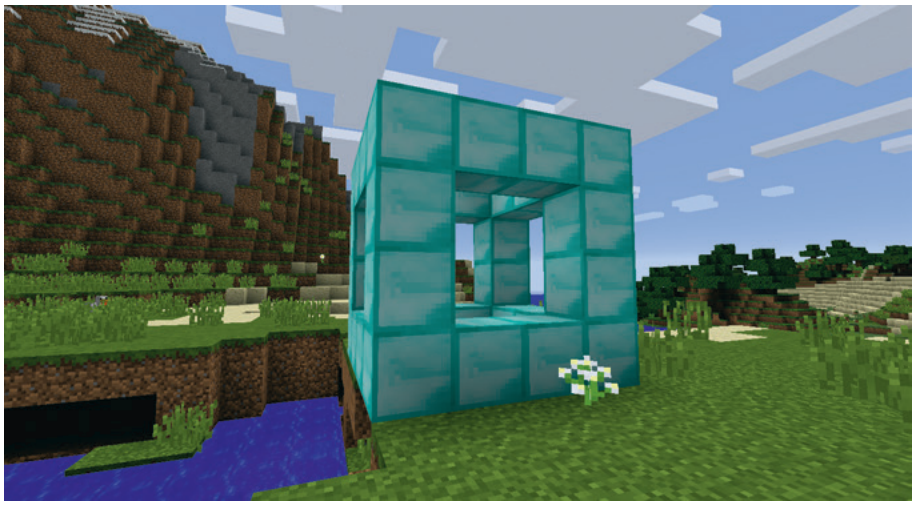

Рис. 10.13. Программа cube.py создала такой куб

На рис. 10.14 показан результат — передняя стена из алмазных блоков.

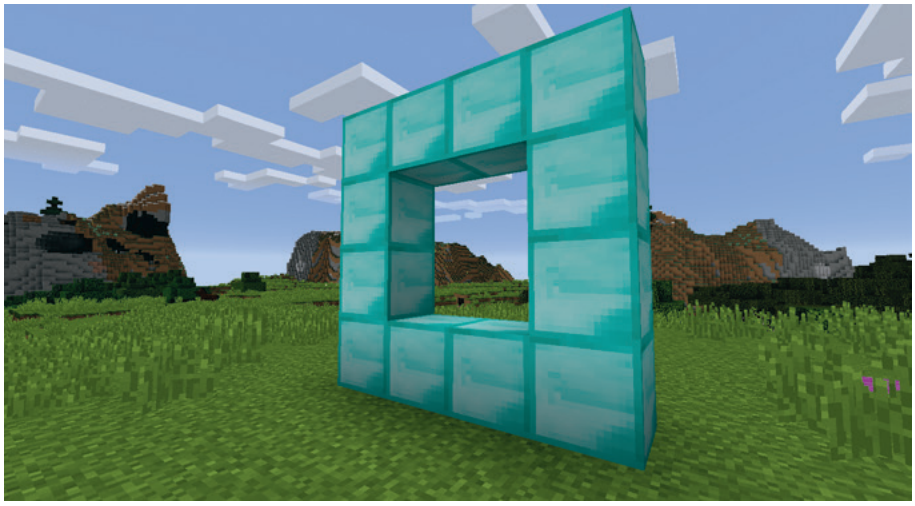

Рис. 10.14. Результат обработки двухмерного списка (индекс 0 в списке `cube`) на первой итерации

После создания блоков из очередного двухмерного списка значение переменной `z` ❸ увеличивается, чтобы сместить следующий «слой» стены на один блок дальше по оси *z*. Это придает конструкции глубину, иначе получилась бы просто стена. Кроме того, нам нужно сбрасывать значение переменной `y` ❹ до первоначального ❶, чтобы блоки одного уровня шли друг за другом. Если не сбрасывать значение `y`, *y*-координата каждого следующего набора блоков будет все больше и больше, и получится странного вида лесенка. На рис. 10.15 показано, как это выглядит.

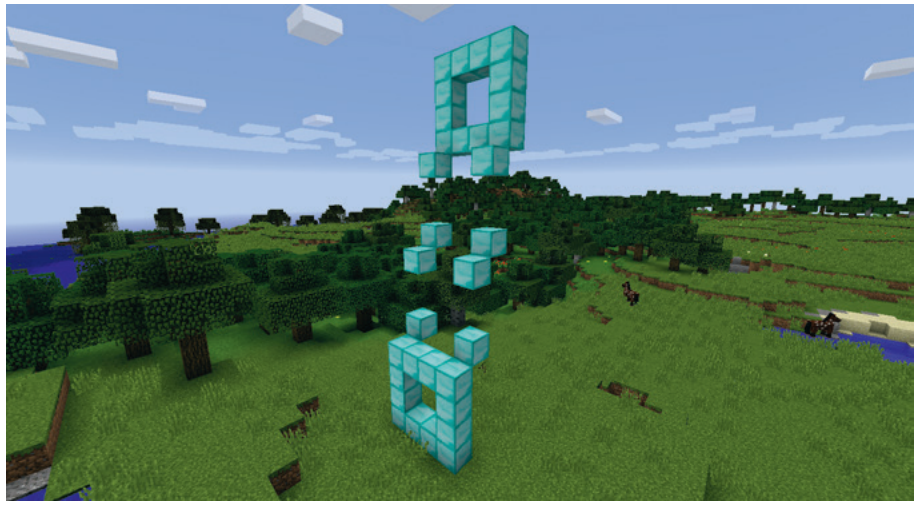

Рис. 10.15. Чтобы такого не произошло, необходимо сбрасывать значение `y`!

На второй итерации внешнего цикла программа создает блоки из списка, который хранится по индексу 1. Выглядит он так:

```
[[57, 0, 0, 57],
 [0, 0, 0, 0],
 [0, 0, 0, 0],
 [57, 0, 0, 57]]
```

В итоге появляется еще одна часть куба, как на рис. 10.16. Затем значение переменной `z` увеличивается на 1 ❸, а значение `y` снова сбрасывается до первоначального значения ❹.

На следующей итерации программа обрабатывает двухмерный список по индексу 2 в списке `cube`:

```
[[57, 0, 0, 57],
 [0, 0, 0, 0],
```

```
 [0, 0, 0, 0],
 [57, 0, 0, 57]]
```

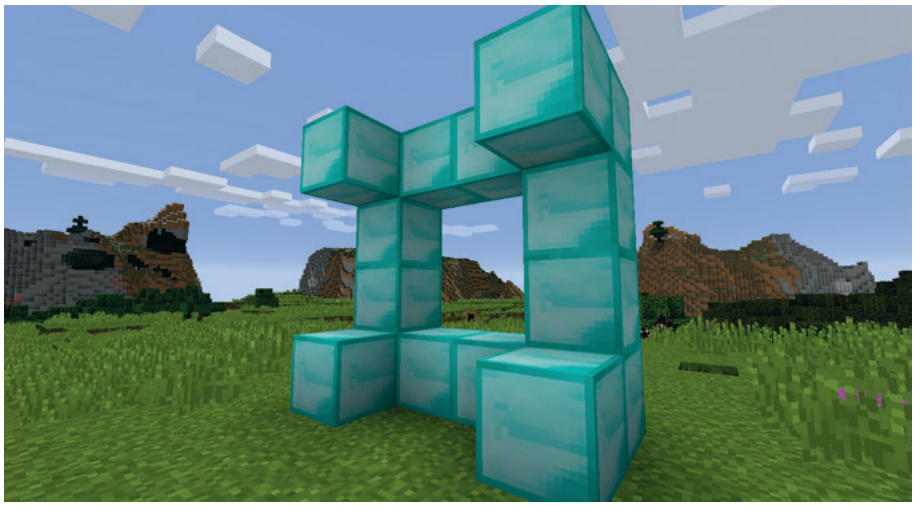

Рис. 10.16. Итог обработки второго двухмерного списка (индекс 1 в списке `cube`)

Результат показан на рис. 10.17. И снова значение `z` возрастает, а значение `y` сбрасывается.

Затем внешний цикл выполняет последнюю, четвертую итерацию, обрабатывая индекс 3 в списке `cube`:

```
[[57, 57, 57, 57],
 [57, 0, 0, 57],
 [57, 0, 0, 57],
 [57, 57, 57, 57]]
```

На следующей итерации программа обрабатывает двухмерный список по индексу 2 в списке `cube`:

```
[[57, 0, 0, 57],
 [0, 0, 0, 0],
 [0, 0, 0, 0],
 [57, 0, 0, 57]]
```

Результат показан на рис. 10.17. И снова значение `z` возрастает, а значение `y` сбрасывается.

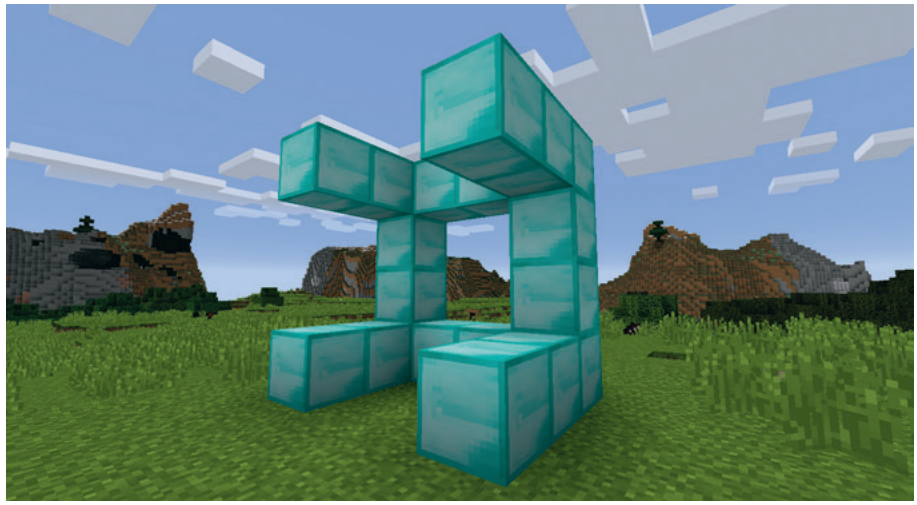

Рис. 10.17. Результат обработки третьего двухмерного списка (индекс 2 в списке `cube`)

На рис. 10.18 показана законченная кубическая конструкция.

Поэкспериментируйте с этой программой: замените идентификатор алмазного блока, попробуйте увеличить размер куба и т. д. — все что захотите!

В следующем разделе я покажу, как обращаться к значениям внутри трехмерного списка. Это поможет усовершенствовать программу.

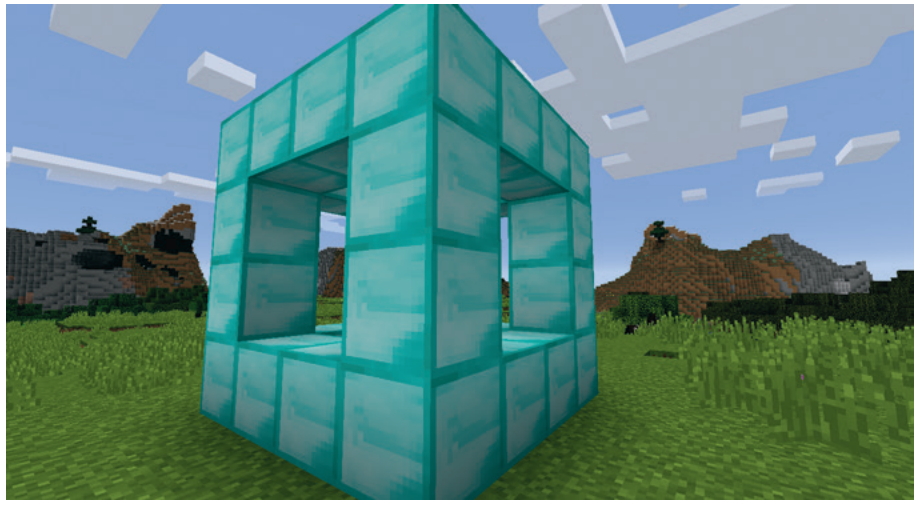

Рис. 10.18. Результат обработки последнего двухмерного списка (индекс 3 в списке `cube`)

Доступ к элементам 3D-списка

Значения внутри трехмерного списка можно менять таким же способом, как и значения внутри двухмерного и одномерного списков, — с помощью индексов, заключенных в квадратные скобки.

Посмотрим на наш трехмерный список для создания алмазного куба.

```
cube = [[[57, 57, 57, 57],
         [57, 0, 0, 57],
         [57, 0, 0, 57],
         [57, 57, 57, 57]],
         #
        [[57, 0, 0, 57],
         [0, 0, 0, 0],
         [0, 0, 0, 0],
         [57, 0, 0, 57]],
         #
        [[57, 0, 0, 57],
         [0, 0, 0, 0],
         [0, 0, 0, 0],
         [57, 0, 0, 57]],
         #
        [[57, 57, 57, 57],
         [57, 0, 0, 57],
         [57, 0, 0, 57],
         [57, 57, 57, 57]]]
```

Допустим, я хочу поменять левый нижний блок в передней части куба на золотой.

Сначала мне нужно получить доступ к элементу списка `cube`, который содержит идентификаторы передних блоков. Ему соответствует индекс 0, поэтому первая часть выражения будет такой:

```
cube[0]
```

Если вывести на экран результат работы этой команды, мы увидим вот что (я добавил переносы строк для лучшей читаемости):

```
[[57, 57, 57, 57],
 [57, 0, 0, 57],
 [57, 0, 0, 57],
 [57, 57, 57, 57]]
```

Этот двухмерный список соответствует передней части куба. Далее мне нужно получить доступ к нижнему ряду — он находится по индексу 3, поэтому я добавлю к выражению [3]:

```
cube[0][3]
```

Если вывести список, хранящийся в этой позиции, мы увидим:

```
[57, 57, 57, 57]
```

Теперь я хочу изменить идентификатор крайнего левого блока с индексом 0. Выходит, окончательное выражение для замены левого нижнего блока на золотой будет таким:

```
cube[0][3][0] = 41
```

Запустив программу *cube.py*, в которую добавлена эта строка, я получу алмазный куб с единственным золотым блоком, как на рис. 10.19.

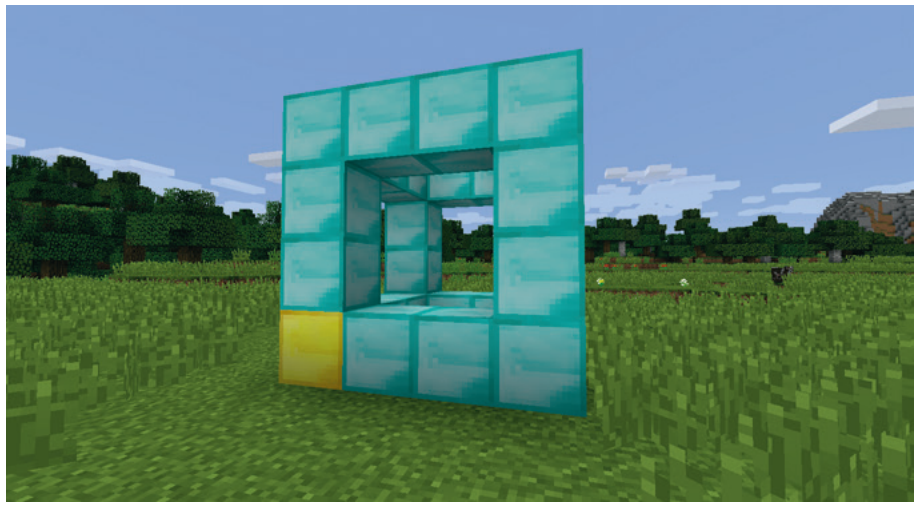

Рис. 10.19. Алмазно-воздушный куб с золотым углом

МИССИЯ 63. КОПИРОВАНИЕ КОНСТРУКЦИЙ

Несомненно, программы для создания конструкций в мире Minecraft экономят уйму времени, однако, если вы похожи на меня, вы все равно потратите немало усилий на то, чтобы украсить свои постройки картинами или предметами мебели. А что если вам захочется создать точную копию какого-нибудь здания? Делать это вручную долго и сложно — чуть зазеваешься, и очередной блок уже не там, где надо… Очевидное решение этой проблемы — написать программу, которая копирует объект игрового мира и создает его дубликат.

Готовая программа должна выполнять две задачи: во-первых, копировать область игрового мира, сохраняя ее в трехмерном списке, а во-вторых, воссоздавать скопированную конструкцию.

Чтобы вам было проще, я подготовил листинг 10.10 с основой кода. Введите его в новый файл и сохраните под именем *duplicateArea.py* в папке *forLoops*.

Duplicate area — дубликат местности

duplicateArea.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

❶ def sortPair(val1, val2):
    if val1 > val2:
        return val2, val1
    else:
        return val1, val2

❷ def copyStructure(x1, y1, z1, x2, y2, z2):
    # Получаем максимальные и минимальные значения x, y и z
    x1, x2 = sortPair(x1, x2)
    y1, y2 = sortPair(y1, y2)
    z1, z2 = sortPair(z1, z2)

    width = x2 - x1
    height = y2 - y1
    length = z2 - z1

    structure = []

    print("Пожалуйста, подождите…")

❸   # Копируем конструкцию

    return structure

❹ def buildStructure(x, y, z, structure):
    xStart = x
    yStart = y

❺    # Воссоздаем конструкцию

# Получаем координаты первого угла
❻ input("Пройдите к первому углу и нажмите Enter в этом окне")
pos = mc.player.getTilePos()
x1, y1, z1 = pos.x, pos.y, pos.z

# Получаем координаты второго (противоположного) угла
❼ input("Пройдите к противоположному углу и нажмите Enter в этом окне")
pos = mc.player.getTilePos()
```

