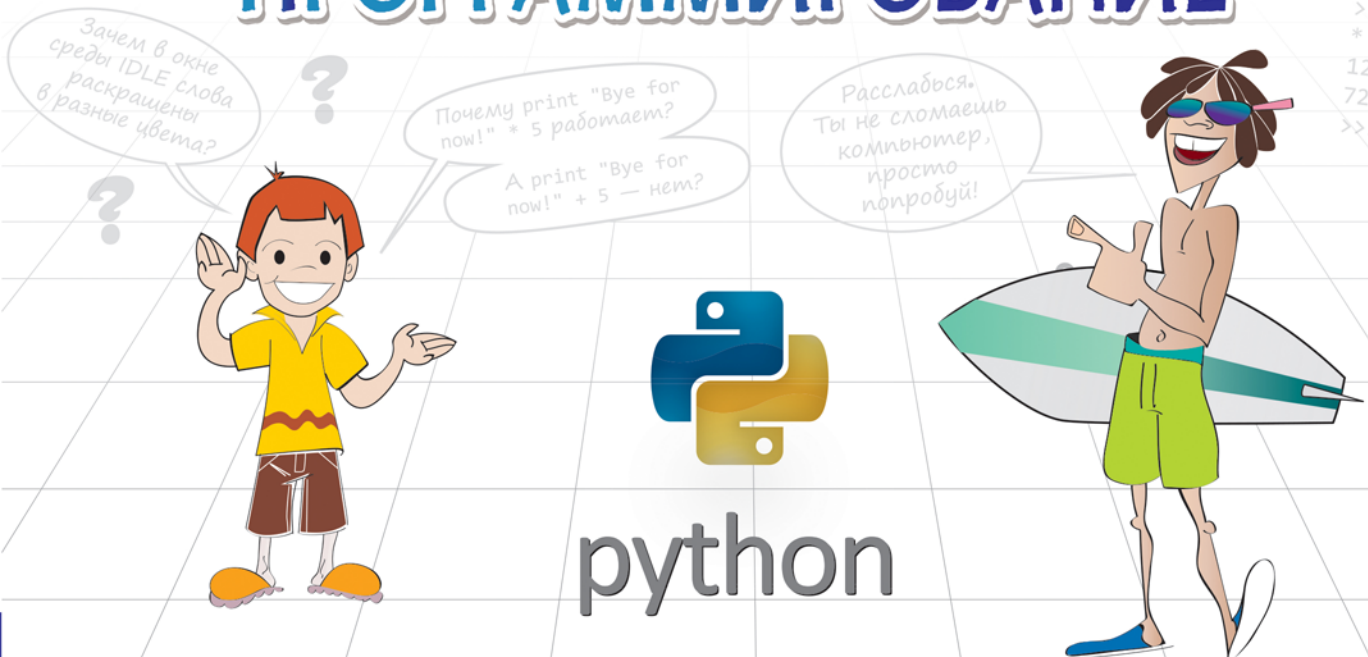
Уоррен Сэнд, Картер Сэнд

8+

Hello World!

ЗАНИМАТЕЛЬНОЕ ПРОГРАММИРОВАНИЕ



python

ПИТЕР

ББК 32.973.2-018
УДК 004.42
С18

У. Сэнд, К. Сэнд

Hello World! Занимательное программирование. — СПб.: Питер, 2016. — 400 с.: ил. — (Серия «Вы и ваш ребенок»).

ISBN 978-5-496-01273-7

Привет! Любишь компьютерные игрушки? А хочешь попробовать написать что-нибудь сам? Представь, как зауважают тебя друзья, когда ты покажешь им игру своей собственной разработки, где при загрузке на экране появится твое имя! «Вот круто!» — будут говорить они, качая ее с твоей странички. И самая красивая девочка в классе, узнав об этом, наконец, обратит на тебя внимание...

Но для этого надо научиться программировать. Это сложно? Ну, на самом деле не очень. Главное — поставить себе цель и ломиться к ней напролом, как носорог через джунгли. Ты наверняка знаешь, что языков программирования существует немало, но мы предлагаем тебе научиться «писать код», как говорят профессионалы, на языке Python (Питон). Он относительно прост в изучении, но обладает всеми необходимыми функциями. Программы на нем получаются быстрыми и легко читаемыми.

Эту книгу по языку Python написали два человека. Взрослый дядька-программист и его сын. Этот сын тоже совсем недавно был подростком, знает, как порой муторно бывает учиться, и поэтому он проследил, чтобы папа объяснялся не очень заумно. Так что если ты никогда не программировал, не беда. Если ты знаком с азами — e-mail, Интернет, mp3, можешь запустить или сохранить файл — ты во всем разберешься. Удачи в освоении!

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ).

ISBN 978-1617290923 англ.
ISBN 978-5-496-01273-7

Заведующая редакцией *Ю. Сергиенко*
Ведущий редактор *Н. Римицан*
Переводчик *И. Рузмайкина*
Литературный редактор *А. Жданов*
Художественный редактор *С. Заматевская*
Корректоры *Л. Казарина, В. Сайко*
Верстка *Л. Родионова*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.13.000 — Книги печатные для детей.
Подписано в печать 16.12.15. Формат 84x108/16.
Бумага офсетная. Усл. п. л. 42,000. Тираж 3000. Заказ 0000
Отпечатано в соответствии с предоставленными материалами в ООО «ИПК Парето-Принт». 170546, Тверская область, Промышленная зона Боровлево-1, комплекс №3А, www.pareto-print.

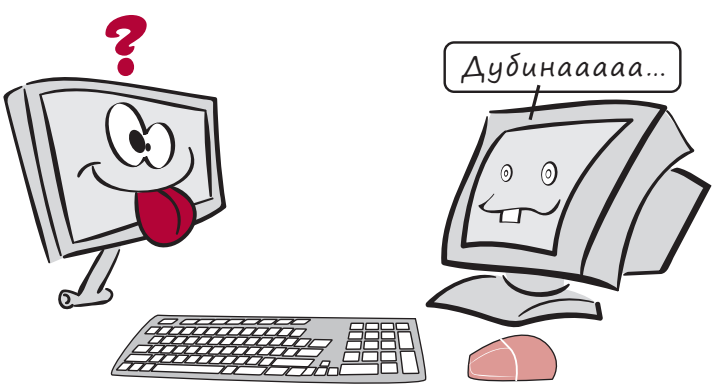
Содержание

Предисловие

Предисловие — это тот самый раздел в начале книги, который все перелистывают, чтобы побыстрее перейти к сути дела, не так ли? Разумеется, можно не читать этот раздел, но кто знает, какую информацию вы рискуете упустить... Тут всего несколько страниц, и я рекомендую вам ознакомиться с их содержимым. Просто на всякий случай.

Что такое программирование

Говоря по-простому, программирование заставляет компьютер выполнять некие действия. Компьютеры — это глупые машины, не имеющие никакого представления о том, как делается что бы то ни было. Вы должны им все объяснить, не упустив при этом ни одной детали.



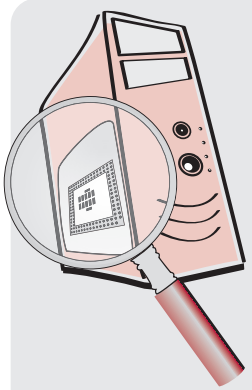
Дав компьютеру правильные инструкции, вы сможете делать множество потрясающих вещей.

НОВЫЕ СЛОВА

Инструкцией называется базовая команда, направленная на выполнение одного конкретного действия.

Компьютерные программы состоят из инструкций. Современные компьютеры прекрасно справляются с самыми сложными задачами, потому что умные программисты уже написали программное обеспечение, объясняющее, как следует поступать. *Программным обеспечением* называется программа или набор программ, запускаемых на вашем, а иногда и на чужом, компьютере, например на веб-сервере.

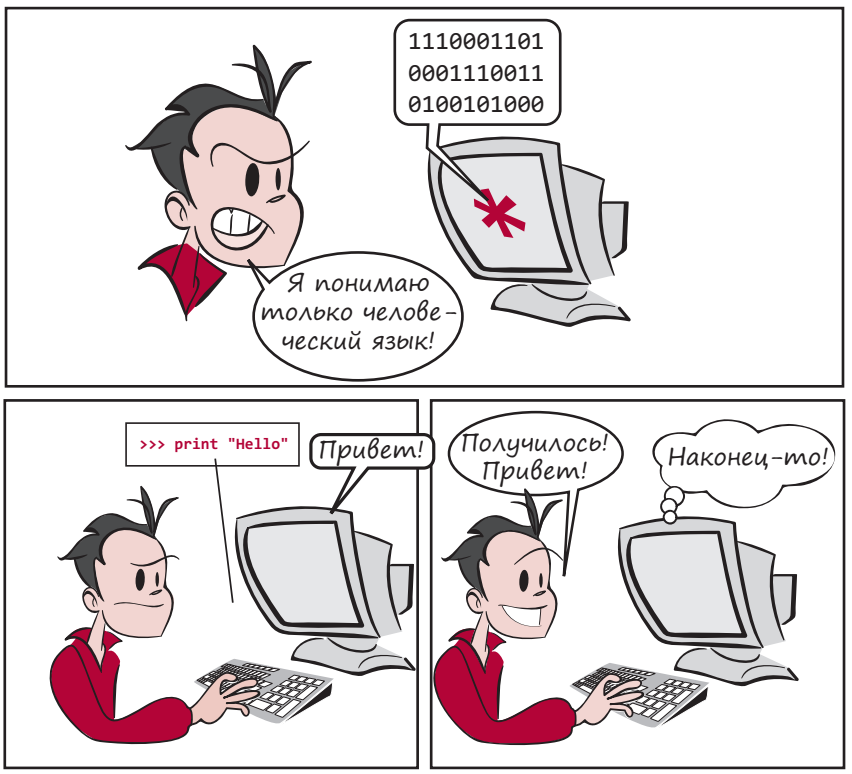
ЧТО ПРОИСХОДИТ ВНУТРИ



Компьютеры «думают» при помощи множества электрических контуров. На базовом уровне эти контуры представляют собой переключатели, которые могут находиться в двух положениях: включено или выключено. Инженеры и специалисты по информатике используют для обозначения этих положений значения 1 и 0. Все эти единицы и нули представляют собой код, называемый двоичным. Этот термин появился потому, что мы имеем дело с «двумя состояниями». Эти состояния — включено и выключено, или 1 и 0.
Вы знаете, что двоичная цифра равна биту?

Python — язык для нас и для компьютера

Все компьютеры понимают только двоичный код. Но люди, как правило, не умеют разговаривать на этом языке. Значит, требуется более простой способ общения с компьютером. Так появились языки программирования. Именно язык программирования позволяет писать понятные нам вещи, которые потом переводятся в понятный компьютеру двоичный код.



Существует множество разных языков программирования. В этой книге вы познакомитесь с одним из них, который называется Python.

- Настоятельно рекомендуем вам воспользоваться программой установки Hello World, которая предоставит вам необходимую для чтения этой книги версию Python. Ее вы найдете на сайте *www.manning.com/books/hello-world-second-edition*.

Зачем изучать программирование

Даже если вы не собираетесь становиться профессиональным программистом (а большинству людей это не требуется), существует множество причин для изучения программирования:

- самая важная причина — ваше желание! Программирование очень интересно и приносит внутреннее удовлетворение и как хобби, и как профессия;
- если вы интересуетесь компьютерами и хотите больше узнать о том, как они работают, как заставить их делать то, что вам нужно, имеет смысл изучить программирование;

- возможно, вы не прочь написать собственную игру или, устав от поисков программы, отвечающей вашим запросам, предпочитаете написать ее своими руками;
- в наши дни компьютеры окружают нас со всех сторон. Скорее всего, вы пользуетесь компьютером в школе или дома, а может быть, и там и там. Изучение программирования поможет вам лучше понять принцип работы компьютера в целом.

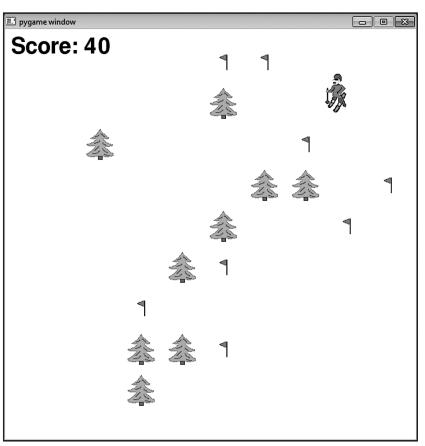
Почему Python

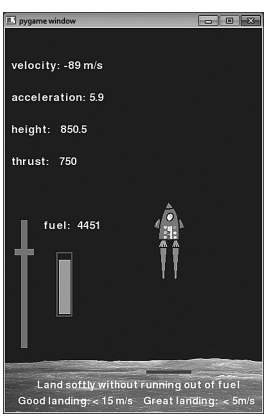
Почему среди множества языков программирования (а их действительно очень много!) я выбрал Python? Вот несколько причин:

- Python изначально создавался таким образом, чтобы его было легко изучать. Написанные на Python программы читаются, пишутся и понимаются намного проще, чем программы на других языках;
- интерпретатор Python абсолютно бесплатен. Вы можете загрузить как его, так и множество полезных и забавных программ, не потратив ни копейки;
- программное обеспечение Python имеет открытый исходный код. А это означает, в частности, что любой пользователь может расширить возможности Python-программ (добавив дополнительную функциональность или упростив решение каких-то задач). Многие пользователи именно так и поступают, в итоге существует большая коллекция доступных для загрузки бесплатных программ;
- Python не игрушка. Этот язык не только хорошо подходит для изучения программирования, но и применяется тысячами профессионалов по всему миру, включая сотрудников NASA и Google. А это значит, что после изучения Python вам не придется переключаться на «настоящий» язык для создания «настоящих» программ. Python позволяет реализовать огромное количество задач;
- программы на Python можно запускать на разных компьютерах. Они работают в Windows PC, Mac и на машинах с операционной системой Linux. В большинстве случаев Python-программа, запущенная у вас дома на машине с Windows, без проблем будет работать с операционной системой Mac OS X. Вы можете читать эту книгу, сидя перед любым компьютером, на который установлен интерпретатор Python (а если вы захотите воспользоваться компьютером, на котором нет интерпретатора Python, его можно загрузить бесплатно);
- мне нравится Python. Я получаю удовольствие от его изучения и применения и надеюсь, вам он тоже понравится.

Для развлечения

Осталось упомянуть всего одну вещь… Одним из самых притягательных развлечений для детей являются компьютерные игры с графикой и звуком. Мы научимся писать собственные игры и добавлять к ним графику и звуковое сопровождение. Вот примеры программ, которые мы собираемся создать:





Я думаю (по крайней мере надеюсь), что вы сочтете изучение основ программирования и создание первых программ делом настолько же увлекательным, как посадка космического корабля на поверхность Луны или управление лыжником, спускающимся по склону. Приятного чтения!

О книге

Эта книга учит основам программирования. Она предназначена для детей, но пригодится и взрослым, решившим узнать, как программируется компьютер.

Никаких особенных навыков программирования для чтения данной книги не требуется, но желательно иметь представление об основах работы с компьютером. Хватит умения пользоваться электронной почтой, искать информацию в Сети, слушать музыку, играть в игры и писать рефераты. Если вы можете запускать программы, открывать и сохранять файлы, у вас не будет проблем.

Что вам потребуется

Эта книга учит программированию на Python. Интерпретатор Python является бесплатным приложением и доступен для загрузки на разных сайтах, в том числе на сайте книги. Для изучения программирования с помощью этой книги вам потребуются следующие вещи.

- Сама книга (разумеется!).
- Компьютер с операционной системой Windows, Mac OS X или Linux. Все представленные в книге примеры кода выполнялись на компьютере с операционной системой Windows. (На сайте книги *www.manning.com/books/hello-world-second-edition* можно найти дополнительные рекомендации для пользователей Mac и Linux.)
- Базовые навыки работы с компьютером (запуск программ, сохранение файлов и т. п.). Если эти вещи представляют для вас сложность, попросите помощи у родителей или у преподавателя.
- Разрешение установить интерпретатор Python на ваш компьютер (от родителей, учителя или другого лица, отвечающего за вашу машину). Мы настоятельно рекомендуем воспользоваться программой установки Hello World. Она обеспечит вас именно той версией Python, которая требуется для чтения данной книги. Программу установки можно найти на сайте *www.manning.com/books/hello-world-second-edition*.

- Желание учиться и экспериментировать, даже если в первое время у вас будет получаться далеко не все.

Что вам не потребуется

Для изучения программирования при помощи этой книги кое-что вам точно не потребуется.

- Покупать какое бы то ни было программное обеспечение. Все, что вам нужно, распространяется бесплатно, и копии этих программ вы всегда найдете на сайте книги *www.manning.com/books/hello-world-second-edition*.
- Уметь программировать. Это книга для начинающих.

Работа с книгой

Вот несколько советов, которые позволят вам извлечь максимальную пользу при чтении данной книги.

- Лично проверяйте все встречающиеся примеры кода.
- Старайтесь набирать программы вручную.
- Отвечайте на контрольные вопросы.
- Не волнуйтесь и получайте удовольствие!

проверка кода

Примеры кода в книге выглядят так:

```
if timsAnswer == correctAnswer:
    print "Вы ответили правильно!"
    score = score + 10
```

Всегда проверяйте их, набирая текст в редакторе. (Я расскажу вам, как это делается.) Можно сесть в большое удобное кресло и прочитать эту книгу от корки до корки. При этом вы даже узнаете что-то о программировании. Но учиться лучше всего на практике, так что пишите код.

установка интерпретатора Python

Читать эту книгу бессмысленно без установленного у вас на компьютер интерпретатора Python. Настоятельно рекомендуем воспользоваться программой установки Hello World, которая обеспечит вас как корректной версией интерпретатора Python, так и другими необходимыми вещами. Программу Hello World можно загрузить с сайта книги: *www.manning.com/books/hello-world-second-edition*.

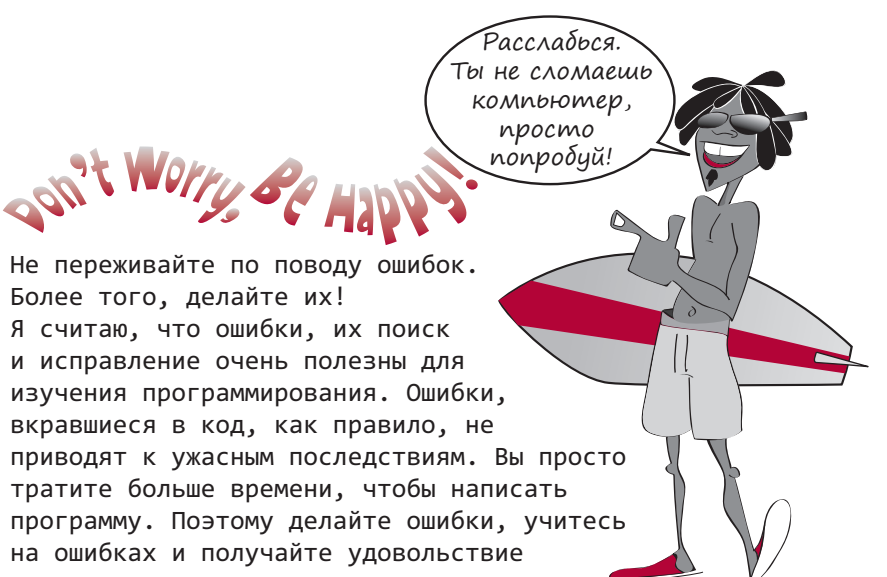
Установив интерпретатор Python другим способом, вы можете оказаться обладателем не той версии, о которой рассказывается в книге, кроме того, у вас не будет ряда необходимых модулей, что может стать причиной разочарования, ведь некоторые программы просто не будут работать нужным образом.

НАБОР КОДА ВРУЧНУЮ

Прилагаемая к книге программа установки скопирует все примеры программ на жесткий диск вашего компьютера (если вы захотите). Ее можно загрузить с сайта книги: *www.manning.com/books/hello-world-second-edition*. Здесь можно найти и загрузить отдельные примеры, но я бы посоветовал вам набирать весь код вручную. Именно в процессе набора кода вы «почувствуете», что такое программирование вообще и Python в частности. (Да и всем нам не помешает дополнительная практика в наборе текста!)

КОНТРОЛЬНЫЕ ВОПРОСЫ

В конце каждой главы вы найдете вопросы, направленные на повторение пройденного материала. Старайтесь отвечать на них как можно подробнее. Если вы зашли в тупик, найдите кого-то, кто разбирается в программировании и сможет вам помочь. Проработайте с этим человеком непонятные моменты — такая практика способствует обучению. По возможности старайтесь не подглядывать в ответы. (Да, в конце книги вы найдете ответы, но старайтесь не злоупотреблять ими.)



Не переживайте по поводу ошибок. Более того, делайте их! Я считаю, что ошибки, их поиск и исправление очень полезны для изучения программирования. Ошибки, вкравшиеся в код, как правило, не приводят к ужасным последствиям. Вы просто тратите больше времени, чтобы написать программу. Поэтому делайте ошибки, учитесь на ошибках и получайте удовольствие от программирования.

Я хотел убедиться, что детям эта книга подходит, что она забавна и проста для понимания. К счастью, мне в этом помогли. Картер — это ребенок, который любит компьютеры и хочет узнать о них побольше. Именно он помог мне убедиться, что я написал все правильно. Когда Картер замечал что-то забавное, необычное или не имеющее смысла, мы вставляли в книгу вот такую картинку.

Что нового во втором издании

Что осталось без изменений. Мы решили оставить интерпретатор Python 2 и не переходить к Python 3. Причины такого шага объясняются в главе 1.

А теперь новшества, появившиеся во втором издании.

- Мы добавили цвет и примечания, объясняющие разницу между Python 2 и Python 3.
- В главу 12 добавлен раздел, посвященный Python-словарям.
- Рассматривая в главе 20 программирование графических интерфейсов, мы перешли от уже не поддерживаемого PythonCard к более распространенному PyQt. Еще это приложение было использовано при создании программы «Виселица» в главе 22 и «Виртуальный питомец» в главе 24.
- В дополнительной главе 25[1] подробно объяснен принцип работы программы Skier, практически без объяснений представленной в главе 10.
- Дополнительная глава 26 посвящена искусственному интеллекту борющихся друг с другом роботов.

Связь с авторами

Свои комментарии и вопросы вы можете направлять по электронной почте *cp4khelp@gmail.com*.

Примечание для родителей и преподавателей

Интерпретатор Python — бесплатная программа с открытым исходным кодом. Загрузить программное обеспечение и необходимые файлы можно с сайта *www.manning.com/books/hello-world-second-edition*. С этими файлами просто работать, они не содержат ни вирусов, ни программ-шпионов.

[1] Главы 25 и 26 доступны для скачивания по адресу http://goo.gl/VPoZ48.

Благодарности

Я не начал бы работу над этой книгой и вряд ли закончил этот труд без вдохновляющего участия моей жены Патрисии. Мы долго пытались найти книгу, которая поддержала бы интерес нашего сына Картера к изучению программирования, и Патрисия сказала: «Ты должен сам ее написать. Это будет замечательный проект, над которым вы сможете работать вместе». И, как это часто бывает, она оказалась совершенно права. Патрисия умеет выявлять в людях их лучшие черты. В итоге мы с Картером стали думать, что включить в такую книгу, придумали структуру глав, примеры программ и попытались сделать все веселым и интересным. Как только работа началась, Картер и Патрисия приложили все силы к достижению конечного результата. Картер был готов отказаться от сказки на ночь, чтобы поработать над книгой. А если процесс по каким-то причинам останавливался, он напоминал мне: «Папа, мы уже несколько дней не пишем книгу!» Картер и Патрисия постоянно повторяли, что любой цели можно достичь, стоит только серьезно взяться за дело. Все члены нашей семьи, включая дочь Кайру, жертвовали совместными развлечениями, пока книга не была закончена. Я хочу сказать им спасибо за терпение и дружескую поддержку, благодаря которой эта книга увидела свет.

Рукопись — только половина дела; нужно еще сделать так, чтобы она нашла своего читателя. Эта книга никогда не была бы опубликована, если бы не энтузиазм и постоянная поддержка Майкла Стивенса из издательства *Manning Publications*. Он с самого начала согласился с необходимостью книги подобного рода. Невозможно переоценить его веру в данный проект и бесконечное терпение, с которым он руководил неопытными авторами. Также я хочу сказать искреннее спасибо остальным сотрудникам издательства, помогавшим мне в реализации замысла, в частности Мэри Пирджис, которая терпеливо координировала все аспекты производственного процесса.

Эта книга никогда не обрела бы столь прекрасный облик без живых и забавных иллюстраций Мартина Муртонена. Его работы однозначно свидетельствуют о творческом потенциале и таланте Мартина. Но даже они не в состоянии передать, какое удовольствие принесла мне наша совместная деятельность.

Однажды я спросил своего друга, Шона Кавану: «Как бы ты написал это на языке Perl?» Шон ответил, что не стал бы этого делать, а предпочел бы воспользоваться Python. Это подвигло меня на изучение нового языка программирования. В процессе учебы Шон отвечал на множество моих вопросов и анализировал первые проекты. Именно он создал и поддерживает программу установки. Я очень ценю его помощь.

Также я хотел бы поблагодарить всех тех, кто помогал готовить рукопись. Это Вибу Чандресекар, Пэм Колхаун, Гордон Колхаун, доктор Тим Купер, Джош Кронемайер, Саймон Кронемайер, Кевин Дрискол, Джеффри Элкнер, Тед Феликс, Девид Гуджер, Лиза Л. Гудьер, доктор Джон Грейсон, Мишель Хаттон, Хорст Дженс, Энди Джадкис, Кейден Кумар, Энтони Лифанте, Шеннон Мэдисон, Кеннет Макдональд, Эван Морис, профессор Александр Реппенинг, Андре Робердж, Кари Дж. Штелпфлуг, Керби Урнер и Брайан Вайнгартен.

Их усилия позволили сделать конечный результат лучше, чем можно было бы ожидать.

Уоррен Сэнд

Я хотел бы поблагодарить Мартина Муртонена, нарисовавшего замечательную карикатуру на меня, маму, позволившую мне сесть за компьютер, когда мне было всего два года, и подавшую идею этой книги, а главное, моего папу, приложившего массу усилий в ходе нашей работы над книгой и научившего меня программировать.

Картер Сэнд

Дополнение ко второму изданию

В обновлении книги «Hello World!» мне помогали многие члены команды, работавшей над первым изданием. Но кроме тех, кого я уже упомянул, мне хотелось бы поблагодарить еще и тех, кто помогал выверять второе издание. Среди них Бен Умс, Брайан Т. Янг, Коди Розброу, Дэйв Бричетти, Элизабет Гордон, Айрис Фаравей, Мэйсон Дженкинс, Рик Гордон, Шон Штебнер и Захари Янг. Огромное спасибо также Игнасио Белтрану-Торресу и Даниэлю Солтису за тщательную корректуру рукописи перед отправкой ее в печать.

Еще мне хотелось бы поблагодарить всех сотрудников издательства *Manning*, благодаря которым второе издание книги «Hello World!» получилось даже лучше первого.

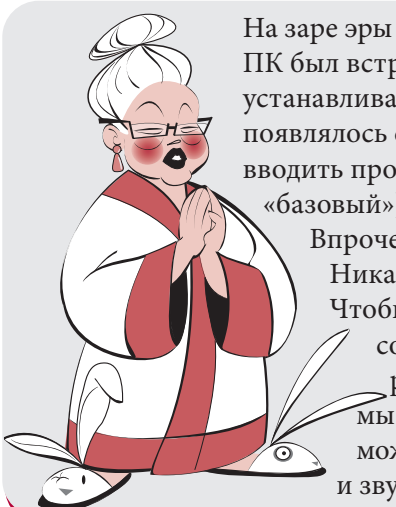
Приступая к работе

Установка Python

Первое, что нужно сделать, — это установить Python на свой компьютер.

Установка Python довольно проста. *Мы настоятельно рекомендуем воспользоваться нашей программой установки Hello World*, предлагающей ту версию Python, которая лучше всего подходит для работы с этой книгой. Вы без труда найдете ее на сайте ***www.manning.com/books/hello-world-second-edition***. При выборе программы установки учитывайте свою операционную систему.

В старые добрые времена



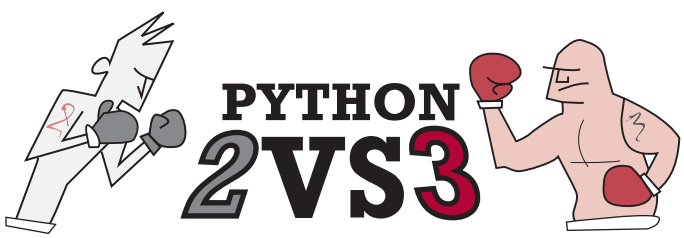
На заре эры персональных компьютеров (ПК) все было просто. Во многие ПК был встроен язык программирования BASIC. Ничего не требовалось устанавливать — достаточно было включить компьютер. На экране появлялось сообщение READY («Готов»), и можно было начинать вводить простые программы (а «basic» дословно и означает «простой», «базовый»). Здорово, правда?

Впрочем, кроме той надписи READY ничего больше и не было. Никаких программ, никаких диалоговых окон, никаких меню. Чтобы компьютер сделал хоть что-нибудь, нужно было писать собственную программу! Не существовало ни текстовых редакторов, ни медиаплееров, ни веб-браузеров, ничего, к чему мы так привыкли сегодня. Не было даже Интернета, в котором можно было бы что-то найти. Не было красивой графики, не было и звукового сопровождения, кроме редкого бипа в случае ошибки.

Существуют версии Python для Windows, Mac OS X и Linux. Все примеры, приведенные в данной книге, рассчитаны на Windows, однако работа с Python в системах Mac OS X или Linux очень схожа. Просто следуйте инструкциям на сайте, чтобы запустить нужную для своей системы программу установки.

Если интерпретатор Python уже установлен, убедитесь, что имеется и ряд «дополнений». На сайте (*www.manning.com/books/hello-world-second-edition*) загляните в раздел, посвященный установке. Самый надежный способ — использовать программу установки с сайта *www.manning.com/books/hello-world-second-edition*.

В этой книге мы используем версию Python 2.7.3. Если вы воспользуетесь упомянутой программой установки с нашего сайта, вы получите именно эту версию. Возможно, вскоре появятся новые версии Python. Однако все примеры в данной книге были проверены именно на версии 2.7.3. Скорее всего, они будут работать и со следующими версиями формата 2.x, но мы не можем предсказывать будущее, поэтому никаких гарантий не даем.



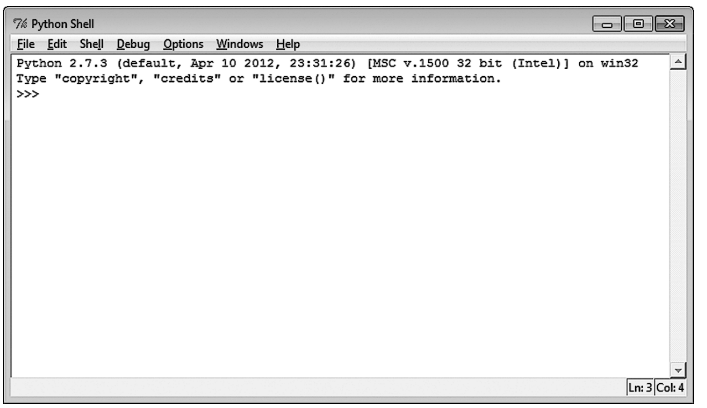
За несколько лет до появления этой книги была выпущена новая версия Python — Python 3. Однако она оказалась не столько усовершенствованной версией уже существующего языка, сколько новым его ответвлением. Поэтому многие не захотели переходить на Python 3, а остались с Python 2. Разработчики Python продолжали выпускать новые версии как Python 2, так и Python 3. К моменту выхода второго издания этой книги последними версиями Python были 2.7.3 и 3.3.0.

В этой книге используется версия 2.7.3, и код, скорее всего, будет работать и в последующих версиях 2.x.

Начинаем знакомство с Python со среды IDLE

Существует несколько способов начать работу с Python. Один из них предполагает применение IDLE, и именно им мы пока что воспользуемся.

В меню *Пуск*, в разделе *Python 2.7*, имеется команда *IDLE (Python GUI)*. Щелкните по ней мышью, чтобы открыть окно среды IDLE. Оно должно выглядеть примерно так:



IDLE (Integrated DeveLopment Environment — интегрированная среда разработки) представляет собой *оболочку* (shell) Python. Оболочка — это, по большому счету, средство взаимодействия с программой с помощью ввода текста. И эта самая оболочка позволяет взаимодействовать с Python — именно поэтому в заголовке окна вы видите надпись *Python Shell*. IDLE — это еще и графический интерфейс пользователя (GUI), поэтому в меню *Пуск* имеется команда *Python GUI*. Впрочем, среда IDLE не ограничивается оболочкой, и совсем скоро мы рассмотрим другие ее аспекты.

Значки >>> на картинке выше — это *приглашение* к вводу команды. Приглашение программа выводит на экран, когда она ждет, чтобы вы что-то ввели. Приглашение >>> говорит нам о том, что интерпретатор Python ожидает от вас дальнейших директив.

НОВЫЕ СЛОВА

Под *графическим интерфейсом пользователя* (Graphical User Interface, GUI) понимается совокупность различных графических элементов, таких как окна, меню, кнопки, полосы прокрутки и т. д. Программы, у которых нет такого интерфейса, называются программами *с текстовым интерфейсом*, *консольными* программами или программами с интерфейсом *командной строки*.

Директивы, пожалуйста!

Давайте дадим интерпретатору Python нашу первую директиву. Установите курсор после приглашения >>> и введите следующее:

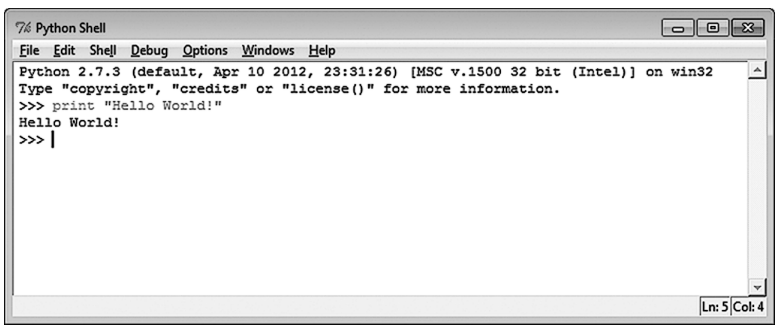
```
print "Hello World!"
```

Затем нажмите клавишу *Enter* (на некоторых клавиатурах она называется *Return*). Клавишу *Enter* необходимо нажимать после ввода каждой строки.

После нажатия клавиши *Enter* на экране должно появиться следующее:

```
Hello World!
>>>
```

На рисунке показано, как это будет выглядеть в окне среды IDLE.