```
    x2, y2, z2 = pos.x, pos.y, pos.z

    # Копируем конструкцию
❽ structure = copyStructure(x1, y1, z1, x2, y2, z2)

    # Задаем координаты копии
❾ input("Перейдите туда, где вы хотите создать конструкцию, ←
        и нажмите Enter в этом окне")
    pos = mc.player.getTilePos()
    x, y, z = pos.x, pos.y, pos.z
    buildStructure(x, y, z, structure)
```

Листинг 10.10. Основа кода для копирования конструкций

Sort pair — отсортировать пару

Этот код состоит из нескольких частей. Функция `sortPair()` ❶ сортирует значения в паре, помещая их в кортеж, — меньшее значение становится первым элементом, а большее — вторым. Если вызвать `sortPair()` с аргументами 9 и 3, эта функция вернет кортеж (3, 9), поскольку 3 меньше 9. Функция `sortPair()` используется для сортировки пар координат *x*, *y* и *z*, чтобы при вычислении значений переменных `width`, `length` и `depth` всегда получались положительные числа.

Copy structure — копировать конструкцию

Build structure — построить конструкцию

Функция `copyStructure()` ❷ копирует конструкцию, однако ее код не закончен ❸. Функция `buildStructure()` ❹ воссоздает скопированную конструкцию, и ее код тоже недописан. Ваша задача — доделать эти две функции.

В программе используется хитрый способ получения координат области копирования, а также координат места, где нужно создать дубликат. Сначала программа вызывает функцию `input()`, предлагая отвести игрока к одному из углов конструкции и нажать ENTER ❻. Функция `input()` приостанавливает работу программы, ожидая ввода, что позволит вам спокойно подвести игрока к нужному месту. Как только вы нажмете ENTER, программа продолжит работу и получит позицию игрока с помощью функции `getTilePos()`. Координаты противоположного угла конструкции программа узнаёт тем же способом ❼. Затем два набора координат передаются в функцию `copyStructure()`, которая использует их при копировании ❽ (если конструкция велика, это может занять некоторое время). Наконец, программа просит отвести игрока в то место, где нужно создать дубликат, и нажать ENTER ❾. Она получает координаты игрока и передает их в функцию `buildStructure()`, которая воссоздает конструкцию по скопированным ранее данным.

Чтобы закончить программу *duplicateArea.py*, допишите код функций `copyStructure()` и `buildStructure()`. Добавьте в тело функции `copyStructure()` три вложенных цикла, которые будут копировать все блоки области, заданной координатами в ее аргументах, в трехмерный список ❸. В тело функции `buildStructure()` также добавьте три вложенных цикла — с их помощью нужно установить в игровом мире

скопированные блоки, идентификаторы которых хранятся в трехмерном списке ❺. Для этого передайте в функцию координаты нового места.

Убедитесь, что программа обрабатывает все три измерения *x*, *y* и *z* внутри копируемой области. Используйте циклы `for` для изменения значений `x`, `y` и `z`.

Программа *duplicateArea.py* довольно велика, но дело того стоит. Закончив ее, вы сможете создавать в мире Minecraft целые города! Я воспользовался программой, чтобы скопировать симпатичную скалу, оригинал которой показан на рис. 10.20.

Рис. 10.20. Эта скала мне понравилась, и я решил ее скопировать

Запустив программу, поставьте игрока снаружи здания (если вы копируете здание) возле одного из его углов и, вернувшись в окно консоли IDLE, нажмите ENTER. На рис. 10.21 мой игрок стоит возле нижнего угла скалы.

Затем поднимите его в воздух и приблизьте к противоположному углу здания (или иного объекта, который вы копируете), нажмите ENTER во второй раз. На рис. 10.22 мой игрок парит в воздухе с противоположной стороны скалы.

В окне консоли IDLE вы увидите сообщение с просьбой подождать, пока конструкция копируется. Отправьте игрока туда, где вам хочется создать дубликат конструкции, и ждите, когда программа выдаст сообщение с запросом места для создания копии (рис. 10.23).

Нажмите ENTER, и прямо перед вами появится копия объекта! На рис. 10.24 показана копия моей скалы.

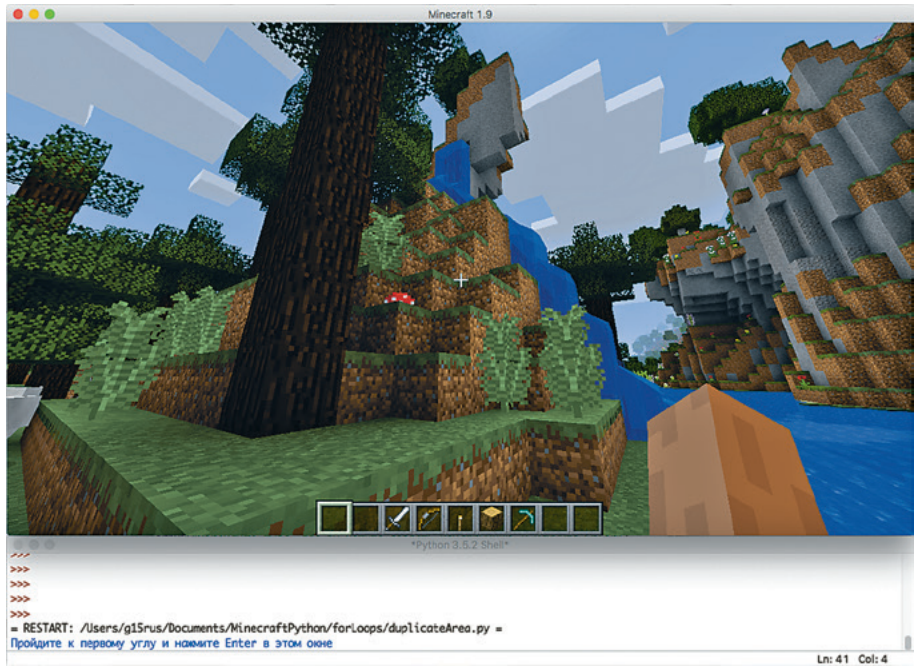

Рис. 10.21. Сначала я подвел игрока к одному из углов конструкции и нажал в окне консоли ENTER

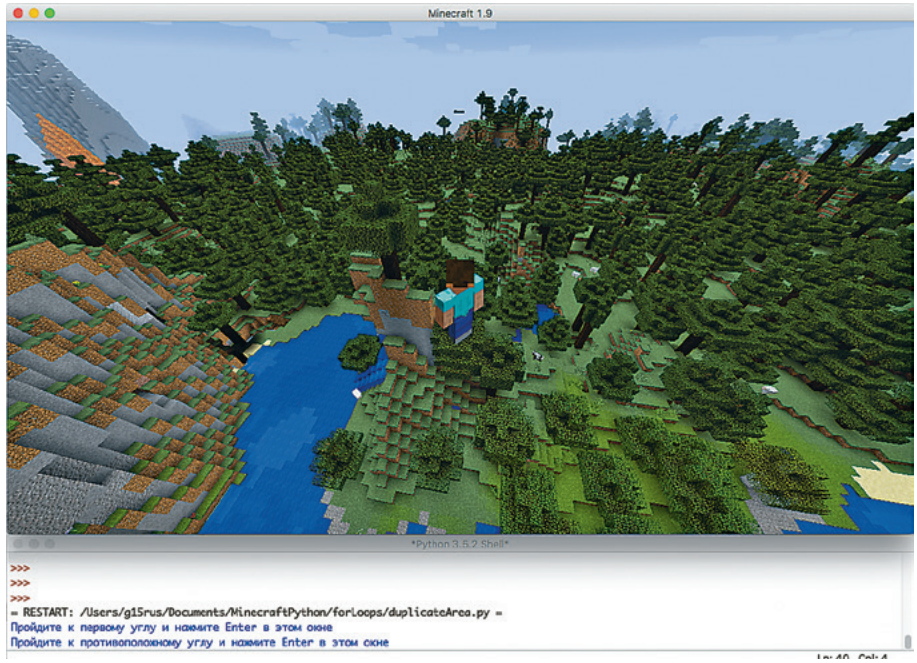

Рис. 10.22. Затем я переместил игрока к противоположному углу и снова нажал ENTER

Рис. 10.23. Я подождал, пока скала скопируется, отвел игрока в то место, где решил создать дубликат, и нажал ENTER

Рис. 10.24. Это копия оригинальной скалы!

Что вы узнали

В этой главе вы узнали довольно много нового. Теперь вы умеете использовать в работе со списками циклы `for` и функцию `range()`, переворачивать содержимое списков, перебирать словари и выходить из циклов с помощью `break`. Также вы научились работать с двухмерными и трехмерными списками, подняв свои возможности по обустройству игрового мира на новый уровень.

От постройки лестниц и пирамид до копирования готовых конструкций и создания пиксель-артов — теперь у вас гораздо больше власти над миром Minecraft! Программы из этой главы — одни из моих любимых. Они, без сомнения, помогут вам при разработке собственных замечательных проектов!

Главы 9 и 10 были посвящены спискам и циклам, которые тесно связаны между собой. В главе 11 вы будете изучать файлы и модули — эти темы имеют непосредственное отношение к функциям. Работая над программами в ходе миссий следующей главы, вы научитесь сохранять конструкции в файлах и загружать их оттуда.

11

КОПИРОВАНИЕ КОНСТРУКЦИЙ С ПОМОЩЬЮ ФАЙЛОВ И МОДУЛЕЙ

Файл — важнейший объект для хранения данных. В файл можно записывать информацию, а затем загружать ее в программу.

До сих пор все созданные вами программы хранили данные исключительно в переменных, значения которых были либо прописаны в коде, либо вводились пользователем с клавиатуры. Конечно, с помощью переменных можно делать потрясающие вещи, но при этом вы ограничены одним сеансом работы и одним миром Minecraft. Файлы позволят сохранять конструкции из вашего мира Minecraft и загружать их в любой другой мир, даже в миры друзей!

В этой главе вы научитесь извлекать данные из файлов и записывать их туда с помощью встроенных в Python функций и двух *модулей*, `pickle` и `shelve`, которые позволяют хранить в файлах самые разные объекты из мира Minecraft.

Модули расширяют возможности программирования на Python. С их помощью легко создавать изображения и делать веб-сайты. В модулях содержатся функции для выполнения тех или иных типичных задач, благодаря которым нам не нужно придумывать собственные решения с нуля.

Также вы познакомитесь с менеджером пакетов `pip` — крайне полезной программой для установки на компьютер новых модулей. Установите с ее помощью модуль `Flask`, и вы создадите простой веб-сайт, который будет подключаться к игре Minecraft и отображать координаты игрока.

Pickle — консервировать

Shelve — складывать

Pip (рекурсивный акроним от Pip Installs Packages) — pip устанавливает пакеты

Flask — пороховница

Работа с файлами

Работая на компьютере, вы постоянно имеете дело с файлами. Файлы хранят тексты, таблицы, изображения, музыку, Python-программы и т. д. Даже эта книга хранилась в текстовом файле, когда я ее писал. Python позволяет создавать и сохранять файлы, а также считывать из них данные, которые можно использовать в программах для Minecraft. Давайте начнем с основ и разберемся, как работать с текстовыми файлами в Python-программах.

Открытие файла

Open — открыть

Прежде чем сделать что-нибудь с файлом, его нужно *открыть*. В Python для этого служит функция `open()`, принимающая два аргумента: путь к файлу и режим доступа к нему. *Путь* — это строка, обозначающая место на компьютере, где хранится файл. А *режим доступа* определяет, может ли программа считывать данные из этого файла и изменять их.

Secret file — секретный файл

Чтобы открыть (или создать) текстовый файл с именем *secretFile.txt*, укажите первым аргументом строку `"secretFile.txt"`:

```
secretFile = open("secretFile.txt", "w")
```

Второй аргумент, `"w"`, задает режим доступа, который определяет, что программе разрешено делать с файлом. В данном случае `"w"` означает, что она может записывать данные в файл *secretFile.txt*.

При вызове функции `open()` с указанием имени файла Python первым делом проверяет, существует ли такой файл. Если это так, программа получит к нему доступ, а если нет, Python создаст файл с таким именем.

Если в пути к файлу не указана *директория* (директория — то же самое, что папка), Python будет искать файл в той директории, где находится сама программа. Если же файл хранится в другом месте, нужно указать это в первом аргументе функции `open()`. Допустим, файл *secretFile.txt* находится в директории *secrets*, тогда путь должен иметь вид `"secrets/secretFile.txt"`:

Secrets — секреты

```
secretFile = open("secrets/secretFile.txt", "w")
```

При этом указанная директория должна существовать, иначе Python выдаст сообщение об ошибке.

Аргумент, определяющий режим доступа, может принимать одно из четырех значений.

w Означает «*только для записи*». Программа может записывать в файл новые данные и перезаписывать уже имеющиеся, но не может считывать их оттуда. Если файл с именем, указанным в первом аргументе `open()`, не существует, Python создаст новый.

Если файл уже существует, то он очистится и станет пустым. Будьте аккуратны!

r Означает «*только для чтения*». Программа может считывать данные из файла, но не может изменять его содержимое. Новый файл в этом режиме не создается.

r+ Означает «*для чтения и записи*». Программа может считывать из файла данные, а также записывать и перезаписывать их. Однако, если файл не существует, новый файл создан не будет — вместо этого программа выдаст ошибку.

a Означает «*для добавления*». Программа может добавлять данные в конец файла, но неспособна менять уже имеющиеся. Считывать данные в этом режиме тоже нельзя. Если файл не существует, будет создан новый.

Все эти режимы доступа предназначены для разных ситуаций. Предположим, вы сохраняете в файл сведения о том, как добраться до найденных вами алмазных копей, и хотите потом использовать эту информацию, не изменив ее случайно. В таком случае нужно применить режим «только для чтения». Если же вы хотите позволить кому-то добавлять в файл новые данные, но не просматривать уже хранящиеся там, подойдет режим «для добавления». Его можно использовать, например, для того чтобы ваши друзья могли делать записи в общем дневнике путешествий, но при этом не видели сведений о спрятанных вами сокровищах!

Далее мы выясним, как записывать данные в файл, а также как закрыть этот файл, чтобы к его содержимому можно было обратиться впоследствии.

Запись данных и сохранение файла

Данные нужно помещать в файл, открытый программой. Это фундаментальный принцип работы с файлами, потому что только таким образом вы сможете записать в файл данные любого типа. Для этого передайте их в качестве аргументов в функцию `write()`.

Write — записать

Давайте откроем файл и запишем в него обычную строку:

```
secretFile = open("secretFile.txt", "w", encoding="utf-8")
secretFile.write("Это секретный файл. Тсс! Чур никому
                 не рассказывать.")
❶ secretFile.close()
```

Первым делом открываем файл с помощью функции `open()`.

Чтобы ваша программа умела считывать текст, набранный кириллицей, и писать в файле по-русски, в функцию `open()` нужно добавить дополнительный аргумент `encoding="utf-8"`. Этот параметр служит своего рода алфавитом, которым программа будет пользоваться, чтобы понимать записи в файле. Прим. науч. ред.

Затем используем точечную нотацию для вызова функции `write()` и передаем ей строку, которую нужно записать в файл *secretFile.txt*. Далее вызываем функцию `close()` ❶ — она сохраняет записанную информацию и закрывает файл. Вызов `close()` очень важен — без этого данные в файл не попадут.

Close — закрыть

Запустите программу, а затем откройте файл *secretFile.txt* в текстовом редакторе, например Блокноте, и убедитесь, что данные успешно сохранились. Измените содержимое строки и запустите программу еще раз. Что произошло? Вместо старого значения в файле должно появиться новое! Измените строку еще раз, при этом вместо `"w"` укажите во втором аргументе функции `open()` значение `"a"`. Запустите код. Что получилось? Здорово, не правда ли?

Чтение данных из файла

Read — читать

Функция `read()` считывает все содержимое открытого прежде файла. Полученные таким образом данные можно использовать в программе, изменять их, а обновленные данные — записывать обратно в файл либо выводить на экран. Так или иначе, сначала вам нужно воспользоваться функцией `read()`.

Не забывайте: прежде чем считывать содержимое файла, его нужно открыть, а закончив работу — закрыть. Сделайте открытие и закрытие файлов своей привычкой, чтобы избежать ошибок в программах!

Show secret file — показать секретный файл

Давайте считаем содержимое файла, а затем выведем на экран данные, полученные с помощью программы *showSecretFile.py* и функций `read()` и `print()`:

showSecretFile.py

```
secretFile = open("secretFile.txt", "r", encoding="utf-8")

❶ print(secretFile.read())
secretFile.close()
```

Сначала мы открываем файл с помощью `open()`, передавая во второй аргумент функции значение `"r"`, чтобы наша программа могла считать данные из файла. Вместо `"r"` можно использовать режим доступа `"r+"`, но, поскольку записывать в файл мы ничего не будем, `"r"` подходит больше. Чтобы вывести содержимое файла на экран, мы используем вызов `secretFile.read()` в качестве аргумента функции `print()` ❶. И наконец, хоть записи в файл и не производилось, его нужно закрыть вызовом `close()`.

Запустите программу и посмотрите, что получится. На экране должно появиться содержимое файла *secretFile.txt*. Итак, теперь вы можете просматривать необходимую информацию без помощи текстового редактора!

Чтение строки из файла

Предположим, у вас есть большой текстовый документ, но вы хотите увидеть только какую-то его часть. В этом вам поможет функция `readline()`. В отличие от `read()`, которая загружает содержимое файла целиком, `readline()` считывает данные по одной строке за раз.

Read line — читать строку

Чтобы испытать функцию `readline()` в действии, сначала добавьте в файл *secretFile.txt* побольше текста. Воспользуйтесь для этого текстовым редактором или вашими новыми знаниями о записи данных в файл! Если выберете Python, добавляйте перед выводимыми строками символы `"\n"` каждый раз, когда захотите напечатать текст с новой строки. Так, если вы запишете в файл строку `"Веселый\nдень\nрождения"`, Python разместит слово `"Веселый"` на одной строке, `"день"` — на следующей и `"рождения"` — на последней, вот так:

```
Веселый
день
рождения
```

Добавив текст в файл *secretFile.txt*, введите в окне программы IDLE следующий код и сохраните его под именем *showSecretLines.py* в новой папке *files*.

Show secret lines — показать секретные строки

Files — файлы

showSecretLines.py

```
secretFile = open("secretFile.txt", "r", encoding="utf-8")

print(secretFile.readline())
print(secretFile.readline())
print(secretFile.readline())

secretFile.close()
```

Прежде чем считывать данные с помощью функции `readline()`, откроем файл вызовом функции `open()`. Чтобы программа могла считать содержимое файла, следует указать режим доступа `"r"` (или `"r+"`). Затем при помощи трех функций `print()` выводим на экран три первые строки из *secretFile.txt*. После этого закрываем файл с помощью `close()`.

Функция `readline()` считывает данные, начиная с первой строки файла. При каждом ее вызове программа автоматически обрабатывает очередную строку. Эта функция очень удобна для вывода нескольких первых строк текстового файла.

Содержимое файла можно преобразовать в список, в котором каждый элемент будет содержать очередную строку. Сделать это позволяет функция `readlines()`. Если вы захотите напечатать строку из середины файла, можно создать цикл `for` и перебирать в нем элементы списка, пока очередь не дойдет до нужной строки.

МИССИЯ 64. ПЕРЕЧЕНЬ ДЕЛ

Если у вас мало свободного времени для игры в Minecraft, строить конструкции вам придется по частям. И чем сложнее конструкция, тем больше вероятность, что ее возведение затянется надолго. Работая с перерывами в несколько дней, вы можете забыть, что начинали делать и чем собирались заняться дальше. Со мной такое происходит постоянно. К счастью, можно написать программу, которая будет напоминать о ваших планах!

Программы в этой миссии создают перечень дел и отображают его в чате. Так вы будете держать перед глазами свои планы и, вернувшись к игре после перерыва, продолжите дело, которым занимались ранее.

Вам предстоит создать две программы: одну — для сохранения в файл перечня дел, другую — для отображения этого перечня на экране. Начнем с программы сохранения.

Часть 1: сохранение записей

Перво-наперво нам нужна программа для сохранения записей в перечне дел. Ее основа показана в листинге 11.1. Для добавления записей воспользуйтесь циклом `while` и функцией `input()`. Введите код в новый файл IDLE и сохраните его под именем *inputToDoList.py* в папке *files*.

Input to do list — внести в перечень дел

inputToDoList.py

```
❶ toDoFile =

❷ toDoList = ""

❸ toDoItem = input("Введите описание дела: ")
```

```
❹ while toDoItem != "выход":
❺     toDoList = toDoList + toDoItem + "\n"
      toDoItem = input("Введите описание дела: ")

❻ # Запишите введенные данные в файл
❼ # Закройте файл
```

Листинг 11.1. Основа кода для сохранения записей в перечне дел

To do list — перечень дел

В начале программы создается пустая строка `toDoList` ❷, где будут храниться все записи перечня. Затем с помощью функции `input()` программа предлагает пользователю ввести описание дела ❸. Далее цикл `while` проверяет, не равна ли введенная строка слову `"выход"` ❹. Если не равна, программа добавляет в перечень дел текст пользователя и переход на новую строку (`"\n"`). Если же пользователь введет `"выход"`, цикл завершится и больше нельзя будет сделать запись в файл.

Вам предстоит добавить в эту программу команды, которые откроют файл и запишут в него строку, хранящуюся в `toDoList`, а затем закроют файл. Чтобы открыть файл, вызовите в начале программы ❶ функцию `open()`. Укажите режим доступа «для записи», а сам файл назовите *toDoList.txt*. Если файла с таким именем не существует, ваша программа создаст новый.

В конце сохраните введенные данные в файл, использовав функцию `write()` для записи значения переменной `toDoList` в переменную `toDoFile` ❻. Не забудьте закрыть файл вызовом функции `close()` ❼.

На рис. 11.1 показано, как выглядит перечень дел, созданный с помощью этой программы. Закончив добавлять записи, я ввел `"выход"`.

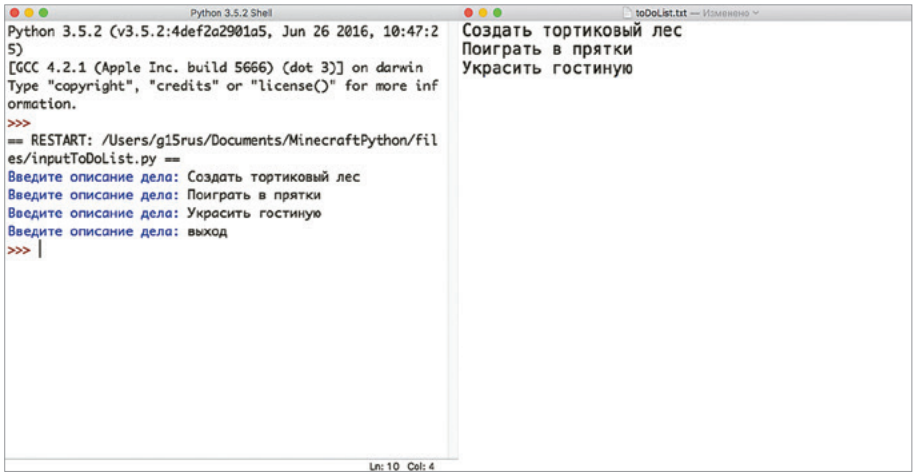

Рис. 11.1. Я добавляю в перечень новые дела

Часть 2: вывод перечня дел на экран

Итак, у вас уже есть программа, которая создает файл с перечнем дел. Осталось построчно вывести этот перечень в чат Minecraft. Основа программы, которая будет это делать, показана в листинге 11.2. Скопируйте код в новый файл и сохраните его под именем *outputToDoList.py* в папке *files*.

Output to do list — вывести перечень дел

outputToDoList.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

❶ toDoList =

for line in toDoList:
❷     # Выводим строку "line" в чат
```

Листинг 11.2. Программа для вывода перечня дел в чат Minecraft

В листинге 11.2 для вывода в чат каждой строки из файла *toDoList.txt* используется цикл `for`. Однако программа не закончена. Допишите ее, добавив открытие файла *toDoList.txt,* — воспользуйтесь для этого функцией `open()` ❶. Укажите режим доступа «для чтения». Затем с помощью функции `postToChat()` ❷ добавьте внутрь цикла `for` код для вывода в чат Minecraft строки `line`.

Line — строка

На рис. 11.2 показан мой перечень дел в чате Minecraft.

Рис. 11.2. Теперь, вернувшись в игру, я вижу, что мне нужно сделать

Модули

Модуль — это набор функций, которые выполняют в программе конкретную задачу: например, делают научные вычисления или создают игру. Во многих языках программирования (и в Python тоже) модуль оформляется в виде файла. Для Python написано великое множество модулей, причем некоторые из них мы уже использовали в этой книге. Minecraft Python API тоже является модулем, и, вводя строку `from mcpi.minecraft import Minecraft`, вы загружаете его в свою программу. Minecraft Python API позволяет программе подключаться к игре, а также содержит готовые функции, которые избавляют от необходимости искать собственные решения.

В комплекте с Python идет набор полезных модулей. Они составляют *стандартную библиотеку* языка Python. Однако помимо них вы можете установить себе на компьютер и другие модули. Подробнее этот вопрос мы рассмотрим в разделе «Установка новых модулей с помощью pip» на с. 307.

А сейчас давайте задействуем модули для решения разных задач. В качестве примера возьмем модуль `pickle`, реализующий более продвинутые способы хранения данных в файлах, чем уже известные вам операции записи и чтения. Итак, знакомьтесь!

Модуль pickle

Модуль `pickle` отлично подходит для хранения в файлах данных со сложной структурой. Если использовать стандартные функции, о которых шла речь в начале этой главы, сохранить в файле словарь или многомерный список будет непростой задачей. Модуль `pickle` значительно ее упрощает.

Однако `pickle` можно использовать и для работы с простыми данными. Например, сохранять и загружать с его помощью числа, не преобразовывая их в строки (что пришлось бы делать при использовании стандартных функций чтения-записи).

Иными словами, с помощью `pickle` можно сохранить значение переменной в файле и считать его оттуда в этой же или другой программе без какой-либо дополнительной обработки. При считывании тип данных останется таким же, каким был при записи, и не важно, строка ли это, целое число, вещественное число или булево значение.

Далее мы научимся импортировать модули на примере `pickle`. Затем вам предстоит применить его для сохранения сложных данных, а именно постройки из мира Minecraft!

Импортирование pickle

Прежде чем использовать какие-либо функции модуля, их нужно загрузить с помощью команды `import`. Вы уже делали это, работая с модулем `time` и его функциями Minecraft Python API.

После загрузки модуля вы можете обращаться к его функциям через точечную нотацию. Для этого введите имя модуля, точку и имя функции, которую хотите вызвать. Давайте загрузим модуль `pickle` и воспользуемся некоторыми его функциями:

```
❶ import pickle

  locations = {'Тимур': 'Лес', 'Артур': 'Горы', 'Анна': 'Город'}

❷ secretFile = open("secretFile.txt", "wb", encoding="utf-8")
❸ pickle.dump(locations, secretFile)
```

Загружаем модуль `pickle` ❶, затем открываем файл *secretFile.txt* в специальном режиме доступа `"wb"` ❷. При обращении к `pickle` всегда следует добавлять к символу, означающему режим доступа, букву `b`. В данном случае `"wb"` позволяет открыть файл для записи данных в специальном нетекстовом формате, который необходим модулю.

Dump — свалить в кучу

Locations — местоположения

Функция `dump()` записывает данные в файл, сохраняя в нем значение указанной переменной ❸. Эта функция принимает два аргумента: данные для записи и открытый файл, в который их нужно поместить. В нашем случае данные — это словарь `locations` с местами дислокации секретных агентов, а записать эти данные нужно в файл *secretFile*. Поскольку `dump()` находится в модуле `pickle`, при вызове этой функции нужно воспользоваться точечной нотацией: `pickle.dump()`.

Load — загрузить

Помимо этого, модуль `pickle` позволяет загружать сохраненные ранее данные с помощью функции `load()`. Эта функция принимает единственный аргумент — файл, из которого производится загрузка, а возвращает полученные данные.

В следующем примере программа считывает из файла словарь `locations`, который мы сохранили туда ранее:

```
  import pickle

❶ secretFile = open("secretFile.txt", "rb", encoding="utf-8")
  locations = pickle.load(secretFile)
```

Сначала мы открываем файл в режиме `"rb"` ❶, что позволяет программе считывать оттуда данные в формате, который использует `pickle`. Затем загружаем из файла содержимое словаря, вызывая для этого функцию `load()`.

После загрузки с этим словарем можно работать так же, как с обычным, содержимое которого задано в программе. Например, можно получить

значение по ключу. Для этого добавьте после вызова `pickle.load()` следующий код:

```
print(locations['Артур'])
```

Программа выведет на экран `"Горы"` — значение, которое соответствует ключу `"Артур"`. Словарь загрузился в точно таком же виде, в каком был на момент записи в файл, и теперь мы можем с ним работать, как с любым другим словарем.

Аналогичным образом в файлы сохраняются и загружаются из них списки и переменные.

Импортирование одиночной функции с помощью from

После загрузки модуля нам становятся доступны все его функции, однако порой в программе нужна лишь одна из них. В таком случае при импортировании следует воспользоваться ключевым словом `from`. Оно позволяет обращаться к функции, не указывая при каждом вызове имя модуля. То есть вместо `модуль.функция()` вы можете писать просто `функция()`.

From — из

Предположим, в вашей программе из всех функций модуля `pickle` нужна только `dump()`. В таком случае можно изменить код из прошлого примера так:

```
❶ from pickle import dump

  locations = {'Тимур': 'Лес', 'Артур': 'Горы', 'Анна': 'Город'}

  secretFile = open("secretFile", "wb", encoding="utf-8")
❷ dump(locations, secretFile)
```

Ключевое слово `from` в первой строке позволяет импортировать из модуля `pickle` только функцию `dump()` ❶. В последней строке кода идет вызов этой функции ❷. Обратите внимание, что теперь обращение к `dump()` производится просто по имени, без точечной нотации.

С помощью `from` возможно импортировать и несколько функций. В этом случае следует разделить их имена запятыми. Если вашей программе нужны `dump()` и `load()` из модуля `pickle`, их можно загрузить так:

```
❶ from pickle import dump, load
  locations = {'Тимур': 'Лес', 'Артур': 'Горы', 'Анна': 'Город'}
  secretFile = open("secretFile", "wb", encoding="utf-8")
```

```
❷ dump(locations, secretFile)

❸ locations = load(secretFile)
  print(locations['Артур'])
```

В первой строке после `from` через запятую указываются функции `dump()` и `load()` ❶. Это значит, что далее в программе их можно использовать без точечной нотации ❷ ❸.

Импортирование всех функций с помощью *

Но это еще не всё. Вы можете загружать и вызывать без точечной нотации все функции модуля. Просто поставьте после ключевого слова `import` звездочку (*), вот так:

```
❶ from pickle import *
  locations = {'Тимур': 'Лес', 'Артур': 'Горы', 'Анна': 'Город'}

  secretFile = open("secretFile", "wb", encoding="utf-8")
❷ dump(locations, secretFile)

❸ locations = load(secretFile)
  print(locations['Артур'])
```

Поскольку здесь с помощью `from` и звездочки импортируются все функции модуля ❶, при вызове `dump()` ❷ и `load()` ❸ не нужно использовать точечную нотацию.

Такой вариант загрузки очень удобен, однако связан с определенным риском. Если вы используете несколько модулей, имена некоторых функций в разных модулях могут оказаться одинаковыми. В таком случае Python не поймет, к какой из функций следует обращаться, и выдаст ошибку. Поэтому, работая с несколькими модулями, лучше не используйте *, а импортировать конкретные, нужные вам функции.

Псевдоним модуля

Порой бывает необходимо переименовать модуль — например, когда его имя неудобно вводить из-за большой длины, сложности запоминания или когда оно совпадает с именем другого модуля.

As — как

В этом случае при загрузке модулю можно дать *псевдоним* — при помощи ключевого слова `as`. В следующем примере модуль `pickle` переименовывается в `p`:

```
import pickle as p
```

Теперь при любом обращении к модулю в программе можно писать не `pickle`, а `p`, вот так:

```
p.dump(locations, secretFile)
```

Короткая запись `p.dump()` вместо длинной `pickle.dump()` позволяет сэкономить время при написании кода.

МИССИЯ 65. СОХРАНЕНИЕ И ЗАГРУЗКА КОНСТРУКЦИИ

Больше всего я люблю сооружать в Minecraft различные конструкции: часами строю дома, замки, деревни и много-много чего еще. Однако при переходе в другую область игрового мира или вообще в другой мир мне приходится оставлять свои творения. Уверен, вам тоже доводилось расставаться с замечательными постройками.

Вот если бы их можно было забрать с собой… Что ж, с модулями `pickle` и Minecraft Python API это возможно!

В ходе миссии вы напишете две программы: одну для сохранения конструкций в игре Minecraft, другую — для их загрузки. Обе они основаны на коде *duplicateArea.py* из главы 10 (с. 277).

Часть 1: сохранение конструкции

Первая программа будет сохранять объект игрового мира в файл. Код копирования показан в листинге 11.3. Введите его в IDLE и сохраните под именем *saveStructure.py* в папке *files*.

Save structure — сохранить конструкцию

saveStructure.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import pickle


def sortPair(val1, val2):
    if val1 > val2:
        return val2, val1
    else:
        return val1, val2


❶ def copyStructure(x1, y1, z1, x2, y2, z2):
    x1, x2 = sortPair(x1, x2)
    y1, y2 = sortPair(y1, y2)
    z1, z2 = sortPair(z1, z2)
```

```
    width = x2 - x1
    height = y2 - y1
    length = z2 - z1

    structure = []

    print("Пожалуйста, подождите…")

    # Копируем конструкцию
    for row in range(height):
        structure.append([])
        for column in range(width):
            structure[row].append([])
            for depth in range(length):
❷               block = mc.getBlockWithData(x1 + column, y1 + row, z1 + depth)
                structure[row][column].append(block)

    return structure

❸ # Получаем координаты первого угла
input("Пройдите к первому углу и нажмите Enter в этом окне")
pos1 = mc.player.getTilePos()

x1 = pos1.x
y1 = pos1.y
z1 = pos1.z

❹ # Получаем координаты второго (противоположного) угла
input("Подойдите к противоположному углу и нажмите Enter в этом окне")
pos2 = mc.player.getTilePos()

x2 = pos2.x
y2 = pos2.y
z2 = pos2.z

❺ structure = copyStructure(x1, y1, z1, x2, y2, z2)

❻ # Сохраняем конструкцию в файл
```

Листинг 11.3. Основа кода для сохранения конструкции в файл

Функция `copyStructure()` копирует область игрового мира, помещая ее координаты в трехмерный список ❶. В качестве аргументов она принимает два набора координат. Я немного изменил эту функцию по сравнению с программой *duplicateArea.py*, использовав вместо `getBlock()` функцию `getBlockWithData()` ❷. Помимо идентификатора блока она возвращает для заданных координат состояние блока. Это нужно, к примеру, для блоков-ступеней, у которых состояние определяет, с какой стороны находится выемка. После копирования те блоки, вид которых зависит от их состояния, останутся такими же, как в оригинальной конструкции.

Get block with data — получить блок с исходными данными

Как вы помните, в этом коде я применил хитрый прием, получив координаты копируемой области через позицию игрока. После запуска программа попросит вас подвести игрока к одному из углов конструкции и нажать ENTER в окне консоли Python ❸. При этом она получит координаты игрока, чтобы использовать их в качестве первого набора координат копируемой области. Затем программа попросит вас подвести игрока к противоположному углу конструкции и снова нажать ENTER для получения второго набора координат ❹. В итоге, чтобы скопировать конструкцию, вам будет достаточно переместить игрока с одного места на другое и при этом не придется вводить координаты с клавиатуры или жестко прописывать их в коде.

Далее координаты передаются в функцию `copyStructure()` ❺, которая и осуществляет копирование. Полученные из этой функции данные сохраняются в переменной `structure`.

Чтобы дописать программу, вам нужно добавить код, открывающий новый файл для работы с `pickle`, — пусть этот файл называется *pickleFile*. Затем напишите код для сохранения конструкции в файл ❻.

На рис. 11.3 показана башня, которую я построил в своем мире Minecraft.

Рис. 11.3. Башня, которую я решил скопировать

Чтобы скопировать башню с помощью *saveStructure.py*, я запустил программу, подвел игрока к одному из углов и нажал ENTER в окне консоли IDLE (рис. 11.4).

Затем я поднял игрока в воздух, переместил к противоположному углу башни и снова нажал ENTER (рис. 11.5).

Повторите мои действия и сохраните с помощью программы *saveStructure.py* какое-нибудь из своих сооружений.

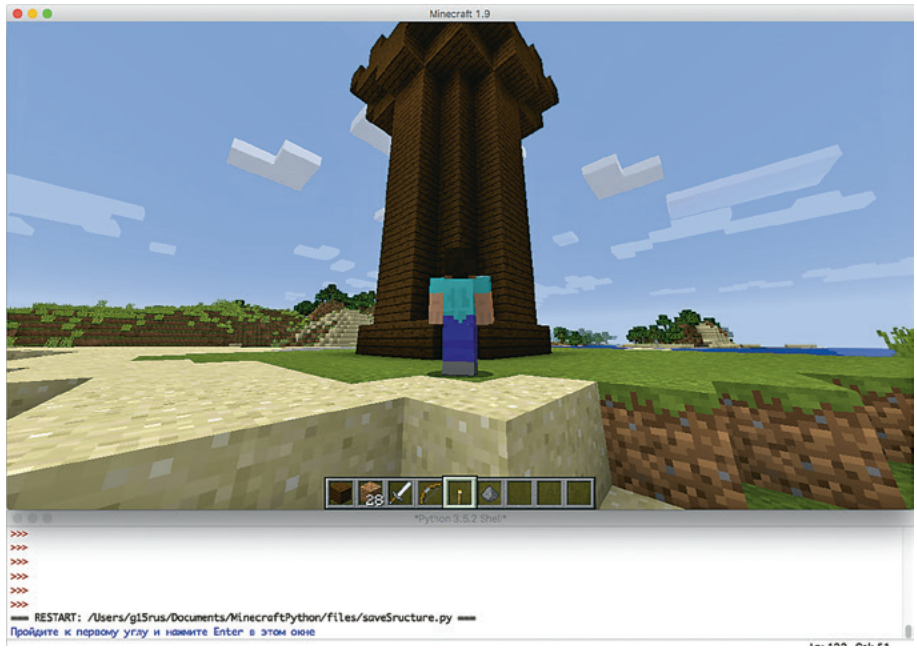

Рис. 11.4. Мой игрок стоит у одного из углов башни

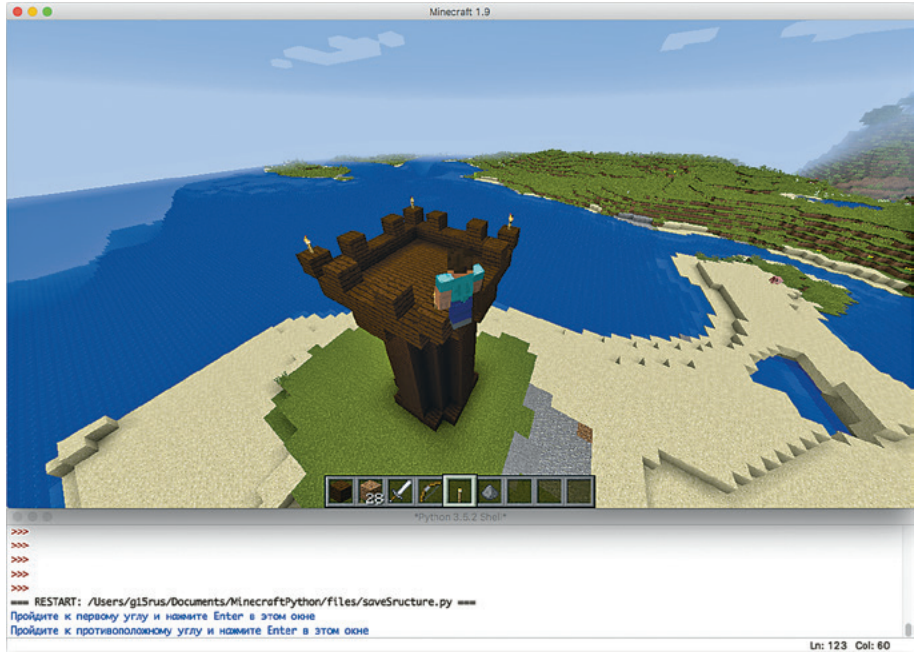

Рис. 11.5. Я переместил игрока к противоположному углу башни

Теперь давайте перейдем ко второй части процесса — загрузке конструкции из файла в игру.

Часть 2: загрузка конструкции

Вторая часть программы должна из файла *pickleFile* загружать в игровой мир конструкцию, копия которой была создана программой *saveStructure.py*. В листинге 11.4 дан фрагмент кода из *duplicateArea.py* (с. 277), воссоздающий конструкцию на основе данных из вложенных списков. Введите код листинга 11.4 в IDLE и сохраните под именем *loadStructure.py* в папке *files*.

Load structure — загрузить конструкцию

loadStructure.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import pickle


❶ def buildStructure(x, y, z, structure):
    xStart = x
    zStart = z
    for row in structure:
        for column in row:
            for block in column:
                mc.setBlock(x, y, z, block.id, block.data)
                z += 1
            x += 1
            z = zStart
        y += 1
        x = xStart

# Открываем файл и загружаем трехмерный список structure
❷ structure =

❸ pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
❹ buildStructure(x, y, z, structure)
```

Листинг 11.4. Этот код должен воссоздавать конструкцию на основе данных из файла, но он не закончен

Функция `buildStructure()` ❶ выполняет бóльшую часть работы — она создает в игровом мире конструкцию, используя для этого четыре аргумента: координаты *x*, *y* и *z*, а также трехмерный список, в котором содержатся параметры конструкции.

Используя модуль `pickle`, загрузите конструкцию в программу и поместите ее в переменную `structure` ❷. Для этого сначала вызовите `open()`, чтобы открыть файл *pickleFile* с сохраненной конструкцией, а затем загрузите конструкцию в переменную `structure` с помощью вызова `pickle.load()`. Затем *pickleFile* желательно закрыть, вызвав для этого функцию `close()`.

Также в листинге 11.4 есть код, использующий позицию игрока для получения координат места, в котором нужно воссоздать конструкцию ❸.

После того как конструкция будет загружена, а координаты заданы, давайте создадим ее копию при помощи вызова функции `buildStructure()` ❹. Она принимает в качестве аргументов координаты места и значение переменной `structure`.

На рис. 11.6 показан результат работы программы. Башня, которую я ранее сохранил в файл, теперь загружена и воссоздана в новом месте. Попробуйте сами! Теперь у вас есть возможность брать свои сооружения с собой!

Рис. 11.6. Посмотрите, это копия моей башни!

Но что если вам захочется воссоздать целую деревню? Вы можете сохранить каждый из домов в отдельный файл с помощью `pickle`, однако это слишком долго. Модуль `pickle` хорош для сохранения одиночных конструкций, но для нескольких зданий он не очень подходит. Здесь вам на помощь придет модуль `shelve` — о нем и поговорим дальше.

Модуль shelve и хранение наборов данных

Модуль `pickle` сохраняет лишь одну конструкцию за раз, однако в некоторых программах может понадобиться сохранить сразу несколько объектов. Если использовать для этого `pickle`, вам придется создавать несколько файлов, что заметно усложнит написание кода.

Модуль `shelve` решает эту проблему, поскольку позволяет хранить несколько конструкций в одном файле. `Shelve` похож на шкаф, на полках которого лежит разная информация.

Открытие файла с помощью shelve

После загрузки модуля `shelve` вы можете использовать его функцию `open()` для открытия файла. Если файла с указанным именем не существует, будет создан новый.

Следующий код открывает файл *locationsFile.db* и присваивает его переменной `shelveFile`:

Locations file — файл с местоположениями

```
import shelve
shelveFile = shelve.open("locationsFile.db")
```

Функция `open()` принимает единственный аргумент — имя файла. При этом вам не нужно указывать режим доступа, файл автоматически открывается для чтения и записи.