Интерпретатор Python выполнил заданную директиву — он напечатал (`print` буквально означает «печатать») заданное сообщение. (Обратите внимание, что в программировании «напечатать» часто означает не только распечатать что-нибудь на принтере, но и вывести текст на экран и даже набрать на клавиатуре.) Вот эта строка для интерпретатора Python и есть *директива*. Итак, мы уже начали программировать! Теперь компьютер в нашей власти!

Кстати, у начинающих программистов есть такая традиция: первое, что они делают, — это выводят на экран сообщение «Hello World!», что в переводе означает «Привет, мир!» Вот откуда появилось название этой книги. Мы тоже следуем давней традиции. Добро пожаловать в мир программирования!

Хороший вопрос! IDLE пытается помочь вам во всем разобраться. Разными цветами выделяются разные части кода. (*Код* — это просто другое название директив, которые вы даете компьютеру на том или ином языке программирования, например на Python.) Мы узнаем, что означают эти цвета, немного попозже.

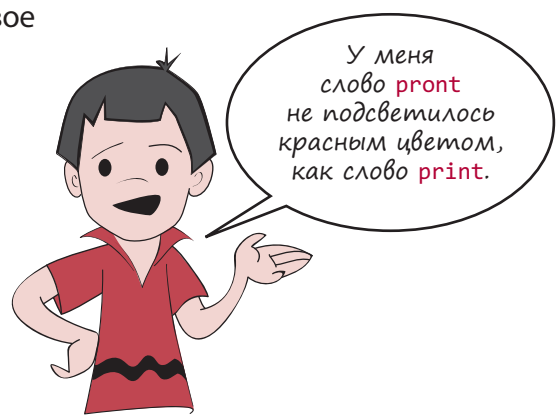
Если ничего не работает

В случае ошибки на экране появляется что-то вроде:

```
>>> pront "Hello World!"
SyntaxError: invalid syntax
>>>
```

Это сообщение об ошибке означает, что введена директива, которую Python не может понять. В данном примере слово `print` написано неправильно (`pront`), и Python не знает, что с ним делать. Если такое случится, попробуйте ввести команду еще раз и обязательно убедитесь, что все слова написаны так же, как в примере.

Все верно. Это произошло потому, что `print` — ключевое слово в Python, а `pront` — нет.



НОВЫЕ СЛОВА

Ключевое слово — это специальное слово, которое является частью Python (его еще называют *зарезервированным* словом).

Взаимодействие с Python

Только что мы работали с Python в интерактивном режиме. Мы ввели команду (директиву), и интерпретатор Python ее тут же *выполнил*.

НОВЫЕ СЛОВА

Выполнить команду, директиву или программу означает то же самое, что и «запустить» ее.

Давайте попробуем выполнить что-нибудь еще в режиме взаимодействия. После приглашения >>> введите такую команду:

```
>>> print 5+3
```

В результате на экране должно появиться следующее:

```
8
>>>
```

Оказывается, Python умеет складывать! Это не должно вас удивлять, ведь компьютеры сильны в арифметике.

Давайте попробуем еще:

```
>>> print 5*3
15
>>>
```

Практически во всех компьютерных программах и языках программирования символ * используется в качестве знака умножения. Этот знак называется *астериск*, или *звездочка*.

Если на уроках математики вы привыкли писать «5 умножить на 3» или «5 × 3», теперь для умножения в Python вам нужно привыкать использовать знак *. (На большинстве клавиатур этот знак находится на той же клавише, что и цифра 8.)

Хорошо, а как насчет этого?

```
>>> print 2345*6789
15920205
>>>
```

Ну а как теперь?

```
>>> print 12345678987654321234567890
* 98765432123456789876543210
12193263200731596000609652202408166072245112635269
>>>
```

Правильно. С помощью компьютера можно производить арифметические операции с очень и очень большими числами.

Вот еще кое-что, на что способен интерпретатор Python:

```
>>> print "cat" + "dog"
Catdog
>>>
```

А попробуйте ввести такую команду:

```
>>> print "Hello " * 20
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
```

Помимо арифметики компьютеры очень хорошо справляются с повторяющимися заданиями. В этом примере мы приказали Python напечатать слово «Hello» 20 раз.

Мы еще поработаем в интерактивном режиме попозже, а пока…

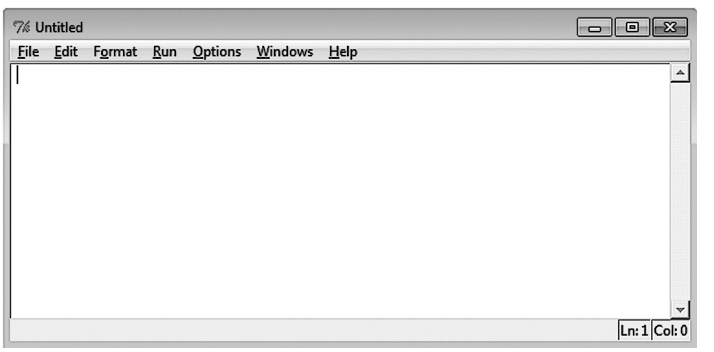
Приступим к программированию

Примеры, которые мы рассматривали до этого, были отдельными директивами для Python (в интерактивном режиме). И хотя они наглядно демонстрируют некоторые возможности Python, это не настоящие программы. Как мы уже отмечали, программа — это набор директив, собранных вместе. Итак, давайте напишем нашу первую программу на Python.

Когда речь идет о пунктах меню, как, например, в случае с *File* ▸ *New*, первая часть (в данном случае *File*) обозначает главное меню. Значок ▸ говорит нам о том, что следующий пункт (в данном случае *New*) находится в меню *File*. Такие обозначения будут встречаться в этой книге и дальше.

Прежде всего программу нужно где-то записать. Если просто вводить программу в интерактивном окне, Python ее не «запомнит». Для этого нам потребуется текстовый редактор (такой как Блокнот в Windows, TextEdit в Mac OS X или vi в Linux), который сумеет сохранить программу на жестком диске. В среде IDLE уже есть встроенный текстовый редактор, и он подходит нам гораздо больше, чем, скажем, Блокнот. Чтобы его открыть, выберите в меню среды IDLE команду *File ▸ New Window*.

Откроется окно, показанное на следующем рисунке. В его заголовке написано *Untitled* («Безымянный»), потому что мы еще не дали название файлу.



Теперь наберем нашу первую программу в редакторе:

Листинг 1.1. Наша первая настоящая программа

```
print "I love pizza!"
print "pizza" * 20
print "yum" * 40
print "I'm full."
```

Когда закончите, сохраните программу, выбрав в меню команду *File ▸ Save* или *File ▸ Save as*. Назовите файл *pizza.py*. Можете сохранить его где угодно (только не забудьте, где, чтобы его можно было легко найти в дальнейшем). Возможно, вам стоит создать отдельную папку для Python-программ. Буквы *.py* в конце имени файла очень важны, потому что они говорят компьютеру, что это программа, написанная на Python, а не какой-то забытый текстовый файл.

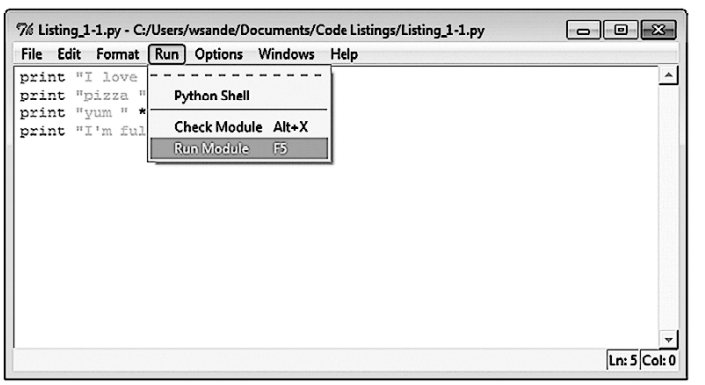
Вы, наверное, уже заметили, что редактор использует различные цвета для разных слов в программе. Некоторые выделены красным, другие — зеленым. Это все

Обратите внимание на заголовок «Листинг 1.1». Все примеры кода, образующие полноценные программы на Python, будут пронумерованы таким образом для того, чтобы вы смогли легко найти их в папке *\examples* или на веб-сайте.

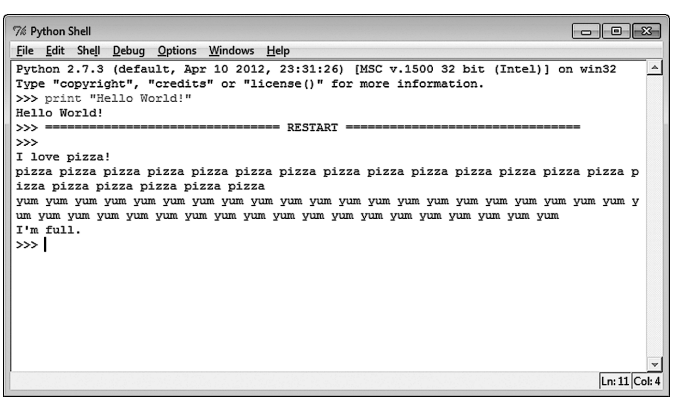
потому, что редактор среды IDLE предполагает, что программа пишется на Python. В таких программах редактор выделяет ключевые слова языка красным, а все, что заключено в кавычки, — зеленым цветом. Это сделано для того, чтобы было проще читать Python-код.

Запуск нашей первой программы

После сохранения своей программы откройте меню *Run* (в IDLE-редакторе) и выберите команду *Run Module* (как показано на следующем рисунке). В результате программа будет запущена.



Окно *Python Shell* (то, что первым появилось при запуске среды IDLE) снова станет активным, и на экране появится что-то наподобие этого:



Строка со словом RESTART говорит о том, что программа была запущена. (Она пригодится, когда вы будете многократно тестировать и запускать свои программы.)

За этой строкой следуют выводимые программой данные. Да, она делает пока не слишком много, но нам удалось добиться от компьютера того, что мы хотели. Наши программы со временем будут становиться все более и более интересными.

Если что-то идет не так

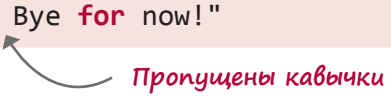
Что произойдет, если в коде обнаружится ошибка и программа не выполнится? Существует два вида ошибок, которые могут произойти. Давайте рассмотрим оба варианта, чтобы знать, что делать в обоих случаях.

синтаксические ошибки

IDLE проверяет вашу программу еще до ее запуска. Если обнаруживается какая-то ошибка, то это, как правило, *синтаксическая ошибка*. Синтаксис — это правила орфографии и грамматики в языке программирования, поэтому синтаксическая ошибка означает, что вы написали что-то, не соответствующее правилам Python.

Приведем пример:

```
print "Hello, and welcome to Python!"
print "I hope you will enjoy learning to program."
print Bye for now!"
```



Мы пропустили кавычки между словом `print` и строкой `"Bye for now!"`.

Если вы попробуете запустить эту программу, то увидите сообщение: «There's an error in your program: invalid syntax» («Ошибка в программе: неверный синтаксис»). После этого нужно проверить свой код, чтобы понять, что с ним не так. Редактор среды IDLE выделит красным цветом место ошибки. Возможно, это не будет сама ошибка, но она должна присутствовать где-то поблизости.

ошибки выполнения

Второй вид ошибок — это те ошибки, которые Python (или IDLE) не может обнаружить до запуска программы. Такие ошибки проявляются только при выполнении программы, поэтому они называются *ошибками выполнения*. Приведем пример подобной ошибки в программе:

```
print "Hello, and welcome to Python!"
print "I hope you will enjoy learning to program."
print "Bye for now!" + 5
```

Если сохранить эту программу и попробовать ее выполнить, она запустится. Первые две строки появятся на экране, но затем последует сообщение об ошибке:

```
>>> ============================ RESTART ============================
>>>
Hello, and welcome to Python!
I hope you will enjoy learning to program.

Traceback (most recent call last):
  File "C:/HelloWorld/examples/error1.py", line 3, in <module>
    print "Bye for now!" + 5
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

Начало сообщения об ошибке

Место ошибки

Неправильный код

Причина ошибки по мнению интерпретатора Python

Строка, которая начинается со слова `Traceback`, — это начало сообщения об ошибке. Следующая строка указывает на место ошибки — название файла и номер строки. Затем идет строка с неверным кодом. Это помогает обнаружить проблему в коде. Последняя часть сообщения об ошибке указывает на то, в чем, собственно, ошибка. Когда мы поближе познакомимся с Python и программированием, нам будет легче понять смысл этого сообщения.

Ну, Картер, это похоже на старую загадку про свиней и ежей. В Python нельзя сложить два объекта разных типов, например текст и число. Именно поэтому команда `print "Bye for now!" + 5` возвращает ошибку. Это все равно что сказать: «Если к пяти свиньям прибавить трех ежиков, сколько чего у меня будет?» У нас будет восемь, но восемь чего? Сложение в данном случае бессмысленно. Однако мы можем умножать практически все, что угодно, чтобы получить большее количество чего-либо. (Если двух ежиков умножить на 5, то у нас получится десять ежиков!) Вот поэтому команда `print "Bye for now!" * 5` работает правильно.

Наша вторая программа

Первая программа ничего особенного не делала. Она просто вывела кое-что на экран. Давайте попробуем сотворить что-нибудь поинтереснее. Код в листинге 1.2 — это простая игра, в которой нужно угадать число. Создайте новый файл в IDLE-редакторе так же, как мы это делали в первый раз, то есть командой *File ▸ New Window*. Введите код из листинга 1.2, а затем сохраните его. Вы можете назвать файл с кодом как угодно, главное, чтобы название заканчивалось расширением *.py*. Ну, например *NumGuess.py*.

В программе всего 18 директивных строк для Python плюс несколько пустых строк, чтобы было удобнее читать. У вас не должно уйти много времени на то, чтобы набрать весь код. Не переживайте, если не поймете, что означает каждая из директив. Совсем скоро мы дойдем и до этого.

Листинг 1.2. Игра «Угадай число»

```
import random
secret = random.randint(1, 99)
guess = 0
tries = 0
print "Эй на палубе! Я Ужасный пират Робертс, и у меня есть секрет!"
print "Это число от 1 до 99. Я дам тебе 6 попыток."
while guess != secret and tries < 6:
    guess = input("Твой вариант?")
    if guess < secret:
        print "Это слишком мало, презренный пес!"
    elif guess > secret:
        print "Это слишком много, сухопутная крыса!"

    tries = tries + 1

if guess == secret:
    print "Хватит! Ты угадал мой секрет!"
else:
    print "Попытки кончились!"
    print "Это число ", secret
```

Выбираем секретное число

Получаем мнение игрока

Разрешаем до 6 попыток

Подсчитываем число попыток

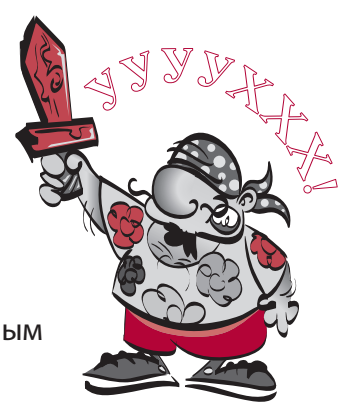
В конце игры выводим сообщение на экран

Когда будете набирать код, обратите внимание на отступы в строках, следующих за командой `while`, и на еще большие отступы в строках, следующих за командами `if` и `elif`. Также обратите внимание на двоеточия в конце некоторых строк. Если поставить двоеточие в нужном месте, редактор поможет вам, начав новую строку с отступа.

Сохраните код и запустите программу командой ***Run ▸ Run Module***, как мы это делали в прошлый раз. Попробуйте поиграть в игру и посмотреть, что получится. Вот что вышло у меня:

```
>>> ====================== RESTART ======================
>>>
Эй на палубе! Я ужасный пират Робертс, и у меня есть секрет!
Это число от 1 до 99. Я дам тебе 6 попыток.
Твой вариант? 40
Это слишком много, сухопутная крыса!
Твой вариант? 20
Это слишком много, сухопутная крыса!
Твой вариант? 10
Это слишком мало, презренный пес!
Твой вариант? 11
Это слишком мало, презренный пес!
Твой вариант? 12
Хватит! Ты угадал мой секрет!
>>>
```

Мне понадобилось 5 попыток, чтобы угадать число 12.



С командами `while`, `if`, `else`, `elif` и `input` мы познакомимся в следующих главах. Но уже сейчас вы, наверное, поняли в общих чертах, как работает программа.

1. Программа выбирает случайное секретное число.
2. Пользователь пытается его угадать.
3. Программа сравнивает каждое предположение с секретным числом: оно больше или меньше?
4. Пользователь продолжает попытки, пока он не угадает число или пока попытки не закончатся.

Когда предположение совпадает с секретным числом, игрок выигрывает.

Что мы узнали

Уф! Мы прошли довольно длинный путь. В этой главе мы:

- установили Python;
- узнали, как запустить IDLE;
- познакомились с интерактивным режимом;
- дали интерпретатору Python несколько директив, и он их выполнил;
- убедились, что Python умеет выполнять арифметические операции (включая операции с очень большими числами!);
- запустили текстовый редактор среды IDLE, в котором написали свою первую программу;
- запустили свою первую программу;
- познакомились с сообщениями об ошибках;
- запустили свою вторую Python-программу — игру «Угадай слово».

Проверь себя[1]

1. Как запустить среду IDLE?
2. Что делает команда `print`?
3. Каким символом в Python обозначается умножение?
4. Что показывает IDLE при запуске программы?
5. Какое еще слово иногда используют вместо слова «запустить» (программу)?

Эксперименты

1. В интерактивном режиме используйте Python для вычисления количества минут в неделе.
2. Напишите короткую программу для вывода на экран трех строк: ваших имени, даты рождения и любимого цвета. Выводимые программой данные должны выглядеть примерно так:

```
Меня зовут Уоррен Сэнд.
Я родился 1 января 1970.
Мой любимый цвет синий.
```

   Сохраните и запустите программу. Если программа сделает что-то не то, что вы от нее ожидаете, или вы получите какие-то сообщения об ошибках, попытайтесь исправить эти ошибки и заставить программу работать должным образом.

[1] Ответы на вопросы из разделов «Проверь себя» доступны для скачивания по адресу http://goo.gl/VPoZ48.

ГЛАВА 2

Запомните это: память и переменные

Что такое программа? Хотя подождите! Разве мы не ответили на этот вопрос в главе 1? Мы говорили, что программа — это последовательность директив для компьютера.

И это действительно так. Но почти все делающие что-то полезное или забавное программы обладают дополнительными характеристиками:

- они поддерживают *ввод* данных;
- они *обрабатывают* введенные данные;
- они обеспечивают *вывод* информации.

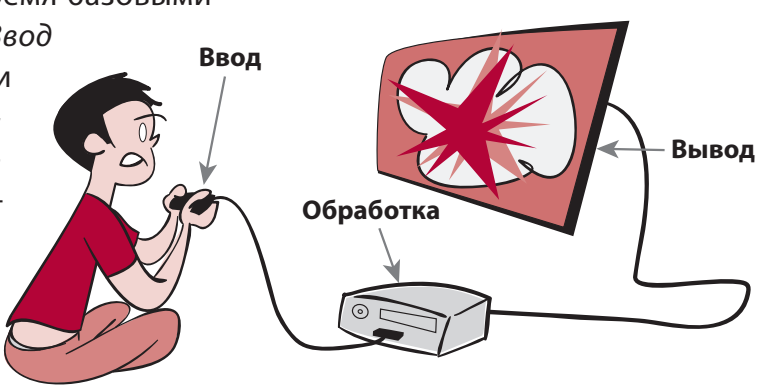
Ввод, обработка, вывод

В нашей первой программе (см. листинг 1.1) не было ни входных данных, ни их обработки. Поэтому она не очень интересна. Результатом ее работы стал вывод сообщения на экран.

Следующая программа — игра в угадывание чисел — будет обладать всеми тремя базовыми элементами.

- Игрок вводит угадываемое число.
- Программа проверяет правильность догадки и считает очередь.
- Программа выводит сообщение с результатом.

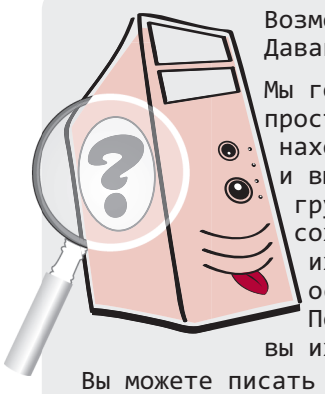
Другим примером программы с тремя базовыми элементами является видеоигра. *Ввод* осуществляется через джойстик или игровой контроллер, в процессе *обработки* программа определяет, удалось ли вам убить монстра, уклониться от огненного шара и перейти на следующий уровень, а *вывод* представляется картинкой на экране и звуком в наушниках.



Итак, запомните последовательность: ввод, обработка, вывод.

А что компьютер делает с вводимыми данными? Чтобы выполнить с ними какие-либо действия, компьютер сначала должен их запомнить, то есть где-то сохранить. Всю информацию, в том числе входные данные (а также саму программу), компьютер хранит в своей памяти.

ЧТО ПРОИСХОДИТ ВНУТРИ



Возможно, вы слышали про память компьютера. Давайте посмотрим, что означает этот термин.

Мы говорили, что компьютер представляет собой просто набор переключателей, которые могут находиться в двух состояниях — включенном и выключенном. Память можно представить как группу таких переключателей, некоторое время сохраняющих свое положение. Вы устанавливаете их в определенную позицию, и они в ней остаются, пока вы что-нибудь не поменяете. Переключатели помнят, в какое положение вы их поставили. Они обладают памятью!

Вы можете писать в память (задавать положение переключателей) или читать из нее (смотреть, в какое положение установлены переключатели, не внося никаких изменений).

Но как объяснить интерпретатору Python, в какое место памяти мы хотим поместить фрагмент данных? И как потом его найти?

Если вы хотите, чтобы программа на Python что-то запомнила и вы смогли этим пользоваться в дальнейшем, нужно присвоить этому имя. Что бы это ни было — число, текст, картинка или музыкальный отрывок, — интерпретатор Python выделит для него место в компьютерной памяти. В дальнейшем сослаться на этот фрагмент данных можно, назвав его имя.

Воспользуемся интерактивным режимом интерпретатора Python и посмотрим более подробно, что представляют собой имена.

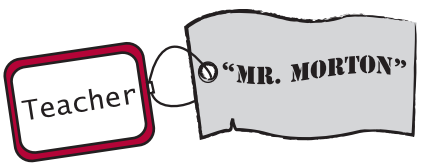
Имена

Вернитесь в окно Python Shell (если вы закрыли среду IDLE после выполнения упражнений главы 1, снова откройте ее). После появления приглашения наберите:

```
>>> Teacher = "Mr. Morton"
>>> print Teacher
```

Запомните, что знаки >>> являются приглашением на ввод Python-команды. Наберите следующий за этим знаком текст и нажмите клавишу *Enter*. Вот что должно получиться:

```
Mr. Morton
>>>
```



Мы только что создали элемент, состоящий из букв "Mr. Morton", и присвоили ему имя Teacher.

Знак равенства (=) заставляет интерпретатор Python выполнить присваивание, то есть «сделать что-то равным чему-то». Вы присвоили набору букв "Mr. Morton" имя Teacher.

Буквы "Mr. Morton" существуют в каком-то фрагменте памяти вашего компьютера. Точное местоположение не имеет значения. Вы объяснили интерпретатору Python, что в дальнейшем будете ссылаться на эти буквы по имени Teacher.

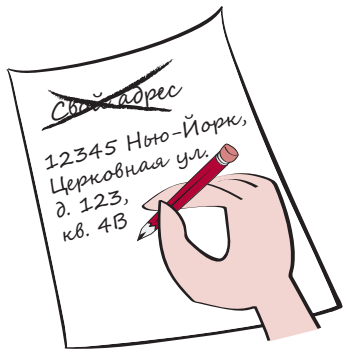
Подобно метке, ярлыку или наклейке, имя предназначено для идентификации объекта.

Кавычки интерпретатор Python понимает в буквальном смысле. И выводит то, что заключено между ними. При отсутствии кавычек интерпретатор определяет значение фрагмента. Это может быть число (например 5), выражение (5 + 3) или имя (Teacher). В данном случае речь идет об имени Teacher, и интерпретатор Python выводит принадлежащее ему значение, то есть набор букв "Mr. Morton".

Картер, если кто-то тебе скажет: «Напиши свой адрес», ты же не будешь писать так:

(А Картер опять шутит...)

Думаю, что ты написал бы так:



Если вы напишете слова `"Свой адрес"`, это выражение поймут буквально. То есть интерпретатор Python понимает только то, что заключено в кавычки. Вот еще один пример:

```
>>> print "53 + 28"
53 + 28
>>> print 53 + 28
81
```

НОВЫЕ СЛОВА

Арифметическим выражением называется комбинация чисел и символов, значение которой может определить интерпретатор Python.

Вычислить означает всего лишь «определить значение».

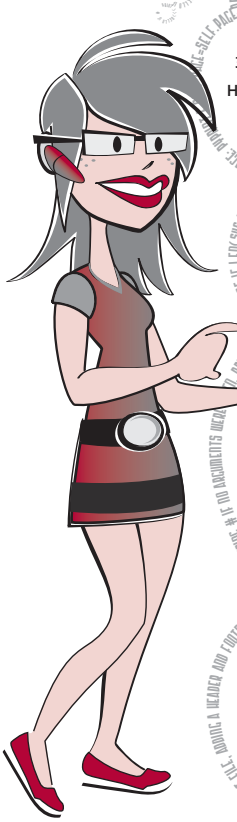
В первом случае кавычки присутствуют, и интерпретатор выводит на экран заключенное в них выражение 53 + 28.

В отсутствии кавычек Python воспринимает 53 + 28 как *арифметическое выражение* и *вычисляет* его значение. В данном случае нужно сложить друг с другом два числа, поэтому на экране появляется их сумма.

Интерпретатор Python сам решает, сколько памяти потребуется для хранения символов и какой частью памяти он воспользуется. Для получения этой информации достаточно снова указать ее имя. Ключевое слово `print` и имя позволят вывести на экран соответствующий элемент (например число или текст).

Переменные можно создавать не только из букв. Можно присвоить имя численному значению. Помните наш предыдущий пример?

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



ДУМАЙ КАК ПРОГРАММИСТ

Когда вы назначаете имени значение (например значение "Mr. Morton" имени Teacher), оно сохраняется в памяти и начинает называться переменной. В большинстве языков программирования говорят, что значение сохранено в переменной. Но Python в этом отношении немного отличается от других языков программирования. Он не сохраняет значения в переменных, а, скорее, дает значениям имена. Некоторые программисты говорят, что в Python отсутствует понятие переменной, а вместо него есть понятие имени. Тем не менее поведение программы в обоих случаях сходное. Но эта книга посвящена программированию в целом, а не только языку Python. Поэтому, говоря об именах Python, мы будем использовать термины «имя» или даже «имя переменной». На самом деле неважно, как вы называете некую сущность, главное, чтобы вы понимали, как она себя ведет и каким образом ее использовать в своих программах. К слову, создатель Python Гвидо ван Россум в своем руководстве пишет: «Знак = используется для присваивания значения переменной». Наверное, он тоже считает,что в Python есть переменные.

ХОРОШИЙ СПОСОБ ХРАНЕНИЯ

Имена в Python напоминают ситуацию с заказами в химчистке. Одежда, помеченная именами владельцев, развешена на плечиках на огромной вращающейся вешалке. Человеку, пришедшему за своим пальто, не нужно знать, где вешалка, на которой оно висит. Он называет приемщику имя и получает вещь. Пальто может находиться не в том месте, куда его повесили при сдаче в чистку. Но это проблемы сотрудников химчистки, а клиенту достаточно назвать имя.

Аналогичным образом обстоят дела и с переменными. Вам неважно, где именно в памяти хранится информация. Можно воспользоваться именем, которое применялось при ее сохранении.

Давайте сделаем то же самое с участием переменных:

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

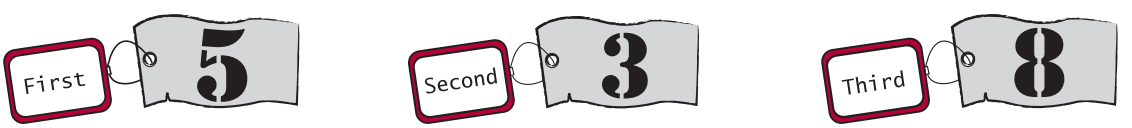
Мы создали два имени `First` и `Second`. Число 5 было присвоено имени `First`, а число 3 — имени `Second`. Затем мы *вывели на экран* сумму этих переменных.

Можно сделать и по-другому:

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

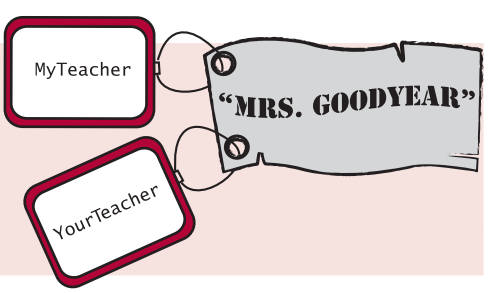
В интерактивном режиме вывести на экран значение переменной можно, просто набрав ее имя, не прибегая к ключевому слову `print`. (В программах такой подход не работает.)

В этом примере суммирование выполнялось не внутри команды `print`. Мы взяли элемент с именем `First` и элемент с именем `Second` и сложили их, попутно создав новый элемент с именем `Third`. `Third` — это сумма переменных `First` и `Second`.

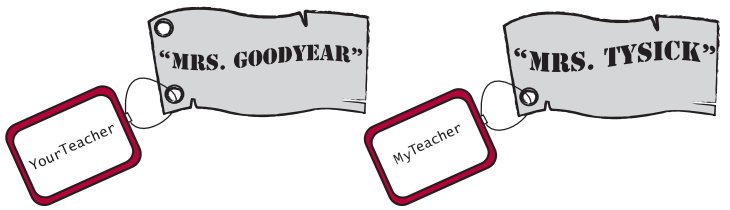


Одному и тому же элементу можно присвоить несколько имен. Проделайте это в интерактивном режиме:

```
>>> MyTeacher = "Mrs. Goodyear"
>>> YourTeacher = MyTeacher
>>> MyTeacher
"Mrs. Goodyear"
>>> YourTeacher
"Mrs. Goodyear"
```



Это все равно что налепить на одну вещь две наклейки. На одной наклейке написано `YourTeacher`, а на другой — `MyTeacher`, но обе они присоединены к элементу `"Mrs. Goodyear"`.



Хороший вопрос, Картер. И ответ на него отрицательный. При этом будет создан новый элемент `"Mrs. Tysick"`. Ярлык `MyTeacher` отсоединится от значения `"Mrs. Goodyear"` и присоединится к значению `"Mrs. Tysick"`. У вас по-прежнему два разных элемента (два ярлыка), но теперь они присоединены к двум разным вещам, а не к одной.

Что такое имя

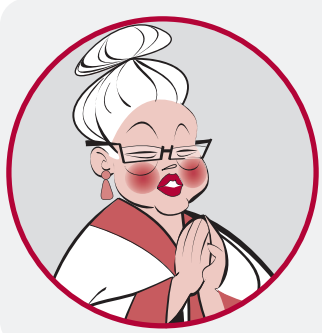
Переменные допустимо именовать как угодно (ну, почти как угодно). Имя может иметь произвольную длину, состоять из цифр и букв, а также символа нижнего подчеркивания (_).

Тем не менее при именовании нужно соблюдать ряд правил. Самым важным является чувствительность к регистру. Важно, какими буквами — прописными или строчными — набрано имя. Поэтому `teacher` и `TEACHER` — два разных имени. Аналогично `first` и `First`.

Кроме того, имя переменной должно начинаться с буквы или с символа подчеркивания. С числа имя начинаться не может. Соответственно, имя `4fun` является недопустимым.

И еще в имени не должно быть пробелов.

В СТАРЫЕ ДОБРЫЕ ВРЕМЕНА



В некоторых ранних языках программирования имена переменных обозначались всего одним символом. При этом существовали компьютеры, поддерживающие только буквы верхнего регистра, то есть существовало всего 26 вариантов названий переменных: A–Z! Если программа требовала более 26 переменных, программисту можно было только посочувствовать.

Числа и строки

Итак, мы умеем создавать переменные для букв (текста) и чисел. Но откуда интерпретатор Python знает, что мы имели в виду числа 5 и 3, а не символы `"5"` и `"3"`? Правильно, все снова зависит от наличия кавычек.

Символ или набор символов (букв, цифр и знаков препинания) называется *строкой*.

Чтобы объяснить интерпретатору Python, что вы создаете строку, нужно заключить символы в кавычки. В Python вы можете пользоваться для этой цели как одинарными, так и двойными кавычками. Допустимы оба варианта:

```
>>> teacher = "Mr. Morton"
>>> teacher = 'Mr. Morton'
```

Двойные кавычки

Одинарные кавычки

Но кавычки в начале и в конце строки должны быть одного типа.

Число без кавычек интерпретатор Python воспринимает как численное значение, а не как символ. Попробуйте сделать так, чтобы почувствовать разницу:

```
>>> first = 5
>>> second = 3
>>> first + second
8
>>> first = '5'
>>> second = '3'
>>> first + second
'53'
```

Без кавычек значения 5 и 3 интерпретируются как цифры, и вы получаете в итоге сумму. Кавычки превращают `"5"` и `"3"` в строки, и на выходе мы видим два «сложенных» друг с другом символа `"53"`.

Еще вы можете объединять друг с другом строки из букв, как было показано в главе 1:

```
>>> print "cat" + "dog"
catdog
```

Обратите внимание, что при подобном объединение строк между ними не оказывается пробела. Строки просто склеиваются друг с другом.

СТРАННОЕ ДЛИННОЕ СЛОВО

Конкатенация

Глагол «складывать» не совсем корректен, когда речь идет о строках. Для обозначения процесса стыковки символов или строк друг с другом с целью получения более длинной строки существует специальный термин. Этот процесс называется не «сложением» (термин, допустимый только для цифр), а *конкатенацией*.

ДЛИННЫЕ СТРОКИ

Для реализации многострочных фрагментов применяется особый вид строки с тремя кавычками. Вот как она выглядит:

```
long_string = """ Много, много птичек запекли в пирог:
Семьдесят синичек, сорок семь сорок.
Трудно непоседам в тесте усидеть, птицы за обедом громко стали петь.
Побежали люди в золотой чертог,
Королю на блюде понесли пирог."""
```

Такие строки начинаются и заканчиваются тремя кавычками. Кавычки могут быть как одинарными, так и двойными, то есть предыдущий фрагмент можно записать и таким образом:

```
long_string = ''' Много, много птичек запекли в пирог:
Семьдесят синичек, сорок семь сорок.
Трудно непоседам в тесте усидеть, птицы за обедом громко стали петь.
Побежали люди в золотой чертог,
Королю на блюде понесли пирог.'''
```

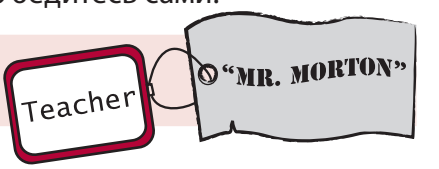
Подобная конструкция крайне полезна, если у вас есть несколько строк текста, который нужно вывести на экран одним фрагментом без разрывов.

НАСКОЛЬКО ОНИ «ПЕРЕМЕННЫЕ»

Переменные получили такое название не просто так. Дело в том, что они меняются! То есть назначенное переменной значение можно изменить. В Python для этого создается новый элемент, которому присваивается старая метка (имя). Именно так мы поступили с переменной `MyTeacher` в последнем разделе. Мы отобрали ярлык `MyTeacher` у строки `"Mrs. Goodyear"` и присоединили его к новой строке — `"Mrs. Tysick"`. То есть мы назначили переменной `MyTeacher` новое значение.

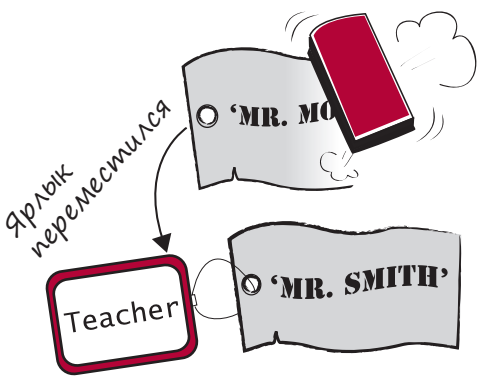
Рассмотрим пример. Помните созданную нами переменную `Teacher`? Если вы не закрыли среду IDLE, ее значение можно будет увидеть на экране. Убедитесь сами:

```
>>> Teacher
'Mr. Morton'
```



Мы можем ее поменять:

```
>>> Teacher = 'Mr. Smith'
>>> Teacher
'Mr. Smith'
```



Мы создали новый элемент `"Mr. Smith"` и присвоили ему имя `Teacher`. Ярлык переместился со старого элемента на новый. Но что случилось со старым элементом `"Mr. Morton"`?

Помните, что элементы могут иметь несколько имен (более одного связанного с ними ярлыка). Если с элементом `"Mr. Morton"` все еще связан какой-то ярлык, он остается в памяти компьютера. Но элемент, с которым не связано никаких ярлыков, Python считает ненужным и удаляет из памяти.

Благодаря этому память освобождается от неиспользуемых элементов. Удаление выполняется автоматически, вам об этом беспокоиться не нужно.

Важно понять, что на самом деле мы не превращали элемент `"Mr. Morton"` в элемент `"Mr. Smith"`. Мы просто переместили ярлык (переназначили имя) с одного элемента на другой. В Python есть сущности (например числа и строки), которые не допускают изменений. Можно присваивать их имена другим элементам (как мы только что сделали), но изменить значение исходного элемента у вас не получится.

В Python присутствуют и другие неизменные сущности. Их мы рассмотрим в главе 12, когда речь пойдет о списках.

Новый я

Переменную также можно приравнять самой себе:

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

Держу пари, что вы сочли эту операцию бесполезной. И вы совершенно правы. Это все равно что утверждать: «Я — это я». Но небольшое изменение может сделать вас совершенно новой сущностью:

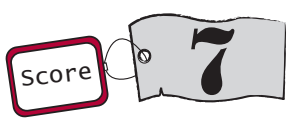
```
>>> Score = Score + 1
>>> print Score
8
```

Меняет переменную Score с 7 на 8

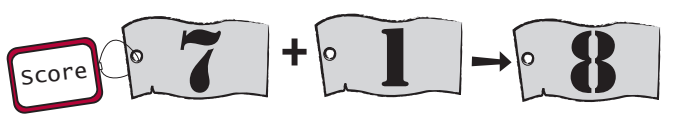
Что в этом случае происходит? В первой строке ярлыку `Score` присваивается значение 7. Затем мы создаем новый элемент `Score + 1`, или 7 + 1. Он имеет значение 8. После этого ярлык `Score` отбирается от старого элемента (7) и присоединяется к новому (8). Фактически переменной `Score` присваивается новое значение.

В выражении присваивания переменная всегда фигурирует слева от знака равенства (=). Но она может появиться и справа. Это крайне полезное свойство, с которым вы столкнетесь во многих программах. Чаще всего оно применяется в операции инкремента переменной (увеличения на определенную величину) или в противоположной операции — декременте (уменьшении на определенную величину).

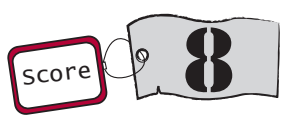
1. Начинаем со `Score = 7`.



2. Создаем новый элемент, добавляя 1 (что дает значение 8).



3. Присваиваем новому элементу имя `Score`.



В результате переменная `Score` меняет свое значение с 7 на 8.

Вот что важно помнить о переменных.

- Переменной в любом месте программы можно присвоить другое значение (ярлык перемещается на новый элемент). Одной из наиболее распространенных ошибок в программировании является изменение не той переменной или изменение нужной переменной не в то время.

  Для предотвращения подобной ситуации имеет смысл пользоваться легко запоминаемыми именами. Ничто не мешает нам написать:

```
t = 'Mr. Morton'
```

или

```
x1796vc47blahblah = 'Mr. Morton'
```

Но в этом случае нам будет сложно запомнить фигурирующие в программе переменные. И при работе с ними с большей вероятностью будут возникать ошибки. Поэтому пользуйтесь осмысленными именами, указывающими на предназначение переменной.

- Имена переменных чувствительны к регистру. Это означает, что программа различает строчные и прописные буквы. И с ее точки зрения, `teacher` и `Teacher` — разные имена.

ДУМАЙ КАК ПРОГРАММИСТ

Профессиональные программисты на Python практически всегда начинают имена переменных с символа нижнего регистра. В других языках программирования может быть принята другая практика. Подобная договоренность является необязательной, и если вы не хотите, можете ей не следовать. Но так как данная книга посвящена Python, мы будем придерживаться данного стиля именования.

Что мы узнали

В этой главе мы узнали:

- как «запоминать», или сохранять, фрагменты данных в памяти компьютера при помощи переменных;
- что переменные допустимо называть «именами» или «именами переменных»;
- что переменные существуют для различных элементов, например для чисел и строк.

Проверь себя

1. Как объяснить интерпретатору Python, что переменная представляет собой не число, а строку (набор символов)?
2. Можно ли изменить значение, присвоенное созданной вами переменной?

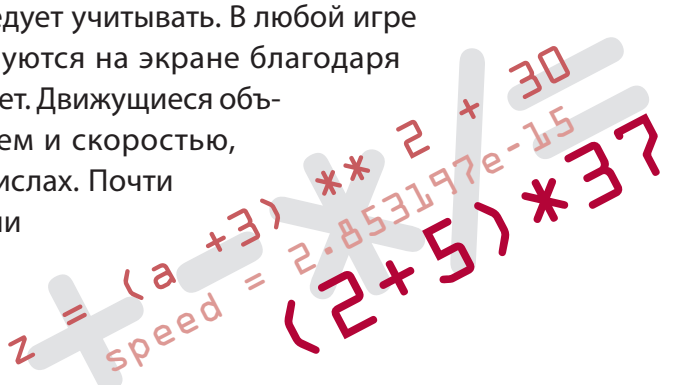
3. Имена `TEACHER` и `TEACHEr` принадлежат одной или разным переменным?
4. Одинаковы ли для интерпретатора Python записи `'Blah'` и `"Blah"`?
5. Считает ли интерпретатор Python, что `'4'` равно 4?
6. Какое из имен недопустимо? Почему?
   1) `Teacher2`
   2) `2Teacher`
   3) `teacher_25`
   4) `TeaCher`
7. `"10"` — это число или строка?

Эксперименты

1. Создайте переменную и присвойте ей число (любое на ваш выбор). Выведите его на экран командой `print`.
2. Отредактируйте значение этой переменной, заменив старое новым или добавив к старому значению некую величину. Отобразите новое значение командой `print`.
3. Создайте еще одну переменную и присвойте ей строку (произвольный текст). Выведите ее на экран командой `print`.
4. В интерактивном режиме вычислите количество минут в неделе, как мы это делали в предыдущей главе. Но на сей раз воспользуйтесь переменными. Создайте переменные `DaysPerWeek` (дней в неделе), `HoursPerDay` (часов в дне) и `MinutesPerHour` (минут в часе) и перемножьте их друг с другом (вы можете придумать собственные варианты имен).
5. Люди всегда говорят, что им не хватает времени. Сколько минут было бы в неделе, если бы сутки длились 26 часов? (Подсказка: измените переменную `HoursPerDay`.)

Базовая математика

Когда мы впервые использовали интерпретатор Python в интерактивном режиме, то увидели, как он выполнял простые математические операции. Посмотрим, что еще Python может делать с числами. Возможно, вы еще этого не поняли, но математика окружает нас со всех сторон! А уж в программировании она применяется повсеместно. Разумеется, это не означает, что программирование могут освоить только знатоки математики, но данный аспект следует учитывать. В любой игре подсчитываются очки. Графики рисуются на экране благодаря числам, задающим их положение и цвет. Движущиеся объекты характеризуются направлением и скоростью, которые опять-таки выражаются в числах. Почти любая интересная программа тем или иным образом использует числа и математику. Поэтому имеет смысл познакомиться с основами математики в Python.



Большая часть того, что вы узнаете в этой главе, применима к другим языкам программирования и даже к другим приложениям, например к электронным таблицам. Математические операции выполняются подобным образом не только в Python.

Четыре основные операции

С некоторыми арифметическими действиями, доступными в Python, мы познакомились в главе 1: сложение выполняется при помощи знака плюс (+), умножение — с помощью знака звездочки (*).

Дефис (-) (также называемый минусом), как несложно догадаться, применяется в Python для вычитания:

```
>>> print 8 - 5
3
```

Так как на клавиатуре отсутствует символ деления (÷), все программы используют для этой цели косую черту (/):

```
>>> print 6/2
3
```

Впрочем, иногда результат деления получается совсем не таким, как вы ожидали:

```
>>> print 3/2
1
```

Как вам это понравится? Вроде бы компьютеры должны разбираться в математике. Ведь все знают, что:

$$3 / 2 = 1.5$$

В чем же проблема?

Как ни странно, в данном случае интерпретатор Python *попытался* проявить интеллект. Однако чтобы это понять, нужно знать, что такое целые и десятичные числа. Если вы не понимаете, чем они отличаются друг от друга, прочитайте следующее примечание.

НОВЫЕ СЛОВА

Целыми (integers) называются числа, которые мы легко можем посчитать, например 1, 2, 3. Разумеется, сюда же относятся 0 и отрицательные числа –1, –2, –3 и т. д.

Десятичные числа (decimal numbers), которые также называют *вещественными* (real numbers), имеют десятичную точку и некое значение после нее, например 1.25, 0.3752 и –101.2.

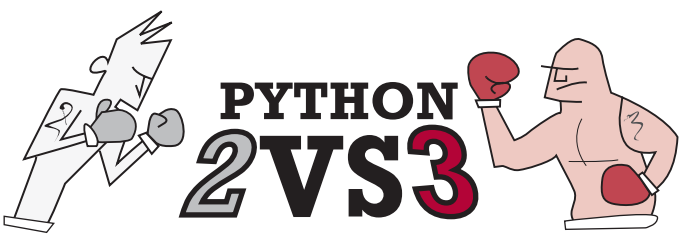
В программировании такие числа еще называют *числами с плавающей точкой* (floating-point numbers). Ведь десятичная точка может смещаться в разные стороны. Например, в переменной данного типа может содержаться число 0.00123456 или 12345.6.

Так как числа 3 и 2 мы ввели как целые, интерпретатор Python решил, что ответ тоже должен быть целым. И округлил ответ 1.5 до ближайшего целого, отбрасывая дробную часть. В результате мы получили 1. То есть происходит деление нацело.

Вот как можно исправить ситуацию:

```
>>> print 3.0 / 2
1.5
```

Если одно из двух чисел указать как десятичное, интерпретатор Python поймет, что в ответе нам требуется десятичное число.



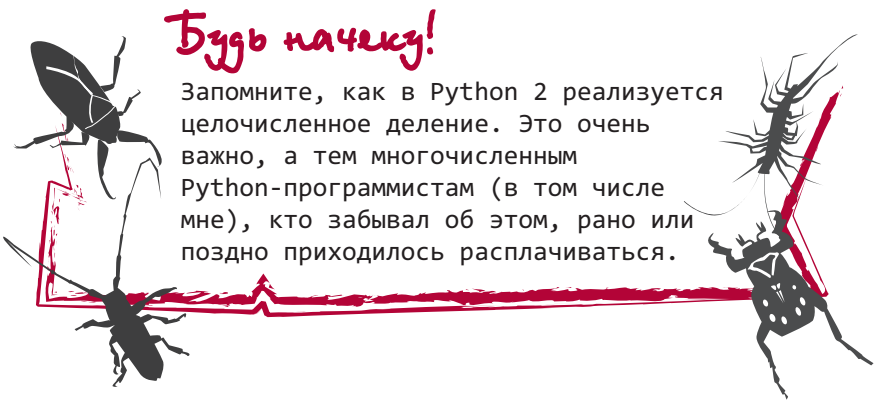
Деление с остатком
В Python 2 выполняется так называемое деление с остатком (floor division). А вот в Python 3 все уже по-другому. В Python 3 косая черта реализует обычную операцию деления:

```
>>> print 3/2
1.5
```

Что же касается деления с остатком, то в Python 3 оно реализуется при помощи двух косых черт:

```
>>> print 3//2
1
```

Это одно из наиболее заметных различий Python 2 и Python 3, которое вызывает сбой во многих программах, написанных на Python 2, при их запуске в Python 3, и наоборот.



Операторы

Символы +, -, * и / называются *операторами*. Потому что они «оперируют» числами. Знак = также является оператором и называется *оператором присваивания*, так как именно он присваивает переменным различные значения.

СТРАННОЕ ДЛИННОЕ СЛОВО

Оператором называется нечто, что оказывает воздействие на другие сущности или оперирует ими. Воздействием может быть присвоение значения, проверка или редактирование одной или нескольких сущностей.

```
myNumber + yourNumber
```

Операнд Оператор Операнд

Символы +, -, * и /, используемые нами для указания арифметических действий (операций), являются *операторами*.

Сущностями, над которыми с помощью операторов производятся операции, называются *операндами*.

Порядок выполнения операций

Какое из следующих двух выражений правильное?

2 + 3 * 4 = 20

2 + 3 * 4 = 14

Все зависит от порядка выполнения операций. Если первым мы выполняем сложение, получится:

2 + 3 = 5 и, соответственно, 5 * 4 = 20.

Если же первым выполняется умножение, мы получим:

3 * 4 = 12 и, соответственно, 2 + 12 = 14.

Корректным является второй вариант. В математике существует так называемый *порядок выполнения операций*, определяющий, какие операторы должны выполняться раньше других, даже если в выражении они появляются позднее.

В нашем примере, несмотря на то что знак + встречается раньше знака *, первым выполняется умножение. В Python используются обычные математические правила, поэтому умножение выполняется раньше сложения. Чтобы убедиться в этом, проделаем в интерактивном режиме следующие действия:

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

Порядок выполнения операций в Python аналогичен порядку выполнения операций, которые вы изучали (или будете изучать) на уроках математики. Первым делом выполняется возведение в степень, затем умножение и деление и только после них — сложение и вычитание.

Для изменения порядка выполнения операций понадобятся скобки. Например так:

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

На этот раз интерпретатор Python первым выполнил сложение 2 + 3 (благодаря скобкам), получив значение 5. Затем было произведено умножение 5 * 4, давшее нам значение 20.

Как видите, это ничем не отличается от привычных для вас уроков математики. В Python (и всех остальных языках программирования) применяются обычные математические правила и привычный порядок выполнения арифметических операций.

Еще два оператора

Я хотел бы познакомить вас с еще двумя операторами. Добавьте их к четырем уже знакомы вам, и вы получите инструментарий, достаточный для написания 99 % программ.

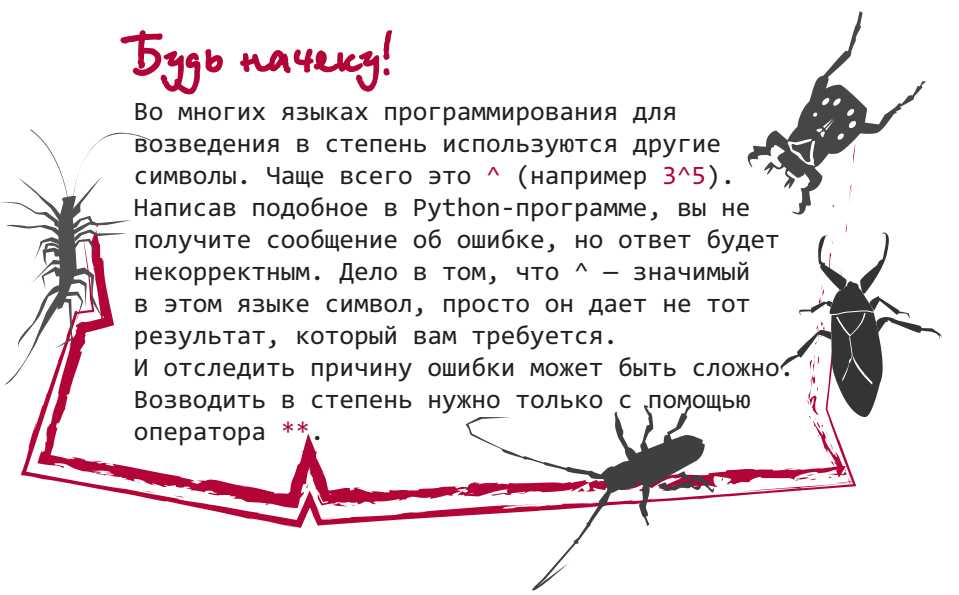
возведение в степень

Если вам нужно умножить цифру 3 на саму себя пять раз, можно написать:

```
>>> print 3 * 3 * 3 * 3 * 3
243
```

Но ведь это то же самое, что и 3^5, или «три в пятой степени». Для возведения в степень в Python используется двойная звездочка:

```
>>> print 3 ** 5
243
```



Возведение в степень лучше использовать вместо многократного умножения в первую очередь потому, что в результате приходится меньше набирать на клавиатуре. Кроме того, оператор ** позволяет возводить в нецелую степень. Например:

```
>>> print 3 ** 5.5
420.888346239
```

Подобную операцию невозможно заменить на умножение.

Остаток целочисленного деления

При первом знакомстве с операцией деления в Python мы обнаружили, что при делении двух целых чисел интерпретатор Python 2 дает в результате также целое число. (В Python 3 это действие реализует оператор `//`.) Это так называемое деление нацело. И ответ в подобных случаях на самом деле состоит из двух частей.

Помните, как вы первый раз познакомились с делением? Если числа невозможно было поделить нацело, появлялся *остаток*:

7 / 2 = 3, в остатке 1.

При выполнении операции 7 / 2 ответ содержит результат деления (в данном случае 3) и остаток (в данном случае 1). При делении в Python двух целых чисел вы получаете только первую часть. А куда девается остаток?

Для получения остатка целочисленного деления существует отдельный оператор (`%`). Вот как он используется:

```
>>> print 7 % 2
1
```

Совместно используемые операторы `/` и `%` позволяют получить обе части результата деления нацело:

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

При делении 7 на 2 получается 3 и 1 в остатке. А при делении с плавающей точкой вы получите десятичное значение:

```
>>> print 7.0 / 2
3.5
```

На самом деле те и другие операторы делают примерно одно и то же. Арифметические операторы соединяются друг с другом цифры, как раньше оператор-телефонист соединял друг с другом телефоны.

Существуют *еще два* оператора, с которыми я хотел бы вас познакомить. И хотя я помню, что сначала речь шла только о двух дополнительных операторах, в данном случае все предельно просто!

Инкремент и декремент

Помните пример из предыдущей главы: `score = score + 1`? Мы назвали эту операцию *приращением*, или *инкрементированием*. Аналогичный пример `score = score - 1` называется *отрицательным приращением*, или *декрементированием*.

Эти операции в программировании выполняются настолько часто, что для них придумали специальные операторы: `+=` (инкремент) и `-=` (декремент).

Вот как они применяются:

```
>>> number = 7
>>> number += 1
>>> print number
8
```

Переменная number увеличилась на 1

Или:

```
>>> number = 7
>>> number -= 1
>>> print number
6
```

Переменная number уменьшилась на 1

В первом случае к числу добавляется единица. (Оно меняется с 7 на 8.) Во втором случае из числа вычитается единица. (Оно меняется с 7 на 6.)

Очень большие и очень маленькие

Помните, как в главе 1 мы перемножали два очень больших числа? Ответ также представлял собой большое число. Иногда в Python такие числа выводятся на экран немного по-другому. Проделайте в интерактивном режиме эту операцию:

```
>>> print 9938712345656.34 * 4823459023067.456
4.79389717413e+025
```

В данном случае не имеет значения, какие именно цифры вы вводите. Подойдут любые большие вещественные числа.

Буква *e* позволяет выводить на экран очень большие числа и называется *экспоненциальным представлением*, или *e-нотацией*. При работе с очень большими (или с очень маленькими) числами крайне неудобно писать все цифры и десятичные разряды. А в математике такие числа встречаются достаточно часто.

К примеру, астрономическая программа, показывающая расстояние от Земли до звезды Альфа Центавра, может вывести на экран 38000000000000000, 38,000,000,000,000,000 или 38 000 000 000 000 000. (Это 38 квадриллионов километров!) И в любом из трех вариантов вам придется помучиться, пересчитывая все нули.

Другим способом отображения этого числа является *экспоненциальное представление*, в котором наряду с десятичными числами фигурируют степени 10. Расстояние до звезды Альфа Центавра в этом случае будет записано как 3.8×10^{16}. (Обратили внимание, что цифра 16 возвышается над строчкой и имеет меньший размер?) Выражение читается как «три целых восемь десятых на десять в шестнадцатой степени» и означает, что вы берете 3.8 и перемещаете десятичную точку на 16 разрядов вправо, добавляя необходимые нули:

3.80000000000000000

Передвиньте десятичную точку на 16 разрядов вправо

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

Экспоненциальное представление замечательно подходит для случаев, когда 16 можно записать как показатель степени, подняв немного вверх и уменьшив его размер, как это было сделано выше. Вы можете этим пользоваться, работая с карандашом и бумагой или с приложением, поддерживающим верхние индексы.

НОВЫЕ СЛОВА

Верхний индекс представляет собой символ или символы, приподнятые над остальным текстом, например 10^{13}. В этом примере 13 — верхний индекс. Обычно он обозначается более мелким шрифтом по сравнению с основным текстом.

На него похож *нижний индекс*, но в этом случае символы располагаются ниже основного текста, например $\log_2$. В данном примере 2 — нижний индекс.

Из-за ограниченной применимости верхних индексов появилась еще одна форма записи — е-нотация. Это всего лишь еще один вид экспоненциального представления.

Е-нотация

В е-нотации наше число выводится на экран как 3.8E16 или как 3.8e16. В данном случае буква E обозначает степень 10. То есть фактически у нас написано 3.8×10^{16}.

В большинстве языков программирования, в том числе и в Python, букву E можно писать как в верхнем (E), так и в нижнем (e) регистре.

При записи очень маленьких чисел, например 0.0000000000001752, применяется возведение в отрицательную степень. То есть можно записать 1.752×10^{-13}, или 1.752e-13. Отрицательная степень означает, что десятичную точку нужно сдвинуть на указанное количество разрядов не вправо, а влево:

0000000000000000 1.752

Подвиньте десятичную точку на 13 разрядов влево

0.00000000000011752 = 1.752e-13

Подобная форма записи позволяет представлять в Python как очень большие, так и очень маленькие числа (более того, в ней можно при необходимости представить любое число). Позднее вы узнаете, как заставить интерпретатор Python вывести на экран числа в подобном представлении.

А пока попробуем воспользоваться им для ввода чисел:

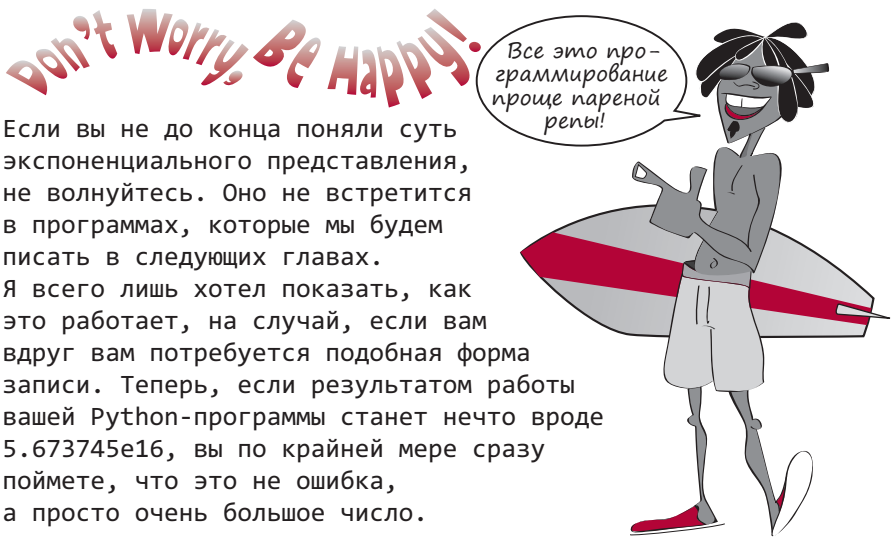
```
>>> a = 2.5e6
>>> b = 1.2e7
>>> print a + b
14500000.0
```

Несмотря на ввод чисел с буквой *e*, ответ выведен в обычной десятичной форме. Дело в том, что интерпретатор Python показывает результат в экспоненциальном представлении, только если число очень маленькое или очень большое (содержит много нулей) или если вы в явном виде затребовали вывести данные в такой форме. Попробуйте сделать так:

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

На этот раз ответ автоматически дается в экспоненциальном представлении, так как глупо выводить на экран число с 73 нулями!

Чтобы отобразить подобным образом, например, число 14 500 000, интерпретатору Python следует дать специальную команду. С ней вы познакомитесь позже (в главе 21).



Если вы не до конца поняли суть экспоненциального представления, не волнуйтесь. Оно не встретится в программах, которые мы будем писать в следующих главах. Я всего лишь хотел показать, как это работает, на случай, если вам вдруг вам потребуется подобная форма записи. Теперь, если результатом работы вашей Python-программы станет нечто вроде 5.673745e16, вы по крайней мере сразу поймете, что это не ошибка, а просто очень большое число.

Возведение в степень, или экспоненциальное представление

Не следует путать возведение чисел в степень и e-нотацию:

- 3**5 означает 3^5, или «три в пятой степени», или 3 * 3 * 3 * 3 * 3, и равняется 243.
- 3e5 означает 3 * 10^5, или «три умножить на десять в пятой степени», или 3 * 10 * 10 * 10 * 10 *10, и равняется 300 000.
- Возведение в степень означает, что вы умножаете число на само себя указанное количество раз. Экспоненциальное представление означает, что вы умножаете его на степень 10.

Некоторые читают и 3e5, и 3**5 как «три в пятой степени», но на самом деле это разные выражения. Впрочем, способ чтения не имеет особого значения при условии, что вы понимаете значение каждого из вариантов записи.

Что мы узнали

В этой главе мы научились:

- выполнять базовые математические операции на языке Python;
- различать целые числа и числа с плавающей точкой;

- возводить в степень;
- вычислять остаток от деления;
- записывать числа в экспоненциальном представлении.

Проверь себя

1. Каким символом в Python обозначается умножение?
2. Какой ответ интерпретатор Python 2 даст при операции `8 / 3`?
3. Как получить остаток при делении `8 / 3`?
4. Как в Python 2 получить результат деления `8 / 3` в десятичной форме?
5. Как еще в Python можно вычислить `6 * 6 * 6 * 6`?
6. Как записать число 17 000 000 в экспоненциальном представлении?
7. Как будет выглядеть число `4.56e-5` в обычном представлении (без символа E)?

Эксперименты

1. Решите следующие задачи в интерактивном режиме или написав небольшую программу:
   1) три человека обедают в ресторане и хотят оплатить вскладчину счет в 35.27 доллара. На чай они собираются оставить 15 %. Какую сумму должен выложить каждый из обедающих?
   2) вычислите площадь и периметр прямоугольной комнаты шириной 12.5 метров и длиной 16.7 метра.
2. Напишите программу для преобразования градусов Фаренгейта в градусы Цельсия. Преобразование осуществляется по формуле C = 5 / 9 * (F – 32). (Подсказка: помните про подводные камни при делении целых чисел!)
3. Как узнать, сколько времени занимает путешествие на автомобиле? Словами формула выражается так: «время путешествия равна расстоянию, деленному на скорость». Напишите программу, которая посчитает, сколько времени займет перемещение на 200 километров со скоростью 80 километров в час, и отобразите ответ на экране.

ГЛАВА 4

Типы данных

Вы уже знаете, что переменной можно присваивать значения (для их сохранения в памяти компьютера) по крайней мере трех типов: целые числа, числа с плавающей точкой и строки. Но в Python допустимы и другие типы данных, с которыми вы познакомитесь чуть позже, а пока мы будем работать с известными нам типами. В этой главе вы научитесь определять принадлежность к конкретному типу, а также узнаете, как получить один тип из другого.

Преобразование типов

Часто возникает необходимость перейти от одного типа к другому. Например, выводимое на экран число сначала может потребоваться преобразовать в текст. Эту операцию за вас выполняет команда `print`, но бывают ситуации, когда нужно поменять тип без вывода информации на экран или осуществить преобразование строки в число (такое преобразование команда `print` выполнять не умеет). Эта операция называется *преобразованием типов* (type conversion). Каким образом она реализуется?

На самом деле интерпретатор Python не «преобразует» типы друг в друга. Он создает из исходного элемента новый элемент нужного вам типа. Вот ряд функций, умеющих выполнять эту операцию:

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

Скобки в конце указывают, что перед нами не *команды* языка Python (такие как `print`), а встроенные в него *функции*.

Подробно функции будут рассмотрены чуть позже, а на данном этапе вам достаточно знать, что подлежащее преобразованию значение указывается *в скобках*. Лучше всего заметно это на примерах. Рассмотрим их в интерактивном режиме оболочки IDLE.

Преобразование целого в десятичное

Возьмем целое число и при помощи функции `float()` создадим на его основе число с плавающей точкой:

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

Обратите внимание на десятичную точку и завершающий 0 при выводе переменной `b`. Именно по этим признакам мы видим, что перед нами не целое, а десятичное число. При этом переменная `a` не меняется, так как на исходное значение функция `float()` не влияет — она создает новое значение.

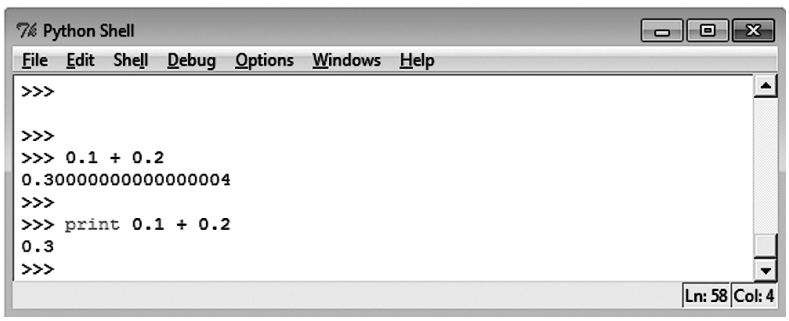
В интерактивном режиме достаточно ввести имя переменной (без команды `print`), и интерпретатор Python выведет на экран ее значение. (Вы уже делали так в главе 2.) Однако в программах это не работает.

Преобразование десятичного в целое

Теперь попробуем сделать обратный переход — возьмем десятичное число и при помощи функции `int()` получим из него целое:

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

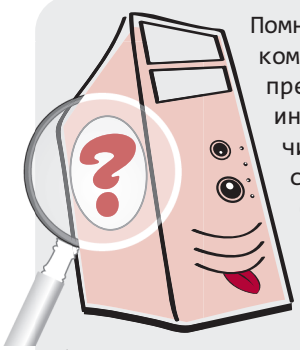
Мы создали новую переменную `d`, которая представляет собой целую часть переменной `c`.



Ой! Как так получилось? Картер, думаю, твой компьютер сошел с ума!

Шучу. На самом деле такому поведению есть объяснение.

ЧТО ПРОИСХОДИТ ВНУТРИ?



Помните, мы говорили о том, что внутри компьютер использует двоичное представление? Именно в таком виде интерпретатор Python сохраняет все числа. Для суммы чисел 0.1 и 0.2 Python создает переменную с плавающей точкой (десятичное число) и количеством двоичных разрядов (битов), достаточным для хранения 15 десятичных знаков. Это двоичное число не равно 0.3, но имеет очень близкое к этому значение. (В нашем случае ошибка составляет 0.000000000000004.) Разница называется ошибкой округления.

При вводе выражения 0.1 + 0.2 в интерактивном режиме интерпретатор Python показывает все знаки после точки. А вот команда print показывает уже тот результат, который вы ожидали увидеть, так как она автоматически округляет и выводит на экран значение 0.3.

Представьте, что вы спрашиваете у прохожего, который час. Он может ответить вам: «двенадцать часов, сорок четыре минуты и пятьдесят три секунды». Но в большинстве случаев вы услышите в ответ: «без четверти час», так как люди знают, что особая точность в подобных ситуациях не требуется. Ошибки округления при работе с вещественными числами возникают во всех языках программирования. Количество показываемых знаков после точки может различаться на разных компьютерах и в разных языках, но базовый метод хранения чисел с плавающей точкой всегда один и тот же. Ошибки округления, как правило, слишком малы, чтобы на них обращать внимание.

Рассмотрим еще один пример:

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

Несмотря на то что 54.99 — это почти 55, целочисленный ответ будет 54. Функция `int()` производит округление, просто отбрасывая дробную часть. Поэтому она дает нам не ближайшее целое, а меньшее из двух целых.

Существует и способ получить ближайшее целое, но с ним вы познакомитесь в главе 21.

Преобразование строк в числа с плавающей точкой

Получить из строки число можно следующим образом:

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

Обратите внимание, что переменная `a` выводится в кавычках. Так интерпретатор Python показывает, что это строка. А для переменной `b` выводится значение с плавающей точкой.

Получаем дополнительную информацию

В предыдущем разделе по наличию кавычек мы определяли, что именно перед нами — число или строка. Но это можно узнать и другим способом.

В Python существует функция `type()`, в явном виде сообщающая тип переменной.

Посмотрим, как она работает:

```
>>> a = '44.2'
>>> b = 44.2
>>> type(a)
```

```
<type 'str'>
>>> type(b)
<type 'float'>
```

Функция `type()` для переменной `a` выводит `type 'str'`, что соответствует *строке*, а для переменной `b` — `type 'float'`. Вам больше не нужно ничего угадывать!

Ошибки при преобразовании типов

Если дать функции `int()` или `float()` что-то отличное от числа, она не сработает. Убедитесь сами:

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

Мы получили сообщение об ошибке, в котором говорится, что интерпретатор Python не знает, как превратить строку `"fred"` в число. А вы знаете?

Применение преобразования типов

Вернемся к программе, которая пересчитывала градусы Фаренгейта в градусы Цельсия в конце предыдущей главы. Помните, что для получения корректного ответа нам требовалось решить проблему с делением целых чисел, и пришлось менять 5 на 5.0 или 9 на 9.0?