Имя файла для модуля `shelve` обязательно должно оканчиваться расширением *.db: locationsFile.db*.

Добавление, изменение и доступ к данным файла при помощи shelve

Модуль `shelve` работает как словарь. Чтобы добавить данные в файл, после имени словаря нужно указать в квадратных скобках ключ, через который будет осуществляться доступ к этим данным. Предположим, что агент Оксана находится на подводной лодке и вы хотите сохранить ее местоположение в словаре `shelveFile`:

```
import shelve
shelveFile = shelve.open("locationsFile.db")
shelveFile['Оксана'] = 'Подводная лодка'
shelveFile.close()
```

Первым делом мы открываем файл. Далее сохраняем значение `"Подводная лодка"` в словаре `shelveFile`, связав его с ключом

"Оксана". Затем при помощи функции close() из модуля shelve переносим сохраненные данные в файл и закрываем его.

Если с указанным ключом в словаре shelveFile было связано другое значение, этот код его обновит. Предположим, Оксана завершила свою миссию и вернулась в штаб. Тогда обновить ее местоположение можно так:

```
import shelve
shelveFile = shelve.open('locationsFile.db')
shelveFile['Оксана'] = 'Штаб'
shelveFile.close()
```

Теперь ключу "Оксана" соответствует значение "Штаб".

Доступ к значениям в модуле shelve можно получить так же, как и в словаре, — с помощью ключей. Чтобы вывести местоположение Оксаны, напишите такой код:

```
import shelve
shelveFile = shelve.open('locationsFile.db')
print(shelveFile['Оксана'])
```

После этого на экране появится строка "Штаб".

Так же как и обычный словарь, модуль shelve может сохранять в файле и загружать в программу данные любых типов: вещественные числа, строки, булевы значения, многомерные списки, словари и т. д. В следующей миссии вам предстоит написать программу, которая будет сохранять и считывать данные из многомерных списков сразу нескольких конструкций!

МИССИЯ 66: СОХРАНЕНИЕ НАБОРА КОНСТРУКЦИЙ

Эта миссия также разделена на две части: в первой мы будем сохранять в файле данные разных конструкций, а во второй — загружать их в программу.

Ваша задача — изменить программы из миссии 65 так, чтобы в их кодах вместо модуля pickle использовался модуль shelve. Также вам предстоит добавить команды, задающие название каждой конструкции. Откройте файлы *saveStructure.py* и *loadStructure.py* и сохраните их под именами *saveCollection.py* и *loadCollection.py*.

Save collection — сохранить коллекцию

Load collection — загрузить коллекцию

Так же как в предыдущей миссии, рассмотрим сначала программу для сохранения, а затем — программу для загрузки конструкций.

Часть 1: запись конструкции в коллекцию

Ниже показана часть оригинального кода *saveStructure.py* с пометками, которые подскажут вам, где нужно внести изменения.

saveCollection.py

```
❶ import pickle

--пропущено--

# Запрашиваем название конструкции
❷ structureName = input("Как вы хотите назвать конструкцию? ")

# Сохраняем конструкцию в файл
❸ pickleFile = open("pickleFile", "wb", encoding="utf-8")
❹ pickleFile.dump(structure)
```

В код добавлена строка ❷, которая запросит название конструкции, если вы захотите сохранить ее с помощью модуля `pickle`. Запустив свою версию программы, я могу ввести, например, `"Дом"` или `"Тортиковый лес"`. У каждой конструкции должно быть уникальное название. Если вы назовете новый объект именем уже имеющегося, данные не сохранятся.

Чтобы программа использовала вместо `pickle` модуль `shelve`, в код нужно внести два изменения. Во-первых, поменять имя импортируемого модуля ❶. Во-вторых, переписать несколько последних строк кода, чтобы конструкция сохранялась с помощью `shelve`, а не `pickle`. Вызвав `shelve.open()`, откройте файл *structuresFile.db* и присвойте его переменной `shelveFile` ❸. Затем сохраните значение переменной `structure` в словаре `shelve`, используя в качестве ключа значение переменной `structureName` ❹. Выглядеть это должно примерно так: `shelveFile[structureName] = structure`. Затем, в последней строке, закройте файл с помощью функции `close()`.

Structures file — файл с конструкциями

Часть 2: загрузка конструкции из коллекции

Теперь нужно изменить файл *loadCollection.py*. Я не привожу часть программы, чтобы сэкономить место и чтобы строки, которые вам предстоит изменить, были заметнее.

loadCollection.py

```
❶ import pickle

--пропущено--
```

```
❷ structure = pickle.load("pickleFile")
❸ structureName = input("Введите название конструкции: ")

  pos = mc.player.getTilePos()
  x = pos.x
  y = pos.y
  z = pos.z

❹ buildStructure(x, y, z, structureDictionary[structureName])
```

Я добавил команду с запросом названия конструкции, которую вы хотите скопировать ❸, а также изменил последнюю строку кода, чтобы данные объекта извлекались из словаря `shelve` и передавались в функцию `buildStructure()`.

В эту программу нужно внести несколько изменений. Во-первых, аналогично программе *saveCollection.py*, вместо `pickle` импортируйте модуль `shelve` ❶. Во-вторых, откройте файл *structuresFile.db*, созданный программой *saveCollection.py*, при помощи `shelve.open()` ❷. Присвойте результат вызова переменной `structureDictionary` ❹. Код должен выглядеть так: `structureDictionary = shelve.load ("structuresFile.db")`.

Structure dictionary — структура словаря

Теперь данные всех конструкций, включая их имена и типы блоков, будут храниться в файле *structuresFile.db*. Благодаря этому для загрузки определенной конструкции перед запуском программы *loadCollection.py* не нужно менять ее код. Следует лишь ввести в ответ на запрос название конструкции, которую вы хотите скопировать.

Давайте посмотрим, как работает *loadCollection.py*. Сначала я запустил программу копирования *saveCollection.py*, подвел игрока к одному из углов конструкции и нажал ENTER (рис. 11.7).

Затем я поднял его в воздух к противоположному углу и снова нажал ENTER (рис. 11.8).

Далее программа попросила меня ввести название конструкции. Как видно на рис. 11.9, я назвал ее `"Тортиковое дерево"`.

Затем я запустил программу *loadCollection.py*, отправил игрока туда, где решил создать копию конструкции, и ввел в окне консоли название объекта (рис. 11.10). Программа тут же, на моих глазах, создала копию тортикового дерева. Чем не волшебство!

Вы можете повторить этот процесс для любого количества зданий или других объектов. Я, например, скопировал еще дом (рис. 11.11). Единожды сохранив конструкцию, вы можете загрузить ее в Minecraft в любой момент! Для этого нужно лишь запустить программу *loadCollection.py* и ввести имя конструкции, указанное при ее сохранении.

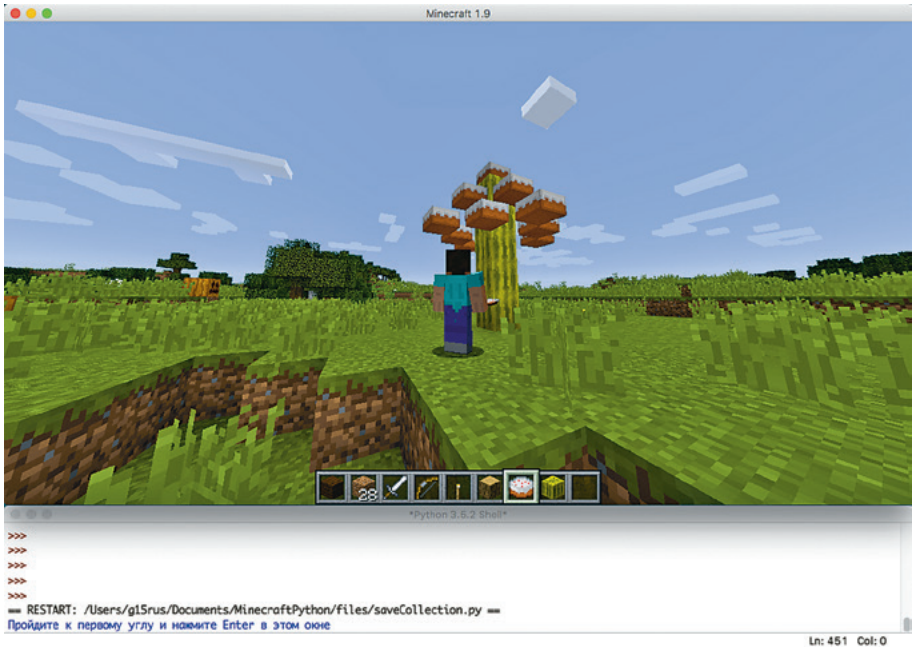

Рис. 11.7. Мой игрок стоит у одного из углов конструкции, которую я хочу скопировать

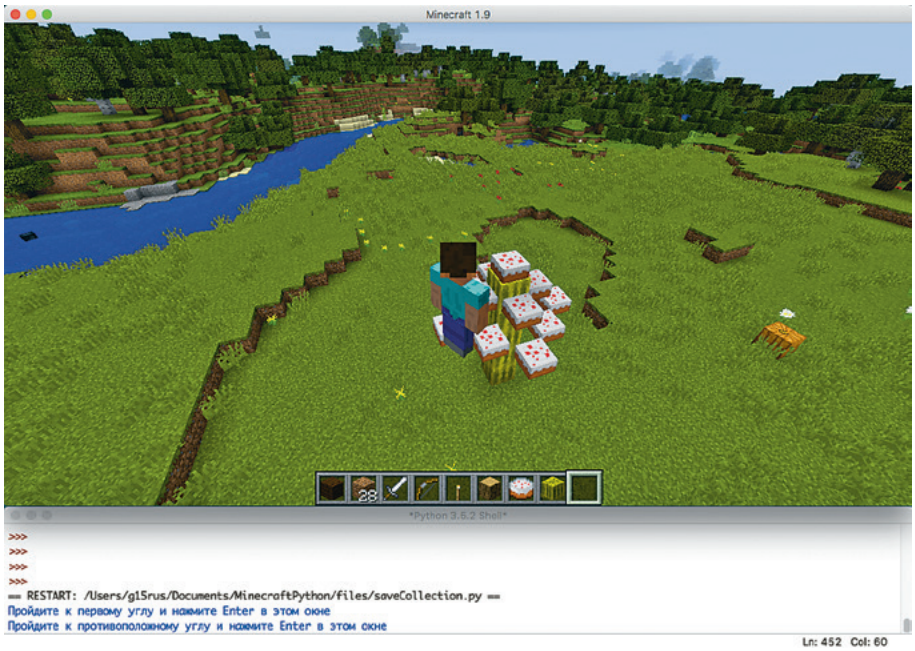

Рис. 11.8. Мой игрок находится у противоположного угла конструкции

Рис. 11.9. Я ввел имя, под которым решил сохранить конструкцию

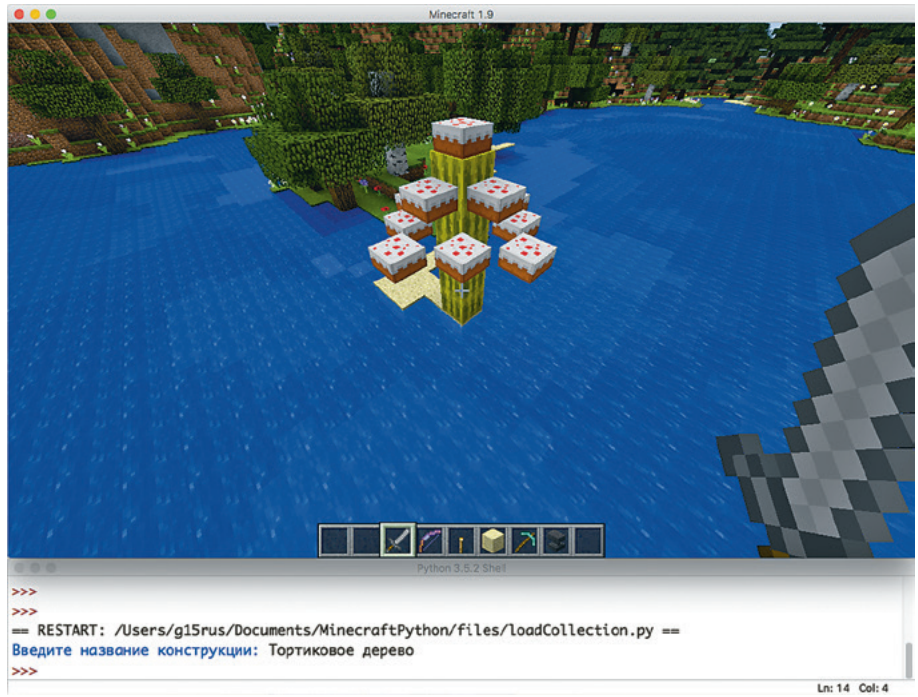

Рис. 11.10. Чтобы создать копию конструкции,
я просто ввожу ее название, и объект возникает передо мной

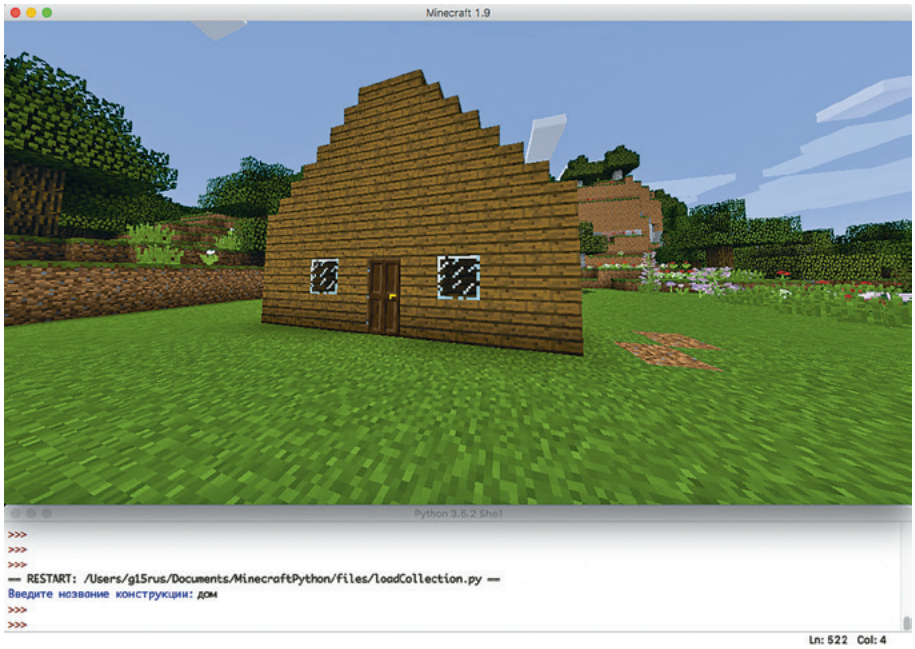

Рис. 11.11. Программу loadCollection.py можно использовать для сохранения и воссоздания самых разных конструкций. В этот раз я скопировал дом

Установка новых модулей с помощью pip

Помимо `pickle` и `shelve` существуют тысячи других модулей, которые вы можете использовать в своих программах. Когда модулей так много, особенно важно правильно их установить. Чтобы облегчить жизнь программистам, в комплекте с Python идет менеджер пакетов под названием `pip`. *Менеджер пакетов* — это программа, которая работает с набором (пакетом) файлов, необходимых для установки других программ, а также отвечает за их обновление и удаление.

Менеджер пакетов `pip` работает с пакетами, которые написаны на языке программирования Python. Его база данных содержит множество полезных модулей. В этом разделе мы разберем, как устанавливать пакеты с помощью `pip`, и покажем это на примере модуля `Flask`, предназначенного для создания настоящих веб-сайтов!

Если вы используете последнюю версию Python 3, `pip` уже есть на вашем компьютере. Если же у вас более ранняя версия, `pip` может быть не установлен. Самый простой способ это исправить — установить последнюю версию Python (см. раздел «Установка Python» на с. 21, если у вас Windows и на с. 35, если у вас Mac).

Итак, давайте посмотрим, как пользоваться `pip`. В зависимости от операционной системы инструкции по работе с `pip` будут немного

различаться — читайте раздел, который соответствует операционной системе вашего компьютера!

Работа с pip в Windows

Для начала работы с `pip` в операционной системе Windows нужно открыть окно командной строки. Оно напоминает окно консоли и позволяет вводить одиночные команды, которые выполняются после нажатия ENTER.

Чтобы открыть окно командной строки, нажмите клавишу WINDOWS или войдите в меню **Пуск** (**Start**) и введите `cmd` в строке поиска. После запуска программы cmd вы увидите черное окно, как на рис. 11.12.

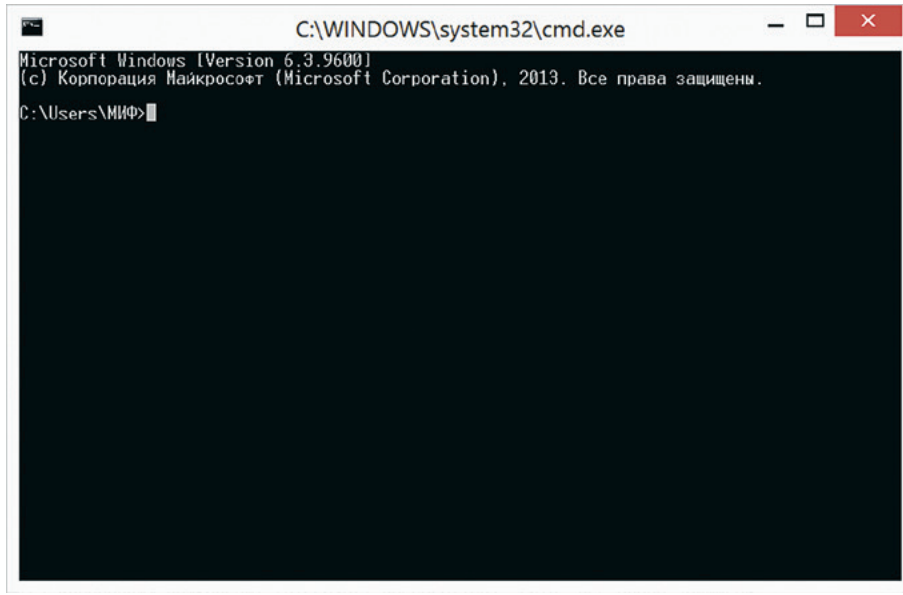

Рис. 11.12. Окно командной строки Windows

Далее нужно ввести команду `pip` и действие, которое вы хотите выполнить. Давайте установим модуль `Flask`. Для этого введите в окне следующую команду:

```
> pip install Flask
```

На сайте каталога пакетов Python (*http://pypi.python.org/*) перечислено немало других модулей, которые можно установить с помощью `pip`.

Работа с pip в Mac OS и Raspberry Pi

Чтобы использовать менеджер пакетов `pip` в Mac OS или Raspberry Pi, перед командой `pip` нужно ввести слово `sudo`. Напишите в окне командной строки:

Sudo (словослияние от superuser do) — выполнить от имени суперпользователя

```
$ sudo pip install Flask
```

Если в результате вы получите сообщение об ошибке, загляните в инструкцию для Mac OS или Raspberry Pi в главе 1 и убедитесь, что Python установлен правильно.

На сайте каталога пакетов Python (*http://pypi.python.org/*) перечислено немало других модулей, которые можно установить с помощью pip.

Модуль для создания веб-сайтов Flask

В этом разделе вы узнаете, как создать простейший веб-сайт с помощью модуля `Flask` и как интегрировать его с игрой Minecraft, чтобы на сайте отображались координаты игрока!

Для создания веб-сайта и управления его работой достаточно лишь нескольких строк кода. Вы пишете код, добавляете информацию про модуль `Flask`, а потом запускаете программу и получаете веб-сайт, который можно открыть в веб-браузере.

Код листинга 11.5 создает простой веб-сайт. Там отображается наиболее важная информация обо мне — имя.

namePage.py

```
from flask import Flask
❶ app = Flask(__name__)


❷ @app.route("/")
  def showName():
❸     return "Крэйг Ричардсон"

❹ app.run()
```

Листинг 11.5. Программа, которая создает веб-сайт с помощью модуля Flask

Для создания веб-сайта первым делом нужно вызвать функцию `Flask()` ❶. Ее аргумент `__name__` указывает на то, что весь Flask-проект содержится в этом файле и искать где-либо другие части программы не нужно. Обратите внимание: в начале и в конце аргумента `__name__` стоят два знака подчеркивания, а не один.

Тег — ключевое слово или маркер, которым пользуются для разметки текста при создании сайтов. *Прим. науч. ред.*

Show name — показать имя

Name page — страница с именем

Run module — запустить модуль

В теге используется декоратор `@app.route()`. *Декораторы* позволяют указывать дополнительную информацию о функциях. В этой программе декоратор `@app.route()` сообщает модулю `Flask`, в какой части веб-сайта нужно использовать функцию `showName()`: аргумент `"/"` означает, что функция должна вызываться на главной странице ❷. Команда `return` определяет, какую информацию нужно отобразить. В нашем случае она возвращает мои имя и фамилию, поэтому на странице сайта появятся именно они ❸. Последняя строка программы запускает `Flask` ❹.

Сохраните эту программу под именем *namePage.py* в папке *files*. Чтобы сделать собственный веб-сайт, поменяйте строку после `return` на другую, какую хотите.

Для запуска веб-сайта выберите в меню IDLE **Run → Run Module**, и программа сгенерирует сайт. Его можно будет открыть в браузере. Чтобы узнать адрес сайта, изучите строку, которую программа выведет при запуске. Моя программа напечатала:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Это означает, что, введя в браузере адрес *http://127.0.0.1:5000/*, я попаду на только что запущенный веб-сайт (рис. 11.13).

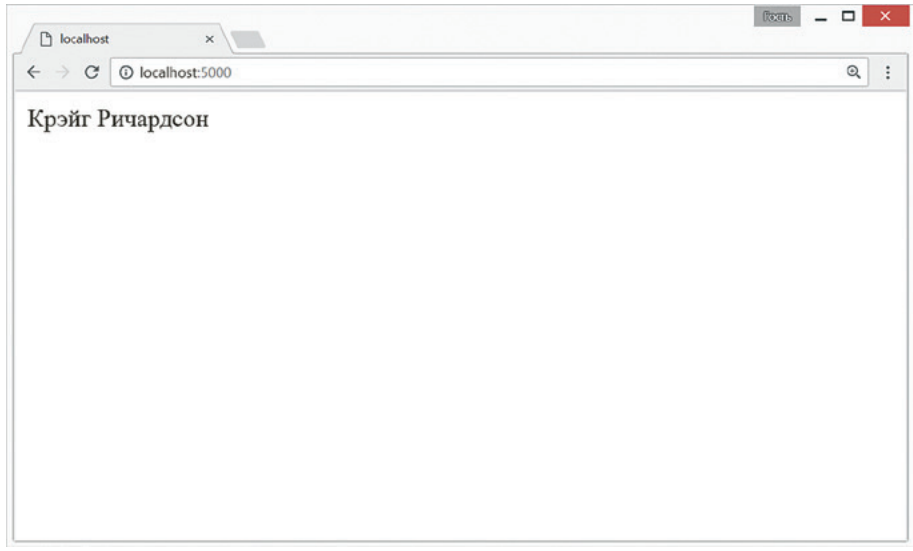

Рис. 11.13. Посмотрите на мой веб-сайт! Вы можете изменить надпись на любую другую

! *Веб-сайт, который создала эта программа, будет доступен только на вашем компьютере. Зайти на него сможете только вы, и из интернета он виден не будет.*

Чтобы остановить программу, перейдите в IDLE и нажмите CTRL + C или выберите в меню **Shell → Restart Shell**.

Restart shell — перезапустить оболочку

В этой главе мы лишь кратко знакомимся с модулем `Flask`*, который позволяет быстро создавать сайты на Python. Если вы хотите узнать об этом больше, прочитайте обучающую статью на английском языке на сайте Flask по ссылке http://flask.pocoo.org/docs/0.12/tutorial/.*

МИССИЯ 67. САЙТ С КООРДИНАТАМИ ИГРОКА

Важным свойством Python является то, что функции из разных модулей легко объединяются в одной программе. Вы уже работали с модулем Minecraft Python API, а теперь познакомились с `Flask`. А вы знали, что их можно использовать одновременно?

В этой миссии вам предстоит объединить Minecraft Python API и `Flask`, чтобы отобразить на веб-странице координаты игрока.

Создайте в IDLE новый файл и сохраните его под именем *positionPage.py* в папке *files*. Ваша задача — получить координаты игрока и отобразить их на главной странице веб-сайта с помощью функции, помеченной тегом `@app.route("/")`. В качестве основы программы можно взять код листинга 11.5. Выводите координаты в формате `"x 10, y 110, z 12"`.

Position page — страница с местоположением

Запустите программу и введите в браузер ссылку на веб-страницу. Здорово, правда? С помощью модуля `Flask` можно создавать сайты с самой разнообразной информацией. И даже загружать эти страницы в интернет, чтобы показать своим друзьям!

Что вы узнали

В этой главе вы узнали, как работать с файлами, научились записывать в них данные и считывать их оттуда с помощью функций стандартной библиотеки Python, так что теперь можете использовать файлы в своих программах. Также вы познакомились с модулями, которые расширили ваши возможности в программировании на языке Python.

Модуль `pickle` позволяет сохранять в файле содержимое одиночной переменной. Он особенно удобен для хранения многомерного списка или словаря, что было бы непросто сделать с помощью функций стандартной библиотеки Python. Модуль `shelve` обладает всеми преимуществами `pickle`, но позволяет хранить в файле несколько значений, используя для этого некое подобие словаря. Также вы узнали, как устанавливать в свои программы новые модули с помощью менеджера пакетов `pip`, и познакомились с `Flask` — модулем для быстрого создания веб-страниц.

Используя эти знания, вы выполнили четыре миссии. В первой создали перечень дел, который можно отобразить в чате Minecraft, чтобы напомнить самому себе о своих же планах. Во второй миссии вы научились сохранять конструкции из игры и воссоздавать их в этом же или других мирах. В третьей миссии использовали доработанные коды программ второй миссии, с помощью которых сохранили все данные конструкций в одном файле. И наконец, в последней миссии вы создали веб-страницу, отображающую текущие координаты игрока.

Вы достигли значительных успехов! Следующая глава будет последней. В ней вы изучите классы и освоите объектно ориентированное программирование — популярный метод, позволяющий использовать код повторно.

12

ОБЪЕКТНО ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ — ЭТО КЛАССНО!

Повторное использование кода позволяет сэкономить силы и время. Вы уже знаете этот способ и применяли его при работе с циклами и функциями, а теперь настало время познакомиться с объектно ориентированным программированием.

Объектно ориентированное программирование (ООП) — это особый подход к написанию программ, при котором функции и переменные объединяются в *классы*. На основе класса можно создать любое количество *объектов,* и для каждого из них будет повторно использоваться код переменных и функций, заданных в классе.

Функцию, которая является частью класса, называют *методом*, а переменную класса — *свойством*.

В этой главе вы познакомитесь с основами ООП и научитесь повторно использовать код с помощью классов. Навыки ООП выводят разработку программ на новый уровень, годится этот подход и для создания игр. В миссиях вам предстоит написать несколько несложных программ. Начнем с постройки обычного здания, а немного погодя создадим уже целый город.

Основы

ООП пользуется большой популярностью и подходит для разработки самых разных программ. Однако понять идею, лежащую в основе этой методологии, не так-то просто. Давайте сравним ее с чем-нибудь более привычным — например, с вами.

Вы человек. У вас есть набор функций, или методов: вы можете есть, дышать, спать, считать до 10 и делать множество других вещей. Помимо этого, у вас есть свойства: имя, возраст, рост, размер ноги и т. д.

У вашей подруги Маши такие же методы: она тоже ест, дышит, спит, считает до 10 и т. д. И свойства у нее те же (имя, возраст и т. д.), хотя значения этих свойств могут быть иными.

По сути, эти методы и свойства есть у всех людей. Можно сказать, что «люди» — это класс. А вы и Маша, конкретные люди, — объекты этого класса.

В ООП объекты еще называют *экземплярами*. Все экземпляры одного класса используют общие методы и свойства, и лишь значения свойств для разных объектов отличаются.

Давайте вернемся к языку Python и создадим класс.

Создание класса

Class — класс
Object — объект

Сначала следует объявить класс, а потом уже — различные его объекты. Для этого нужно ввести ключевое слово `class`, указать имя нового класса, а после него, в скобках, — `object` (я объясню, что это значит, в разделе «Наследование классов» на с. 332):

```
class ClassName(object):
    def __init__(self):
        # Тело метода __init__
```

Классам принято давать имена, начинающиеся с заглавной буквы. Так их проще отличать от функций, названия которых, как правило, начинаются со строчной.

Init (*сокр.* от initialization) — инициализация
Self — собственный

При создании нового класса в него нужно добавить метод `__init__()`, указав в качестве аргумента значение `self`. Аргумент `self` должен быть у каждого метода класса — это ссылка на объект, для которого данный метод вызывается. В методе `__init__()` вы пишете код, который выполняется при создании объекта, — *инициализации*.

Cat — кот

Создадим класс `Cat`, а затем — несколько объектов этого класса. Пусть у `Cat` будет два свойства — имя (name) и вес (weight). При этом каждый объект данного класса будет обладать своей парой значений этих свойств. Откройте окно программы IDLE, введите код следующего листинга и сохраните файл под именем *catClass.py* в новой папке *classes*.

Classes — классы

catClass.py

```
class Cat(object):
❶    def __init__(self, name, weight):
❷        self.name = name
❸        self.weight = weight
```

Метод `__init__()` в этом коде принимает три аргумента ❶. Первый — `self` — обязательный для всех методов класса. А второй и третий (`name` и `weight`) — дополнительные и используются для задания свойств каждого нового объекта.

В последних строках мы должны объявить свойства `name` ❷ и `weight` ❸, присвоив им значения одноименных аргументов. Для этого обратимся к точечной нотации. Свойства всегда задаются через `self`: тогда Python понимает, что они принадлежат классу.

А теперь давайте научимся создавать объекты класса.

Создание объектов

Инициализация объекта напоминает объявление переменной. При инициализации нужно ввести имя нового объекта, знак равно и имя класса, после которого в скобках указать аргументы, как при вызове функции.

Представим, что мы завели кота и назвали его Пушок. Чтобы создать объект класса `Cat` с именем `Пушок`, добавим в конец программы *catClass.py* такую строку кода (обратите внимание, что отступа в начале нет).

catClass.py

```
class Cat(object):
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

fluff = Cat("Пушок", 4.5)
```

Количество аргументов, которые мы передаем при вызове класса и создании объекта, зависит от метода `__init__()` данного класса. Мы передали `__init__()` два аргумента: один содержит имя `(Пушок)`, другой — вес `(4.5)`. Аргумент `self` передавать не нужно — Python добавляет его автоматически.

При создании объекта вызывается *конструктор*. Называется он так потому, что собирает объект, словно из деталей конструктора. В наших примерах конструктором является специальный метод `__init__()` — к нему не нужно обращаться, он автоматически

Fluff — Пушок

вызывается при создании каждого объекта данного класса. Например, код `fluff = Cat("Пушок", 4.5)` вызывает метод `__init__()`, который создает объект класса `Cat` и помещает его в переменную `fluff`.

Далее вы узнаете, как получить доступ к свойствам объекта `fluff`.

Доступ к свойствам

Чтобы получить информацию об объекте, можно обратиться к его свойствам. Выведем на экран значение свойства `weight` объекта `fluff`, добавив в конец программы *catClass.py* следующий код.

catClass.py

```
print(fluff.weight)
```

После запуска программа должна напечатать число `4.5`, поскольку это значение свойства `weight` было указано при создании объекта. Обратите внимание: мы пишем имя объекта — `fluff`, а через точку — имя свойства — `weight`. Точечная нотация необходима для обращения к свойствам объекта.

Значение свойства можно изменить так же, как значение обычной переменной, — с помощью знака «равно» (=). Давайте обновим информацию о массе нашего Пушка, который отъелся за праздники и теперь весит 5 килограммов. Присвоим свойству `weight` объекта `fluff` значение 5.

catClass.py

```
fluff.weight = 5
```

Теперь каждый раз, когда вы будете обращаться к свойству `weight` объекта `fluff`, оно будет содержать значение 5.

А сейчас пора воспользоваться новыми знаниями о классах и объектах, чтобы сделать в мире Minecraft что-нибудь интересненькое!

МИССИЯ 68. ОБЪЕКТЫ-МЕСТА

При написании программ, предложенных в этой книге, вам уже доводилось сохранять координаты разных областей игрового мира: дома, замка, дворца или чего-то еще. Тогда вы использовали переменные, списки, кортежи и словари.

Работать с информацией такого рода позволяет и ООП. В частности, вы можете использовать для хранения координат объекты.

Всё в мире Minecraft имеет координаты *x*, *y* и *z*, но значения этих координат для каждого места разные. Создав класс `Location`, вы сможете хранить значения *x*, *y* и *z* в объектах этого класса. Так будет легче присматривать за своими чудесными сооружениями, ведь вы всегда сможете получить доступ к их координатам, чтобы телепортировать туда игрока.

Location — местоположение

В листинге 12.1 показана основа кода для создания класса `Location`. Введите его в новый файл IDLE и сохраните под именем *locationClass.py* в папке *classes*.

locationClass.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

❶ class Location(object):
      def __init__(self, x, y, z):
❷         self.x = x
❸         # Объявите здесь свойства y и z

❹ bedroom = Location(64, 52, -8)
❺ mc.player.setTilePos(bedroom.x, bedroom.y, bedroom.z)
```

Листинг 12.1. Недописанный код для создания класса Location

Для создания класса я использовал ключевое слово `class`, далее указал его имя `Location` ❶. В строке ❹ находится команда инициализации объекта `bedroom`, в котором хранятся координаты спальни в моем мире Minecraft. Далее с помощью метода `setTilePos()` игрок телепортируется в спальню, то есть в место с координатами, равными значениям свойств `x`, `y` и `z` объекта `bedroom` ❺. Однако программа недописана. Вам предстоит закончить код метода `__init__()` и объявить свойства `y` и `z`, присвоив им значения, которые переданы в аргументы метода `__init__()`. Я уже написал команду для свойства `x` ❷, а добавить такие же для `y` и `z` — ваша задача ❸. Не забудьте указать в строке ❹ координаты собственной спальни!

Bedroom — спальня

На рис. 12.1 видно, что программа перенесла моего игрока в спальню.

БОНУСНОЕ ЗАДАНИЕ: ДОМ, МИЛЫЙ ДОМ

В какие еще комнаты вы хотели бы телепортировать игрока? Чтобы мгновенно перемещать его по дому, куда вам захочется, создайте другие объекты класса Location.

Рис. 12.1. Программа телепортировала моего игрока в спальню

Что такое методы

Класс содержит методы — функции, к которым имеют доступ все объекты. Это позволяет использовать код повторно, ведь единожды созданный метод работает для всех объектов класса.

Чтобы создать метод, *определите* его с помощью ключевого слова `def` — так же, как вы создавали функции раньше, только теперь в теле класса. Давайте изменим класс `Cat` из программы *catClass.py*, так чтобы кот мог есть. Для этого создадим метод класса с именем `eat()`. Добавьте в программу следующий код (после серых строк).

Eat — ест

catClass.py

```
class Cat(object):
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def eat(self, food):
        self.weight = self.weight + 0.05
        print(self.name + " ест " + food)
```

Обратите внимание: определение метода `eat()` вводится с отступом четыре пробела, а его тело — с отступом восемь пробелов, чтобы Python понял: этот метод относится к классу.

Food — еда

Подобно функциям, методы могут принимать аргументы. В данном случае метод `eat()` принимает аргумент `food`, в котором содержится

информация, что именно ест кот. Метод `eat()` увеличивает вес животного (свойство `weight`) на 0,05 кг и выводит на экран сообщение, что кот ест.

После создания объекта можно вызвать любой метод, который задан для его класса. Например, метод `eat()` объекта `fluff`. Для этого добавьте в конец программы *catClass.py* следующий код.

catClass.py

```
fluff = Cat("Пушок", 4.5)
fluff.eat("рыбу")
```

Серым цветом показана инициализация объекта `fluff`. Далее мы вызываем метод `eat()`, присваивая ему аргумент `"рыбу"`. После запуска программы на экране появится:

```
Пушок ест рыбу
```

Итак, Пушок доволен — он ест рыбу. Напоминаю, что метод `eat()` также увеличивает значение свойства `weight`. Добавьте в программу после вызова `eat()` вывод этого свойства на экран.

Eat and sleep — ест и спит

Также методы можно вызывать из других методов. Давайте создадим в классе `Cat` еще один метод — `eatAndSleep()`. Он будет вызывать метод `eat()`, а затем выводить сообщение, что кот теперь спит. Добавьте следующий код в *catClass.py* сразу после вызова `eat()` (не забудьте про отступы, чтобы новый метод попал в тело класса).

catClass.py

```
    def eatAndSleep(self, food):
        self.eat(food)
        print(self.name + " теперь спит…")
```

Чтобы вызвать метод изнутри класса, которому он принадлежит, перед его именем нужно указать `self`. В данном случае вызов выглядит так: `self.eat()`. Обратите внимание, что снаружи класса методы вызываются иначе — с указанием имени объекта.

Добавьте в код *catClass.py* вызов метода `eatAndSleep()` объекта `fluff` (это должна быть последняя строка программы).

catClass.py

```
fluff.eatAndSleep("рыбу")
```

После запуска программы вы увидите:

```
Пушок ест рыбу
Пушок теперь спит…
```

Вот программа целиком. Проверьте, находится ли каждая часть вашего кода на своем месте:

```
class Cat(object):
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def eat(self, food):
        self.weight = self.weight + 0.05
        print(self.name + " ест " + food)

    def eatAndSleep(self, food):
        self.eat(food)
        print(self.name + " теперь спит…")

fluff = Cat("Пушок", 4.5)
print(fluff.weight)
fluff.eat("рыбу")
fluff.eatAndSleep("рыбу")
```

А теперь используем новые знания на практике!

МИССИЯ 69. ДОМ-ПРИЗРАК

Самое приятное в программировании для Minecraft на Python — возможность придумать нечто необычное и начать писать для этого код. Сначала некоторые задумки кажутся бесполезными, но, написав несколько строк кода, вы можете получить интересный результат.

Однажды я подумал, что было бы забавно создать в игре дом-призрак, который то появляется, то снова исчезает. Каждые 30 секунд дом мог бы возникать в каком-нибудь новом месте.

Ghost house — дом-призрак

Вот первая версия программы для создания дома-призрака. Введите в новый файл IDLE код листинга 12.2 и сохраните его под именем *ghostHouse.py* в папке *classes*.

ghostHouse.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
```

```
import time

❶ class Building(object):
❷     def __init__(self, x, y, z, width, height, depth):
          self.x = x
          self.y = y
          self.z = z

          self.width = width
          self.height = height
          self.depth = depth

❸     def build(self):
          mc.setBlocks(self.x, self.y, self.z,
                       self.x + self.width, self.y + self.height,
                       self.z + self.depth, 4)

          mc.setBlocks(self.x + 1, self.y + 1, self.z + 1,
                       self.x + self.width - 1, self.y + self.height - 1,
                       self.z + self.depth - 1, 0)
❹         # Определите методы buildDoor() и buildWindows()

❺     def clear(self):
          mc.setBlocks(self.x, self.y, self.z,
                       self.x + self.width, self.y + self.height,
                       self.z + self.depth, 0)

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

❻ ghostHouse = Building(x, y, z, 10, 6, 8)
ghostHouse.build()

time.sleep(30)

ghostHouse.clear()
❼ ghostHouse.x = 8
```

Листинг 12.2. Класс Building создает дом

В коде листинга 12.2 создается класс `Building` ❶ с методом `__init__()`, который позволяет задать координаты и размеры дома ❷. В строке ❻ создается объект класса `Building` с именем `ghostHouse`. Дом появляется в игре при вызове метода `build()` ❸, а затем, через 30 секунд, таинственно исчезает благодаря вызову метода `clear()` ❺. Проблема лишь в одном: на дом эта постройка не слишком похожа. Сейчас она напоминает скорее огромную каменную коробку.

Building — здание

Build — построить

Clear — очистить

Build door — сделать дверь

Build windows — сделать окна

Ваша задача — сделать постройку похожей на дом, ведь коробки-призраки не так страшны, как дома-призраки. Для этого определите метод `buildDoor()`, который создает проем в передней стене здания, и метод `buildWindows()`, добавляющий окна. Вызывайте их из метода `build()`, чтобы все части дома создавались одновременно ❹.

Закончив с методами, переместите дом на новое место, изменив его координаты *x*, *y* и *z* и добавив в код еще пару вызовов `build()` и `clear()`. Я уже начал это делать, поменяв значение `x` ❼.

После запуска программы дом-призрак должен внезапно появиться, а через 30 секунд — исчезнуть и возникнуть уже в новом месте. Выглядит жутковато!

На рис. 12.2 показан мой дом-призрак.