```
cel = 5.0 / 9 * (fahr - 32)
```

Другим способом это можно сделать при помощи функции `float()`:

```
cel = float(5) / 9 * (fahr - 32)
```

Или так:

```
cel = 5 / float(9) * (fahr - 32)
```

Попробуйте и убедитесь сами!

Что мы узнали

В этой главе мы узнали:

- как осуществлять преобразование типов (или, точнее, создавать элементы одного типа из элементов другого) при помощи функций: `str()`, `int()` и `float()`;

- как вывести на экран значение, не пользуясь командой `print`;
- как проверить тип переменной с помощью функции `type()`;
- что такое ошибки округления и как они возникают.

Проверь себя

1. В какую сторону — в бóльшую или в меньшую — округляется результат при преобразовании вещественного числа в целое с помощью функции `int()`?
2. Будет ли это работать в вашей программе, преобразующей температуру?

```
cel = float(5 / 9 * (fahr - 32))
```

   А как насчет этого:

```
cel = 5 / 9 * float(fahr - 32)
```

   Если это не работает, то почему?

3. (Дополнительный вопрос) Можно ли, не прибегая ни к каким функциям, кроме `int()`, округлить число не в меньшую, а в бóльшую сторону? (Например, сделать так, чтобы 13.2 округлялось до 13, а 13.7 — до 14.)

Эксперименты

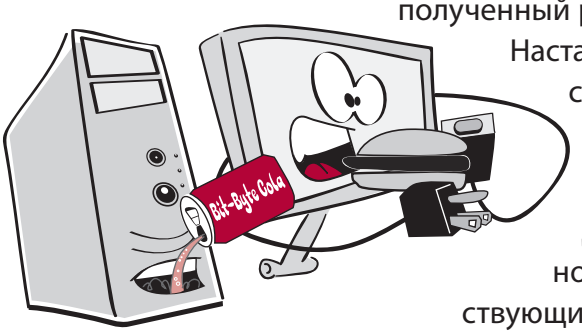
1. При помощи функции `float()` превратите строку `'12.34'` в число. Убедитесь, что в результате действительно получается число!
2. При помощи функции `int()` превратите число 56.78 в целое. В бóльшую или в меньшую сторону происходит округление в данном случае?
3. При помощи функции `int()` создайте из строки целое число. Убедитесь, что в результате получается именно целое число!

ГЛАВА 5

Ввод

До этого момента вся цифровая информация программ, производящих вычисления, указывалась непосредственно в коде. Например, в программе преобразования температуры, которую вам предлагали написать в главе 3, преобразуемая температура помещалась в код. А это означает, что для преобразования другой температуры придется менять программу.

Нельзя ли сделать так, чтобы после запуска программы пользователь мог указывать ту температуру, которую он хочет преобразовать? Ранее мы уже говорили о том, что для каждой программы характерны три аспекта: ввод, обработка и вывод. Наша первая программа занималась только выводом данных. Программа преобразования температуры производила некоторую обработку (собственно пересчет температуры) и выводила полученный результат, при этом о вводе данных речи не шло.



Настало время наделить наши программы третьей способностью — способностью *ввода* (input) данных. Это процедура, позволяющая вставить новую информацию в уже работающую программу.

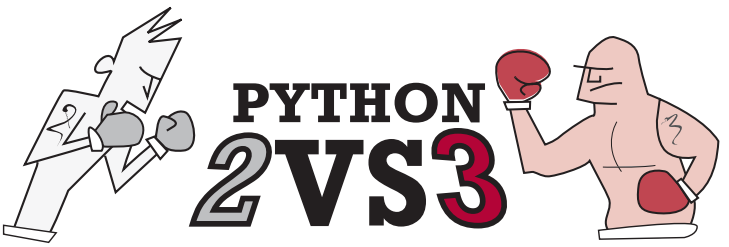
Данная процедура позволяет писать более разнообразные и интересные программы, взаимодействующие с пользователем.

В Python существует встроенная функция `raw_input()`, предназначенная для получения данных от пользователя. И в этой главе вы научитесь применять ее для своих нужд.

Функция raw_input()

Функция `raw_input()` получает от пользователя строку. Обычно данные просто вводятся с клавиатуры.

Итак, к уже знакомым вам встроенным Python-функциям `str()`, `int()`, `float()` и `type()`, которые мы изучили в главе 4, добавилась еще одна. Подробно о функциях мы будем говорить позже, а пока достаточно запомнить, что вставляя их в свои программы, вы должны добавлять круглые скобки.



Необработанные входные данные
В версии Python 3 функция `raw_input()` называется `input()`. Она полностью аналогична функции `raw_input()` из версии Python 2.

Вот пример применения этой функции:

```
someName = raw_input()
```

Эта команда дает пользователю возможность ввести строку, которой будет присвоено имя `someName`.

Вставим ее в программу. Создайте в IDLE новый файл и добавьте туда код из листинга 5.1.

Листинг 5.1. Получение строки через функцию raw_input

```
print "Введите свое имя: "
somebody = raw_input()
print "Привет, ", somebody, "как дела?"
```

Сохраните и запустите эту программу в IDLE, чтобы посмотреть, как она работает. Вы должны получить примерно такой результат:

```
Введите свое имя:
Уоррен
Привет, Уоррен, как дела?
```

Я ввел свое имя, а программа назначила его переменной `somebody`.

Команда print и запятая

Обычно пользователя информируют коротким сообщением, показывающим, какие данные ему следует ввести:

```
print "Введите ваше имя: "
```

После этого при помощи функции `raw_input()` можно получить ответ:

```
someName = raw_input()
```

Результатом запуска этих строк кода и ввода моего имени будет:

```
Введите ваше имя:
Уоррен
```

Если вы хотите, чтобы вводимый пользователем ответ оказался в одной строке с информационным сообщением, поставьте после инструкции `print` запятую. Вот так:

```
print "Введите ваше имя: ",
someName = raw_input()
```

Обратите внимание, что запятая ставится после кавычек.

Результат запуска этого кода будет выглядеть так:

```
Введите ваше имя: Уоррен
```

Запятая объединяет в одну строку данные, выводимые разными инструкциями `print`. Она как бы предлагает: «не перескакивайте на следующую строку после вывода этого фрагмента». Именно так мы поступили в последней строке листинга 5.1.

Введите в IDLE-редактор и запустите код из листинга 5.2.

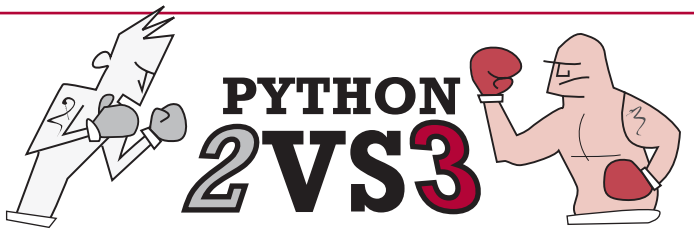
Листинг 5.2. Зачем нужна запятая

```
print "Меня",
print "зовут",
print "Дэйв."
```

После запуска кода вы получите:

```
Меня зовут Дэйв.
```

Обратили внимание, что после слов в кавычках пробелов нет, а результат выводится с пробелами? Интерпретатор Python добавляет пробелы при объединении инструкций `print` в одну строку.



Конец запятой
В версии Python3 трюк с объединением при помощи запятой выводимых на экран результатов не работает. Кроме того, в этой версии выводимые командой `print()` данные должны помещаться в скобки. Поэтому, если вы пользуетесь Python 3, листинг 5.2 будет выглядеть так:

```
print("Меня", end=" ")
print("зовут", end=" ")
print("Дэйв.", end=" ")
```

Я рад, что ты спросил!
Именно об этом я хотел поговорить.

ПРИГЛАШЕНИЕ НА ВВОД ДАННЫХ

Показать приглашение на ввод данных можно и более простым способом. Функция `raw_input()` и сама может выводить на экран данные, поэтому прибегать к инструкции `print` нет необходимости:

```
someName = raw_input ("Введите ваше имя: ")
```

Можно сказать, что функция `raw_input()` обладает встроенной командой `print`. И с этого момента мы будем пользоваться сокращенной формой записи.

Ввод чисел

Итак, теперь вы знаете, как при помощи функции `raw_input()` осуществить ввод строки. А что делать, если вам нужно ввести число? Как-никак разговор про ввод данных возник при обсуждении новой версии программы преобразования температуры.

Впрочем, если вы читали главу 4, ответ вам известен. Можно взять функцию `int()` или `float()` и превратить предоставленную вам функцией `raw_input()` строку в число. Это будет выглядеть так:

```
temp_string = raw_input()
fahrenheit = float(temp_string)
```

Сначала при помощи функции `raw_input()` мы получаем вводимые пользователем данные в виде строки. А затем, применив функцию `float()`, мы превращаем ее в число. Получив температуру в виде числа с плавающей точкой, мы помещаем ее в переменную `fahrenheit`.

Впрочем, все это можно сделать и короче, всего за один шаг:

```
fahrenheit = float(raw_input())
```

Эта строка выполняет те же самые действия. Получает у пользователя строку и превращает ее в число. Но теперь для этого требуется меньше кода.

Воспользуемся полученными сведениями для нашей программы преобразования температуры. Попробуйте запустить программу из листинга 5.3.

Листинг 5.3. Преобразование температуры с помощью функции raw_input()

```
print "Эта программа преобразует градусы Фаренгейта в градусы Цельсия"
print "Введите температуру в градусах Фаренгейта: ",
fahrenheit = float(raw_input())
celsius = (fahrenheit - 32) * 5.0 / 9
print "Это",
print celsius,
print "градусов Цельсия"
```

Для получения от пользователя температуры задействуйте функцию float(raw_input())

Обратите внимание на запятые в конце этих строк

Последние три строки из листинга 5.3 можно объединить в одну:

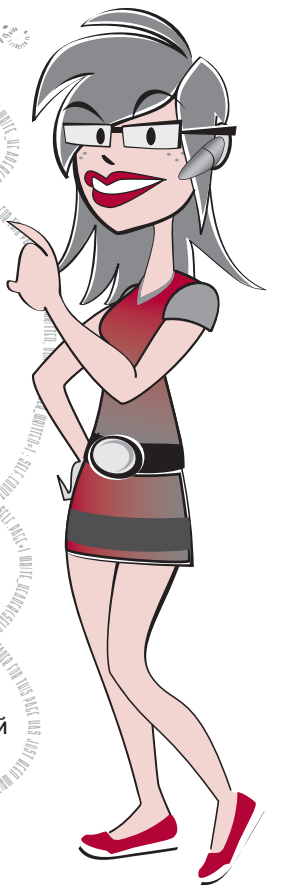
```
print "Это", celsius, "градусов Цельсия"
```

Это сокращенный вариант записи трех инструкций `print`, которые у нас были изначально.

Думай как программист

Существует и другой способ ввода чисел. В Python 2 есть функция `input()`, сразу дающая вам числа, а значит, избавляющая от необходимости выполнять преобразование при помощи функции `int()` или `float()`. Именно ее мы применяли в главе 1 в программе, угадывающей цифры, так как это самый простой способ получить от пользователя число. Однако для единообразия в дальнейшем мы будем работать только с функцией `raw_input()`. Кроме того, в версии Python 3 функция `input()` отсутствует. Там осталась только функция `raw_input()`.

Все еще больше запутывает тот факт, что функция, которая в версии Python 2 называлась `raw_input()`, в Python 3 получила имя `input()`, а ведь это уже знакомая вам функция, дающая результат в виде строк. Так как вы умеете преобразовывать строки в числа, в версии Python 2 рекомендую пользоваться функцией `raw_input()` вместо функции `input()`.



Совместное использование функций int() и raw_input()

Если вы хотите, чтобы все вводимые пользователем числа были целыми (без десятичных знаков), для преобразования можно задействовать функцию `int()`:

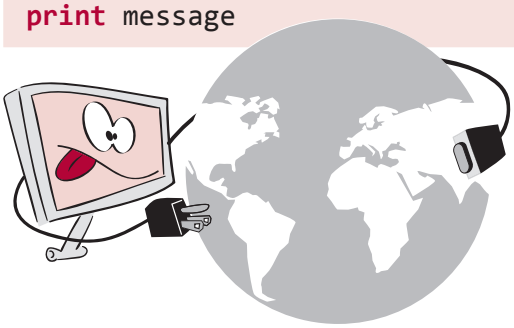
```
response = raw_input("Сколько в вашем классе студентов: ")
numberOfStudents = int(response)
```

Ввод данных из Интернета

Обычно данные вводит работающий с программой пользователь. Но существуют и другие способы ввода. Данные можно получить с жесткого диска вашего компьютера (как это сделать, вы узнаете в главе 22) или из Интернета. При соединении с Интернетом вы можете запустить программу из листинга 5.4. Она открывает файл с сайта нашей книги и показывает содержащееся внутри этого файла сообщение.

Листинг 5.4. Полученные данных из файла в Интернете

```
import urllib2
file = urllib2.urlopen('http://manning.com/data/message.txt')
message = file.read()
print message
```



Да, именно так. Всего четыре строки кода заставляют ваш компьютер зайти во Всемирную паутину, найти файл на сайте нашей книги и показать его. Запустив эту программу (при условии, что вас есть соединение с Интернетом), вы увидите сообщение.

ДУМАЙ КАК ПРОГРАММИСТ

В зависимости от установленной операционной системы (Windows, Linux или Mac OS X) после запуска программы из листинга 5.4 в конце каждой строки могут появиться квадратики или символы \r. Дело в том, что разные операционные системы по-разному отмечают конец текстовой строки. В Windows (а до этого в MS-DOS) используются два символа: CR (Carriage Return — возврат каретки) и LF (Line Feed — перевод строки), в Linux — только LF, а в Mac OS X — только CR. Некоторые программы не обращают на такое внимания, а другие, и в их числе IDLE-редактор, приходят в замешательство, не обнаружив ожидаемого ими символа конца строки. В этом случае на экран выводится маленький квадратик, который означает «я не понимаю этого символа». Появление этих квадратиков зависит от используемой вами операционной системы и способа запуска программы (в IDLE или другим методом).

Если вы запускаете эту программу со школьного компьютера, она может не сработать. В публичных местах соединение с Интернетом осуществляется через так называемые прокси-серверы. Прокси-сервер — это дополнительный компьютер, напоминающий мостик между Интернетом и вашей школой. Настройки прокси-сервера могут оказаться такими, что программа просто не соединится с Интернетом. Тогда попробуйте запустить ее с компьютера, обладающего прямым соединением с Интернетом.

Что мы узнали

В этой главе мы научились:

- вводить текст при помощи функции `raw_input()`;
- добавлять к функции `raw_input()` приглашения на ввод данных;
- вводить числа, комбинируя функции `int()` и `float()` с функцией `raw_input()`;
- при помощи запятой выводить несколько фрагментов в одну строку.

Проверь себя

1. Если пользователь введет 12, к какому типу данных будет принадлежать переменная `answer`? Она будет строкой или числом?

```
answer = raw_input()
```

2. Как заставить функцию `raw_input()` показать приглашение на ввод данных?
3. Как при помощи функции `raw_input()` осуществить ввод целого числа?
4. Как при помощи функции `raw_input()` получить число с плавающей точкой (десятичное)?

Эксперименты

1. Создайте в интерактивном режиме две переменные — одну для вашего имени, а вторую для вашей фамилии. Затем при помощи одной инструкции `print` выведите их на экран в одну строку.
2. Напишите программу, которая сначала спрашивает ваше имя, потом вашу фамилию, а затем выводит на экран сообщение с вашими личными данными.
3. Напишите программу, которая запрашивает размеры прямоугольной комнаты (в сантиметрах) и выводит на экран площадь закрывающего весь пол ковра.
4. Напишите программу, которая делает все перечисленное в задании 3, запрашивая стоимость ковра за квадратный метр. После выведите на экран следующие данные:
   - общую площадь ковра в квадратных сантиметрах;
   - общую площадь ковра в квадратных метрах (1 квадратный метр = 10 000 квадратных сантиметров);
   - итоговую цену ковра.
5. Напишите программу для счета мелких денег. Она должна спрашивать:
   - «Сколько у вас монет по 50 копеек?»;
   - «Сколько у вас монет по 10 копеек?»;
   - «Сколько у вас монет по 5 копеек?»;
   - «Сколько у вас монет по 1 копейке?».

   После этого на экране должна появиться общая сумма.

ГЛАВА 6

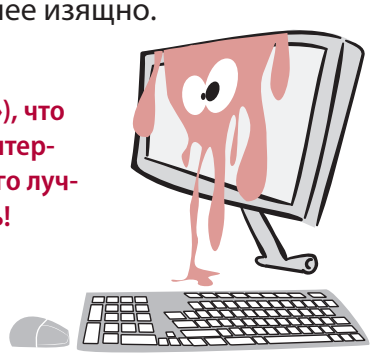
Графические интерфейсы пользователя

До этого момента мы выполняли ввод и вывод в виде обычного текста в окне IDLE. Однако современные компьютеры и программы активно используют графику. Было бы здорово добавить графику и в наши программы. В этой главе мы будем учиться создавать простые графические интерфейсы пользователя, чтобы придать вашим программам более привычный вид — с окнами, кнопками и т. п.

Что такое GUI?

Аббревиатура GUI расшифровывается как Graphical User Interface (*графический интерфейс пользователя*). Благодаря GUI действия пользователя не ограничиваются вводом текста и выводом на экран в текстовом виде результатов работы программы. Он получает возможность с помощью мыши и клавиатуры работать с различными графическими элементами — окнами, кнопками, текстовыми полями и т. п. До сих пор мы имели дело с программами в *текстовом режиме* (text-mode), то есть с так называемыми интерфейсами *командной строки* (command-line). Графический интерфейс предлагает альтернативный способ взаимодействия с программой. Наличие GUI не влияет на набор базовых этапов, работа программы все так же делится на ввод, обработку и вывод данных. Просто первый и последний этапы реализуются более изящно.

Кстати, аббревиатура GUI произносится как «гуи» («gooey»), что в переводе означает «сладкий», «липкий». Графический интерфейс пользователя компьютеру нужен, а вот липкое на него лучше не проливать, а то вам будет сложно программировать!



Наш первый GUI-интерфейс

На самом деле вы уже неоднократно пользовались разными GUI-интерфейсами. Веб-браузер является GUI-интерфейсом. Среда разработки IDLE — это тоже GUI. А теперь

вы собираетесь разработать собственный GUI-интерфейс. В этом нам поможет модуль интерпретатора Python, который называется EasyGui.

Вы пока еще не знаете, что такое модули (о них речь пойдет в главе 15), но это всего лишь средство добавить к интерпретатору Python функциональность, не встроенную в него по умолчанию.

Если для установки Python вы пользовались прилагаемой к книге программой установки, значит, модуль EasyGui у вас уже имеется. В противном случае его можно загрузить с сайта *easygui.sourceforge.net/*.

Установка EasyGui

Вам потребуется загрузить файл *easygui.py* или содержащий его zip-архив. Для установки этот файл следует поместить туда, где его сможет обнаружить интерпретатор Python. Где же находится это место?

Маршруты интерпретатора Python

На жестком диске есть ряд мест, в которых интерпретатор Python ищет свои модули. В Windows, Mac OS X и Linux они располагаются по разным адресам. Но если поместить файл *easygui.py* туда, где установлен сам интерпретатор Python, он без проблем будет обнаружен. Поэтому найдите на жестком диске своего компьютера папку *Python27* и скопируйте туда файл *easygui.py*.

Приступаем к созданию GUI-интерфейса

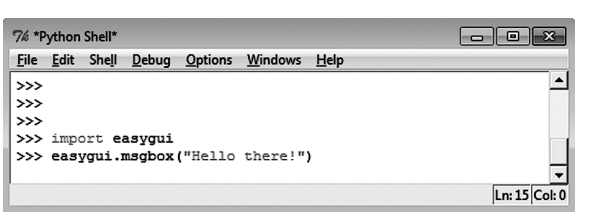
Запустите IDLE и в интерактивном режиме наберите следующее:

```
>>> import easygui
```

Так вы проинформируете интерпретатор Python о своем намерении воспользоваться модулем EasyGui. Если сообщение об ошибке не появится, значит, модуль был успешно обнаружен. В случае же сообщения об ошибке следует зайти на сайт книги (*www.manning.com/books/hello-world-second-edition*) и прочитать дополнительную инструкцию.

Теперь создадим простое информационное окно с кнопкой *OK*:

```
>>> easygui.msgbox("Hello There!")
```



Функция `msgbox()` модуля EasyGui создает информационное окно. В большинстве случаев имена функций модуля EasyGui представляют собой сокращения английских слов (в данном случае — message box).

Результат применения функции `msgbox()` выглядит примерно так:

Щелчок по кнопке *OK* закрывает информационное окно.

Ввод в GUI-интерфейсе

Вы только что познакомились с одним из вариантов вывода данных в GUI-интерфейсе — это информационное диалоговое окно. А каким образом осуществляется ввод?

Вы щелкнули по кнопке *OK* после запуска в интерактивном режиме предыдущего фрагмента кода. После этой операции в оболочке, терминале или окне командной строки вы должны были увидеть следующее:

```
>>> import easygui
>>> easygui.msgbox("Hello there!")
'OK'
```

Строкой `'OK'` интерпретатор Python и модуль EasyGui сообщают вам, что пользователь щелкнул по кнопке *OK*. Модель EasyGui всегда предоставляет обратную связь, информируя о действиях пользователя в GUI-интерфейсе, — на какой кнопке был сделан щелчок, какие данные были набраны и т. п. Этой реакции модуля можно присвоить имя (назначить ее переменной). Попробуйте сделать так:

```
>>> user_response = easygui.msgbox("Hello there!")
```

Щелкните по кнопке *OK*, чтобы закрыть диалоговое окно. Затем наберите следующее:

```
>>> print user_response
OK
```

Теперь действия пользователя — щелчок по кнопке *OK* — зафиксированы в переменной с именем `user_response`. Посмотрим на другие способы ввода данных при работе с модулем EasyGui. Информационное окно, которое вы видели, представляет собой частный пример объекта, называемого *диалоговым окном* (dialog box). Диалоговые окна являются элементами GUI-интерфейса, предназначенными для сообщения пользователю некой информации или для получения от него данных. Вводимая информация может быть щелчком по кнопке (например *OK*), именем файла или каким-то текстом (строкой).

Окно msgbox модуля EasyGui представляет собой диалоговое окно с сообщением и единственной кнопкой *OK*. Но существуют и другие виды диалоговых окон с большим количеством кнопок и другими элементами.

Выбор на свой вкус

В качестве иллюстрации различных способов ввода пользовательских данных при работе с модулем EasyGui рассмотрим процесс выбора любимого мороженого.

Диалоговое окно с набором кнопок

Создадим диалоговое окно (напоминающее информационное окно) с несколькими кнопками. Для этого предназначен специальный элемент buttonbox, представляющий собой *окно с кнопками* (button box). На этот раз вместо работы в интерактивном режиме мы напишем программу.

Откройте в IDLE новый файл и введите текст листинга 6.1.

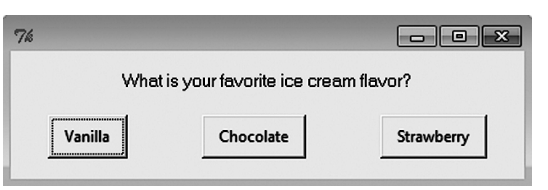
Листинг 6.1. Ввод данных при помощи кнопок

```
import easygui
flavor = easygui.buttonbox("What is your favorite ice cream flavor?",
                 choices = ['Vanilla', 'Chocolate', 'Strawberry'] )
easygui.msgbox ("You picked " + flavor)
```

Список вариантов

В квадратные скобки помещен так называемый *список* (list). О списках речь пойдет в главе 12, а пока просто перепишите этот код, чтобы модуль EasyGui смог сделать свою работу.

Сохраните файл (я назвал его *ice_cream1.py*) и запустите его. Вот что у вас получится:



А затем, в зависимости от выбранного вами варианта, появится следующее окно:

Как это работает? Метка кнопки, на которой щелкнул пользователь, интерпретируется как *ввод* данных. Мы помещаем эти данные в переменную — здесь она называется `flavor`. Это все равно что прибегнуть к функции `raw_input()`, просто пользователь не набирает данные, а щелкает по кнопке. Именно для этого и предназначены GUI-интерфейсы.

Окно выбора

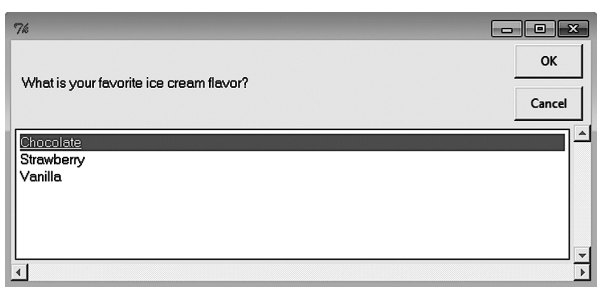
Предложим пользователю еще один способ выбора. В модуле EasyGui есть элемент `choicebox` — это *окно выбора* (choice box), позволяющее выбирать из набора вариантов. Пользователь находит нужный и щелкает по кнопке *OK*.

Для вставки этого элемента в программу слегка отредактируем листинг 6.1: заменим `buttonbox` на `choicebox` (листинг 6.2).

Листинг 6.2. Ввод данных при помощи списка

```
import easygui
flavor = easygui.choicebox("What is your favorite ice cream flavor?",
                  choices = ['Vanilla', 'Chocolate', 'Strawberry'] )
easygui.msgbox ("You picked " + flavor)
```

Сохраните программу из листинга 6.2 и запустите ее. Вы должны увидеть вот такое окно:



После выбора любимого сорта мороженого и щелчка по кнопке *OK* вы увидите уже знакомое вам информационное окно. Имейте в виду, что теперь вы можете выбирать любимое мороженое не только щелчком мыши, но и нажатием клавиш со стрелками на клавиатуре.

Щелчок по кнопке *Cancel* завершает работу программы и показывает сообщение об ошибке. Дело в том, что последняя строка программы ожидает ввода текста (например `Vanilla`), а если вы предпочтете воспользоваться кнопкой *Cancel*, эта операция отменяется.

У меня было ровно то же самое. А так как большое окно с набором вариантов нам не подходит, я пошел на небольшую хитрость! Я отредактировал файл *easygui.py*, добавив туда возможность уменьшить размер окна, чтобы оно кра-

сиво смотрелось в книге. Вы можете обойтись и без этого, но если хотите, я расскажу, как добиться такого же эффекта. Должен предупредить, что это не так-то просто!

1. Найдите в файле *easygui.py* строку, начинающуюся со слов `def __choicebox` (в моей версии файла это была строка 934). Помните, что большинство редакторов показывают номера строк кода ближе к нижней части окна.
2. Примерно на 30 строк ниже (в районе строки 970) вы увидите примерно такой код:

```
root_width = int((screen_width * 0.8))
root_height = int((screen_height * 0.5))
```

   Измените 0.8 на 0.4, а 0.5 — на 0.25. Сохраните отредактированный файл *easygui.py*. При следующем запуске программы окно с набором вариантов будет иметь меньший размер.

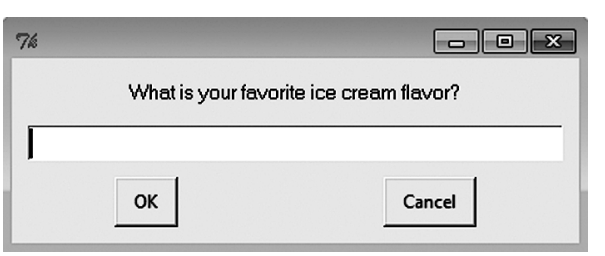
Текстовый ввод

Приведенные примеры позволяли пользователю выбирать из предложенных вами вариантов. А почему бы нам не добавить в программу аналог функции `raw_input()`, позволяющий пользователю ввести собственный вариант ответа? В модуле EasyGui есть элемент `enterbox` — *поле ввода* (enter box), — специально предназначенный для таких случаев. Посмотрим, как это выглядит на практике (листинг 6.3).

Листинг 6.3. Ввод данных в поле ввода

```
import easygui
flavor = easygui.enterbox("What is your favorite ice cream flavor?")
easygui.msgbox ("You entered " + flavor)
```

После запуска программы вы увидите вот такое окно:



Когда вы введете название вашего любимого мороженого и щелкнете по кнопке *OK*, эта информация появится в уже знакомом вам информационном окне.

Фактически вы имеете дело с аналогом функции `raw_input()`, который принимает от пользователя текст (строку).

Ввод по умолчанию

В некоторых ситуациях ответ пользователя можно предугадать, так как некоторые варианты вводятся чаще других. В этом случае можно говорить про вариант, предлагаемый

по умолчанию (default). Он экономит время пользователя, автоматически предлагая наиболее распространенный вариант. В результате пользователю потребуется набирать свой вариант, только если он отличается от предложенного.

В листинге 6.4 показано, как вставить в поле ввода вариант, предлагаемый по умолчанию.

Листинг 6.4. Вариант по умолчанию

```
import easygui
flavor = easygui.enterbox("Твое любимое мороженое?",
                          default = 'Ванильное')
easygui.msgbox ("Вы указали " + flavor)
```

Это вариант, предлагаемый по умолчанию

Теперь при запуске программы вы обнаружите, что в поле уже введен вариант «Ванильное». Вместо него при желании можно ввести любой другой вариант, но если вы действительно больше всего любите ванильное мороженое, вам остается только щелкнуть по кнопке *OK*.

А как насчет цифр?

Чтобы с помощью модуля EasyGui ввести число, всегда можно воспользоваться полем ввода, преобразовав полученную строку при помощи функции `int()` или `float()`, как мы это делали в главе 4.

Кроме того, модуль EasyGui обладает элементом `integerbox`, представляющим собой *поле ввода целых чисел* (integer box). Можно задать верхний и нижний пределы вводимых данных.

Но в это поле нельзя вводить числа с плавающей точкой (десятичные). Для этой операции применяется обычное поле ввода, и полученная строка затем преобразуется в число с помощью функции `float()`.

И снова игра... угадай число

В главе 1 мы написали простую программу для угадывания чисел. Теперь попробуем сделать это еще раз, но уже с использованием модуля EasyGui (листинг 6.5).

Листинг 6.5. Игра в угадывание чисел с применением модуля EasyGui

```
import random, easygui
secret = random.randint(1, 99)
guess = 0
tries = 0
easygui.msgbox("""AHOY! I'm the Dread Pirate Roberts,
and I have a secret!
```

Выбираем секретное число

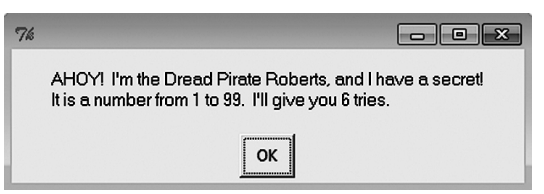
```
It is a number from 1 to 99. I'll give you 6 tries.""")
while guess != secret and tries < 6:
    guess = easygui.integerbox("What's yer guess, matey?")
    if not guess: break
    if guess < secret:
        easygui.msgbox(str(guess) + " is too low, ye scurvy dog!")
    elif guess > secret:
        easygui.msgbox(str(guess) + " is too high, landlubber!")
    tries = tries + 1
if guess == secret:
    easygui.msgbox("Avast! Ye got it! Found my secret, ye did!")
else:
    easygui.msgbox("No more guesses! The number was " + str(secret))
```

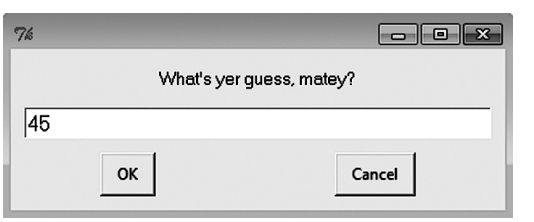
Получаем вариант игрока

Разрешаем до 6 попыток

Выводим сообщение о конце игры

Эта программа содержит пока еще не изученные вами элементы языка Python, поэтому просто введите ее и посмотрите, как все работает. После запуска вы увидите вот такие окна:





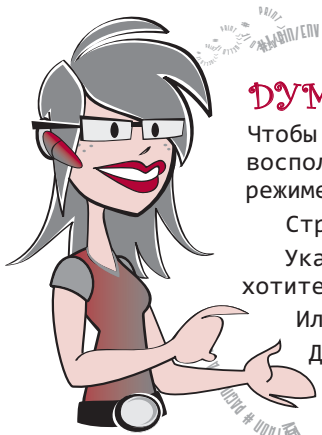
С инструкциями `if`, `else` и `elif` вы познакомитесь в главе 7, а с циклом `while` — в главе 8. В главе 15 вы узнаете о модуле `random`, а поработать с ним сможете в главе 23.

Другие GUI-элементы

В модуле EasyGui есть и другие GUI-элементы: списки, позволяющие выбирать несколько вариантов, диалоговые окна, предназначенные для ввода имен файлов, и т. п. Но на данном этапе вам хватит изученного в этой главе материала.

Модуль EasyGui значительно облегчает генерацию простых графических интерфейсов пользователя, автоматически выполняя ряд сложных операций. Чуть позже вы познакомитесь с альтернативным более гибким способом создания GUI-интерфейсов, предоставляющим вам больше возможностей для контроля.

Подробнее рассмотреть модуль EasyGui можно на странице *easygui.sourceforge.net*.



Думай как программист

Чтобы узнать больше о расширениях языка Python, воспользуйтесь справочной системой. В интерактивном режиме введите: **>>>help()**.

Строка приглашения приобретет вид: **help >**.

Укажите имя элемента, по поводу которого вы хотите получить справку. Например: **help> time.sleep**.

Или: **help> easygui.msgbox**.

Для выхода из справочной системы в обычный интерактивный режим введите команду **quit**:

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

Что мы узнали

В этой главе мы научились:

- создавать простые GUI-интерфейсы при помощи модуля EasyGui;
- отображать сообщения в информационном окне — элементе `msgbox`;
- вводить данные при помощи кнопок, списков выбора и полей ввода, то есть элементов `buttonbox`, `choicebox`, `enterbox`, `integerbox`;
- задавать для текстовых полей значения, предлагаемые по умолчанию;
- пользоваться встроенной справочной системой Python.

Проверь себя

1. Как в модуле EasyGui вызвать информационное окно?
2. Как при работе с модулем EasyGui ввести данные в виде строки (некоего текста)?
3. Как при работе с модулем EasyGui ввести целое число?
4. Как при работе с модулем EasyGui ввести число с плавающей точкой (десятичное)?
5. Что такое значение, предлагаемое по умолчанию? Приведите пример такого значения.

Эксперименты

1. Измените программу преобразования температуры из главы 5, сделав так, чтобы ввод и вывод осуществлялись через GUI, а не через функции `raw_input()` и `print`.
2. Напишите программу, которая последовательно запрашивает ваши индекс, страну, город, улицу, номер дома, номер квартиры и имя (через диалоговые окна EasyGui). Программа должна выводить на экран почтовый адрес:

```
124681 Россия, Москва
ул. Ленина дом 1 кв. 1
Иван Иванов
```

ГЛАВА 7

Решения, решения

В первых главах вы познакомились с некоторыми базовыми строительными блоками программ. Теперь вы умеете писать программы, обеспечивающие *ввод*, *обработку* и *вывод* данных. Более того, ввод и вывод можно красиво оформить при помощи GUI. Вводимые данные можно сохранить в переменной, доступной для дальнейшего использования и обработки. Пришло время познакомиться со способами управления ходом выполнения программы.

Программа, все время выполняющая одни и те же действия, скучна и не очень полезна. Программы должны уметь принимать *решения* по поводу своих действий. Поэтому мы добавим к известным нам средствам *обработки* ряд приемов принятия решений.

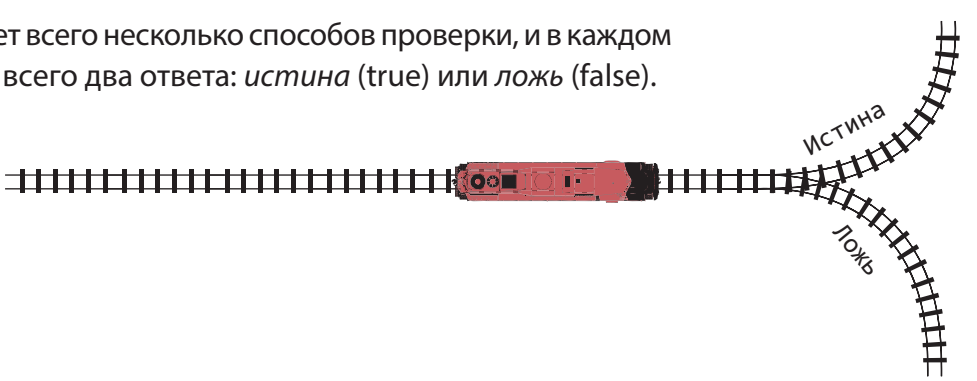
Проверки, проверки

Программы должны уметь выполнять разные действия в зависимости от введенных данных. Вот несколько примеров:

- *если* Тим дает верный ответ, добавляем единицу к его результату;
- *если* Джейн попадает в инопланетянина, должен прогреметь взрыв;
- *если* файл отсутствует, появляется сообщение об ошибке.

Для принятия решения программа проверяет, истинно или ложно определенное *условие*. В первом из приведенных примеров условием является верный ответ.

В Python существует всего несколько способов проверки, и в каждом случае возможны всего два ответа: *истина* (true) или *ложь* (false).



Вот вопросы, которые интерпретатор Python задает при проверке.

- Два элемента равны?
- Один элемент меньше другого?
- Один элемент больше другого?

Но подождите. Ведь таким способом проверить, верен ли ответ, невозможно, по крайней мере напрямую. Это означает, что нам нужно описать процедуру проверки языком, понятным интерпретатору Python.

Когда мы хотим узнать, правильно ли ответил Тим, мы, вероятно, знаем как верный ответ, так и ответ Тима. И можем написать нечто вроде этого:

НОВЫЕ СЛОВА

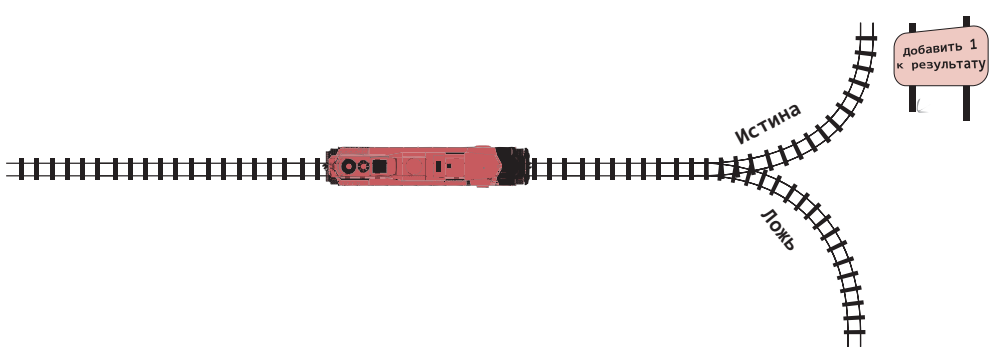
Проверка условий и принятие решений по результатам этой проверки называется *ветвлением* (branching). Программа таким способом выбирает, по какой из возможных ветвей ей двигаться.

В случае правильного ответа Тима переменные будут равны друг другу, и *условие* окажется *истинным*. Неверный ответ означает неравенство двух переменных, и *условие* оказывается *ложным*.

В Python проверка условия осуществляется при помощи ключевого слова `if`:

```
if timsAnswer == correctAnswer:
    print "Ты ответил правильно!"
    score = score + 1
print "Спасибо за игру."
```

Эти две строки, имеющие отступ от верхней и нижней строк, формируют «блок» кода



Двоеточие (:) в конце строки с инструкцией `if` сообщает интерпретатору Python, что дальше находится *блок* команд. В блок входят все строки, расположенные *с отступом* от строки с инструкцией `if`, вплоть до следующей строки *без* отступа.

Если условие *истинно*, выполняется весь расположенный ниже *блок*. В предыдущем коротком примере *блок* инструкций, относящихся к инструкции `if` в первой строке, составляют вторая и третья строки.

НОВЫЕ СЛОВА

Блоком (block) кода называют объединенные друг с другом строки. Они всегда связаны с определенной частью программы (например, с инструкцией if). В Python блоки кода формируются при помощи отступов.

Давайте поговорим об *отступах* и *блоках* кода более подробно.

Отступы

В некоторых языках программирования отступы являются делом личного вкуса, и вы можете вообще обходиться без них. Однако в Python это неотъемлемая часть кода. Именно отступ сообщает интерпретатору Python, где начинается и где заканчивается блок кода.

Некоторым инструкциям в Python, например инструкции `if`, именно блок кода сообщает, какие действия следует предпринять. В случае инструкции `if` блок кода информирует интерпретатор Python, как действовать, *если* условие истинно.

НОВЫЕ СЛОВА

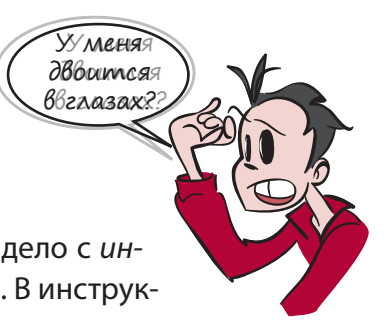
Отступ означает небольшое смещение строки кода вправо. В начале такой строки находятся несколько пробелов, за счет которых она на несколько символов отстоит от левого края.

Размер отступа не имеет значения, главное, чтобы весь блок был сдвинут на одно и то же расстояние. По *соглашению* Python для отступа блоков кода используются четыре пробела. В своих программах имеет смысл придерживаться именно этого соглашения.

Их тут два?

Разве в инструкции `if` должны быть два знака равенства (`if timsAnswer == correctAnswer`)? Да, такая запись является корректной, и вот почему.

Можно сказать: «Пять плюс четыре равно девять», а можно спросить: «Если к пяти прибавить четыре, получится девять?». В первом случае мы имеем инструкцию, а во втором — вопрос.



НОВЫЕ СЛОВА

Соглашение означает всего лишь то, что многие программисты поступают именно так.

В Python мы тоже имеем дело с *инструкциями* и *вопросами*. В инструк-

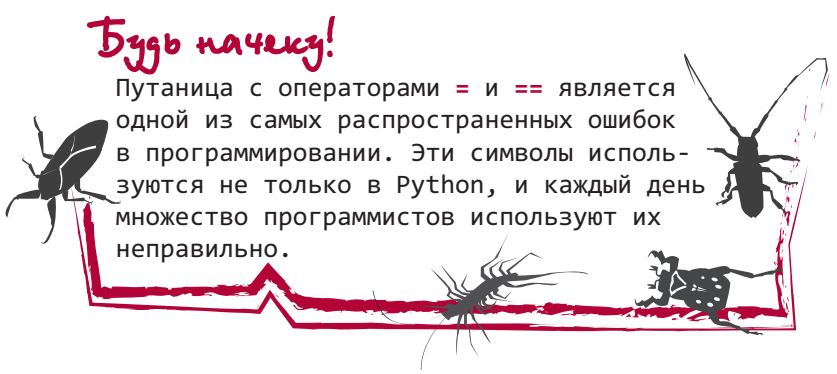
ции переменной присваивается значение. В вопросе проверяется, равна ли переменная определенному значению. В первом случае вы задаете некий элемент (выполняя присваивание, то есть делая его равным). Во втором вы проверяете этот элемент (равен он или нет?). Поэтому в Python мы используем два разных символа.

Вы уже видели, как при помощи знака равенства (=) переменным присваиваются значения. Вот еще несколько примеров:

```
correctAnswer = 5 + 3
temperature = 35
name = "Bill"
```

Для проверки двух элементов на равенство интерпретатор Python использует удвоенный знак равенства (==). Вот так:

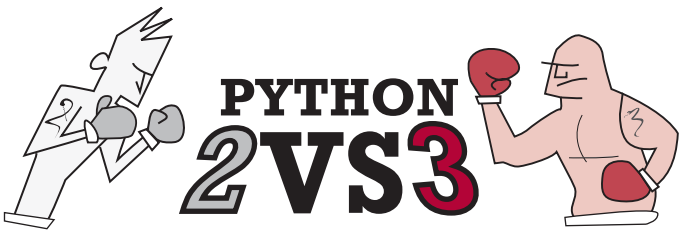
```
if myAnswer == correctAnswer:
if temperature == 40:
if name == "Fred":
```



Проверка также называется *сравнением*. Соответственно, двойной знак равенства называют *оператором сравнения* (comparison operator). Помните, в главе 3 мы уже говорили об *операторах*, то есть о специальных символах, которые оперируют стоящими рядом значениями? В данном случае операция представляет собой проверку двух значений на равенство.

Другие виды проверок

К счастью для нас, остальные операторы сравнения запомнить проще: меньше чем (<), больше чем (>) и не равно (!=). Последний оператор также можно записать в виде <>, но большинство программистов пользуются формой записи !=. Еще можно скомбинировать оператор > или < с оператором =, получив операторы больше или равно (>=) и меньше или равно (<=). Некоторые из них вы могли видеть на уроках математики.



Не равно
В версии Python 3 запись `<>` для оператора неравенства уже не поддерживается. Значит, можно использовать только оператор `!=`.

Еще можно объединить два оператора больше чем или меньше чем и проверить, попадает ли число в определенный интервал:

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

В данном случае проверяется, попадает ли переменная age в промежуток от 8 до 12, исключая граничные значения. Условие будет выполнено при равенстве переменной age значению 9, 10 или 11 (или 8.1, или 11.6, и т. п.). Если же мы хотим включить в проверку и граничные значения 8 и 12, следует написать:

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

НОВЫЕ СЛОВА

Операторы сравнения еще называют *операторами отношения* (relational operators), так как они проверяют отношения двух сторон: равно или не равно, больше чем или меньше чем. А саму процедуру сравнения еще называют *проверкой условия* (conditional test), или *логической проверкой* (logical test). В программировании слово «логический» относится к действиям, результат которых может быть истинным или ложным.

В листинге 7.1 показан пример программы, в которой применяются сравнения. Откройте в IDLE-редакторе новый файл, введите программу и сохраните ее под именем ***compare.py***. Затем выполните команду ***Run***. Попробуйте запустить ее несколько раз, используя разные числа. Проверьте варианты, когда первое число больше второго, когда первое число меньше второго и когда оба числа равны друг другу.

Листинг 7.1. Применение операторов сравнения

```
num1 = float(raw_input("Введите первое число: "))
num2 = float(raw_input("Введите второе число: "))
if num1 < num2:
    print num1, "меньше чем", num2
if num1 > num2:
    print num1, "больше чем", num2
if num1 == num2:
    print num1, "равно", num2
if num1 != num2:
    print num1, "не равно", num2
```

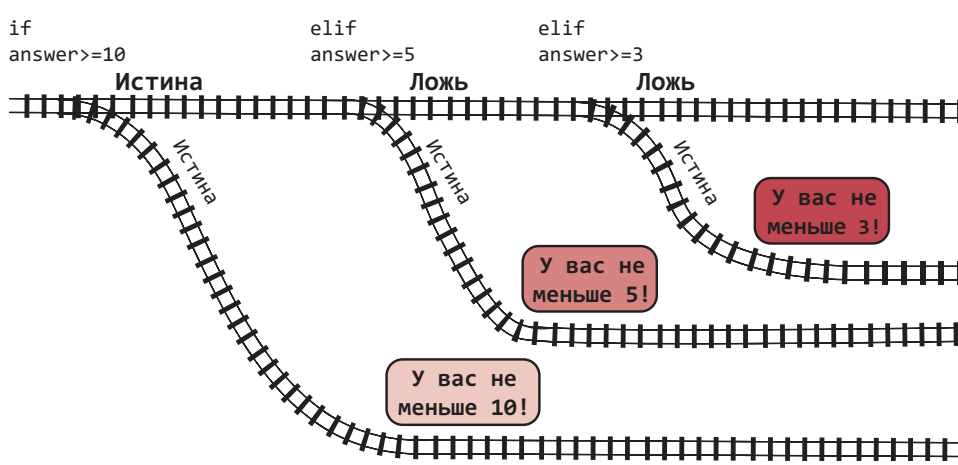
Здесь должен использоваться двойной знак равенства

Если условие ложно

Вы уже знаете, как заставить интерпретатор Python выполнить некие действия при соблюдении поставленного условия. А что делать, если условие *ложно*? У вас три возможности.

- *Провести еще одну проверку*. Если результат первой проверки оказался *ложным*, можно заставить интерпретатор Python рассмотреть другое условие. Для этого применяется ключевое слово `elif` (что является сокращением от «else if» — «в противном случае, если»):

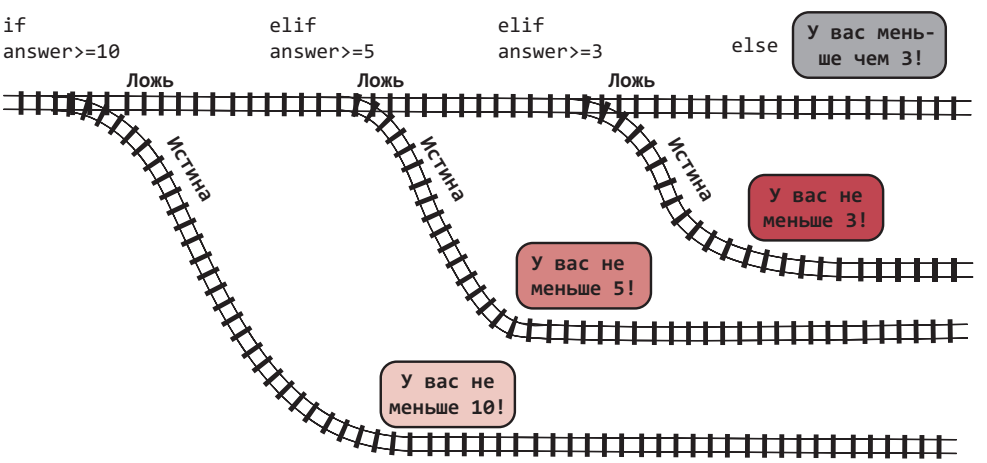
```
if answer >= 10:
    print "У вас не меньше 10!"
elif answer >= 5:
    print "У вас не меньше 5!"
elif answer >= 3:
    print "У вас не меньше 3!"
```



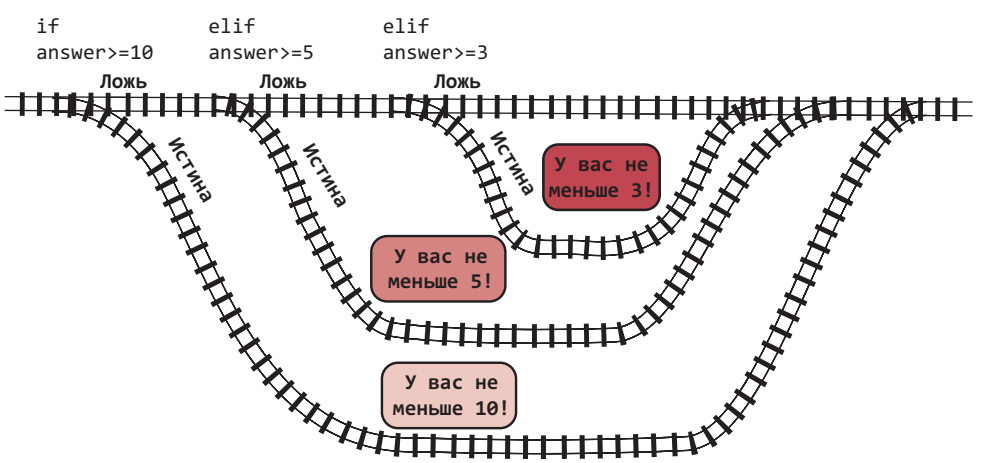
Количество инструкций `elif` после инструкции `if` может быть произвольным.

- Выполнить *какие-то другие действия*, если результаты всех проверок оказались *ложными*. Для этого существует ключевое слово `else`. Оно всегда фигурирует в самом конце, после инструкций `if` и `elif`:

```
if answer >= 10:
    print "У вас не меньше 10!"
elif answer >= 5:
    print " У вас не меньше 5!"
elif answer >= 3:
    print " У вас не меньше 3!"
else:
    print "У вас меньше, чем 3."
```



- *Двигаться дальше*. Если после блока `if` вы не поместили никаких других инструкций, программа перейдет к выполнению следующей строки кода (если таковая присутствует) или завершится (если кода больше нет).



Попробуйте написать программу на основе приведенного здесь кода, добавив в начало строку для ввода числа:

```
answer = float(raw_input ("Введите число от 1 до 15"))
```

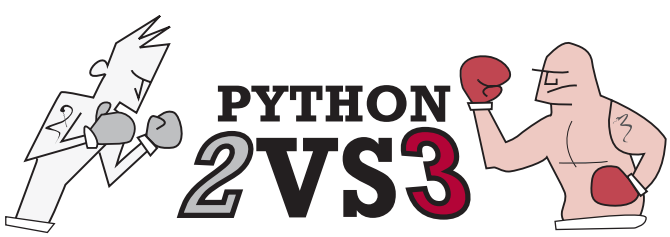
Не забудьте сохранить файл (на этот раз выберите имя самостоятельно). Запустите его несколько раз с разными входными данными, чтобы посмотреть, как работает программа.

Проверка нескольких условий

Как быть в ситуации, когда у вас есть несколько условий? Предположим, вы хотите написать игру для детей от восьми лет, которые при этом учатся по крайней мере в третьем классе. То есть должны соблюдаться сразу два условия. Вот один из способов их проверки:

```
age = float(raw_input("Сколько вам лет?: "))
grade = int(raw_input("В каком классе вы учитесь?: "))
if age >= 8:
    if grade >= 3:
        print "Вы можете играть."
else:
    print "Извините, вам нельзя играть."
```

Обратите внимание, что первая строка с инструкцией `print` сдвинута вправо не на четыре, а на восемь пробелов. Дело в том, что каждой инструкции `if` должен соответствовать собственный блок, поэтому каждый из них имеет свой отступ.



Памятка
Напоминаем, что в версии Python 3 нужно поменять **`raw_input()`** на **`input()`**, а аргумент команды **`print`** следует поместить в скобки: **`print("Вы можете играть.")`**

Ключевое слово and

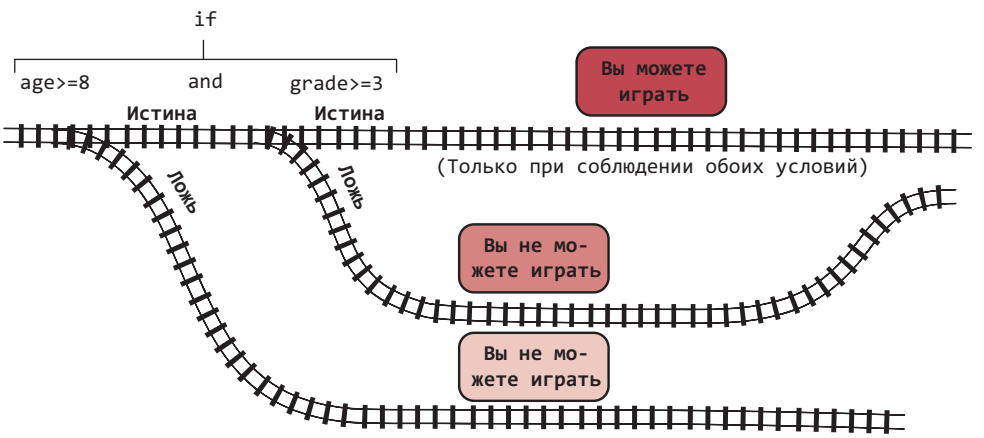
Последний фрагмент кода прекрасно работает, но данный результат можно получить и более коротким способом. Условия следует объединить так:

```
age = float(raw_input("Сколько вам лет?: "))
grade = int(raw_input("В каком классе вы учитесь?: "))
```

```
if age >= 8 and grade >= 3:
    print "Вы можете играть."
else:
    print "Извините, вам нельзя играть."
```

Объединение условий ключевым словом and

Мы объединили два условия при помощи ключевого слова and. Оно означает, что следующий ниже блок выполняется только при соблюдении обоих условий.



Ключевое слово and объединяет произвольное число условий:

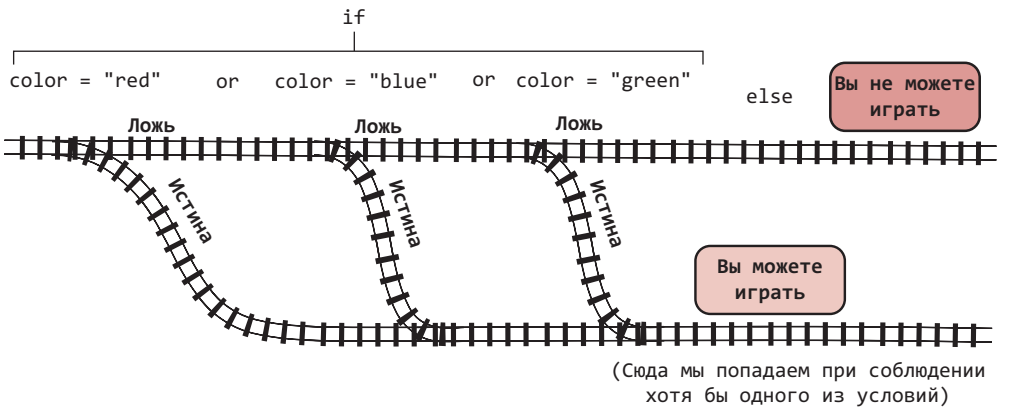
```
age = float(raw_input("Сколько вам лет?: "))
grade = int(raw_input("В каком классе вы учитесь?: "))
color = raw_input("Ваш любимый цвет?: ")
if age >= 8 and grade >= 3 and color == "зеленый":
    print "Вам можно играть в эту игру."
else:
    print "Извините, вам нельзя играть."
```

Если у вас есть несколько условий, для истинности инструкции `if` все они должны быть истинными. Однако существуют и другие способы объединения условий.

Ключевое слово or

Ключевое слово or также применяется для объединения условий. Но в этом случае для выполнения блока достаточно истинности хотя бы одного из них:

```
color = raw_input("Ваш любимый цвет?: ")
if color == "красный" or color == "синий" or color == "зеленый":
    print "Вам можно играть в эту игру."
else:
    print "Извините, вам нельзя играть."
```



Ключевое слово not

Сравниваемый компонент можно перевернуть с ног на голову при помощи ключевого слова `not`. Например:

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

Эта строка полностью аналогична такой строке:

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

В обоих случаях блок выполняется в случае, если задан возраст от 8 лет и выше.

В главе 4 вы познакомились с *математическими операторами* `+`, `-`, `*` и `/`. В этой главе вы узнали об *операторах сравнения* `<`, `>`, `==` и т. п. Ключевые слова `and`, `or` и `not` также являются операторами.

Они называются *логическими операторами* и используются для модификации сравниваемых значений, которые могут объединяться (`and`, `or`) или инвертироваться (`not`) — менять свое значение на противоположное. В табл. 7.1 перечислены все уже известные вам операторы.

Можете положить закладку на эту страницу, чтобы к таблице можно было легко вернуться в любой момент.

Что мы узнали?

В этой главе мы познакомились с:

- проверкой условий и операторами отношения;
- отступами и блоками кода;
- объединением условий при помощи ключевых слов `and` и `or`;
- изменением значения условия на обратное при помощи ключевого слова `not`.

Таблица. 7.1. Список математических операторов и операторов сравнения

Оператор	Название	Описание
Математические операторы		
=	Присваивание	Присваивает имени (переменной) значение
+	Сложение	Складывает два числа. Может применяться для объединения строк
–	Вычитание	Вычитает одно число из другого
+=	Инкремент	Добавляет к числу единицу
–=	Декремент	Уменьшает число на единицу
*	Умножение	Перемножает два числа
/	Деление	Делит одно число на другое. Если оба числа целые, результатом будет целая часть без остатка
%	Получение остатка от деления	Дает остаток от деления двух целых чисел
**	Возведение в степень	Возводит число в степень. Число и показатель степени могут быть целыми и числами с плавающей точкой
Операторы сравнения		
==	Равенство	Проверяет равенство двух элементов
<	Меньше чем	Проверяет, меньше ли первое число второго
>	Больше чем	Проверяет, больше ли первое число второго
<=	Меньше или равно	Проверяет, что первое число меньше или равно второму
>=	Больше или равно	Проверяет, что первое число больше или равно второму
!=	Неравенство	Проверяет неравенство двух элементов. (Могут использоваться оба оператора.)
<>		

Проверь себя

1. Каким будет результат работы такой программы?

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

2. Каким будет результат работы той же программы, если переменной `my_number` присвоить значение 25?
3. Как будет выглядеть инструкция `if`, проверяющая, что число больше 30, но меньше или равно 40?
4. Как будет выглядеть инструкция `if`, проверяющая, в каком регистре — верхнем или нижнем — пользователь ввел букву Q?

Эксперименты

1. В магазине распродажа. На товары за 10 долларов и меньше скидка 10 %, а на товары дороже 10 долларов — 20 %. Напишите программу, которая будет запрашивать цену товара и показывать размер скидки (10 или 20 %) и итоговую цену.

2. Футбольная команда набирает девочек от 10 до 12 лет. Напишите программу, которая будет запрашивать возраст и пол претендента, используя для обозначения пола буквы `m` (от male — мужчина) и `f` (от female — женщина). Программа должна выводить на экран сообщение, информирующее, подходит ли претендент для вступления в команду.

   Дополнительное задание: программа должна запрашивать возраст только для претендентов женского пола.

3. Путешествуя на автомобиле, вы заехали на заправку. До следующей заправки 200 километров. Напишите программу, которая будет определять, нужно ли вам заправляться или же можно подождать до следующей станции.

   Программа должна спрашивать:

   - Каков размер вашего бензобака в литрах?
   - Сколько горючего в бензобаке (в процентах)?
   - Сколько километров проходит автомобиль на одном литре бензина?

   Результат работы программы должен выглядеть примерно так:

```
Размер бензобака: 60
Заполненность в процентах: 40
Км на литр: 10
Вы можете проехать еще 240 км.
Следующая заправка через 200 км.
Можно подождать следующей заправки.
```

   Или:

```
Размер бензобака: 60
Заполненность в процентах: 30
Км на литр: 8
Вы можете проехать еще 144 км
Следующая заправка через 200 км.
ЗАПРАВЬТЕСЬ СЕЙЧАС!
```

   Добавьте к программе погрешность в 5 литров на случай не совсем точных показаний расхода топлива.

4. Напишите программу, для работы с которой пользователю потребуется ввести пароль. Вы-то пароль знаете (так как он содержится в самом коде). А вот вашим друзьям придется либо спрашивать его у вас, либо пытаться подобрать, либо как следует изучить Python, чтобы найти нужную информацию в коде программы!

   Программа может быть любой. Возьмите готовую или напишите простейшую, при вводе корректного пароля выводящую на экран сообщение «Вот вы и вошли!»

ГЛАВА 8

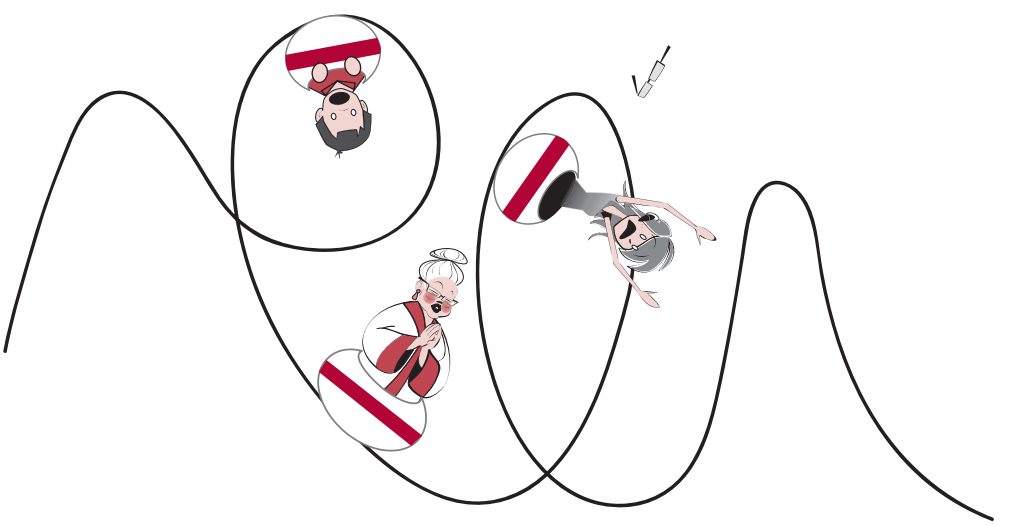
Циклы

Большинство людей не любят раз за разом выполнять одни и те же действия, поэтому возникает вопрос: почему бы не переложить эту задачу на компьютер? Компьютеры не устают, им не бывает скучно, то есть они отлично подходят для решения повторяющихся задач. В этой главе вы узнаете, как заставить компьютер повторять одни и те же действия.

Компьютерные программы часто снова и снова делают одно и то же. Этот процесс называется *выполнением цикла* (looping).

Существуют две основных разновидности циклов:

- циклы, повторяющиеся определенное число раз. Их называют *счетными циклами* (counting loops);
- циклы, повторяющиеся до наступления определенного события. Их называют *условным циклами* (conditional loops), так как они продолжаются, пока соблюдается некое условие.



Счетные циклы

Счетные циклы называют также циклами `for`, так как во многих языках, в том числе в Python, для создания таких циклов применяется ключевое слово `for`.

Попробуем применить такой цикл на практике. Откройте в IDLE новое окно, выбрав в меню команду *File ▸ New* (как мы делали в самой первой программе), и введите туда код из листинга 8.1.

Листинг 8.1. Очень простой цикл for

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

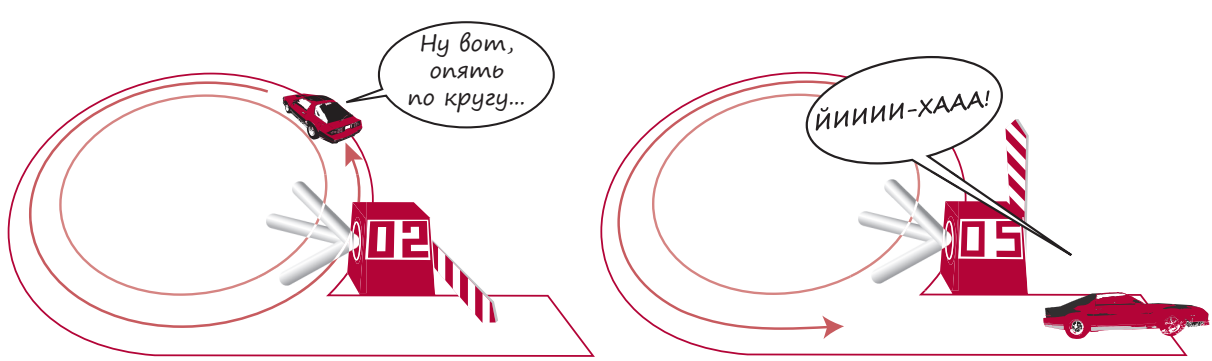
Сохраните файл под именем *Loop1.py* и запустите программу. (Для этого можно выбрать в меню команду *Run ▸ Run Module* или нажать клавишу *F5*.)

Вы увидите нечто вот такое:

```
>>> ================ RESTART ================
>>>
hello
hello
hello
hello
hello
```

Кажется, у нас появилось эхо. Программа вывела слово «hello» пять раз, хотя инструкция `print` у нас всего одна. Почему же так получилось? Первая строка (`for looper in [1, 2, 3, 4, 5]:`) в переводе означает следующее:

1) переменная `looper` сначала имеет значение 1 (то есть `looper` = 1);
2) цикл выполняет расположенный ниже блок команд один раз для каждого значения из списка (Списком в данном случае являются цифры в квадратных скобках.);



3) на каждом следующем шаге цикла переменной `looper` присваивается следующее значение из списка.

Вторая строка (`print "hello"`) представляет собой блок, который интерпретатор Python выполняет на каждом проходе цикла. Циклу `for` блок кода необходим, чтобы программа знала, какие действия ей нужно выполнять на каждом проходе. Этот блок (часть кода с отступом) называется *телом цикла*. (Вспомните, что говорилось про отступы блоков в предыдущей главе.)

НОВЫЕ СЛОВА

Каждый проход цикла называется *итерацией*.

Попробуем выполнить еще какие-нибудь действия. Пусть программа выводит на экран не одно и то же значение, а разные (листинг 8.2).

Листинг 8.2. Разные действия на разных проходах цикла

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

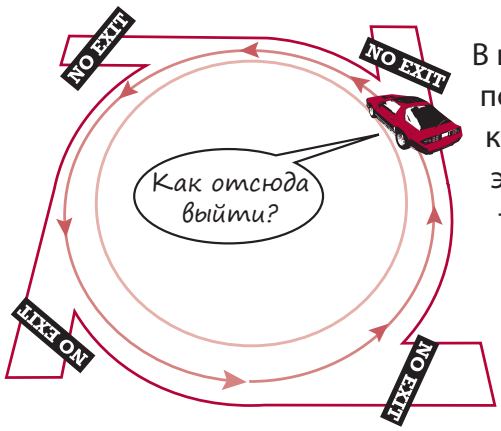
Сохраните файл под именем *Loop2.py* и запустите. Результат будет выглядеть так:

```
>>> ================ RESTART ================
>>>
1
2
3
4
5
```

На этот раз вместо пятикратного вывода слова `hello` мы показываем значение переменной `looper`. На каждом проходе цикла эта переменная принимает очередное значение из списка.

ЦИКЛ ВЫХОДИТ ИЗ-ПОД КОНТРОЛЯ

Со мной это тоже случалось, Картер! С циклами, вышедшими из-под контроля (еще их называют бесконечными). Периодически сталкивается любой программист. Для остановки Python-программы в произвольный момент (даже во время бесконечного цикла) следует нажать комбинацию клавиш *Ctrl*+*C*. Это означает, что мы нажимаем и удерживаем клавишу *Ctrl* и одновременно нажимаем клавишу *C*. Данный прием вам еще пригодится!



В играх и графических программах циклы используются постоянно. Им нужно то и дело получать данные с мыши, клавиатуры или игрового контроллера, обрабатывать эти данные и обновлять экран. Когда мы начнем писать такие программы, в них будет множество циклов. И может возникнуть ситуация, когда программа в какой-то точке не сможет выйти из цикла, а значит, нам нужно знать, как действовать в этом случае!

Зачем нужны квадратные скобки

Возможно, вы обратили внимание на квадратные скобки. В них заключен перечень значений цикла. Квадратные скобки и запятые между числами в Python являются атрибутами *списков* (lists). О списках мы подробно поговорим в главе 12, а пока достаточно знать, что это «контейнер» для хранения группы элементов. Здесь элементами являются цифры — значения, которые переменная `looper` принимает на различных итерациях цикла.

Применение счетного цикла

Давайте найдем циклам более полезное применение. Например выведем на экран таблицу умножения. Для этого внесем в нашу программу небольшие изменения (листинг 8.3).

Листинг 8.3. Вывод таблицы умножения на 8

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

Сохраните файл под именем ***Loop3.py*** и запустите программу. Вы увидите следующее:

```
>>> ================== RESTART ==================
>>>
1 умножить на 8 = 8
2 умножить на 8 = 16
3 умножить на 8 = 24
4 умножить на 8 = 32
5 умножить на 8 = 40
```

Теперь вы видите, каким мощным инструментом являются циклы. Без них для получения аналогичного результата пришлось бы писать вот такую программу:

```
print "1 умножить на 8 =", 1 * 8
print "2 умножить на 8 =", 2 * 8
```

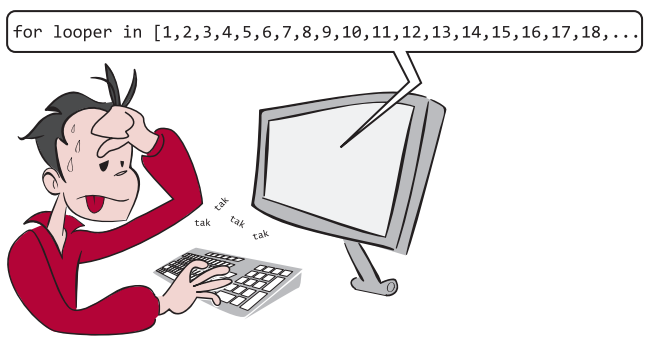
```
print "3 умножить на 8 =", 3 * 8
print "4 умножить на 8 =", 4 * 8
print "5 умножить на 8 =", 5 * 8
```

Для получения более длинной таблицы умножения (например до значения 10 или даже 20) эту программу пришлось бы делать существенно длиннее, а вот программа с циклом практически не изменится (просто в списке появятся дополнительные цифры). Циклы многое упрощают!

Функция range()

В предыдущем примере цикл повторялся всего 5 раз:

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



А что делать, если цикл нужно запустить 100 или 1000 раз? Сколько же придется печатать!

Но существует возможность облегчить себе жизнь. Функция `range()` позволяет указывать только начальное и конечное значения, создавая за вас все остальные значения диапазона. В листинге 8.4 мы выведем таблицу умножения при помощи функции `range()`.

Листинг 8.4. Цикл с функцией range()

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

Сохраните этот файл под именем *Loop4.py* и запустите программу. (Можно выбрать в меню команду *Run ▸ Run Module* или нажать клавишу *F5*.) Вот что из этого получится:

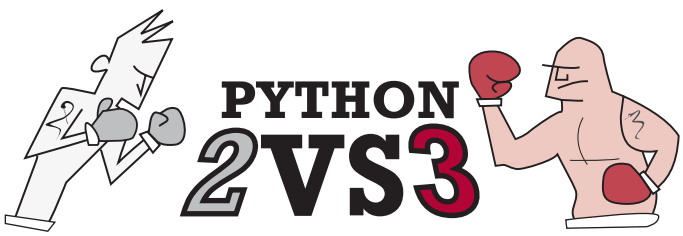
```
>>> ================ RESTART ================
>>>
1 умножить на 8 = 8
2 умножить на 8 = 16
3 умножить на 8 = 24
4 умножить на 8 = 32
```

Мы получили почти тот же самый результат, что и в первом случае… но последняя итерация почему-то не выполнилась! Что же случилось?

Дело в том, что функция `range (1, 5)` дает нам список `[1, 2, 3, 4]`. Вы сами можете проверить это в интерактивном режиме:

```
>>> print range(1, 5)
[1, 2, 3, 4]
```

Почему не 5? Просто функция `range()` работает именно таким образом. Она формирует список, начиная с первого числа и заканчивая *перед* последним. Эту особенность следует учитывать, изменяя диапазон для получения нужных нам значений.



Диапазон
В версии Python 3 вы получите немного другой результат:
`>>> print(range(1, 5))`
`range(1,5)`
Дело в том, что в Python 3 функция `range()` создает не список чисел, а интерфейс `iterable`. Именно он реализует итерации цикла. В цикле `for` эта функция работает так же, как в предыдущем случае. Просто внутри она немного другая.

Листинг 8.5 демонстрирует нам отредактированную версию программы, в которой таблица умножения на 8 выводится до значения 10 включительно.

Листинг 8.5. Вывод таблицы умножения на 8 до 10 включительно

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

Вот что мы получим после ее запуска:

```
>>> ================== RESTART ==================
>>>
1 умножить на 8 = 8
2 умножить на 8 = 16
3 умножить на 8 = 24
4 умножить на 8 = 32
5 умножить на 8 = 40
6 умножить на 8 = 48
7 умножить на 8 = 56
```

```
8 умножить на 8 = 64
9 умножить на 8 = 72
10 умножить на 8 = 80
```

В программе из листинга 8.5 функция `range(1, 11)` дает нам список чисел от 1 до 10 включительно, при этом выполняется по одной *итерации* для каждого числа из списка. На каждом проходе цикла переменная `looper` принимает следующее значение из списка.

В данном случае мы назвали переменную цикла `looper`, но вы можете выбрать для нее любое имя, которое вам нравится.

Имена переменных цикла

Переменная цикла ничем не отличается от любой другой переменной. В ней нет ничего особенного — это всего лишь имя для значения. И неважно, что вы используете ее как счетчик итераций.

Ранее говорилось, что имена переменных должны описывать их назначение. Именно поэтому в предыдущем примере я выбрал имя `looper`. Но для переменных цикла иногда делается исключение. Дело в том, что в программировании существует соглашение (напоминаю, что это означает общепринятую практику) использовать для переменных цикла буквы `i`, `j`, `k` и т. д.

Так как многие называют свои переменные цикла `i`, `j` и `k`, программисты к этому привыкли. Можно выбирать и другие имена, как это сделали мы. Но буквы `i`, `j` и `k` не стоит использовать для переменных, не занятых в цикле.

В старые добрые времена

Почему для циклов зарезервированы буквы i, j и k? Дело в том, что раньше программы использовались для математических расчетов, а в математике буквы a, b, c и x, y, z уже зарезервированы для других целей. Кроме того, в популярных языках программирования переменные i, j и k всегда являются целыми — им нельзя присваивать другой тип. Счетчиками цикла также всегда являются целые числа, поэтому программисты выбирали для этой цели переменные i, j и k, и это стало общепринятой практикой.

С учетом данного соглашения наша программа приобретет следующий вид:

```
for i in range(1, 5):
    print i, "умножить на 8 =", i * 8
```

Работать она будет совершенно так же. (Проверьте и убедитесь!)

Выбор имен для переменных цикла — это дело *стиля*. Этот выбор влияет на внешний вид программ, а не на их работу. Однако следование стилю, привычному другим программистам, облегчает чтение, понимание и отладку ваших программ. Кроме того, у вас выработается привычка к данному стилю, благодаря чему вам будет проще разбираться в чужих программах.

УПРОЩЕНИЕ ФУНКЦИИ RANGE()

В функции `range()` вовсе не обязательно указывать два числа, как мы это делали в листинге 8.5. Можно обойтись одним:

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

Это аналогично записи:

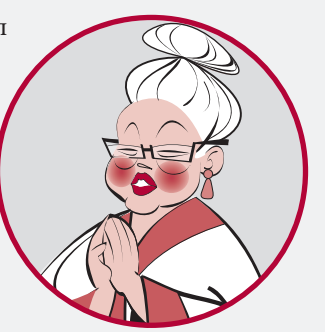
```
for i in range(0, 5):
```

В обоих случаях получается список цифр: `[0, 1, 2, 3, 4]`.

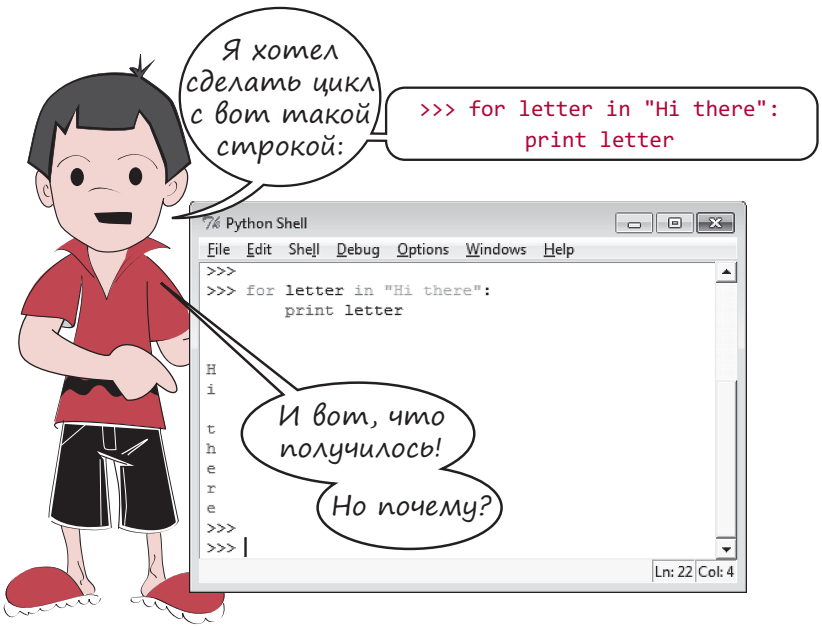
На самом деле многие программисты начинают циклы с 0, а не с 1. Написав `range(5)`, вы получите 5 итераций цикла, что очень легко запомнить. Просто следует знать, что первое значение `i` равно 0, а не 1, а последнее 4, а не 5.

В СТАРЫЕ ДОБРЫЕ ВРЕМЕНА

Почему большинство программистов начинают цикл с 0, а не с 1?
Раньше некоторые начинали цикл с 1, а некоторые с 0. Те и другие приводили весьма изощренные аргументы, споря о том, какой способ лучше. Но в конце концов победили сторонники второго варианта. С тех пор большинство начинают циклы с 0, но вы можете действовать так, как вам будет удобно. Просто помните о необходимости редактировать верхнюю границу диапазона для получения нужного количества итераций.



Что ж, Картер, ты познакомился со свойством строк. Текстовая строка похожа на *список символов*. Ты уже знаешь, что счетные циклы осуществляют *итерации* на основе *списков*. Это означает, что можно построить цикл на основе строки. Каждый ее символ соответствует одной итерации цикла. Поэтому при выводе на экран переменной цикла, которую Картер в своем примере назвал `letter`, мы будем выводить символы строки по одному за проход. Так как каждая инструкция `print` начинает новую строчку, каждая буква оказывается на собственной строке.



Эксперименты и попытки идти разными путями, как в данном случае поступил Картер, являются замечательным способом обучения!

Пошаговый отсчет

До этого момента на каждой итерации мы прибавляли к переменной цикла единицу. А что делать, если мы хотим прибавлять на каждом шаге 2? Или 5, или даже 10? А как насчет обратного отсчета?

В функцию `range()` можно добавить еще один *аргумент*, позволяющий менять величину шага, который по умолчанию равен 1.

НОВЫЕ СЛОВА

Аргументами называются значения, которые при использовании функций указываются в скобках. Мы говорим о передаче аргумента в функцию. Также в этом контексте применяется термин *параметр*, то есть можно сказать «передать параметр». Как и функции, аргументы и параметры рассматриваются в главе 13.

Рассмотрим несколько циклов в интерактивном режиме. После ввода первой строки кода с двоеточием на конце IDLE автоматически делает отступ, так как для цикла требуется отдельный блок кода. Завершив ввод блока, дважды нажмите клавишу ***Enter***.

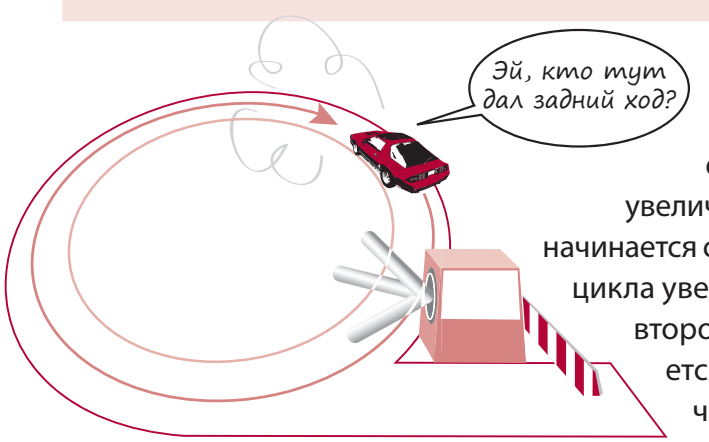
```
>>> for i in range(1, 10, 2):
    print i
1
3
5
7
9
```

Мы добавили к функции `range()` третий параметр — цифру 2. Теперь переменная цикла меняется с шагом 2. Попробуем другой вариант:

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

Теперь шаг равен 5. А как насчет обратного отсчета?

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



Сделав третий параметр функции `range()` отрицательным, мы заставляем переменную цикла *уменьшаться*, а не увеличиваться. Надеюсь, вы помните, что отсчет начинается с первого указанного числа и переменная цикла увеличивается (или уменьшается), при этом второе указанное число в процесс не включается, поэтому в нашем примере последним числом будет 2, а не 1.

Мы можем написать программу для таймера обратного отсчета. Достаточно добавить еще пару строк. Откройте новое окно в IDLE и введите код из листинга 8.6.

Листинг 8.6. Готовы к взлету?

```
import time
for i in range(10, 0, -1):
    print i
    time.sleep(1)
print "ПУСК!"
```

Обратный отсчет

Задержка в одну секунду

Не волнуйтесь по поводу тех элементов программы, которые мы пока не рассматривали, таких как `import`, `time` и `sleep`. Мы познакомимся с ними в следующих главах. А пока просто наберите программу из листинга 8.6 и посмотрите, как она работает. Особое внимание следует обратить на строку `range (10, 0, -1)`, которая заставляет цикл считать от 10 до 1.

Нецифровой отсчет

Во всех предыдущих примерах в качестве переменной цикла фигурировало число. В терминах программирования цикл *осуществлял перебор* списка чисел. Но список может состоять из чего угодно. Как вы уже видели в эксперименте Картера, можно взять список символов (строку). А можно воспользоваться списком строк или еще чем-нибудь.

Как это работает, лучше всего посмотреть на примере. Запустите программу из листинга 8.7 и посмотрите, что получится.

Листинг 8.7. Кто из них самый крутой?

```
for cool_guy in ["Губка Боб Квадратные Штаны", "Человек-паук",
"Джастин Тимберлейк", "Мой папа"]:
    print cool_guy, "самый крутой парень!"
```

Здесь в цикле перебирается не список чисел, а список текстовых строк. Поэтому переменная цикла называется не `i`, а `cool_guy`. На каждой итерации цикла она приобретает новое значение. Это тоже своего рода *счетный цикл*, несмотря на отсутствие списка чисел. Интерпретатор Python *подсчитывает* количество элементов списка, чтобы понять, сколько итераций цикла он должен сделать. (На этот раз выходные данные я показывать не буду, вы сами их увидите, запустив программу.)

А что делать, если вы заранее не знаете, сколько итераций вам потребуется? Что, если списка доступных значений просто не существует? Именно об этом пойдет речь в следующем разделе!

Раз уж речь зашла о циклах...

Вы только что познакомились с циклом `for`, или *счетным циклом*. Но существует и другая разновидность — это цикл `while`, или *условный цикл*.

Цикл `for` замечательно работает, если вы заранее знаете, сколько итераций вам требуется сделать. Но иногда нужно, чтобы цикл выполнялся до наступления некоего события, и количество итераций в этом случае заранее оценить просто нереально. И здесь вам на помощь приходит цикл `while`.

В предыдущей главе вы узнали, что такое условия и как выполнить их проверку при помощи инструкции `if`. Вместо подсчета количества итераций цикл `while` также проводит проверку, пытаясь определить, когда ему следует прекратить свою работу. Именно поэтому его называют *условным циклом*. Он работает, пока не будет выполнено некое условие.

По сути, цикл `while` до конца своего выполнения постоянно спрашивает: «Я уже завершился?.. Я уже завершился?.. Я уже завершился?..». И так продолжается до момента, когда условие перестает быть истинным.

Условный цикл в Python реализуется при помощи ключевого слова `while`. Пример программы с таким циклом демонстрирует листинг 8.8. Запустите эту программу, чтобы посмотреть, как она работает. (Помните, что сначала вам нужно воспользоваться командой ***Save***, а затем — командой ***Run***.)

Листинг 8.8. Условный цикл

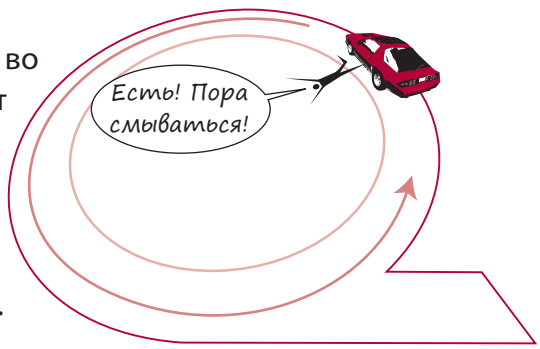
```
print "Введите 3, чтобы продолжить и любой другой символ для выхода."
someInput = raw_input()
while someInput == '3':
    print "Спасибо за 3. Вы очень добры."
    print " Введите 3, чтобы продолжить
и любой другой символ для выхода."
    someInput = raw_input()
print "Это не 3, поэтому я завершаю работу."
```

Цикл продолжается, так как someInput ='3'

Тело цикла

Эта программа постоянно просит пользователя ввести данные. *Пока* пользователь вводит цифру 3, условие соблюдается и цикл продолжается. Ввод любого *другого* символа нарушает условие продолжения цикла, и цикл завершается.

Вмешательство в работу цикла



Иногда возникает необходимость вмешаться в цикл во время его работы, еще до того как цикл `for` завершит подсчет или цикл `while` достигнет условия выхода. Это можно сделать двумя способами: перейти на следующую итерацию цикла при помощи инструкции `continue` или остановить итерации при помощи инструкции `break`. Посмотрим более подробно, как это происходит.

Прыжок вперед — инструкция continue

Если вы хотите остановить текущую итерацию и перейти к следующей, вам поможет инструкция `continue`. Ее действие демонстрирует листинг 8.9.

Листинг 8.9. Применение в цикле инструкции continue

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

Вот как будет выглядеть результат работы такой программы:

```
>>> ================== RESTART ==================
>>>
i = 1 Привет, как твои дела?

i = 2 Привет, как твои дела?

i = 3 Привет, как
i = 4 Привет, как твои дела?

i = 5 Привет, как твои дела?
```

Обратите внимание, что на третьей итерации цикла (когда `i == 3`) его тело выполняется не полностью — происходит переход к следующей итерации (`i == 4`). Это результат вмешательства инструкции `continue`. Аналогичным образом она функционирует в цикле `while`.

Выход из цикла — инструкция break

А что делать, если вы хотите вообще выйти из цикла, не завершая счет или не дожидаясь условия выхода? Здесь вам поможет инструкция `break`.

Изменим строку 6 в листинге 8.9, заменив `continue` на `break`, и вновь запустим программу:

```
>>> ================== RESTART ==================
>>>
i = 1 Привет, как твои дела?
i = 2 Привет, как твои дела?
i = 3 Привет, как
```

На этот раз цикл не пропускает часть итерации 3, а вообще завершается. Это результат выполнения инструкции `break`. Аналогичным способом он функционирует в цикле `while`.

Многие программисты считают применение инструкций `break` и `continue` *плохим тоном*. Лично я так не считаю, и хотя все равно я их практически не использую, я решил рассказать вам об инструкциях `break` и `continue` на случай, если они вам понадобятся.

Что мы узнали

В этой главе мы познакомились с:

- циклом `for` (известным как счетный цикл);
- функцией `range()`, упрощающей использование счетных циклов;
- возможностью менять шаг в функции `range()`;
- циклом `while` (известным как условный цикл);
- переходом к следующей итерации при помощи инструкции `continue`;
- выходом из цикла при помощи инструкции `break`.

Проверь себя

1. Сколько итераций сделает такой цикл?

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

2. Сколько итераций сделает такой цикл? И какое значение будет у переменной `i` на каждой итерации?

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

3. Какой список чисел даст вам функция `range(1,8)`?
4. Какой список чисел даст вам функция `range(8)`?
5. Какой список чисел даст вам функция `range(2,9,2)`?
6. Какой список чисел даст вам функция `(10,0,-2)`?
7. Какое ключевое слово позволяет остановить текущую итерацию цикла и перейти к следующей?
8. Когда завершается цикл `while`?

Эксперименты

1. Напишите программу, выводящую на экран таблицу умножения. Первым делом она должна спрашивать, для какого числа требуется вывести таблицу. Результат будет выглядеть примерно так:

```
Для какого числа нужна таблица умножения?
5
Вот ваша таблица:
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

2. Скорее всего, в этой программе вы использовали цикл `for`. Так поступило бы большинство программистов. Но для практики перепишите ее с циклом `while`.

   Если же задачу из первого задания вы решили при помощи цикла `while`, перепишите ее с циклом `for`.

3. Добавьте к программе дополнительные детали, уточнив, до какого множителя следует выводить таблицу умножения. Результат должен выглядеть примерно так:

```
Для какого числа нужна таблица умножения?
7
До какого множителя вы хотите дойти?
12
Вот ваша таблица:
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
7 x 11 = 77
7 x 12 = 84
```

   Это можно сделать, используя цикл `for`, цикл `while` или сразу оба цикла.

ГЛАВА 9

Только для вас — комментарии

До этого момента все, что мы набирали в тексте наших программ (в том числе в интерактивном режиме), представляло собой команды для компьютера. Но в программу имеет смысл включать также примечания, описывающие, что она делает и как она работает. Это поможет вам (или кому-то другому), глядя на программу спустя некоторое время, понять принцип ее работы.

На компьютерном языке такие примечания называются *комментариями*.

Добавление комментариев

Комментарии предназначены только для чтения, к командам для компьютера они отношения не имеют. То есть они являются частью сопроводительной *документации*, и компьютер при выполнении программы их игнорирует.

В Python существует два способа добавления комментариев.

НОВЫЕ СЛОВА

Документацией называется информация, описывающая программу и принцип ее работы. Комментарии являются частью документации, но могут быть и другие части, не входящие в код и сообщающие, к примеру:

- зачем написана программа (ее предназначение);
- кто ее написал;
- для кого она предназначена (целевая аудитория);
- как она устроена и многое другое.

Чем больше и чем сложнее программа, тем больше у нее сопроводительной документации.

Встроенная в интерпретатор Python справочная система, о которой шла речь в одной из врезок главы 6, также относится к документации. Она нужна, чтобы помочь пользователям — таким, как вы, — понять, как работает Python.

Однострочные комментарии

Любую строку можно превратить в комментарий, поместив перед ней символ # (иногда его еще называют знаком фунта):

```
# Это комментарий в программе на языке Python
print 'Это не комментарий'
```

Если запустить эти две строки, получится следующий результат:

```
Это не комментарий
```

Первая строка при запуске программы игнорируется. Комментарий, начинающийся с символа #, предназначен только для вас и для других людей, которые будут читать этот код.

Комментарии в конце строки

Комментарий можно поместить после строки кода. Вот так:

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

Комментарий начинается после символа #. Все, что находится до этого символа, представляет собой обычный код.

Многострочные комментарии

Иногда комментарий может занять несколько строк. В этом случае в начале каждой строки следует поместить символ #. Вот так:

```
# ***************
# Это программа, иллюстрирующая использование комментариев в Python
# Ряд звездочек визуально отделяет комментарий
# от остальной части кода
# ***************
```

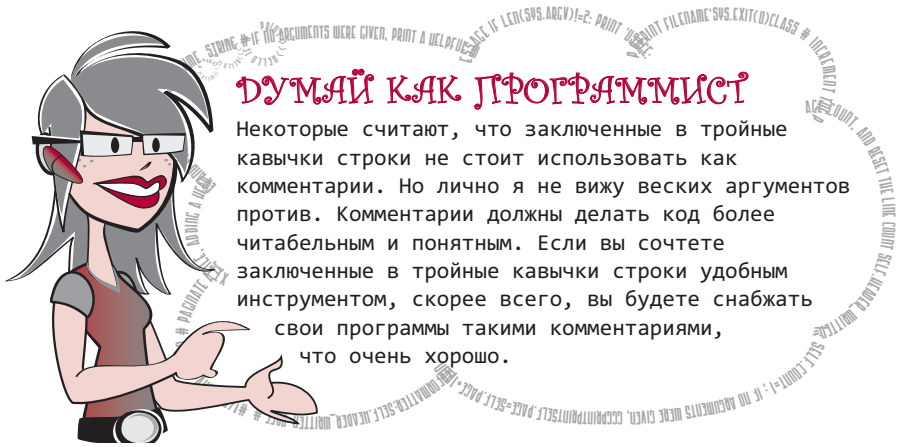
Многострочные комментарии позволяют визуально выделять разделы кода при чтении программы. В них можно описывать происходящее в каждом из разделов. Расположенный в начале программы многострочный комментарий обычно содержит имя автора, название программы, дату ее создания или обновления и любую другую информацию, которую вы сочтете необходимым туда включить.

Тройные кавычки

Существует еще один способ создания элемента, который в Python будет функционировать как многострочный комментарий. Достаточно создать безымянную строку и заключить ее в тройные кавычки. В главе 2 мы говорили о том, что тройные кавычки позволяют объединить между собой группу текстовых строк. Значит, вы можете сделать так:

```
""" Это комментарий, который занимает несколько
строк и заключен в тройные кавычки.
Это не совсем комментарий, но он
Имеет сходное поведение.
"""
```

Эта строка не имеет имени и программа ничего с ней не «делает», никакого воздействия на работу программы она не оказывает. Поэтому она может использоваться как комментарий, хотя, строго говоря, в терминах языка Python комментарием она не является.



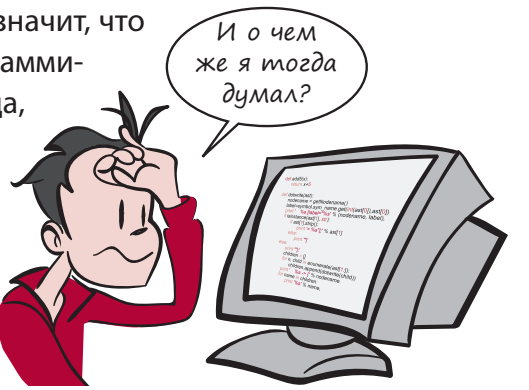
При вводе комментариев в IDLE-редактор вы обнаружите, что они выделяются отдельным цветом. Это сделано, чтобы облегчить вам чтение кода.

Большинство редакторов кода выделяют комментарии цветом (как и остальные части кода). По умолчанию комментарии в IDLE окрашены в красный цвет. Так как заключенные в тройные кавычки строки с точки зрения интерпретатора Python настоящими комментариями не являются, они будут выделены другим цветом. В IDLE они зеленые, поскольку по умолчанию строки выделяются именно этим цветом.

Стиль комментариев

Итак, теперь вы умеете добавлять комментарии. Но что в них писать? Так как на работу программы они не влияют, я скажу, что это дело вкуса. Фактически вы можете писать

что угодно (или вообще ничего не писать). Но это не значит, что комментарии не имеют значения. Большинство программистов начинают осознавать их важность в момент, когда, обратившись к программе, созданной ими несколько недель, месяцев или даже лет назад, обнаруживают, что не могут ничего в ней понять! Иногда такое случается даже с программами, написанными буквально вчера. Обычно это происходит из-за недостатка комментариев, объясняющих принцип работы программы. В момент создания кода все может казаться совершенно очевидным, но вернувшись к программе через некоторое время, вы не сможете в ней разобраться.



Жестких правил написания комментариев не существует, но я советую добавлять их столько, сколько нужно. В конце концов, чем больше, тем лучше. Лучше написать слишком много комментариев, чем слишком мало. Постепенно вы подберете оптимальный объем и количество комментариев.

Комментарии в книге

В приведенных в тексте примерах кода вы найдете не так уж много комментариев. Дело в том, что вместо них мы используем *примечания* — маленькие заметки рядом с кодом. Но если вы загрузите примеры кода из папки *\examples* на нашем сайте, вы обнаружите, что все листинги снабжены комментариями.

Превращение инструкции в комментарий

Еще при помощи комментариев можно на время исключить из программы часть кода. Ведь все комментарии игнорируются:

```
#print "Привет"
print "Мир"
>>> =============== RESTART ================
>>>
Мир
```

Строка `print "Привет"` превращается в комментарий. В результате она не выполняется, и слово «Привет» на экран не выводится.

Это полезно при отладке программы, когда вы хотите запустить только некоторые ее части. Достаточно поместить в начале строки, которую вы хотите исключить, знак `#` или заключить фрагмент такого кода в тройные кавычки.

Большинство редакторов кода, в том числе IDLE-редактор, снабжены инструментами, позволяющими быстро превращать в комментарии целые блоки кода и так же быстро возвращать их в обычное состояние. В IDLE-редакторе они находятся в меню ***Format***.

Что мы узнали

В этой главе мы выяснили, что:

- комментарии предназначены только для вас (и других людей), а не для компьютера;
- при помощи комментариев можно исключить из программы целые фрагменты кода;
- тройные кавычки, объединяющие группу текстовых строк, можно использовать для задания комментариев особого вида.

Проверь себя

Так как комментарии являются очень простым элементом, мы сделаем небольшой перерыв и не будем задавать проверочных вопросов.

Эксперименты

Вернитесь к программе преобразования температуры (из раздела «Эксперименты» в главе 3) и снабдите ее комментариями. Запустите программу и убедитесь, что в ее работе ничего не изменилось.

Время поиграть

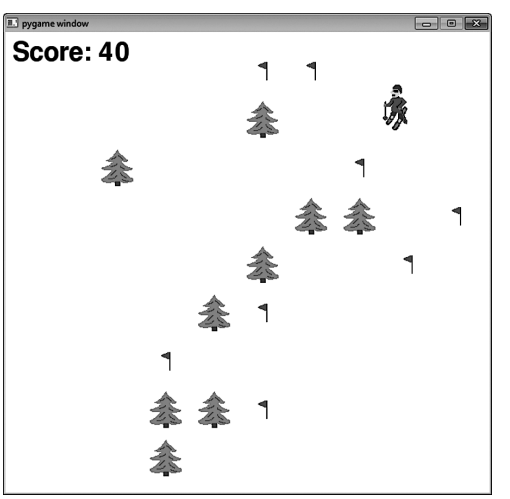
Одной из традиций при изучения программирования является ввод кода, в котором вы ничего не понимаете. Честное слово!

Тем не менее иногда эта процедура позволяет почувствовать, как все работает, даже если вы не в состоянии понять каждую строчку и каждое ключевое слово. Именно так было в главе 1 с игрой в угадывание чисел. Сейчас мы снова сделаем это, на этот раз написав более длинную и более интересную программу.

Лыжник

Лыжник — простая программа по мотивам игры SkiFree. (Прочитать про эту игру можно здесь: *ru.wikipedia.org/wiki/SkiFree*.)

Вы спускаетесь вниз по холму, огибая деревья и собирая флаги. Один флаг дает 10 очков. Врезавшись в дерево, вы теряете 100 очков.



После запуска программы вы должны увидеть вот такой экран:

За графику в игре отвечает модуль Pygame. (Модули будут подробно рассматриваться в главе 15.) Если вы пользовались программой установки, прилагаемой к книге, этот модуль у вас уже есть. В противном случае его можно загрузить с сайта *www.pygame.org*. Модулю Pygame посвящена глава 16.

Для работы программы вам потребуются следующие графические файлы:

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

Они в папке *\examples\skier* (если вы запускали нашу программу установки) или на сайте книги. Поместите их в ту же папку, в которой вы будете сохранять программу. Иначе интерпретатор Python не сможет найти эти файлы, и программа работать не будет.

Код игры представлен в листинге 10.1. Это длинный листинг, около 100 строк кода (плюс пустые строки, чтобы упростить чтение), но не пожалейте времени на его набор. Листинг снабжен примечаниями, объясняющими назначение фрагментов кода. Обратите внимание, что в имени функции `__init__` по два нижних подчеркивания с каждой стороны.