Рис. 12.2. Дом-призрак то появляется, то исчезает

БОНУСНОЕ ЗАДАНИЕ: НАСТОЯЩИЙ ДОМ

Сейчас дом-призрак выглядит довольно примитивно и мало похож на настоящий дом. С помощью полученных знаний измените код метода `build()`, чтобы дом выглядел интереснее.

Методы, возвращающие значение

Подобно функциям, методы могут возвращать значения (в том числе значения свойств объекта) с помощью команды `return`. Допустим, мы хотим знать: сколько весит Пушок не в килограммах, а в граммах. В килограмме 1000 граммов, а значит, чтобы получить ответ, нужно умножить значение свойства `weight` на 1000. В программе *catClass.py* добавьте в класс `Cat` метод `getWeightInGrams()`.

Get weight in grams — получить вес в граммах

catClass.py

```
class Cat(object):
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def getWeightInGrams(self):
        return self.weight * 1000
```

Чтобы вывести на экран значение, которое возвращает метод `getWeightInGrams()`, нужно создать объект и затем вызвать этот метод. В следующем примере мы используем объект `fluff` в качестве аргумента функции `print`, чтобы программа вывела на экран вес кота в граммах.

catClass.py

```
fluff = Cat("Пушок", 4.5)
print(fluff.getWeightInGrams())
```

Теперь программа после запуска должна вывести:

```
4500
```

В следующей миссии мы доработаем программу с домом-призраком, определив метод, возвращающий информацию о здании.

МИССИЯ 70. ЗАМОК-ПРИЗРАК

Я возвел в своем мире Minecraft немало построек, и у каждой есть название: пляжный домик, ферма, склад, дворец, подводный дворец, подземный дворец и т. д. Проблема в одном: эти названия хранятся лишь в моей голове!

Как вы знаете из миссии 69 (с. 320), в состав класса могут входить самые разные свойства — например, координаты и размер здания. Для названия тоже можно завести отдельное свойство!

Давайте дадим дому-призраку имя, и пусть Python его запомнит. Мы изменим класс `Building` из программы *ghostHouse.py*, добавив в него еще один метод, который будет возвращать название дома. Введите код листинга 12.3 в новый файл IDLE и сохраните его под именем *ghostCastle.py* в папке *classes*.

Ghost castle — замок-призрак

ghostCastle.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

❶ class NamedBuilding(object):
❷     def __init__(self, x, y, z, width, height, depth, name):
          self.x = x
          self.y = y
          self.z = z

          self.width = width
          self.height = height
          self.depth = depth

❸         self.name = name

      def build(self):
          mc.setBlocks(self.x, self.y, self.z,
                       self.x + self.width, self.y + self.height,
                       self.z + self.depth, 4)

          mc.setBlocks(self.x + 1, self.y + 1, self.z + 1,
                       self.x + self.width - 1, self.y + self.height - 1,
                       self.z + self.depth - 1, 0)

      def clear(self):
          mc.setBlocks(self.x, self.y, self.z,
                       self.x + self.width, self.y + self.height,
                       self.z + self.depth, 0)

❹     def getInfo():
          # Добавьте сюда тело метода getInfo()
```

```
pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
ghostCastle = NamedBuilding(x, y, z, 10, 16, 16, "Замок-призрак")
ghostCastle.build()
❺ mc.postToChat(ghostCastle.getInfo())

time.sleep(30)

ghostCastle.clear()
```

Листинг 12.3. Класс NamedBuilding похож на класс Building, но имеет дополнительное свойство для названия постройки и метод, который возвращает информацию о ней

Named building — здание с названием

Get info — получить информацию

Первым делом я поменял имя класса на `NamedBuilding`, чтобы не путать его с классом `Building` из предыдущей миссии ❶. В конструкторе я добавил еще один аргумент и новое свойство `name` ❷. Дополнительный аргумент позволит присвоить имя объекту при его создании, после чего конструктор сохранит это имя в свойстве `name` ❸.

В этой миссии вам предстоит добавить в класс `NamedBuilding` метод `getInfo()`, который возвращает название и координаты здания. Я уже ввел определение этого метода ❹ — вам осталось лишь написать его код. Чтобы вывести в чат Minecraft строку, которую возвращает метод `getInfo()`, в строке ❺ я вызываю этот метод для объекта `ghostCastle`. Если координаты замка (134, 66, 1250), метод `getInfo()` должен вернуть строку `"Замок-призрак имеет координаты 134, 66, 250"`.

На рис. 12.3 показан результат работы моей программы. Замок-призрак выше дома-призрака из миссии 69, но в остальном очень на него похож, поскольку код метода `build()` для обоих зданий одинаков. Однако вы можете изменить программу так, чтобы ваша постройка больше напоминала замок.

БОНУСНОЕ ЗАДАНИЕ: ТЕПЛЫЙ ПРИЕМ

Было бы здорово видеть в чате описание каждого здания, куда заходит ваш игрок, не правда ли? Что ж, сделать это возможно, хотя и не слишком просто. В качестве отправной точки можно взять код *shower.py* (папка *ifStatements*) из миссии 32 (с. 158). Пусть программа определяет координаты игрока и, если он находится в здании, вызывает метод `getInfo()`.

Рис. 12.3. Описание замка-призрака в чате

Создание нескольких объектов

Можно создать несколько объектов одного класса, указывая разные имена при вызове конструктора (напоминаю: конструктором называется метод `__init__()`). Представьте, что мы подобрали кошку по имени Стелла и теперь они с Пушком дружат. Откройте программу *catClass.py* и введите следующий код, чтобы создать объект `stella`.

catClass.py

```
class Cat(object):
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

fluff = Cat("Пушок", 4.5)
stella = Cat("Стелла", 3.9)
```

Теперь у нас два объекта — `fluff` и `stella`. Они обладают одним и тем же набором свойств, но значения этих свойств разные.

Добавьте в *catClass.py* следующий код, выводящий имена наших питомцев.

catClass.py

```
print(fluff.name)
print(stella.name)
```

Запустив программу, вы увидите:

```
Пушок
Стелла
```

Оба этих объекта имеют доступ к одним и тем же методам, а значит, оба могут использовать метод `eat()`. Добавьте в *catClass.py* следующий код.

catClass.py

```
fluff.eat("рыбу")
stella.eat("торт")
```

В результате программа выведет:

```
Пушок ест рыбу
Стелла ест торт
```

Написав код для создания класса, совсем несложно создать множество объектов, относящихся к этому классу. Давайте попробуем!

МИССИЯ 71. ПОСЕЛОК-ПРИЗРАК

Что может быть страшнее дома-призрака? Правильно, два дома-призрака! Но ведь три дома-призрака еще страшнее! А если их будет много? Надо перестать об этом думать, иначе я ночью не усну.

В ходе миссии 69 (с. 320) вы написали код, создающий класс для постройки исчезающего дома. Теперь ваша задача — создать несколько объектов этого класса. Python запомнит все их свойства и методы, избавив от необходимости каждый раз заново вводить данные. Вы можете создать столько домов, сколько захотите, и с легкостью заставить их то появляться, то исчезать.

Итак, в этой миссии вам необходимо создать не менее четырех домов-призраков. Пусть через заданное количество секунд все дома исчезнут, а затем появятся в новом месте — получится настоящий поселок-призрак!

Откройте программу *ghostHouse.py* в IDLE — мы возьмем ее код за основу. Одно здание создается примерно так.

ghostHouse.py

```
ghostHouse = Building(17, 22, -4, 10, 6, 8)
ghostHouse.build()

time.sleep(30)

ghostHouse.clear()
```

Ghost village — поселок-призрак

Shop — магазин

Сохраните программу под новым именем *ghostVillage.py*, а затем создайте несколько объектов класса `Building`, чтобы получился поселок. В листинге 12.4 я уже добавил второе здание, назвав его `shop`. Также я объявил переменные `x`, `y` и `z`, в которых хранятся координаты игрока, полученные с помощью вызова функции `player.getTilePos()`, — так вам будет проще возвести дома вокруг игрока.

ghostVillage.py

```
pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
ghostHouse = Building(x, y, z, 10, 6, 8)
shop = Building(x + 12, y, z, 8, 12, 10)
# Создайте еще больше объектов-зданий

ghostHouse.build()
shop.build()
# Постройте здания, вызывая метод build()

time.sleep(30)

ghostHouse.clear()
shop.clear()
```

Листинг 12.4. Основа кода для постройки нескольких домов поселка-призрака

На рис. 12.4 показан мой поселок-призрак. Через 30 секунд после запуска программы все здания внезапно исчезают.

Свойства класса

Иногда нужно присвоить свойству значение, одинаковое для всех объектов класса. Указывать это значение при создании каждого объекта слишком неудобно, однако вы можете сделать свойство *предопределенным* — тогда все объекты данного класса смогут его использовать.

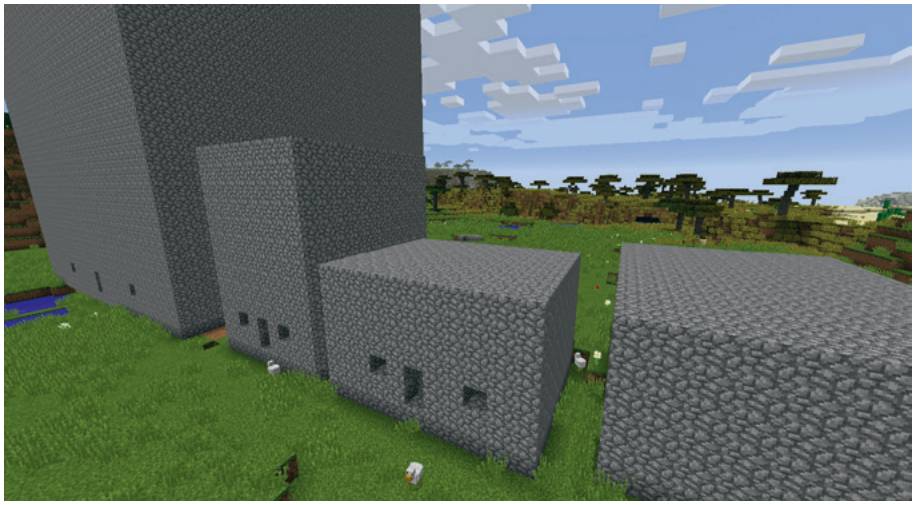

Рис. 12.4. Призрачные здания призрачного поселка!

Когда множество объектов одного класса используют общее значение свойства, его называют *свойством класса*. Допустим, владелец всех наших объектов-котов — Крэйг (это я). Можно дополнить класс `Cat` из программы *catClass.py* свойством с именем `owner`, присвоив ему значение `Крэйг`.

Owner — владелец

catClass.py

```
class Cat(object):
    owner = "Крэйг"

    def __init__(self, name, weight):
        self.name = name
        self.weight = weight
```

Как видите, слово `self` перед именем свойства класса не указывается. В данном примере `owner` — свойство класса, а `self.name` — обычное свойство. Кроме того, свойство класса нужно объявлять вне метода `__init__()`.

Обращаться к свойствам класса можно так же, как и к простым свойствам. Чтобы узнать, кто владелец Пушка, выведем на экран значение `owner`, словно это обычное свойство объекта `fluff`.

catClass.py

```
fluff = Cat("Пушок", 4.5)
print(fluff.owner)
```

Программа должна напечатать "Крэйг". Сделав то же самое для объекта stella, мы получим такой же результат, ведь значения свойств класса одинаковы для всех принадлежащих этому классу объектов.

catClass.py

```
stella = Cat("Стелла", 3.9)
print(stella.owner)
```

Этот код также выведет "Крэйг".

Однако для конкретных объектов значение свойства класса можно изменять, причем на остальные объекты класса это никак не повлияет. Предположим, мой друг Мэтью забрал Стеллу себе, и теперь имя владельца нужно исправить.

catClass.py

```
stella.owner = "Мэтью"
print(stella.owner)
print(fluff.owner)
```

Когда мы выведем значение свойства owner объекта stella, то увидим на экране "Мэтью", но для fluff значение owner останется прежним — "Крэйг".

После внесения всех этих изменений программа *catClass.py* должна выглядеть так.

catClass.py

```
class Cat(object):
    owner = "Крэйг"

    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def eat(self, food):
        self.weight = self.weight + 0.05
        print(self.name + " ест " + food)

    def eatAndSleep(self, food):
        self.eat(food)
        print(self.name + " теперь спит…")

    def getWeightInGrams(self):
        return self.weight * 1000
```

```
fluff = Cat("Пушок", 4.5)
print(fluff.owner)
stella = Cat("Стелла", 3.9)
print(stella.owner)

print(fluff.weight)
fluff.eat("рыбу")
fluff.eatAndSleep("рыбу")

print(fluff.getWeightInGrams())
print(fluff.name)
print(stella.name)

fluff.eat("рыбу")
stella.eat("торт")

stella.owner = "Мэтью"
print(stella.owner)
print(fluff.owner)
```

Итак, вы уже знаете, для чего можно использовать объекты, а теперь давайте расширим их возможности с помощью наследования.

Наследование

Наследование позволяет одним классам делить методы и свойства с другими классами. К примеру, утки — это семейство птиц. У них должны быть такие же методы, как у других птиц (они могут летать, есть и т. д.), и такой же набор свойств (вес, размах крыльев…). Поэтому можно сказать, что класс «утки» наследует методы и свойства класса «птицы». Взаимосвязь этих классов показана на рис. 12.5.

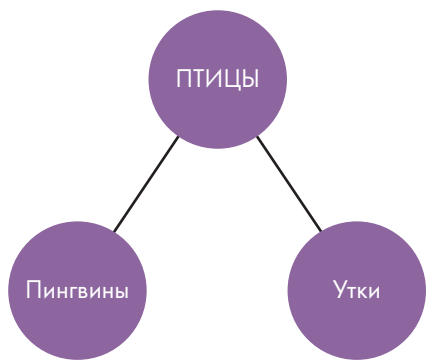

Рис. 12.5. И пингвины, и утки — разновидности птиц

Класс, от которого другие классы наследуют свойства и методы, называется *базовым*. А класс, наследующий от базового, — *производным*.

Наследование позволяет использовать общий код для похожих, но все же имеющих различия классов. Пингвины — тоже семейство птиц, однако не все птицы могут плавать под водой, как пингвины. Класс «пингвины» — производный от класса «птицы», но с дополнением, позволяющим пингвинам плавать. Так и создаются производные классы — вы используете ключевые особенности базового класса, чтобы не писать код заново, а затем добавляете необходимые для производного класса методы и свойства.

Наследование классов

Когда производный класс наследует от базового, он может использовать все его методы и свойства, а также добавлять свои, не требуя изменений в коде базового класса.

Давайте покажем это на примере птиц и начнем с базового класса `Bird`. Создайте в IDLE новый файл *birdClass.py* и введите следующий код.

Bird — птица

birdClass.py

```
❶ class Bird(object):
❷     def __init__(self, name, wingspan):
          self.name = name
          self.wingspan = wingspan

❸     def birdcall(self):
          print("чик-чирик")

❹     def fly(self):
          print("хлоп-хлоп")
```

Мы создали базовый класс `Bird` ❶, который — обратите внимание — унаследован от класса `object`. Класс `object` является главным базовым классом, от которого происходят все остальные. Таким образом, даже если у вас несколько уровней наследования, на самом верхнем уровне обязательно должен быть `object`. Используйте его как базовый, если не хотите наследовать свой класс ни от какого другого.

Метод `__init__()` класса `Bird` принимает два аргумента для задания двух свойств: имени птицы (`name`) и размаха ее крыльев (`wingspan`) ❷. Также в классе есть два метода: `birdcall()` ❸ и `fly()` ❹. Сейчас эти методы просто выводят сообщения с характерными для птиц звуками.

Birdcall — птичий щебет

Fly — летает

Garden bird — парковая птица

Создайте в той же программе объект класса `Bird` — `gardenBird`.

birdClass.py

```
gardenBird = Bird("Джеффри", 12)
gardenBird.birdcall()
gardenBird.fly()
```

Этот код выведет:

```
чик-чирик
хлоп-хлоп
```

Теперь, когда у нас есть базовый класс, можно создать производный, который получит все методы и свойства базового, а также будет обладать собственными методами. Этим и займемся.

Добавление новых методов в производный класс

Давайте создадим в программе *birdClass.py* производный от класса `Bird` класс для пингвинов — назовем его `Penguin`. Поскольку пингвины умеют плавать под водой, добавим в `Penguin` метод `swim()`:

Penguin — пингвин

Swim — плавает

birdClass.py

```
class Penguin(Bird):
    def swim(self):
        print("умеет плавать")
```

Чтобы создать класс, наследующий не от `object`, а от базового класса `Bird`, укажите имя этого базового класса в скобках. Обратите внимание, что я не стал создавать в классе `Penguin` метод `__init__()`. Дело в том, что этот метод наследуется от класса `Bird`. Давайте воспользуемся методом `__init__()` базового класса, чтобы создать пингвина Сару, а затем проверим работу метода `swim()`.

birdClass.py

```
sarahThePenguin = Penguin("Сара", 10)
sarahThePenguin.swim()
```

Этот код выведет:

```
умеет плавать
```

Также объекты класса `Penguin` могут вызывать методы `fly()` и `birdcall()`, унаследованные от класса `Bird`.

birdClass.py

```
sarahThePenguin.fly()
sarahThePenguin.birdcall()
```

Программа выведет:

```
хлоп-хлоп
чик-чирик
```

Однако хлопанье крыльев (`хлоп-хлоп`) и чириканье (`чирик-чирик`) не подходят для пингвинов — ведь эти птицы не летают, а их голоса больше похожи на кряканье. Дойдя до раздела «Переопределение методов и свойств» на с. 336, вы узнаете, как исправить это несоответствие, переопределив унаследованные методы.

Однако сначала давайте вернемся в Minecraft и создадим еще несколько зданий-призраков, применив для этого прием наследования.

МИССИЯ 72. ГОСТИНИЦА-ПРИЗРАК

И дома, и гостиницы являются зданиями: в них есть двери, окна, лестницы и стены. В сущности, гостиницы — всего лишь красивые дома со множеством уютных комнат и ухоженной территорией.

Можно ли создать гостиницу-призрак, взяв за основу код, написанный для дома-призрака? Базовые особенности и там и там одинаковые, поэтому будем считать, что главное отличие кода гостиницы-призрака от кода дома-призрака — в дополнительных методах, которые мы используем, чтобы создать ковры в комнатах и добавить цветники по периметру здания. То есть класс гостиницы-призрака можно унаследовать от класса дома-призрака — останется лишь добавить пару методов для цветов и ковров.

Ghost hotel — гостиница-призрак

Fancy building — роскошное здание

Upgrade — облагородить

Создайте в IDLE новый файл и сохраните его под именем *ghostHotel.py* в папке *classes*. Скопируйте в него код класса `Building` из программы *ghostHouse.py*.

Создайте новый класс `FancyBuilding`, который наследует от класса `Building`. В классе `FancyBuilding` должен быть метод `upgrade()`, размещающий внутри здания ковры, а снаружи — цветы.

В листинге 12.5 показана моя версия кода, хотя вы можете обустроить свою гостиницу иначе.

ghostHotel.py

```
# Создайте здесь класс FancyBuilding

    def upgrade(self):
        # Ковер
        mc.setBlocks(self.x + 1, self.y, self.z + 1,
                     self.x + self.width - 1, self.y,
                     self.z + self.depth - 1, 35, 6)

        # Цветы
        mc.setBlocks(self.x - 1, self.y, self.z - 1, self.x - 1,
                     self.y, self.z + self.depth + 1,
                     37)
        mc.setBlocks(self.x - 1, self.y, self.z - 1,
                     self.x + self.width + 1, self.y, self.z - 1,
                     37)
        mc.setBlocks(self.x + self.width + 1, self.y, self.z - 1,
                     self.x + self.width + 1, self.y,
                     self.z + self.depth + 1,
                     37)
        mc.setBlocks(self.x - 1, self.y, self.z + self.depth + 1,
                     self.x + self.width + 1, self.y,
                     self.z + self.depth = 1,
                     37)

# Создайте объект класса FancyBuilding
# Вызовите методы build() и upgrade()
```

Листинг 12.5. Метод класса FancyBuilding добавляет ковер и цветы

Объявив класс `FancyBuilding` и добавив новый метод, создайте объект `ghostHotel`. Постройте гостиницу, определив метод `build()`, а затем добавьте характерные для гостиницы детали, определив метод `upgrade()`.

На рис. 12.6 показана моя роскошная гостиница-призрак.

БОНУСНОЕ ЗАДАНИЕ: НАСТОЯЩИЙ ПОСЕЛОК

В ходе миссии 71 вы создали поселок-призрак, все здания которого выглядят одинаково. В жизни такое встречается редко. Измените программу *ghostVillage.py*, добавив в нее несколько классов, которые будут наследовать от `Building`. Это могут быть классы `Shop`, `Hospital`, `Restaurant`.

Shop — магазин

Hospital — больница

Restaurant — ресторан

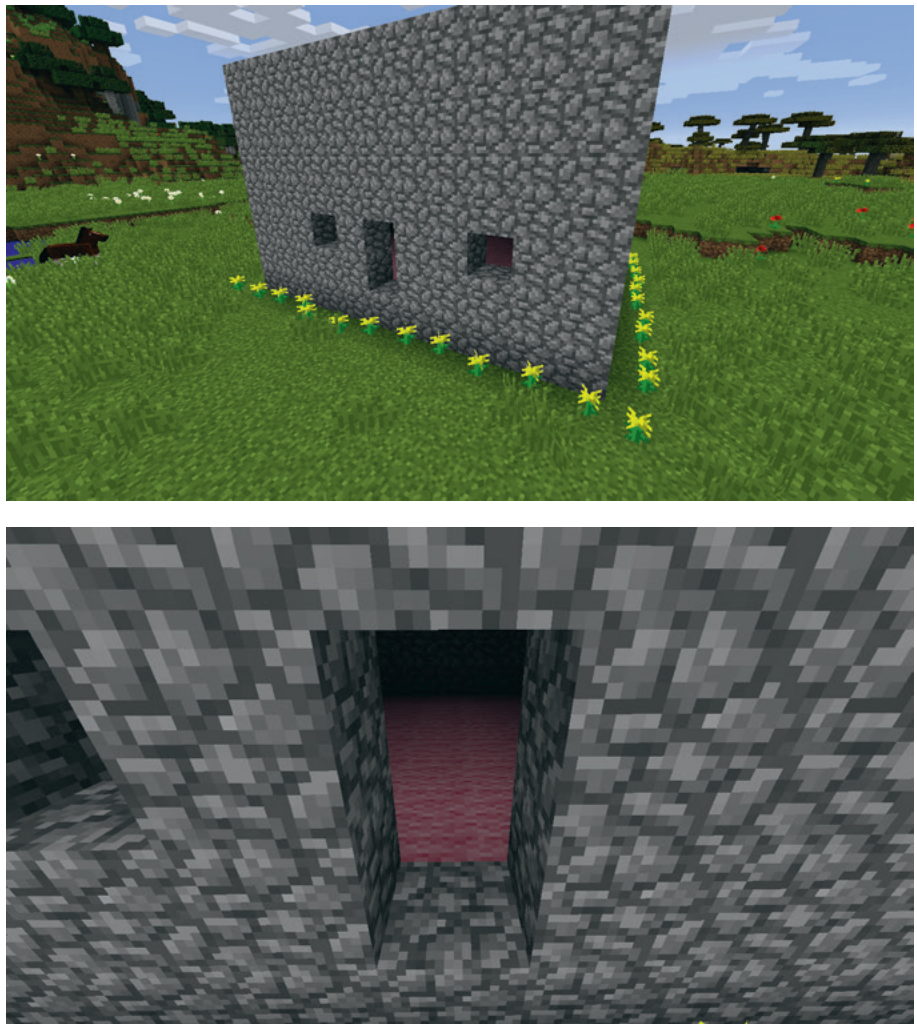

Рис. 12.6. Взгляните на эти цветы и ковры!

Переопределение методов и свойств

В производном классе можно *переопределять* методы и свойства базового класса. При этом метод производного класса будет называться так же, как и метод базового, но вести себя он может по-другому.

В разделе «Наследование» на с. 331 мы создали классы `Bird` и `Penguin`. Класс `Penguin` наследует от `Bird`, получая все его методы. Однако пингвины не умеют летать, и голос их похож на кряканье больше, чем на чириканье, а значит, следует изменить поведение методов

`fly()` и `birdcall()`. Откройте программу *birdClass.py* и добавьте в нее следующий код.

birdClass.py

```
class Penguin(Bird):
    def swim(self):
        print("умеет плавать")

❶   def birdcall(self):
        print("что-то вроде кря-кря")

❷   def fly(self):
        print("Пингвины не летают :(")
```

Я добавил в класс `Penguin` методы `birdcall()` ❶ и `fly()` ❷. Поскольку имена этих методов совпадают с именами методов базового класса `Bird`, произойдет переопределение методов базового класса.

Вызовем эти методы, добавив в *birdClass.py* следующий код:

birdClass.py

```
sarahThePenguin.fly()
sarahThePenguin.birdcall()
```

Теперь, запустив программу, вы увидите:

```
Пингвины не летают :(
что-то вроде кря-кря
```

Переопределение метода базового класса меняет поведение этого метода для объектов производного класса, но не для объектов базового класса. Поэтому пингвины в нашей программе летать не будут, но другие птицы, которые наследуют от `Bird`, сохранят эту способность.

Также в производном классе можно переопределить метод `__init__()`. Это значит, что при создании объектов производного класса будут выполняться не те действия и задаваться не те свойства, что были у объектов базового класса.

Давайте создадим в программе *birdClass.py* класс `Parrot`, наследующий от `Bird`. Попугаи бывают разных цветов, поэтому добавим в метод `__init__()` еще один аргумент с названием цвета.

Parrot — попугай

birdClass.py

```
class Parrot(Bird):
❶    def __init__(self, name, wingspan, color):
        self.name = name
        self.wingspan = wingspan
        self.color = color
```

Я определил в классе `Parrot` новый метод `__init__()`, который отличается от аналогичного метода в классе `Bird` дополнительным аргументом `color` ❶.

Color — цвет

Теперь, создав новый объект класса `Parrot`, мы получим доступ к свойству `color`. Также мы можем вызывать методы `birdcall()` и `fly()`, поскольку они унаследованы от базового класса `Bird`.

birdClass.py

```
freddieTheParrot = Parrot("Фредди", 12, "синий")
print(freddieTheParrot.color)
freddieTheParrot.fly()
freddieTheParrot.birdcall()
```

Этот код выведет:

```
синий
хлоп-хлоп
чик-чирик
```

Запомните: можно переопределить любой унаследованный от базового класса метод, даже метод `__init__()`. Это дает большу́ю свободу при управлении объектами, их методами и свойствами.

После всех изменений, которые мы внесли в *birdClass.py*, программа должна выглядеть так.

birdClass.py

```
class Bird(object):
    def __init__(self, name, wingspan):
        self.name = name
        self.wingspan = wingspan

    def birdcall(self):
        print("чик-чирик")

    def fly(self):
```

```
        print("хлоп-хлоп")


class Penguin(Bird):
    def swim(self):
        print("умеет плавать")

    def birdcall(self):
        print("что-то вроде кря-кря")

    def fly(self):
        print("Пингвины не летают :(")


class Parrot(Bird):
    def __init__(self, name, wingspan, color):
        self.name = name
        self.wingspan = wingspan
        self.color = color

gardenBird = Bird("Джеффри", 12)
gardenBird.birdcall()
gardenBird.fly()

sarahThePenguin = Penguin("Сара", 10)
sarahThePenguin.swim()
sarahThePenguin.fly()
sarahThePenguin.birdcall()

freddieTheParrot = Parrot("Фредди", 12, "синий")
print(freddieTheParrot.color)
freddieTheParrot.fly()
freddieTheParrot.birdcall()
```

В следующей миссии вам предстоит поупражняться в переопределении методов и свойств класса.

МИССИЯ 73. ДЕРЕВО-ПРИЗРАК

Вы построили уже немало призрачных зданий — настало время поднять мастерство на новый уровень и создать дерево-призрак. Прекрасная идея, но как ее воплотить? Класс `Building` предназначен для зданий с полом и потолком, а у деревьев пола и потолка, как правило, нет. Однако не унывайте! Нашему горю легко помочь, переопределив методы базового класса `Building`.

Как и призрачные здания, дерево-призрак будет появляться и исчезать при вызове методов `build()` и `clear()`, но работать эти методы будут иначе, ведь дерево не похоже на дом. Итак, вам предстоит

создать класс, наследующий от класса `Building`, а затем переопределить его методы `build()` и `clear()`.

Чтобы вам было с чего начать, я взял функцию создания дерева из программы *forest.py* (с. 190) и скопировал ее в листинг 12.6. Введите этот код в IDLE и сохраните под именем *ghostTree.py* в папке *classes*.

Ghost tree — дерево-призрак

ghostTree.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

# Скопируйте сюда код из программы ghostHouse.py
# Создайте класс Tree

❶ def growTree(x, y, z):
    """Создает дерево в указанных координатах"""
    wood = 17
    leaves = 18

    # Ствол
    mc.setBlocks(x, y, z, x, y + 5, z, wood)

    # Листва
    mc.setBlocks(x - 2, y + 6, z - 2, x + 2, y + 6, z + 2, leaves)
    mc.setBlocks(x - 1, y + 7, z - 1, x + 1, y + 7, z + 1, leaves)

# Вызовите здесь методы build() и clear() класса Tree
```

Листинг 12.6. Функция создания дерева

Tree — дерево

Чтобы доделать программу, скопируйте и вставьте в нее код класса `Building` из файла *ghostHouse.py* (с. 320). Затем создайте новый класс `Tree`, унаследовав его от `Building`. Добавьте в класс `Tree` методы `build()` и `clear()`, тем самым переопределив такие же методы базового класса `Building`, чтобы получилось дерево, а не дом. Не забудьте использовать в окончательном коде метода `growTree()` аргумент `self` для доступа к свойствам объекта ❶.

Затем создайте объект класса `Tree` `ghostTree`. Вызовите его метод `build()`, подождите немного и затем вызовите метод `clear()`, чтобы дерево таинственным образом исчезло.

На рис. 12.7 показан результат работы моей программы.

БОНУСНОЕ ЗАДАНИЕ: ПРИЗРАЧНЫЙ ЛЕС

Измените код программы *ghostTree.py* так, чтобы она создавала целый призрачный лес. Как думаете, что за сокровища можно там найти?

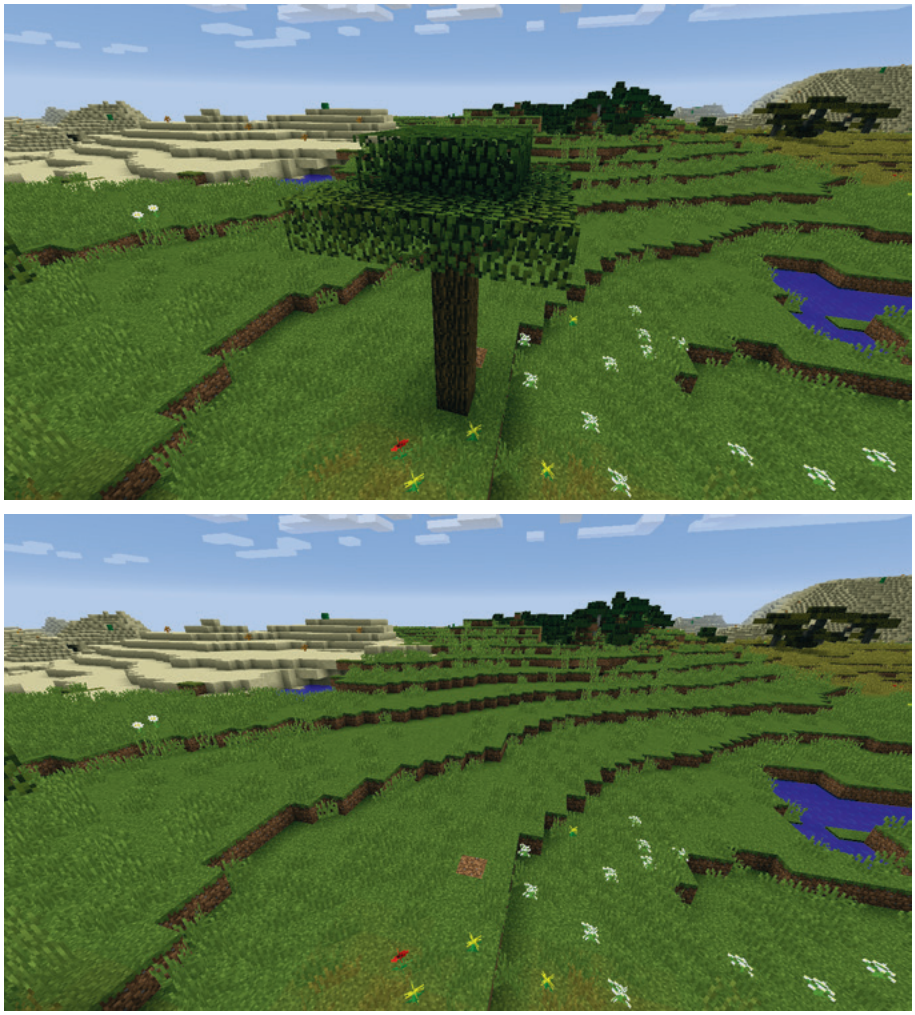

Рис. 12.7. Какое страшное дерево!

Что вы узнали

Вы только что изучили основы объектно ориентированного программирования — одного из самых эффективных подходов к созданию программ! Полученные навыки можно использовать не только в игре Minecraft — они пригодятся в любой области программирования, с которой вы решите познакомиться дальше!

ПОСЛЕСЛОВИЕ

Итак, настал важный момент: вы дочитали книгу до конца. Немаленькое вышло приключение — и для вас, и для меня. Пока я работал над текстом, успел отрастить не одну бороду, пожить в трех разных городах и выяснить, что у бананов нет семян. Если честно, работа над книгой доставила мне уйму удовольствия, и, хотя временами она была сложна и утомительна, я не сдавался, потому что хотел, чтобы люди прочитали книгу.

Вы очень многое узнали с тех пор, как впервые открыли ее: изучили основы языка Python и создали немало программ, творящих в мире Minecraft настоящие чудеса. Вы узнали про переменные, математические операции, строки, пользовательский ввод, булевы значения, познакомились с конструкцией `if`, циклами `while` и `for`, функциями, списками и словарями, модулями, файлами и классами. Возможно, сначала вы и были новичком, однако нынешнего багажа знаний хватит, чтобы решать на языке Python продвинутые задачи.

Как бы вы ни распорядились этими навыками в будущем, я искренне желаю вам удачи. Когда-то программирование было для меня просто хобби, а в итоге превратилось в полноценную работу, чему я очень рад!

Если мы когда-нибудь встретимся, я с удовольствием пожму вам руку как своему коллеге!

УСТРАНЕНИЕ НЕПОЛАДОК

Для работы с этой книгой вам понадобится Minecraft версии 1.8 или 1.9, Python версии 3.5 или выше и Java версии 7 или выше. Если у вас возникли проблемы с настройкой программ, скорее всего, дело в том, что программы устарели. Чтобы это проверить, следуйте инструкциям ниже.

Если вы убедились, что с версиями программ все в порядке, но ошибки все равно возникают, обратитесь к дальнейшим разделам этого приложения.

Та ли версия Minecraft у вас установлена?

Чтобы игра Minecraft могла подключиться к серверу Spigot, ее версия должна в точности соответствовать версии сервера. Однако новые версии Minecraft выходят довольно часто, и, если вы обновите игру до последней версии, сервер уже не сможет с ней работать. Чтобы обеспечить совместимость Spigot и Minecraft, следует настроить профиль игры.

Как это сделать — читайте на с. 26, если у вас установлена операционная система Windows, на с. 39, если у вас Mac OS, и на с. 45, если вы пользуетесь Raspberry Pi.

Та ли версия Python у вас установлена?

Выполнив указанные действия, проверьте, установлен ли у вас Python версии 3.5.0 или выше.

Если выяснится, что на вашем компьютере установлены и Python 2, и Python 3, деинсталлируйте Python 2 либо перейдите к разделу «Что если на моем компьютере стоят две версии Python?» на с. 345.

Для Windows

Убедитесь, что у вас установлен Python подходящей версии, — для этого откройте меню **Пуск** и выберите **Все программы (All Programs)** или **Программы (Programs)** либо воспользуйтесь поиском по слову «Python». Номер версии может быть указан в названии пункта меню. Если же в меню номера нет, запустите IDLE — номер версии Python будет показан в окне консоли.

Если номер установленной версии ниже, чем 3.5.0, ее следует обновить до последней версии — для этого следуйте инструкциям из раздела «Установка Python» на с. 21.

Для Mac OS

1. Кликните по иконке файлового менеджера Finder в панели Dock и введите `Terminal` в строке поиска. Кликните по значку **Terminal**, чтобы открыть окно терминала.

2. Введите в окне команду `python -V` (обратите внимание, что буква V — заглавная) и нажмите ENTER.

3. На экране появится номер версии Python. Если он ниже, чем 3.5.0, вам следует обновить программу до последней версии — для этого следуйте инструкциям из раздела «Установка Python» на с. 35.

Та ли версия Java у вас установлена?

Выполнив указанные действия, проверьте, действительно ли у вас установлена Java версии 7 или выше.

Для Windows

Выяснить, какая у вас версия, можно так:

1. Откройте меню **Пуск**.

2. Выберите **Все программы** (**All Programs**).

3. В папке *Java* кликните **About Java**.

Есть и другой способ узнать версию Java:

1. Кликните по кнопке **Пуск (Start)** или нажмите клавишу WINDOWS на клавиатуре и введите `cmd` в строке поиска. Найдите программу *cmd* и запустите ее, кликнув по названию.

2. Введите команду `java -version` и нажмите ENTER. Вы увидите сообщение вида `java version "1.8.0_72"`. Нужный вам номер версии — число, стоящее после первой точки (в данном случае это 8). Если ваш номер версии ниже 7 (или если на компьютере вообще нет Java), установите последнюю версию, следуя инструкциям из раздела «Установка Java» на с. 23.

Для Mac OS

1. Кликните по иконке файлового менеджера Finder в панели Dock и введите `Terminal` в строке поиска. Кликните по значку **Terminal**, чтобы открыть окно терминала.

2. Введите в окне терминала команду `java -version` и нажмите ENTER.

3. Вы увидите сообщение вида `java version "1.8.0_72"`. Нужный вам номер версии — число, стоящее после первой точки (в данном случае это 8). Если ваш номер версии ниже 7, установите последнюю версию, следуя инструкциям из раздела «Установка Java» на с. 36.

Что если на моем компьютере стоят две версии Python?

Если вы хотите, чтобы у вас были установлены и Python 3, и Python 2, но при этом у вас старая версия программы *Install_API*, этот файл нужно изменить для работы с Python 3.

Minecraft Python API работает и со старыми версиями Python, однако *для запуска программ из этой книги необходимо использовать Python 3* — иначе при выполнении функции `print()` и некоторых других операций компьютер выдаст ошибку. При этом наряду с Python 3 вы можете оставить и Python 2.

Для Windows

Чтобы изменить файл *Install_API*, выполните следующие шаги:

1. Откройте папку *Minecraft Tools* и найдите в ней файл *Install_API*.

2. Кликните по нему правой кнопкой мыши и выберите **Изменить (Edit).**

3. Когда файл откроется, найдите в нем следующую строку:

```
python -m pip install minecraftPythonAPI.zip
```

4. Если этой строки в файле нет, значит, у вас установлен файл последней версии и менять ничего не нужно. В противном случае отредактируйте строку, чтобы она выглядела так:

```
py –3 -m pip install minecraftPythonAPI.zip
```

5. Сохраните файл и закройте его.

6. Сделайте двойной клик по файлу *Install_API*, чтобы установить модуль Minecraft Python API.

Для Mac OS

Чтобы изменить файл *Install_API*, выполните следующие шаги:

1. Откройте папку *Minecraft Tools* и найдите в ней файл *Install_API*.

2. Кликните по этому файлу, удерживая клавишу CONTROL, и выберите **Edit**.

3. Когда файл откроется, найдите в нем следующую строку:

```
python -m pip install minecraftPythonAPI.zip
```

4. Если этой строки в файле нет, значит, у вас установлен файл последней версии и менять ничего не нужно. В противном случае отредактируйте строку, заменив «python» на «python3», вот так:

```
python3 -m pip install minecraftPythonAPI.zip
```

5. Сохраните файл и закройте его.

6. Сделайте двойной клик по файлу *Install_API*, чтобы установить модуль Minecraft Python API.

После этого убедитесь, что вы используете подходящую версию IDLE — для запуска программ из этой книги подходит только версия IDLE3.

Start_Server, файл не найден (для Windows)

При попытке запустить файл *Start_Server* может появиться окно с сообщением, что Windows не может найти файл *C:\Users\Petr\Документы* или что-нибудь в таком роде.

Эту ошибку можно обойти. Откройте папку *server* и двойным кликом запустите файл *start.bat* — теперь сервер должен работать без ошибок. Делайте так каждый раз, когда вам понадобится запустить сервер.

Есть и другой способ решения этой проблемы — скачать последнюю версию установочных файлов со страницы книги *http://mif.to/minecraft/*.