Листинг 10.1. Игра Skier

```
import pygame, sys, random

skier_images = ["skier_down.png", "skier_right1.png",
                "skier_right2.png", "skier_left2.png",
                "skier_left1.png"]

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

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

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

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

Создаем лыжника

Поворачиваем лыжника

Двигаем лыжника влево и вправо

Создаем деревья и флаги

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

Обеспечиваем прокрутку игрового фона

Удаляем препятствия, ушедшие за верхнюю границу экрана

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

Создаем экран со случайным образом расположенными деревьями и флагами

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

Обновляем экран

```
pygame.init()
screen = pygame.display.set_mode([640,640])
clock = pygame.time.Clock()
skier = SkierClass()
speed = [0, 6]
obstacles = pygame.sprite.Group()
map_position = 0
points = 0
create_map()
font = pygame.font.Font(None, 50)
running = True
while running:
clock.tick(30)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
```

Готовим все к запуску

Запускаем основной цикл

Обновляем графику 30 раз в секунду

Проверяем нажатие клавиш или закрытие окна

```
                speed = skier.turn(-1)
            elif event.key == pygame.K_RIGHT:
                speed = skier.turn(1)
    skier.move(speed)

    map_position += speed[1]

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

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

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

Проверяем нажатие клавиш или закрытие окна

Перемещаем лыжника

Прокручиваем экран

Создаем новый экран с декорациями

Проверяем столкновения с деревьями и взятие флагов

Показываем счет

Код этого листинга можно найти в папке *\examples\skier*. И если вам трудно набирать код или вы не хотите тратить на это время, можете воспользоваться этим файлом. Однако набор кода даст вам намного больше, чем открытие и просмотр готового листинга.

В следующих главах вы познакомитесь со всеми ключевыми словами и приемами, используемыми в игре Лыжник. Кроме того, в конце книги вы найдете целую главу, в которой разбирается принцип работы этой программы.

Эксперименты

Все, что вам нужно сделать в этой главе, — это набрать код из листинга 10.1 и запустить его на выполнение. Если после этого появится сообщение об ошибке, изучите его и попробуйте определить, где именно вкралась ошибка. Удачи!

Вложенные циклы и переменные циклов

Вы уже видели, что в тело цикла (которое представляет собой блок кода) можно вставлять разные фрагменты кода, состоящие из собственных блоков. В программе для угадывания чисел из главы 1 можно увидеть следующее:

```
while guess != secret and tries < 6:
    guess = input("Твой вариант?")
    if guess < secret:
        print "Это слишком мало, презренный пес!"
    elif guess > secret:
        print "Это слишком много, сухопутная крыса!"
    tries = tries + 1
```

Блок цикла while

Блок if

Блок elif

Внешний блок связан с циклом `while`, а темно-серым цветом выделены блоки инструкций `if` и `elif`, вложенные в блок цикла `while`.

Аналогичным образом можно поместить один цикл внутрь другого. Такие циклы называются *вложенными*.

Вложенные циклы

Помните таблицу умножения, программу для вывода которой вы писали в разделе «Эксперименты» главы 8? Без части кода, отвечающей за ввод, ее можно представить так:

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

Допустим, нам нужно вывести таблицу умножения для трех множителей одновременно. Подобные задачи прекрасно решаются при помощи *вложенных циклов*. То есть нам нужно поместить один цикл внутрь другого. В этом случае на каждой итерации внешнего цикла внутренний будет делать *все* свои итерации.

Для вывода трех таблиц умножения нужно вставить исходный цикл (выводящий таблицу умножения для одного числа) в цикл, который будет запускаться три раза. Это заставит программу вывести на экран три таблицы вместо одной. Соответствующий код представлен в листинге 11.1.

Листинг 11.1. Вывод таблиц умножения сразу для трех множителей

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

Внутренний цикл выводит одну таблицу

Внешний цикл запускается три раза со значениями 5, 6, 7

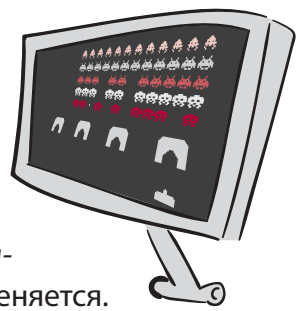
Обратите внимание, что внутренний цикл и инструкцию `print` нужно сдвинуть на четыре пробела относительно начала внешнего цикла `for`. Программа будет выводить таблицу умножения на 5, на 6 и на 7 значений от 1 вплоть до 10:

```
>>> ==================== RESTART ====================
>>>
1 x 5 = 5
2 x 5 = 10
3 x 5 = 15
4 x 5 = 20
5 x 5 = 25
6 x 5 = 30
7 x 5 = 35
8 x 5 = 40
9 x 5 = 45
10 x 5 = 50

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

1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
```

```
6 x 7 = 42
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70
```

Все это может показаться вам крайне скучным, но происходящее во вложенных циклах хорошо иллюстрируют вывод на экран обычных звездочек и их последующий подсчет. Именно этим мы займемся в следующем разделе.

Переменные циклов



Фиксированные значения, которые вы использовали в функции `range()`, называются *константами*. Использование констант в функции `range()` цикла `for` при каждом запуске программы приводит к тому, что цикл выполняется одно и то же количество раз. В этом случае говорят, что количество итераций являются *жестко запрограммированным*, так как оно определено в коде программы и никогда не меняется. Но такое поведение требуется далеко не всегда.

Иногда нужно сделать так, чтобы количество итераций цикла определялось пользователем или другой частью программы. В этом случае не обойтись без переменной.

К примеру, пусть вы пишете игру, в которой происходит война с космическими пришельцами. Так как игрок постоянно убивает все новых и новых врагов, нужно постоянно обновлять экран. Требуется счетчик, который будет фиксировать количество оставшихся инопланетян. И после обновления экрана нужно будет в цикле перебрать оставшихся пришельцев, чтобы нарисовать их изображения. Их количество будет меняться каждый раз, когда игрок убивает очередного противника.

Так как вы пока не умеете рисовать на экране пришельцев, ограничимся в качестве примера простой программой, использующей переменную цикла:

```
for i in range(1, numStars):
    print '*',
>>> ====================== RESTART ======================
>>>
Сколько звезд вам нужно? 5
* * * *
```

Программа спрашивает пользователя, сколько звезд он хочет увидеть на экране, и использует переменную цикла, чтобы показать указанное количество. Точнее, почти указанное количество! Ведь мы попросили пять звезд, а получили всего четыре! Мы забыли,

что цикл for останавливается в одном шаге от второго числа в функции range. Поэтому к числу, указанному пользователем, нужно добавить 1:

```
numStars = int(raw_input ("Сколько звезд вам нужно? "))
for i in range(1, numStars + 1):
    print '*',
```

Добавляем 1, чтобы пользователь, попросивший 5 звезд, увидел на экране 5 звезд

Аналогичный результат можно получить, заставив цикл считать с нуля, а не с единицы. (Мы говорили об этом в главе 8.) Это распространенная практика, и в следующей главе вы с ней столкнетесь. Вот как в этом случае будет выглядеть программа:

```
numStars = int(raw_input ("Сколько звезд вам нужно? "))
for i in range(0, numStars):
    print '*',
>>> ====================== RESTART ====================
>>>
Сколько звезд вам нужно? 5
* * * * *
```

Переменные вложенных циклов

Попробуем изменить количество итераций во вложенном цикле. То есть мы напишем цикл, внутри которого один или несколько циклов используют в функции range() переменную. Рассмотрим пример (листинг 11.2).

Листинг 11.2. Вложенный цикл с переменным числом итераций

```
numLines = int(raw_input ('Сколько строк со звездами вам нужно? '))
numStars = int(raw_input ('Сколько звезд должно быть в строке? '))
for line in range(0, numLines):
    for star in range(0, numStars):
        print '*',
    print
```

Попробуйте запустить эту программу, чтобы понять, имеет ли она смысл. Вы должны получить примерно такой результат:

```
>>> ============================ RESTART ============================
>>>
Сколько строк со звездами вам нужно? 3
Сколько звезд должно быть в строке? 5
*****
*****
*****
```

В первых двух строках выясняется у пользователя, сколько строк со звездами он хочет видеть на экране и сколько звезд должна содержать каждая строка. Ответы пользователя запоминаются в переменных `numLines` и `numStars`. Затем начинают работать два цикла:

- внутренний цикл (`for star in range (0, numStars):`) показывает звезды и выполняется для каждой звезды;
- внешний цикл (`for line in range (0, numLines):`) выполняется для каждой строки звезд.

Вторая команда `print` начинает вывод новой строки звезд. Без нее из-за запятой в первой команде `print` все звезды выводились бы в одну линию. Можно вложить цикл в уже вложенный цикл (выполнить *двойное вложение*), как показано в листинге 11.3.

Листинг 11.3. Блоки звезд с двойным вложенным циклом

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

Вот результат работы такой программы:

```
>>> ====================== RESTART ======================
>>>
Сколько блоков звезд вам нужно? 3
Сколько строк в каждом блоке? 4
Сколько звезд в строке? 8
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *

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

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

В данном случае говорят о «трех уровнях вложенности».

Больше переменных во вложенных циклах

В листинге 11.4 демонстрируется усложненная версия программы из листинга 11.3.

Листинг 11.4. Усложненная версия блоков звезд

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

Формулы, задающие число строк и звезд

Вот результат работы программы:

```
>>> ====================== RESTART ======================
>>>
Сколько блоков звезд вам нужно? 3
* * * *

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

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

В листинге 11.4 через переменную внешнего цикла задается диапазон внутренних циклов. Поэтому если раньше у нас в каждом блоке было одно и то же число строк, а в каждой строке — одно и то же число звезд, теперь эти параметры меняются на каждой итерации внешнего цикла.

Можно создавать циклы произвольного уровня вложенности. Но при этом порой бывает непросто следить за происходящим, поэтому имеет смысл выводить на экран значения переменных цикла, как показано в листинге 11.5.

Листинг 11.5. Вывод переменных циклов во вложенных циклах

```
numBlocks = int(raw_input('Сколько блоков звезд вам нужно? '))
for block in range(1, numBlocks + 1):
```

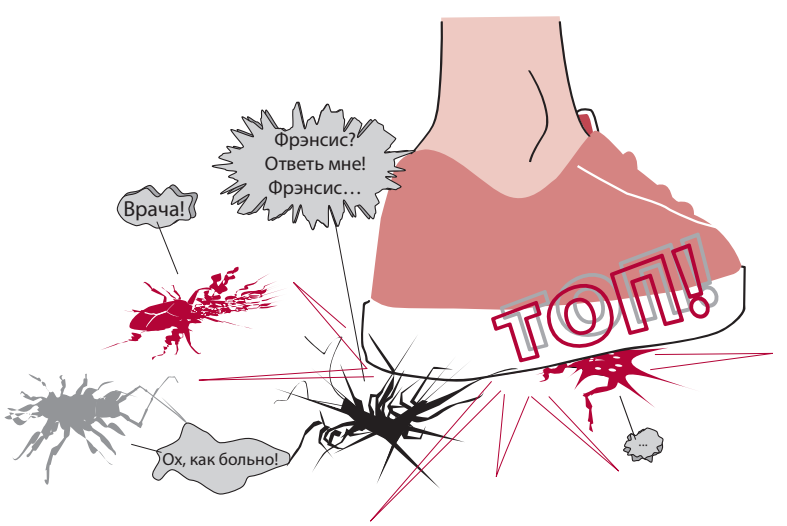
```
    print 'блок = ', block
    for line in range(1, block * 2 ):
        for star in range(1, (block + line) * 2):
            print '*',
        print ' строка = ', line, 'звезд = ', star
    print
```

Вывод переменных

Вот результат работы программы:

```
>>> ====================== RESTART ======================
>>>
Сколько блоков звезд вам нужно? 3
блок = 1
* * * строка = 1 звезд = 3

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

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

Вывод значений переменных может помочь во многих ситуациях, а не только при работе с циклами. Это один из общепринятых методов ликвидации багов (ошибок) в коде.



Применение вложенных циклов

Ну и зачем нам нужны все эти вложенные циклы? А затем, что они, к примеру, позволяют найти все доступные *перестановки* и *сочетания* в серии решений.

НОВЫЕ СЛОВА

Перестановка — это математическое понятие, означающее уникальный способ комбинирования некоего множества объектов. Почти аналогичное значение имеет термин «сочетание». Просто в случае *сочетания* порядок следования объектов не имеет значения, в то время как при перестановках за ним строго следят. К примеру, я попросил вас выбрать три числа от 1 до 20, и вы выбрали:

- 5, 8, 14;
- 2, 12, 20;

и т. п. Если мы попытаемся сделать список всех перестановок из трех чисел от 1 до 20, то следующие два варианта должны считаться разными:

- 5, 8, 14;
- 8, 5, 14.

Так происходит как раз потому, что в случае перестановок порядок следования элементов имеет значение. А вот в случае списка сочетаний следующие три варианта будут рассматриваться как одинаковые:

- 5, 8, 14;
- 8, 5, 14;
- 8, 14, 5.

Ведь в случае сочетаний порядок следования элементов не имеет значения.

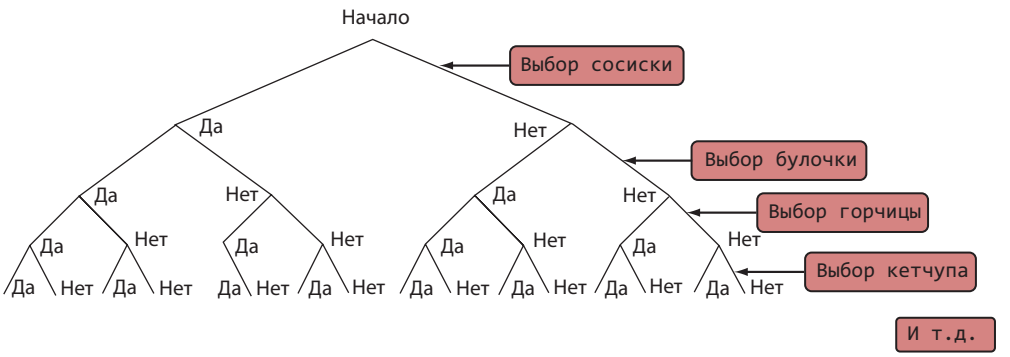
Проще всего объяснить это на примере. Представьте, что вы открыли палатку с хот-догами и хотите повесить рядом плакат, демонстрирующий, как по номеру можно заказать любое сочетание сосиски, булочки, кетчупа, горчицы и лука. Для этого нам сначала нужно определить все возможные сочетания.

Эту задачу можно представить в виде так называемого *дерева решений*. Вот как оно выглядит для задачи с хот-догами.

В каждой точке принятия решения есть только два варианта: «Да» и «Нет». Все пути вниз по дереву описывают разные комбинации компонентов хот-дога. Выделенный мною путь говорит «Да» сосиске, «Нет» булочке, «Да» горчице и «Да» кетчупу.

А сейчас при помощи вложенных циклов мы сделаем список всех комбинаций — всех путей в дереве решений. Так как у нас пять точек принятия решения, дерево имеет пять

уровней, а значит, нам потребуется пять вложенных циклов. (Дерево решений на рисунке имеет всего четыре уровня.)



Введите код из листинга 11.6 в IDLE-редактор и сохраните файл под именем ***hotdog1.py***.

Листинг 11.6. Варианты хот-дога

```
print "\tСосиска \tБулочка \tКетчуп\tГорчица\tЛук"
count = 1
for dog in [0, 1]:
    for bun in [0, 1]:
        for ketchup in [0, 1]:
            for mustard in [0, 1]:
                for onion in [0, 1]:
                    print "#", count, "\t",
                    print dog, "\t", bun, "\t", ketchup, "\t",
                    print mustard, "\t", onion
                    count = count + 1
```

Цикл горчицы
Цикл кетчупа
Цикл сосиски
Цикл булочки
Цикл лука

Видите, как циклы вставляются один в другой? Именно в этом проявляется вся суть вложенных циклов — перебор одних вариантов внутри других вариантов:

- внешний цикл для сосиски (`dog`) запускается дважды;
- цикл для булочки (`bun`) запускается дважды на каждой итерации цикла для сосиски. Значит, он работает $2 \times 2 = 4$ раза;
- цикл для кетчупа (`ketchup`) запускается дважды на каждой итерации цикла для булочки. Значит, всего он будет работать $2 \times 2 \times 2 = 8$ раз.

И т. д. Самый внутренний цикл (то есть расположенный в коде ниже всего цикл `onion`) запускается $2 \times 2 \times 2 \times 2 \times 2 = 32$ раза. Он покрывает все возможные сочетания. Значит, у нас есть 32 возможных варианта. Запустив программу из листинга 11.6, вы получите примерно такой результат:

```
>>> ========================== RESTART ==========================
>>>
            Сосиска Булочка Кетчуп  Горчица Лук
# 1         0       0       0       0       0
# 2         0       0       0       0       1
# 3         0       0       0       1       0
# 4         0       0       0       1       1
# 5         0       0       1       0       0
# 6         0       0       1       0       1
# 7         0       0       1       1       0
# 8         0       0       1       1       1
# 9         0       1       0       0       0
# 10        0       1       0       0       1
# 11        0       1       0       1       0
# 12        0       1       0       1       1
# 13        0       1       1       0       0
# 14        0       1       1       0       1
# 15        0       1       1       1       0
# 16        0       1       1       1       1
# 17        1       0       0       0       0
# 18        1       0       0       0       1
# 19        1       0       0       1       0
# 20        1       0       0       1       1
# 21        1       0       1       0       0
# 22        1       0       1       0       1
# 23        1       0       1       1       0
# 24        1       0       1       1       1
# 25        1       1       0       0       0
# 26        1       1       0       0       1
# 27        1       1       0       1       0
# 28        1       1       0       1       1
# 29        1       1       1       0       0
# 30        1       1       1       0       1
# 31        1       1       1       1       0
# 32        1       1       1       1       1
```

Пять вложенных циклов перебрали все возможные сочетания переменных `dog`, `bun`, `ketchup`, `mustard` и `onion`.

В листинге 11.6 выравнивание осуществлялось при помощи символа табуляции. Он записывается как `\t`. *Форматирование* выводимых результатов мы пока не обсуждали, но если вы хотите узнать об этом больше, загляните в главу 21.

Переменная `count` использовалась для нумерации сочетаний. Например, хот-дог, состоящий из булочки и горчицы, оказался под номером 27. Разумеется, некоторые из сочетаний не имеют смысла. (Хот-дог без булочки, но с горчицей и кетчупом выглядит несколько странно.) В то же время еще никто не отменял правила «Клиент всегда прав!».

Подсчет калорий

Так как в наши дни многие озабочены здоровым питанием, добавим в наш список комбинаций информацию о калорийности. (Допускаю, что вы совершенно не озабочены тем, сколько калорий потребляете, но вот вашим родителям это наверняка не безразлично!) Воспользуемся математическими возможностями языка Python, с которыми мы познакомились в главе 3.

Мы уже знаем, какие элементы входят в каждую комбинацию. Теперь выясним калорийность каждого элемента. После чего нужно будет сложить их в самом внутреннем цикле. Вот код, задающий калорийность каждого варианта:

```
dog_cal = 140
bun_cal = 120
mus_cal = 20
ket_cal = 80
onion_cal = 40
```

Осталось выполнить сложение. Мы знаем, что каждое сочетание в нашем меню содержит 0 или 1 штуку каждого элемента. Поэтому достаточно умножить это количество на соответствующую калорийность:

```
tot_cal = (dog * dog_cal) + (bun * bun_cal) + \
          (mustard * mus_cal) + (ketchup * ket_cal) + \
          (onion * onion_cal)
```

Так как в соответствии с приоритетом операций сначала выполняется умножение, а только потом сложение, скобки в данном случае не требуются. Я добавил их, чтобы было проще понять происходящее.

Соберем все вместе и напишем новую версию программы с подсчетом калорийности наших хот-догов (листинг 11.7).

Листинг 11.7. Программа создания хот-догов со счетчиком калорий

```
dog_cal = 140
bun_cal = 120
ket_cal = 80
mus_cal = 20
onion_cal = 40
```

Список калорий для каждого компонента хот-дога

```
print "\tСосиска \tБулочка \tКетчуп\tГорчица\tЛук \tКалории"
count = 1
for dog in [0, 1]:
    for bun in [0, 1]:
        for ketchup in [0, 1]:
            for mustard in [0, 1]:
                for onion in [0, 1]:
                    total_cal = (bun * bun_cal)+(dog * dog_cal) + \
                        (ketchup * ket_cal)+(mustard * mus_cal) + \
                            (onion * onion_cal)
                    print "#", count, "\t",
                    print dog, "\t", bun, "\t", ketchup, "\t",
                    print mustard, "\t", onion,
                    print "\t", total_cal
                    count = count + 1
```

Вывод заголовков

Цикл dog — это внешний цикл

Подсчет калорий во внутреннем цикле

Вложенные циклы

ДЛИННЫЕ СТРОКИ КОДА

Обратили внимание на символы в виде обратной косой черты (\) на концах строк в предыдущем фрагменте кода? В случае, когда длинное выражение невозможно уместить в одну строку, этот символ сообщает интерпретатору Python: «Строка еще не кончилась. Учти, что следующая строка является продолжением этой». В данном случае, добавив две обратные косые черты, мы разбили одно длинное выражение на три коротких строки. Обратная косая черта, которая называется *символом переноса строки*, присутствует в ряде языков программирования.

Кроме того, выражение можно заключить в дополнительные скобки. Тогда его можно будет расположить на нескольких строках, не прибегая к символам переноса:

```
tot_cal = ((dog * dog_cal) + (bun * bun_cal) +
           (mustard * mus_cal) + (ketchup * ket_cal) +
           (onion * onion_cal))
```

Запустите в IDLE программу из листинга 11.7. Вот что в результате получится:

```
>>> =========================== RESTART ============================
>>>
            Сосиска Булочка Кетчуп  Горчица Лук     Калории
# 1         0       0       0       0       0       0
# 2         0       0       0       0       1       40
# 3         0       0       0       1       0       20
# 4         0       0       0       1       1       60
# 5         0       0       1       0       0       80
# 6         0       0       1       0       1       120
# 7         0       0       1       1       0       100
```

```
# 8          0       0       1       1       1       140
# 9          0       1       0       0       0       120
# 10         0       1       0       0       1       160
# 11         0       1       0       1       0       140
# 12         0       1       0       1       1       180
# 13         0       1       1       0       0       200
# 14         0       1       1       0       1       240
# 15         0       1       1       1       0       220
# 16         0       1       1       1       1       260
# 17         1       0       0       0       0       140
# 18         1       0       0       0       1       180
# 19         1       0       0       1       0       160
# 20         1       0       0       1       1       200
# 21         1       0       1       0       0       220
# 22         1       0       1       0       1       260
# 23         1       0       1       1       0       240
# 24         1       0       1       1       1       280
# 25         1       1       0       0       0       260
# 26         1       1       0       0       1       300
# 27         1       1       0       1       0       280
# 28         1       1       0       1       1       320
# 29         1       1       1       0       0       340
# 30         1       1       1       0       1       380
# 31         1       1       1       1       0       360
# 32         1       1       1       1       1       400
```

Только представьте, как утомительно было бы вручную подсчитывать калорийность всех этих сочетаний даже с использованием калькулятора! Гораздо веселее написать программу, которая все сделает за вас. Циклы и немного математики позволяют решить задачу на языке Python быстро и просто!

Что мы узнали

В этой главе мы познакомились с:

- вложенными циклами;
- переменными циклов;
- перестановками и сочетаниями;
- деревьями решений.

Проверь себя

1. Как в Python реализуется цикл с переменной цикла?
2. Как в Python реализуется вложенный цикл?

3. Сколько звездочек выведет на экран следующий код?

```
for i in range(5):
    for j in range(3):
        print '*',
    print
```

4. Как будет выглядеть результат работы программы из задания 3?
5. У нас есть четырехуровневое дерево решений, и на каждом уровне предлагается два возможных варианта. Сколько всего существует вариантов (путей вниз по дереву решений)?

Эксперименты

1. Помните таймер обратного отсчета, который мы написали в главе 8? Вот как он выглядел:

```
import time
for i in range (10, 0, -1):
    print i
    time.sleep(1)
print "ПУСК!"
```

Напишите эту программу с использованием переменной цикла. Она должна спрашивать пользователя, с какого момента следует начать обратный отсчет:

```
Таймер обратного отсчета: Сколько секунд? 4
4
3
2
1
ПУСК!
```

2. Доработайте предыдущую программу, чтобы рядом с каждым числом она выводила соответствующее количество звездочек:

```
Таймер обратного отсчета: Сколько секунд? 4
4 * * * *
3 * * *
2 * *
1 *
ПУСК!
```

Подсказка: скорее всего, вам потребуется вложенный цикл.

ГЛАВА 12

Списки и словари

Как вы уже знаете, Python позволяет сохранять в памяти различные значения и извлекать их оттуда с помощью имен. До этого момента мы сохраняли *строки* и *числа* (как *целые*, так и с *плавающей точкой*). Но иногда необходимо сохранить целую группу, или *коллекцию* (collection), элементов. Это позволяет оперировать всей коллекцией одновременно и упрощает отслеживание групп элементов. Одной из разновидностей коллекции является *список* (list), другой — *словарь* (dictionary). Именно о списках и словарях пойдет речь в этой главе. Вы узнаете, как они создаются, модифицируются и применяются в программах.

Списки крайне полезны и фигурируют во множестве программ. Они будут часто появляться в примерах кода в следующих главах, когда мы перейдет к изучению графики и начнем разрабатывать игры. Ведь многие графические объекты в играх хранятся именно в виде списков.

Что такое список

Если я попрошу составить список членов вашей семьи, может получиться что-то такое:

На языке Python это выглядит так:

```
family = ['Мама', 'Папа', 'Я', 'Братик']
```

А если попросить вас написать свои счастливые числа, может получиться нечто такое:

2, 7, 14, 26, 30

На языке Python это выглядит так:

```
luckyNumbers = [2, 7, 14, 26, 30]
```

В данном случае как `family`, так и `luckyNumbers` — примеры Python-списков, а входящие в них отдельные объекты называются *элементами* (items). Как видите, Python-списки не сильно отличаются от привычных для вас реальных списков. Квадратные скобки показывают начало и конец списка, а запятые отделяют элементы друг от друга.

Создание списка

Как `family`, так и `luckyNumbers` — это переменные разных видов. Мы уже сталкивались с переменными для чисел и строк, и ничто не мешает нам создать переменную для списка.

Чтобы создать список, как и любую другую переменную, используем присваивание некоего значения. Мы это уже видели на примере переменной `luckyNumbers`. Кроме того, можно создать пустой список:

```
newList = []
```

В квадратных скобках нет ни одного элемента, то есть список является пустым. Но какая польза от пустого списка? Зачем он может кому-то понадобиться?

Дело в том, что часто возникает ситуация, когда вы заранее не знаете, что именно попадет в список. Неизвестны ни количество элементов, ни их состав. Известно только, что вы будете пользоваться списком. А если у вас есть пустой список, туда можно добавить что угодно. Давайте посмотрим, как это делается.

Добавление элементов

Для добавления элементов в список служит метод `append()`. Запустите в интерактивном режиме этот фрагмент кода, формирующий список друзей:

```
>>> friends = []
>>> friends.append('Дэвид')
>>> print friends
```

Создаем новый, пустой список

Добавляем в список элемент 'Дэвид'

Вот какой результат вы получите:

```
['Дэвид']
```

Попробуйте добавить еще один элемент:

```
>>> friends.append('Мэри')
>>> print friends
['Дэвид', 'Мэри']
```

Имейте в виду, что добавлять элементы вы можете только в уже существующий список (пустой или нет). Поэтому первым делом его нужно создать. Это все равно что замеши-

вание теста: нельзя взять и начать смешивать все ингредиенты, сначала вам потребуется кастрюля. Без нее вы только запачкаете стол!

НОВЫЕ СЛОВА

Английское слово *append* обозначает добавление чего-то в конец. Именно поэтому оно выбрано для названия метода, добавляющего в список элементы. При этом добавленные элементы оказываются в конце списка.

Что это за точка

Почему между словами `friends` и `append()` стоит точка? Это первый предвестник большой темы, посвященной объектам. Объекты будут рассматриваться в главе 14, а пока дадим простое объяснение.

Объектами (objects) являются многие элементы языка Python. Для выполнения какого-либо действия с объектом нужно указать его имя, поставить точку и добавить наименование операции. Поэтому, чтобы *добавить* в список `friends` новый элемент, мы пишем:

```
friends.append(что-то)
```

Содержимое списков

Списки могут содержать любые данные, которые можно сохранять в Python. В этот перечень входят числа, строки, объекты и даже другие списки. Элементы списка не обязаны принадлежать к одному типу. Это означает, что в одном списке могут одновременно содержаться, например, числа и строки. Это будет выглядеть примерно так:

```
my_list = [5, 10, 23.76, 'Привет', myTeacher, 7, another_list]
```

Давайте создадим список с простыми элементами, например с буквами латинского алфавита, чтобы при изучении свойств списков было проще понять, что именно происходит. Введите в интерактивном режиме:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
```

Доступ к элементам списка

Доступ к отдельному элементу списка происходит по номеру его индекса. Индексы начинаются с 0, поэтому первым элементом нашего списка, то есть элементом `letters[0]`, будет буква *a*:

```
>>> print letters[0]
a
```

Попробуем еще раз:

```
>>> print letters[3]
d
```

Почему индекс начинается с нуля

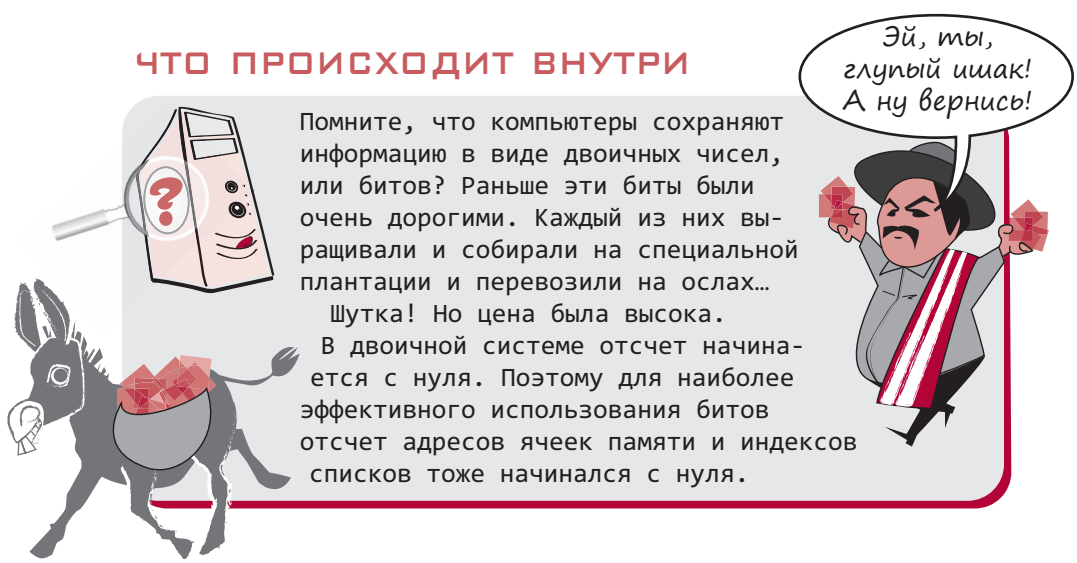
Индекс (index) показывает позицию чего-то. То есть если в очереди вы значитесь четвертым, у вас будет индекс 4. Но если вы четвертый человек в Python-списке, ваш индекс будет равен 3, так как индексация Python-списков начинается с нуля!

Этот вопрос программисты, инженеры и прочие компьютерные специалисты обсуждают с момента появления компьютеров. Мне не хочется углубляться во все эти тонкости, поэтому я отвечу просто: «Потому что так принято» и перейду к следующей теме…

Хорошо, хорошо! В следующих абзацах вы узнаете, почему индекс начинается с 0, а не с 1.

Вы привыкнете отсчитывать индексы с нуля. Это распространенная практика в программировании.

Что происходит внутри



Помните, что компьютеры сохраняют информацию в виде двоичных чисел, или битов? Раньше эти биты были очень дорогими. Каждый из них выращивали и собирали на специальной плантации и перевозили на ослах…

Шутка! Но цена была высока.

В двоичной системе отсчет начинается с нуля. Поэтому для наиболее эффективного использования битов отсчет адресов ячеек памяти и индексов списков тоже начинался с нуля.

Срез списка

Индексы позволяют обеспечить доступ сразу к нескольким элементам списка. Эта операция называется *получением среза* (slicing) списка:

```
>>> print letters[1:4]
['b', 'c', 'd']
```

Как и в случае с функцией `range()` в цикле `for`, при получении среза извлечение элементов начинается с первого указанного индекса и заканчивается *перед* вторым. Именно поэтому в предыдущем примере мы получили три, а не четыре элемента. Запомните, что получаемое в результате среза количество элементов равно разнице между индексами (4 – 1 = 3, то есть мы получаем три элемента).

Есть еще один момент, который нужно учитывать при получении среза списков: в результате создается другой список (обычно меньшего размера). Этот меньший список называют *срезом* (slice) исходного, который при этом остается неизменным. Срез является частичной *копией* оригинала.

Посмотрите, какие разные результаты получаются в этом фрагменте кода:

```
>>> print letters[1]
b
>>> print letters[1:2]
['b']
```

В первом случае нам показывают элемент списка. Во втором нам возвращают список с одним элементом. Кажется, что разница совсем невелика, но о ней нужно помнить. Для получения *элемента* списка указывается его индекс. А для получения одноэлементного *среза* списка через *двоеточие* указываются индексы соседних элементов.

Чтобы лучше понять разницу, запустите следующий фрагмент кода:

```
>>> print type(letters[1])
<type 'str'>
>>> print type(letters[1:2])
<type 'list'>
```

Мы вывели на экран тип (`type`) каждого элемента и убедились, что в первом варианте мы получаем сам элемент, то есть в данном случае *строку* (`str`), а во втором — *список* (`list`).

Меньший список, создаваемый в результате получения среза, представляет собой копию фрагмента исходного списка. Вы можете его редактировать, и это никак не повлияет на исходный список.

Сокращенная запись

Для получения срезов существуют сокращенные варианты записи кода. Нельзя сказать, чтобы они сильно экономили время, но программисты, как правило, люди ленивые и предпочитают не тратить усилия понапрасну. Поэтому я считаю, что вы должны знать о существующих сокращениях, чтобы распознать их в чужом коде и понять, что происходит. Это очень важное умение, так как именно анализ чужого кода помогает при освоении нового языка программирования в частности и программирования в целом.

Если срез должен начинаться с первого элемента списка, поставьте двоеточие и укажите нужное количество элементов. Вот так:

```
>>> print letters[:2]
['a', 'b']
```

Обратите внимание на отсутствие цифры перед двоеточием. При такой записи вы получите все элементы с начала списка до элемента с указанным вами индексом (но не включая этот элемент). Аналогичным способом делается срез конца списка:

```
>>> print letters[2:]
['c', 'd', 'e']
```

Число, за которым следует двоеточие, даст вам фрагмент от элемента с указанным индексом до конца списка.