Ошибка ConnectionRefusedError (для Mac OS)

При попытке выполнить в Python команду `mc = Minecraft.create()` вы можете увидеть такое сообщение об ошибке:

```
ConnectionRefusedError: [Errno 61] Connection refused
```

либо похожее сообщение такого вида:

```
ConnectionRefusedError: [Errno 10061] No connection could be made because
the target machine actively refused it.
```

Как правило, причиной ошибки является устаревшая версия Java. Чтобы исправить эту проблему, обратитесь к инструкциям из раздела «Установка Java» на с. 36.

Если после переустановки Java ошибка не исчезнет, возможно, старая версия Java используется только в командной строке. Исправить это можно так:

1. Кликните по значку поиска в правом верхнем углу экрана.
2. Введите в строке поиска `Java Preferences`.
3. Откройте окно настроек. Убедитесь, что флажок стоит только напротив версии Java 8.

После изменения настроек проблема должна решиться. Если ошибка по-прежнему возникает, деинсталлируйте все старые версии Java и переустановите Java 8.

После запуска Install_API ничего не происходит (для Windows)

При запуске файла *Install_API* на пятом шаге из раздела «Установка Minecraft Python API и Spigot» (с. 25) на экране может появиться пустое черное окно.

Если такое случилось, возможно, всему виной ошибка в программе *pip*, которая используется для установки Minecraft Python API. Чтобы обойти эту ошибку, установите модуль Minecraft Python API из командной строки, вот так:

1. Откройте папку *Minecraft Tools* и найдите в ней файл *minecraftPythonAPI.zip*.
2. Кликните по адресной строке проводника, в окне которого открыта папка, и скопируйте находящийся там текст (кликнув правой кнопкой и выбрав **Копировать (Copy)** либо нажав CTRL + C). Адрес должен выглядеть примерно так:

```
C:\Users\user\Minecraft Python\Minecraft Tools
```

3. Откройте меню **Пуск** и введите в строке поиска `PowerShell`. Запустите программу, кликнув **PowerShell**.
4. В командной строке PowerShell введите `cd`, а затем, кликнув правой кнопкой мыши, вставьте скопированный ранее адрес папки *Minecraft Tools*.
5. Заключите только что вставленный текст в одинарные кавычки. Целиком строка должна выглядеть примерно так:

```
cd 'C:\Users\user\Minecraft Python\Minecraft Tools'
```

6. Нажмите ENTER. Теперь вы сможете выполнять из PowerShell команды, которые работают с содержимым папки *Minecraft Tools*.
7. Введите команду установки модуля Minecraft Python API:

```
python -m pip install minecraftPythonAPI.zip
```

8. Нажмите ENTER. Теперь модуль API должен установиться без проблем.

Ошибка с правами при установке Minecraft Python API (для Mac OS)

На шаге 8 из раздела «Установка Minecraft Python API и Spigot» (с. 37) может возникнуть следующая ошибка:

```
The directory '/Users/YourUserName/Library/Caches/pip/http' or its parent
directory is not owned by the current user and the cache has been
disabled. Please check the permissions and owner of that directory. If
executing pip with sudo, you may want sudo's -H flag.
```

Проверьте имена всех папок на пути к файлу *Install_API* и удалите из этих имен пробелы следующим образом:

1. Удерживая CONTROL, кликните по значку папки, в имени которой есть пробелы.
2. Выберите в контекстном меню **Переименовать (Rename)** и удалите пробелы из имени папки. Внимание — сам жесткий диск (Macintosh HD) переименовывать не надо!

После этого попробуйте еще раз запустить файл *Install_API*.

На некоторых «Маках» сообщение об ошибке появится, даже если модуль API установлен правильно. Если вы увидите в тексте сообщения следующую строку, значит, модуль Minecraft Python API установлен корректно и сообщение можно проигнорировать:

```
Requirement already satisfied (use --upgrade to upgrade): py3minepi…
```

ИДЕНТИФИКАТОРЫ БЛОКОВ

Там, где указано два числа, второе число обозначает состояние блока. Блоки, отмеченные звездочкой (*), есть в версии Minecraft для Raspberry Pi.

196	Акациевая дверь
126, 4	Акациевая плита
125, 4	Акациевая плита (двойная)
162	Акация
157	Активирующие рельсы
57*	Алмазный блок
56*	Алмазная руда
1, 5*	Андезит
103*	Арбуз
7*	Бедрок
160	Белая стеклянная панель
95*	Белое стекло
38, 6	Белый тюльпан
17, 2*	Береза
194	Березовая дверь
126, 2	Березовая плита
125, 2	Березовая плита (двойная)
5, 2*	Березовые доски
18, 2*	Березовые листья
160, 9	Бирюзовая стеклянная панель
35, 9*	Бирюзовая шерсть
95, 9*	Бирюзовое стекло
171, 9	Бирюзовый ковер
22*	Блок лазурита
165	Блок слизи
148	Большегрузная весовая пластина
44, 3*	Булыжная плита
43, 3*	Булыжная плита (двойная)
4*	Булыжник
67*	Булыжные ступеньки
58*	Верстак
147	Весовая пластина
154	Воронка
158	Выбрасыватель
175, 2	Высокая трава
1, 6*	Гладкий андезит
1, 2*	Гладкий гранит
1, 4*	Гладкий диорит
179, 2	Гладкий красный песчаник
24, 2	Гладкий песчаник

	82*	Глина
	34	Головка поршня
	160, 3	Голубая стеклянная панель
	35, 3*	Голубая шерсть
	95, 3*	Голубое стекло
	38, 3	Голубой василек
	171, 3	Голубой ковер
	13*	Гравий
	1, 1*	Гранит
	39*	Гриб
	19	Губка
	151	Датчик дневного света
	197	Дверь из темного дуба
	195	Дверь из тропического дерева
	72	Деревянная нажимная плита
	44, 2*	Деревянная плита
	43, 2*	Деревянная плита (двойная)
	2*	Дерн
	46*	Динамит
	46, 1*	Динамит, ручная детонация
	1, 3*	Диорит
	5, 4	Доски из акации
	5, 5	Доски из темного дуба
	5, 3	Доски из тропического дерева
	17*	Древесина дуба
	17, 3	Древесина тропического дерева
	64*	Дубовая дверь
	107*	Дубовая калитка
	126	Дубовая плита
	125	Дубовая плита (двойная)
	5*	Дубовые доски
	18*	Дубовые листья
	53*	Дубовые ступеньки
	85*	Дубовый забор
	193	Еловая дверь
	126, 1	Еловая плита
	125, 1	Еловая плита (двойная)
	5, 1*	Еловые доски
	6, 1*	Еловый саженец
	17, 1*	Ель
	71*	Железная дверь
	15*	Железная руда
	101	Железные прутья
	42*	Железный блок
	167	Железный люк
	160, 4	Желтая стеклянная панель
	35, 4*	Желтая шерсть
	95, 4*	Желтое стекло
	171, 4	Желтый ковер
	98, 1*	Замшелые каменные кирпичи
	48*	Замшелый булыжник
	160, 13	Зеленая стеклянная панель
	35, 13*	Зеленая шерсть

	95, 33*	Зеленое стекло
	117	Зельеварка
	3*	Земля
	41*	Золотой блок
	14*	Золотая руда
	129	Изумрудная руда
	133	Изумрудный блок
	178	Инверсный датчик дневного света
	127	Какао-бобы
	81*	Кактус
	3, 1*	Каменистая земля
	70	Каменная нажимная плита
	44*	Каменная плита
	43*	Каменная плита (двойная)
	98*	Каменные кирпичи
	1*	Камень
	142	Картофель
	155, 2*	Кварцевая колонна
	44, 7*	Кварцевая плита
	43, 7*	Кварцевая плита (двойная)
	156*	Кварцевые ступеньки
	155*	Кварцевый блок
	45*	Кирпичи
	44, 4*	Кирпичная плита
	43, 4*	Кирпичная плита (двойная)
	108*	Кирпичные ступеньки
	47*	Книжная полка
	171	Ковер
	160, 12	Коричневая стеклянная панель
	95, 12*	Коричневое стекло
	35, 12*	Коричневая шерсть
	171, 12	Коричневый ковер
	118	Котел
	160, 14	Красная стеклянная панель
	35, 14*	Красная шерсть
	95, 14*	Красное стекло
	171, 14	Красный ковер
	12, 1	Красный песок
	179	Красный песчаник
	38, 4	Красный тюльпан
	26*	Кровать
	131	Крюк
	111	Кувшинка
	21*	Лазуритовая руда
	79*	Лед
	65*	Лестница
	29	Липкий поршень
	161	Листья акации
	161, 1	Листья темного дуба
	18, 3	Листья тропического дерева
	106	Лоза
	38, 2	Лук-батун
	96	Люк
	38	Мак

138	Маяк
32	Мертвый куст
110	Мицелий
19, 1	Мокрая губка
141	Морковь
169	Морской фонарь
40*	Мухомор
100	Мухоморный блок
115	Нарост Незера
68*	Настенная табличка
153	Незер-кварцевая руда
87	Незерит
44, 6*	Незеритовая плита
43, 6*	Незеритовая плита (двойная)
112*	Незеритовые кирпичи
114*	Незеритовые ступеньки
25	Нотный блок
172	Обожженная глина
60*	Обработанная почва
155, 1*	Обработанный кварцевый блок
179, 1	Обработанный красный песчаник
24, 1	Обработанный песчаник
49*	Обсидиан
51	Огонь
99	Огромный гриб
37*	Одуванчик
160, 1	Оранжевая стеклянная панель
35, 1*	Оранжевая шерсть
95, 1*	Оранжевое стекло
171, 1	Оранжевый ковер
38, 5	Оранжевый тюльпан
31, 2*	Папоротник
30*	Паутина
12*	Песок
88	Песок душ
24*	Песчаник
44, 1*	Песчаниковая плита
43, 1*	Песчаниковая плита (двойная)
128*	Песчаниковые ступеньки
61*	Печь
175, 5	Пион
44, 5*	Плита из каменных кирпичей
43, 5*	Плита из каменных кирпичей (двойная)
182	Плита из кварцевого песчаника
181	Плита из кварцевого песчаника (двойная)
126, 5	Плита из темного дуба
125, 5	Плита из темного дуба (двойная)
126, 3	Плита из тропического дерева
125, 3	Плита из тропического дерева (двойная)

	174	Плотный лед
	171, 2	Пурпурный ковер
	3, 2*	Подзол
	175	Подсолнух
	119	Портал в Энд
	90	Портал в Незер
	33	Поршень
	98, 2*	Потрескавшиеся каменные кирпичи
	168	Призмарин
	168, 1	Призмариновый кирпич
	84	Проигрыватель
	160, 2	Пурпурная стеклянная панель
	35, 2*	Пурпурная шерсть
	95, 2*	Пурпурное стекло
	59*	Пшеница
	23	Раздатчик
	62*	Разожженная печь
	120	Рамка портала в Энд
	175, 3	Раскидистый папоротник
	52	Рассадник монстров
	132	Растяжка
	73*	Редстоуновая руда
	152	Редстоуновый блок
	94	Редстоуновый повторитель (вкл)
	93	Редстоуновый повторитель (выкл)
	76	Редстоуновый факел (вкл)
	75	Редстоуновый факел (выкл)
	124	Редстоуновый фонарь (вкл)
	123	Редстоуновый фонарь (выкл)
	98, 3	Резные каменные кирпичи
	66	Рельсы
	28	Рельсы с датчиком
	160, 6	Розовая стеклянная панель
	35, 6*	Розовая шерсть
	95, 6*	Розовое стекло
	171, 6	Розовый ковер
	175, 4	Розовый куст
	38, 7	Розовый тюльпан
	38, 8	Ромашка
	69	Рычаг
	6, 4	Саженец акации
	6, 2*	Саженец березы
	6*	Саженец дуба
	6, 5	Саженец темного дуба
	6, 3	Саженец тропического дерева
	83*	Сахарный тростник
	91	Светильник Джека
	160, 5	Светло-зеленая стеклянная панель
	35, 5*	Светло-зеленая шерсть
	95, 5*	Светло-зеленое стекло
	171, 5	Светло-зеленый ковер
	160, 8	Светло-серая стеклянная панель


35, 8* Светло-серая шерсть


95, 8* Светло-серое стекло


171, 8 Светло-серый ковер


89* Светокамень


74* Светящаяся редстоуновая руда


160, 7 Серая стеклянная панель


35, 7* Серая шерсть


95, 7* Серое стекло


171, 7 Серый ковер


95, 11* Синее стекло


171, 11 Синий ковер


38, 1 Синяя орхидея


160, 11 Синяя стеклянная панель


35, 11* Синяя шерсть


175, 1 Сирень

78* Снег

80* Снежный блок


170 Сноп сена


20* Стекло

102* Стеклянная панель


116 Стол зачарований


9* Стоячая вода


11* Стоячая лава


109* Ступеньки из каменного кирпича


54* Сундук


63* Табличка


8* Текущая вода


10* Текущая лава


171, 13 Темно-зеленый ковер


162, 1 Темный дуб


168, 2 Темный призмарин


92 Торт


31, 1* Трава


86 Тыква


173 Угольный блок


16* Уголь


50* Факел


160, 10 Фиолетовая стеклянная панель


35, 10* Фиолетовая шерсть


95, 10* Фиолетовое стекло

171, 10 Фиолетовый ковер

18, 1* Хвоя

140 Цветочный горшок

105* Черенок арбуза

104 Черенок тыквы

35, 15* Черная шерсть

95, 15* Черное стекло

171, 15 Черный ковер

35* Шерсть

121 Эндерняк

27 Энергорельсы

122 Яйцо дракона

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

ОБ АВТОРЕ

крэйг Ричардсон — разработчик программного обеспечения и преподаватель языка Python. Работал в организации Raspberry Pi Foundation, преподавал информатику в старших классах, провел множество семинаров, посвященных созданию Python-программ для игры Minecraft.

О ТЕХНИЧЕСКОМ РЕДАКТОРЕ

Джон Лутц — учитель математики. Он работает в новоорлеанской муниципальной школе, где ведет внеклассные уроки по языку Scratch, роботостроению на базе Arduino и 3D-печати. Джон участвовал в разработке и внедрении программы по информатике в своей школе, и теперь она продолжает развиваться, вовлекая новые яркие умы в искусство кодинга. Поучаствовав в работе над этой книгой, Джон написал программу, которая уничтожает всех зомби-детей в его мире Minecraft.

БЛАГОДАРНОСТИ

Огромное спасибо сотрудникам издательства No Starch Press: Рили Хоффман, Хэйли Бэйкер, Тайлеру Ортману и Дженнифер Гриффит-Делгадо, а также исключительно ответственному техническому рецензенту Джону Лутцу.

Спасибо Дэвиду Уэйлу и Мартину О'Хэнлону, которые очень помогли мне в разрешении технических вопросов. Также я хочу поблагодарить компанию Mojang за бесплатную версию Minecraft: Pi Edition (с оригинальной реализацией Minecraft Python API). Спасибо людям, посвятившим свое свободное время разработке Spigot и CanaryMod, — без них эта книга никогда не была бы написана. То же касается замечательных людей, адаптировавших Minecraft API для Python 3, а также Алекса Брэдбери, работавшего с операционной системой Raspbian.

Если вам доведется повстречать Дэвида Уэйла, Мэтью Тиммонса Брауна, Дэвида Хонесса, Рэйчел Рэйнс, Эндрю Робинсона или Дженни Бреннан, наградите их аплодисментами за помощь в проведении семинаров по Minecraft и Python. Также аплодисментов заслуживают Тим Ричардсон, Майкл Хорн, Алан О'Донахью и Лора Диксон — за организацию мероприятий, привлекающих молодежь к участию в этих семинарах.

Если бы не Брайан Кортил, миссия, в которой используется Flask, вышла бы гораздо менее интересной. А Шарлотта Годли очень помогла мне, одолжив свой «Мак», чтобы я мог написать инструкцию по установке программ в Mac OS.

И наконец, я безмерно благодарен моим друзьям, семье и коллегам за поддержку на разных этапах моего затворничества.

РЕСУРСЫ

Посетите страницу сайта, посвященную книге, *http://mif.to/minecraft/*. Там вы найдете тексты программ для миссий из этой книги и установочные файлы.

Технические требования

Вот список программ, которые понадобятся вам при работе с этой книгой.

Если вы используете операционную систему Windows 7, 8 или 10

- Официальная, платная версия Minecraft, которую можно купить на сайте *https://minecraft.net/ru-ru/*.
- Python 3 — дистрибутив можно бесплатно скачать по адресу *http://www.python.org/downloads/*.
- Java — дистрибутив можно бесплатно скачать по адресу *https://www.java.com/ru/download/*.
- Установочные файлы к этой книге — архив можно скачать бесплатно на сайте *http://mif.to/minecraft/* (содержит Minecraft Python API и Spigot).

Подробные инструкции смотрите в разделе «Установка и настройка программ для Windows» на с. 20.

Если вы используете Mac OS X 10.10 или более новую версию

- Официальная, платная версия Minecraft, которую можно купить на сайте *https://minecraft.net/*.
- Python 3 — дистрибутив можно бесплатно скачать по адресу *https://www.python.org/downloads/*.
- Java — дистрибутив можно бесплатно скачать по адресу *http://www.oracle.com/technetwork/java/javase/downloads/index.html*.

- Установочные файлы к этой книге — архив можно бесплатно скачать на сайте *http://mif.to/minecraft/* (содержит Minecraft Python API и Spigot).

Подробные инструкции смотрите в разделе «Установка и настройка программ для Mac OS» на с. 33.

Если вы используете Raspberry Pi

Вам не нужно ничего устанавливать — бесплатная версия Minecraft идет в комплекте с Raspberry Pi! Вы можете узнать об этом больше на англоязычном сайте *http://www.raspberrypi.org/*. Подробные инструкции смотрите в разделе «Установка и настройка программ для Raspberry Pi» на с. 44.

Максимально полезные книги от издательства «Манн, Иванов и Фербер»

Заходите в гости: http://www.mann-ivanov-ferber.ru/

Наш блог: http://blog.mann-ivanov-ferber.ru/

Мы в Facebook: http://www.facebook.com/mifbooks

Мы ВКонтакте: http://vk.com/mifbooks

Предложите нам книгу:

http://www.mann-ivanov-ferber.ru/about/predlojite-nam-knigu/

Ищем правильных коллег:

http://www.mann-ivanov-ferber.ru/about/job/

Издание для досуга
Для среднего и старшего школьного возраста

Крэйг **Ричардсон**

Программируем с Minecraft

Создай свой мир с помощью Python

Главный редактор *Артем Степанов*
Руководитель направления *Анастасия Троян*
Ответственный редактор *Анна Шахова*
Литературный редактор *Дарья Николаева*
Научный редактор *Георгий Гаджиев*
Дизайн обложки *Сергей Хозин*
Верстка *Елена Бреге*
Корректоры *Надежда Болотина,*
Юлия Молокова, Наталья Витько

ИГРАЙ В MINECRAFT И УЧИСЬ ПРОГРАММИРОВАТЬ

СОЗДАЙ СВОЙ УДИВИТЕЛЬНЫЙ МИР С ПОМОЩЬЮ PYTHON

Вам не страшны криперы, глубокие пещеры и высокие горы? А знаете ли вы, что меч можно превратить в волшебную палочку, дворец – возвести в мгновение ока, а тайные ходы легко открываются нажатием секретной кнопки?

Книга «Программируем с Minecraft» позволит творить эти и многие другие чудеса с помощью Python – языка программирования, которым пользуются миллионы людей, от профи до новичков!

Выполняйте пошаговые инструкции, и вы:

– научитесь сохранять в переменных разные типы данных;
– освоите принцип действия функций;
– узнаете, как проверять условия при помощи булевых значений, операций сравнения и логических операций;
– познакомитесь с циклами while и for;
– поработаете со списками, кортежами и словарями;
– научитесь создавать файлы, записывать и считывать из них данные;
– поймете, в чем прелесть объектно ориентированного программирования.

При этом в вашем арсенале появится большое количество работающих программ, навык программирования на Python и радость от того, что вы можете создавать собственные миры!

Книга подойдет детям от 10 лет, а также всем, кто хочет начать программировать с нуля или не мыслит жизни без Minecraft.

Автор книги Крэйг Ричардсон – разработчик программного обеспечения и преподаватель языка Python. Он работал в Raspberry Pi Foundation, преподавал информатику в старших классах, провел множество семинаров, посвященных созданию Python-программ для Minecraft.

МИФ ДЕТСТВО

Детские книги на сайте
mann-ivanov-ferber.ru

facebook.com/mifdetstvo
vk.com/mifdetstvo
instagram.com/mifdetstvo

no starch press

ISBN 978-5-00100-819-4