Можно вообще не указывать никаких чисел, поместив в скобки только двоеточие. В этом случае вы получите список целиком:

```
>>> print letters[:]
['a', 'b', 'c', 'd', 'e']
```

Помните, мы говорили о том, что при получении среза создается копия? Это значит, что код `letters[:]` формирует точную копию списка. А это очень полезно, если вы хотите внести в список какие-либо изменения, оставив оригинал нетронутым.

Изменение элементов

Индекс дает нам доступ к изменению отдельных элементов списка:

```
>>> print letters
['a', 'b', 'c', 'd', 'e']
>>> letters[2] = 'z'
>>> print letters
['a', 'b', 'z', 'd', 'e']
```

Однако индекс невозможно использовать для добавления в список новых элементов. Например, пусть у нас есть список из пяти элементов с индексами от 0 до 4. При этом выполнить вот такую операцию нельзя:

```
letters[5] = 'f'
```

Это просто не сработает. (Если хотите, попробуйте.) Это все равно что пытаться изменить вещь, которой еще не существует. О способах добавления элементов в список мы поговорим в следующем разделе, но сначала вернем наш список к исходному виду:

```
>>> letters[2] = 'c'
>>> print letters
['a', 'b', 'c', 'd', 'e']
```

Методы добавления элементов

Вы уже знаете, что добавить элемент в список можно методом `append()`. Но существуют и другие методы. Всего их три: `append()`, `extend()` и `insert()`.

- Метод `append()` добавляет один элемент в конец списка.
- Метод `extend()` добавляет несколько элементов в конец списка.
- Метод `insert()` добавляет один элемент в произвольное место списка, необязательно в конец. Место вы выбираете сами.

Добавление в конец методом append()

С методом `append()` вы уже знакомы. Он добавляет в конец списка один элемент:

```
>>> letters.append('n')
>>> print letters
['a', 'b', 'c', 'd', 'e', 'n']
```

Добавим еще один элемент:

```
>>> letters.append('g')
>>> print letters
['a', 'b', 'c', 'd', 'e', 'n', 'g']
```

Обратите внимание, что буквы не упорядочены. Дело в том, что новые буквы метод `append()` помещает в самый конец. Если вам нужен упорядоченный список, следует прибегнуть к сортировке. С приемами сортировки вы познакомитесь чуть позже.

Расширение списка методом extend()

Метод `extend()` добавляет в конец списка несколько элементов:

```
>>> letters.extend(['p', 'q', 'r'])
>>> print letters
['a', 'b', 'c', 'd', 'e', 'n', 'g', 'p', 'q', 'r']
```

Обратите внимание, что аргументом метода `extend()` является список. А так как список заключается в квадратные скобки, после слова `extend` указываются как обычные, так и квадратные скобки.

Все содержимое списка, переданного вами в метод `extend()`, добавляется в конец оригинального списка.

Вставка элемента методом insert()

Метод `insert()` добавляет единичный элемент в произвольное место списка. Место вставки указываете вы:

```
>>> letters.insert(2, 'z')
>>> print letters
['a', 'b', 'z', 'c', 'd', 'e', 'n', 'g', 'p', 'q', 'r']
```

Мы добавили букву *z* в позицию с индексом 2. В списке этот индекс соответствует третьей слева позиции (так как отсчет начинается с нуля). Буква *c*, изначально находившаяся в третьей позиции, сдвигается вправо и оказывается четвертой. Все прочие элементы списка также сдвигаются на одну позицию вправо.

Разница между методами append() и extend()

Иногда методы `append()` и `extend()` выглядят одинаково, но функционируют они по-разному. Вернемся к нашему исходному списку. Для начала воспользуемся методом `extend()`:

```
>>> letters = ['a','b','c','d','e']
>>> letters.extend(['f', 'g', 'h'])
>>> print letters
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Теперь попробуем сделать то же самое методом `append()`:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> letters.append(['f', 'g', 'h'])
>>> print letters
['a', 'b', 'c', 'd', 'e', ['f', 'g', 'h']]
```

Что произошло? Как мы уже говорили, метод `append()` добавляет в список только один элемент. Как же получилось, что он добавил три элемента? Он этого не делал. Он добавил один элемент, который представляет собой список, содержащий три элемента. Поэтому внутри итогового списка появились дополнительные квадратные скобки. Напоминаем, что в список можно включать элементы любых типов, в том числе другие списки. Именно это мы и видим в данном случае.

Метод `insert()` отличается от метода `append()` только тем, что вставляет новый элемент в указанное вами место, в то время как метод `append()` всегда добавляет его в конец списка.

Удаление элементов

Что делать, если вам нужно удалить элемент из списка? Для этого существует три метода: `remove()`, `del` и `pop()`.

Метод remove()

Метод `remove()` удаляет из списка выбранный вами элемент:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> letters.remove('c')
>>> print letters
['a', 'b', 'd', 'e']
```

Вам не нужно знать, где именно в списке находится удаляемый элемент. Достаточно, чтобы он там был. Попытка удаления несуществующего элемента приводит к сообщению об ошибке:

```
>>> letters.remove('f')
Traceback (most recent call last):
    File "<pyshell#32>", line 1, in <module>
```

```
        letters.remove('f')
ValueError: list.remove(x): x not in list
```

Как проверить наличие в списке определенного элемента? Об этом вы скоро узнаете. Но сначала рассмотрим остальные способы удаления элементов.

Метод del

Метод `del` удаляет элемент по его индексу. Вот так:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> del letters[3]
>>> print letters
['a', 'b', 'c', 'e']
```

В данном случае мы удалили четвертый элемент (с индексом 3), то есть букву *d*.

Метод pop()

Метод `pop()` извлекает из списка последний элемент и возвращает его вам. После этого ему можно присвоить имя:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> lastLetter = letters.pop()
>>> print letters
['a', 'b', 'c', 'd']
>>> print lastLetter
e
```

Метод `pop()` можно использовать и с индексом:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> second = letters.pop(1)
>>> print second
b
>>> print letters
['a', 'c', 'd', 'e']
```

В данном случае мы извлекли второй элемент (с индексом 1), то есть букву *b*. Элемент был сохранен в переменной `second` и одновременно удален из списка `letters`.

Если в скобках ничего не указано, метод `pop()` дает вам последний элемент, одновременно удаляя его из списка. Если в скобках стоит какое-нибудь число, метод `pop(n)` выдает элемент с указанным индексом, удаляя его из списка.

Поиск в списке

Как найти в списке нужный элемент? При работе со списками часто возникает необходимость:

- проверить, присутствует ли там некий элемент;
- определить его позицию в списке (то есть индекс).

Ключевое слово in

Проверить, присутствует ли в списке определенный элемент, помогает ключевое слово `in`. Вот как это делается:

```
if 'a' in letters:
print "найден символ 'a' в списке letters"
else:
print "символ 'a' в списке letters не обнаружен"
```

Запись `'a' in letters` — часть *булевого*, или *логического*, выражения. Оно возвращает значение `True`, если в списке присутствует буква *a*, и значение `False` в противном случае.

Новые слова

Булевым называется арифметическое выражение, использующее только два значения: 1 и 0, или true и false. Оно было предложено математиком Джорджем Булем и применяется при объединении истинных и ложных условий (представляемых как 1 и 0) с помощью операторов and, or и not. О них речь шла в главе 7.

Введите этот фрагмент кода в интерактивном режиме:

```
>>> 'a' in letters
True
>>> 's' in letters
False
```

Код сообщает нам, что в списке `letters` присутствует буква *a*, но нет буквы *s*. Теперь можно скомбинировать ключевое слово `in` с методом `remove()` и написать код, который не покажет сообщения об ошибке, даже если в списке нет нужного значения:

```
if 'a' in letters:
    letters.remove('a')
```

Этот код удаляет элемент из списка только в том случае, если он там присутствует.

Поиск индекса

Местоположение элемента в списке определяет метод `index()`:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> print letters.index('d')
3
```

Теперь мы знаем, что буква *d* имеет индекс 3, то есть в списке она фигурирует на четвертой позиции. Подобно методу `remove()`, метод `index()`, не обнаружив в списке нужного элемента, выдает сообщение об ошибке, поэтому его тоже имеет смысл использовать вместе с ключевым словом `in`:

```
if 'd' in letters:
    print letters.index('d')
```

Циклический просмотр списка

В начале своего знакомства с циклами вы видели, как цикл перебирает значения из списка. Еще вы узнали о функции `range()` и воспользовались ею для быстрой генерации списка перебираемых значений. Вы выяснили, что эта функция генерирует список чисел.

Но в цикле можно перебирать элементы любого списка, ограничиваться списком чисел вовсе не обязательно. Представьте, что нам нужно вывести на экран список букв по одной букве на строке. Для этого можно написать такой код:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> for letter in letters:
        print letter
a
b
c
d
e
```

На сей раз переменной цикла является `letter`. (До этого в качестве переменных цикла фигурировали переменные `looper`, а также `i`, `j` и `k`.) Цикл перебирает все значения из списка, и на каждой итерации рассматриваемый элемент сохраняется в переменной `letter` и выводится на экран.

Сортировка списков

Списки относятся к *упорядоченным* коллекциям. Это означает, что их элементы располагаются в определенном порядке и за каждым закреплено его место (индекс). Порядок размещения элементов сохраняется, пока вы не измените его методом

`insert()`, `append()`, `remove()` или `pop()`. Но иногда после этих операций элементы оказываются не в том порядке, который вам нужен. А значит, перед дальнейшей работой со списком его требуется подвергнуть *сортировке*.

Для этого существует метод `sort()`:

```
>>> letters = ['d', 'a', 'e', 'c', 'b']
>>> print letters
['d', 'a', 'e', 'c', 'b']
>>> letters.sort()
>>> print letters
['a', 'b', 'c', 'd', 'e']
```

Метод `sort()` автоматически располагает строки в алфавитном порядке, а числа упорядочивает по возрастанию.

Важно понимать, что метод `sort()` редактирует сам список. То есть он вносит изменения в оригинал, а *не* создает для вас новую отсортированную копию. Поэтому вы не можете написать:

```
>>> print letters.sort()
```

Такая строка не даст никакого результата. Эту операцию нужно проводить в два этапа:

```
>>> letters.sort()
>>> print letters
```

Сортировка в обратном порядке

Существует два способа сортировки списков в обратном порядке. Можно провести обычную сортировку, а затем поменять порядок следования элементов на *обратный*. Например, так:

```
>>> letters = ['d', 'a', 'e', 'c', 'b']
>>> letters.sort()
>>> print letters
['a', 'b', 'c', 'd', 'e']
>>> letters.reverse()
>>> print letters
['e', 'd', 'c', 'b', 'a']
```

В этом фрагменте кода вы видите новый метод `reverse()`, который меняет порядок размещения элементов списка на обратный.

Можно также добавить в метод `sort()` параметр, заставляющий выполнить сортировку в порядке убывания (от самого большого к самому маленькому):

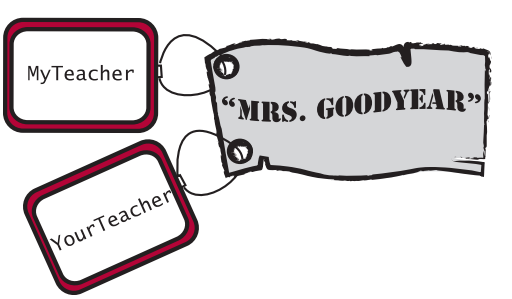
```
>>> letters = ['d', 'a', 'e', 'c', 'b']
>>> letters.sort (reverse = True)
>>> print letters
['e', 'd', 'c', 'b', 'a']
```

Этот параметр, который называется `reverse`, делает именно то, что вам нужно, — инициирует сортировку списка в обратном порядке.

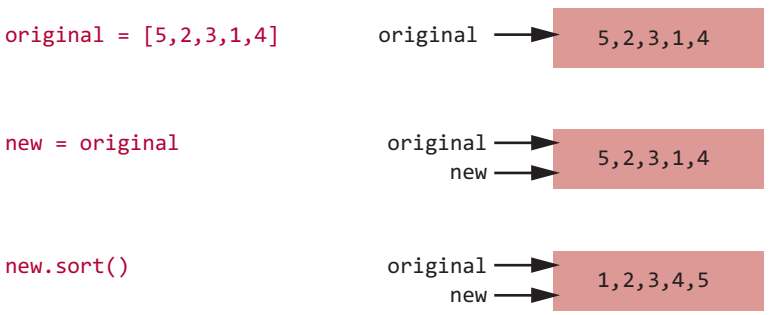
Еще раз напоминаю, что все рассмотренные нами в этом разделе средства сортировки и изменения порядка влияют на оригинал списка. То есть вы теряете исходный порядок размещения элементов. Если вы предпочитаете его сохранить, подвергнув сортировке *копию* списка, можно получить срез списка без указания параметров. Этот прием мы рассматривали чуть раньше:

```
>>> original_list = ['Том', 'Джеймс', 'Сара', 'Фред']
>>> new_list = original_list[:]
>>> new_list.sort()
>>> print original_list
['Том', 'Джеймс', 'Сара', 'Фред']
>>> print new_list
['Джеймс', 'Сара', 'Том', 'Фред']
```

Я рад, что ты это спросил, Картер. Если помнишь, давным-давно, когда ты только начинал знакомиться с именами и переменными (в главе 2), мы говорили, что запись `name1 = name2` означает, что ты присваиваешь уже имеющемуся объекту новое имя. Помнишь эту картинку?

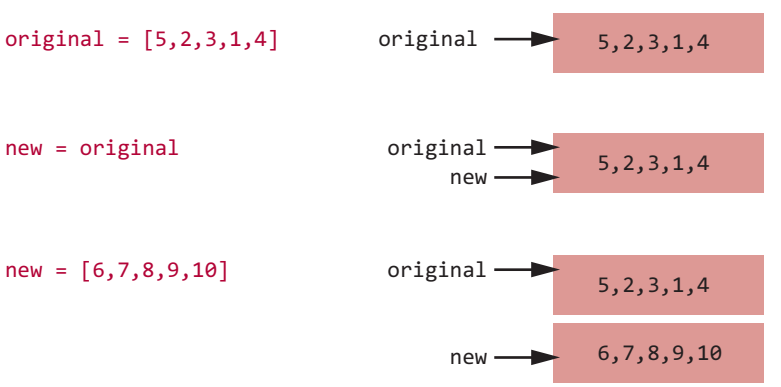


То есть присваивая элементу другое имя, мы просто добавляем к нему еще один ярлык. В приведенном Картером примере переменные `new_list` и `original_list` ссылались на один и тот же список. В итоге список можно редактировать (например сортировать) по любому из имен. Но при этом список у нас всего один. Это выглядит примерно так:



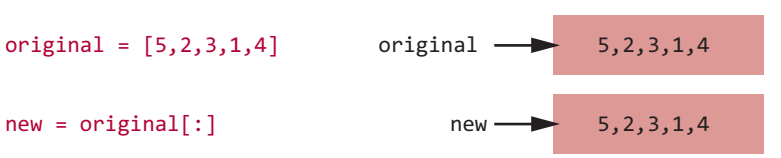
Мы подвергли сортировке список `new`, но при этом сортировке подвергся также список `original`, так как `new` и `original` — имена одного и того же списка.

Конечно, можно перевесить ярлык `new` на новый список:



Именно это мы делали со строками и числами в главе 2.

Это означает, что если вам нужна *копия* списка, нельзя ограничиваться строкой `new = original`. Проще всего в этом случае получить полный срез списка, как я сделал раньше: `new = original[:]`. Такая запись означает «копируем весь список от первого до последнего элемента». После этого мы получим:



Теперь у нас есть два разных списка. Мы создали копию и присвоили ей имя new. И сортировка одного списка больше не будет приводить к автоматической сортировке второго.

Функция sorted()

Существует еще один способ получения отсортированной копии без нарушения порядка размещения элементов в исходном списке. В Python для этой цели используется функция `sorted()`. Вот как она работает:

```
>>> original = [5, 2, 3, 1, 4]
>>> newer = sorted(original)
>>> print original
[5, 2, 3, 1, 4]
>>> print newer
[1, 2, 3, 4, 5]
```

Функция `sorted()` дает вам *отсортированную копию* исходного списка.

Изменяемое и неизменное

Если помните, в главе 2 мы говорили о том, что на самом деле изменить число или строку нельзя. Мы можем поменять только назначенную этому числу или строке *переменную* (другими словами, переместить ярлык). При этом списки относятся к одному из *изменяемых типов* языка Python. Вы только что сами убедились, что в них можно добавлять элементы, сортировать, менять порядок размещения элементов и удалять.

Переменные бывают *изменяемыми* и *неизменными*. В Python числа и строки относятся к первому типу, а списки — ко второму.

Кортежи

Иногда нам требуются списки, не допускающие изменения. Есть ли в Python подобные списки? Конечно же! Существует такой тип объектов, как *кортеж* (tuple), который является именно неизменным списком. Вот как он создается:

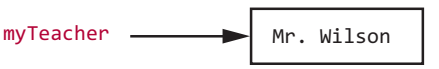
```
my_tuple = ("красный", "зеленый", "синий")
```

Здесь мы используем обычные, а не квадратные скобки, как было бы в случае списка. Так как кортежи не допускают изменения, их невозможно сортировать, в них нельзя добавить элемент или что-то из них удалить. После своего создания кортеж никак не меняется.

Списки списков: таблицы данных

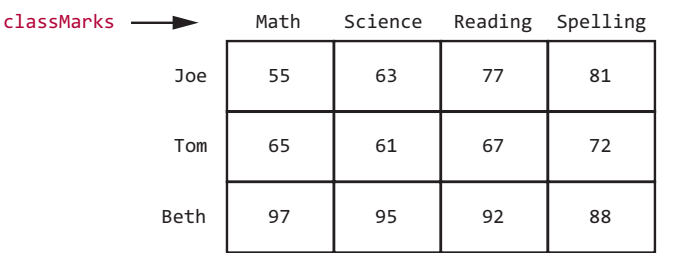
Пытаться представить, как именно в программе хранятся данные, лучше всего с листом бумаги и ручкой.

Вот переменная `myTeacher` (мой учитель) с единственным значением (имя учителя):



А список (в данном случае — список имен моих друзей) напоминает строку значений, связанных друг с другом:

Но иногда нам требуется *таблица* из строк и столбцов (это отметки моих одноклассников по математике, естествознанию, чтению и правописанию):



Как сохранить таблицу данных? Вы уже знаете, что для хранения нескольких элементов у нас служат списки. Поэтому отметки каждого ученика можно поместить в список:

```
>>> joeMarks = [55, 63, 77, 81]
>>> tomMarks = [65, 61, 67, 72]
>>> bethMarks = [97, 95, 92, 88]
```

Еще можно создать список для каждого предмета:

```
>>> mathMarks = [55, 65, 97]
>>> scienceMarks = [63, 61, 95]
>>> readingMarks = [77, 67, 92]
>>> spellingMarks = [81, 72, 88]
```

Но что делать, если вы хотите сохранить всю эту информацию в одной *структуре данных*?

Чтобы создать единую структуру данных для оценок учеников по всем предметам, можно написать такой код:

```
>>> classMarks = [joeMarks, tomMarks, bethMarks]
>>> print classMarks
[[55, 63, 77, 81], [65, 61, 67, 72], [97, 95, 92, 88]]
```

НОВЫЕ СЛОВА

Структурой данных (data structure) называется средство сбора, сохранения и представления данных в программе. К структурам данных относятся переменные, списки и другие элементы, с которыми вы пока не знакомы. Термин «структура данных» на самом деле относится к способу группирования данных в программе.

Это даст вам список элементов, в котором каждый элемент также будет списком. То есть мы создаем список из других списков. Каждый элемент списка `classMarks` представляет собой список.

Более того, список `classMarks` можно создать и без предварительного создания списков `joeMarks`, `tomMarks` и `bethMarks`. Вот как это делается:

```
>>> classMarks = [ [55,63,77,81], [65,61,67,72], [97,95,92,88] ]
>>> print classMarks
[[55, 63, 77, 81], [65, 61, 67, 72], [97, 95, 92, 88]]
```

Теперь выведем содержимое нашей структуры данных на экран. Список `classMarks` состоит из трех элементов, по одному для каждого ученика. Поэтому мы сможем просмотреть их с помощью цикла и ключевого слова `in`:

```
>>> for studentMarks in classMarks:
        print studentMarks
[55, 63, 77, 81]
[65, 61, 67, 72]
[97, 95, 92, 88]
```

Цикл обеспечил перебор списка `classMarks`. Переменная цикла в данном случае называется `studentMarks`. На каждой итерации цикла на экран выводится один элемент списка. Этот элемент представляет собой список с оценками конкретного ученика. (Списки учеников были созданы ранее.)

Обратите внимание, как это похоже на показанную ранее таблицу. То есть у нас есть структура данных, в которой можно сохранить всю информацию одним блоком.

ЗНАЧЕНИЯ ОТДЕЛЬНЫХ ЭЛЕМЕНТОВ

Как получить доступ к сохраненным в таблице (нашем списке списков) значениям? Мы уже знаем, что оценки первого ученика (`joeMarks`) хранятся в списке, который является первым элементом списка `classMarks`. Попробуем сделать так:

```
>>> print classMarks[0]
[55, 63, 77, 81]
```

`classMarks[0]` — список оценок по четырем предметам студента по имени Joe. И нам нужно получить оттуда единственное значение. Как поступить в этом случае? Воспользоваться вторым индексом.

Третья из оценок ученика Joe (оценка за чтение) имеет индекс 2. Поэтому, чтобы вывести ее на экран, напишем так:

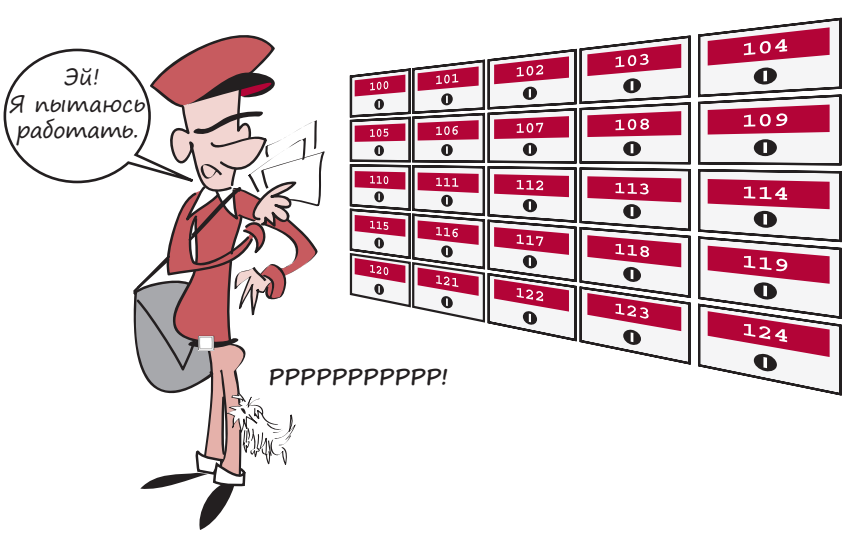
```
>>> print classMarks[0][2]
77
```

Такая запись дает нам первый элемент в списке `classMarks` (с индексом 0), который представляет собой список оценок ученика по имени Joe, и третий элемент из этого списка (с индексом 2), который представляет собой оценку за чтение. Если в коде вы встречаете имя, за которым следуют два набора квадратных скобок, например `classMarks[0][2]`, скорее всего, вы имеете дело со списком списков.

classMarks →

	Math	Science	Reading	Spelling
Joe	55	63	77	81
Tom	65	61	67	72
Beth	97	95	92	88

Список `classMarks` на самом деле ничего не «знает» об именах Joe, Tom и Beth и о том, что такое Math, Science, Reading и Spelling. Это мы пометили их подобным способом, так как изначально собирались сохранить в виде списка. А для интерпретатора Python это всего лишь нумерованные места в списке. Это все равно что абонентские ящики на почте: имен вы на них не найдете, только номера. Почтальон следит за тем, чтобы письма попадали в нужные ящики, а адресат знает, какой из ящиков принадлежит именно ему.



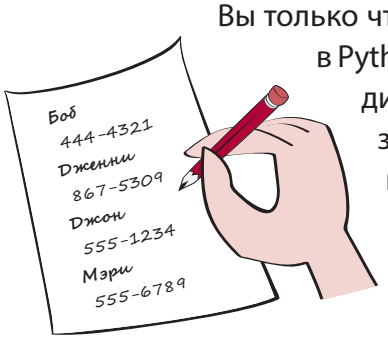
Вот более точный вид меток таблицы `classMarks`:

classMarks →	[0]	[1]	[2]	[3]
classMarks[0]	55	63	77	81
classMarks[1]	65	61	67	72
classMarks[2]	97	95	92	88

Теперь мы сразу видим, что результат 77 оказался в элементе `classMarks[0][2]`.

При создании программы, в которой данные хранятся в структуре `classMarks`, пришлось бы следить за тем, в каких строке и столбце располагается каждый из фрагментов. Нам, как почтальонам, пришлось бы следить за тем, в ящике с каким номером находится каждый фрагмент данных.

Словари



Вы только что познакомились с таким средством хранения наборов данных в Python, как список. Но в программировании часто возникает необходимость собрать элементы в группу таким образом, чтобы каждому значению было сопоставлено другое значение. Аналогично тому, как телефонная книга связывает имена с номерами телефонов, словарь связывает слово с его определением.

В Python *словарь* (dictionary) обеспечивает связывание двух элементом друг с другом. Элементы в этом случае называются *ключом* (key) и *значением* (value). Каждый элемент, или запись в словаре, имеет ключ и значение. Возможно, вы уже слышали про *пары ключ — значение*. Словарь представляет собой коллекцию таких пар.

В качестве простого примера можно рассмотреть список телефонных номеров. Предположим, вам нужно сохранить телефоны друзей. Поиск телефонов будет осуществляться по именам. (Вам повезло, и у друзей с одинаковыми именами у вас просто нет.) Имя в этом случае будет *ключом* (элементом, по которому будет выполняться поиск информации), а номер телефона — *значением* (элементом, который вы ищете).

Рассмотрим один из способов создания такого словаря на языке Python. Для начала создадим пустой словарь:

```
>>> phoneNumbers = {}
```

Точно так же мы создавали список, только в данном случае в выражении задействованы фигурные, а не квадратные скобки.

Добавим первую запись:

```
>>> phoneNumbers["Джон"] = "555-1234"
```

Вот что получится, если теперь вывести содержимое нашего словаря на экран:

```
>>> print phoneNumbers
{'Джон': '555-1234'}
```

Сначала выводится ключ, потом после двоеточия следует значение. Наличие кавычек в данном случае обусловлено тем, что как ключ, так и значение являются строками (но они могут принадлежать и к другому типу данных).

То же самое можно сделать и короче:

```
>>> phoneNumbers = {"Джон": "555-1234"}
```

Внесем в список дополнительные имена. В случае списков мы пользовались методом `append()`, но словари лишены этого удобного инструмента. Поэтому нам остается в явном виде указывать новые ключи и соответствующие им значения:

```
>>> phoneNumbers["Мэри"] = "555-6789"
>>> phoneNumbers["Боб"] = "444-4321"
>>> phoneNumbers["Дженни"] = "867-5309"
```

Посмотрим на содержимое нашего словаря:

```
>>> print phoneNumbers
{'Боб': '444-4321', 'Джон': '555-1234', 'Мэри': '555-6789', 'Дженни':
'867-5309'}
```

А теперь то, зачем мы, собственно, создавали словарь — поиск элементов. Попробуем найти что-нибудь по имени:

```
>>> print phoneNumbers["Мэри"]
'555-6789'
```

Обратите внимание, что ключ, значение которого мы хотим найти, заключен в квадратные скобки, в то время как для словаря используются фигурные.

Словарь во многом напоминает список. Оба типа являются *коллекциями*, то есть они дают возможность собирать вместе элементы разных типов.

Вот чем они похожи.

- Как списки, так и словари могут содержать элементы любого типа (даже другие списки и словари), что дает вам возможность составлять коллекции чисел, строк, объектов и даже других коллекций.
- Как для списков, так и для словарей существуют методы поиска отдельных элементов.

А вот чем они различаются.

- Списки являются *упорядоченными*. Порядок, в котором вы помещаете элементы в список, сохраняется. Но при этом списки можно подвергать сортировке. Словари же *не упорядочены*. Выведя на экран добавленные в словарь элементы, вы можете обнаружить, что они выводятся вовсе не в том порядке, в котором вы их туда добавляли.
- Доступ к элементам списков осуществляется по индексам, а доступ к элементам словарей — по ключам:

```
>>> print myList[3]
'eggs'
>>> print myDictionary["Джон"]
'555-1234'
```

Как мы уже упоминали, многие элементы в Python являются объектами, в том числе списки и словари. И для словарей, как и для списков, определены различные методы, а также уже знакомая вам точечная нотация.

Метод `keys()` создает для вас список всех ключей:

```
>>> phoneNumbers.keys()
['Боб', 'Джон', 'Мэри', 'Дженни']
```

Соответственно, метод `values()` генерирует список всех значений:

```
>>> phoneNumbers.values()
['444-4321', '555-1234', '555-6789', '867-5309']
```

Элементы, напоминающие Python-словари, есть и в других языках программирования. Их обычно называют *ассоциативными массивами* (так как они ассоциируют друг с другом ключи и значения). Вы могли слышать и другое название — *хэш-таблица*.

Элементы словарей, как и элементы списков, могут принадлежать к любому типу, в том числе к простым типам (целое число, число с плавающей точкой, строка), коллекциям (список, словарь) или составным типам (объект).

Да, возможны словари, составленные из других словарей, точно так же как возможны списки, составленные из других списков. Впрочем, это утверждение верно только для значений, а вот на ключи накладываются некоторые ограничения. Ранее мы говорили об *изменяемых* и *неизменных* типах. Так вот, ключи словарей могут принадлежать только к неизменным типам (логические, целые, числа с плавающей точкой, строки и кортежи). Список или словарь в качестве ключа использовать нельзя, так как оба принадлежат к изменяемым типам.

Мы отмечали, что словари, в отличие от списков, не упорядочены. Обратите внимание, что добавленный в словарь третьим номер Боба при выводе содержимого словаря идет первым. Для такого элемента, как словарь, не существует понятия порядка размещения элементов, поэтому их сортировка лишена смысла. Но иногда возникает необходимость вывести на экран содержимое словаря в определенном порядке. Как вы помните, списки *допускают* сортировку, поэтому, как только вы получите список ключей, к нему можно применить метод `sorted()` и отобразить элементы, упорядоченные по ключам:

```
>>> for key in sorted(phoneNumbers.keys()):
        print key, phoneNumbers[key]
Боб 444-4321
Дженни 867-5309
Джон 555-1234
Мэри 555-6789
```

Это уже знакомая вам функция `sorted()`, которой мы пользовались при работе со списками. Если подумать, все так и должно быть, так как коллекция словарных ключей представляет собой именно список.

А что делать, если нужно вывести список, упорядоченный по значениям, а не по ключам? В нашем примере это будет сортировка по номерам телефонов, от меньших чисел к большим. В случае словарей нам доступен только однонаправленный поиск. То есть вы можете искать значения по ключам, но не наоборот. И произвести сортировку значений несколько сложнее. Это возможно, просто придется приложить больше усилий:

```
>>> for value in sorted(phoneNumbers.values()):
    for key in phoneNumbers.keys():
        if phoneNumbers[key] == value:
            print key, phoneNumbers[key]
```

```
Боб 444-4321
Джон 555-1234
Мэри 555-6789
Дженни 867-5309
```

В данном случае после сортировки списка значений мы брали каждое значение и в цикле перебирали все ключи, пока не обнаруживали тот, которому оно соответствует.

Со словарями можно проделывать и другие действия.

- Для удаления элемента служит команда `del`:

```
>>> del phoneNumbers["Джон"]
>>> print phoneNumbers
{'Боб': '444-4321', 'Мэри': '555-6789', 'Дженни': '867-5309'}
```

- Удаление всех элементов (очистка словаря) выполняется методом `clear()`:

```
>>> phoneNumbers.clear()
>>> print phoneNumbers
{}
```

- Проверка наличия в словаре определенного ключа выполняется с помощью ключевого слова `in`:

```
>>> phoneNumbers = {'Боб': '444-4321',
                    'Мэри': '555-6789', 'Дженни': '867-5309'}
>>> "Боб" in phoneNumbers
True
>>> "Барб" in phoneNumbers
False
```

Словари часто используются в Python-коде. Разумеется, это далеко не полный список возможностей Python-словарей. Но по крайней мере вы получили общее представление о данном типе объектов, можете использовать их в собственных программах и распознавать, встречая в чужом коде.

Что мы узнали

В этой главе мы узнали:

- что такое списки;
- как добавить элемент в список;
- как удалить элемент из списка;
- как проверить, содержит ли список определенное значение;

- как произвести сортировку списка;
- как получить копию списка;
- что такое кортеж;
- что такое списки списков;
- что такое Python-словари.

Проверь себя

1. Назовите минимум два способа добавления элементов в список.
2. Назовите минимум два способа удаления элементов из списка.
3. Назовите два способа получения отсортированной копии списка с сохранением оригинала.
4. Как проверить, присутствует ли в списке определенное значение?
5. Как определить положение определенного значения в списке?
6. Что такое кортеж?
7. Как создать список списков?
8. Как извлечь из списка списков один элемент?
9. Что такое словарь?
10. Как добавить в словарь элемент?
11. Как в словаре найти элемент по ключу?

Эксперименты

1. Напишите программу, которая будет запрашивать у пользователя пять имен. Имена должны сохраняться в списке и выводиться в конце программы. Примерно вот так:

```
Введите 5 имен (после каждого нажимайте клавишу Enter):
Тони
Пол
Ник
Майкл
Кевин
Это имена: Тони Пол Ник Майкл Кевин
```

2. Измените программу из предыдущего задания, сделав так, чтобы она выводила на экран как исходный, так и отсортированный список.
3. Еще раз измените программу, чтобы она выводила на экран только третье из введенных имен:

```
Третье введенное вами имя: Ник
```

4. И еще раз модифицируйте программу, предоставив пользователю возможность заменять одно из имен. Сначала он должен указать, какое имя нужно заменить, а потом ввести новое имя. В конце программы выведите новый список на экран:

```
Введите 5 имен (после каждого нажимайте клавишу Enter):
Тони
Пол
Ник
Майкл
Кевин
Это имена: Тони Пол Ник Майкл Кевин
Какое имя нужно заменить? (1-5): 4
Новое имя: Питер
Это имена: Тони Пол Ник Питер Кевин
```

5. Напишите словарь, который даст пользователю возможность вводить слова и их определения, а затем искать их. Убедитесь, что пользователю сообщается об отсутствии слова в словаре. После запуска программа должна работать так:

```
Добавить или искать слово (a/l)? a
Введите слово: компьютер
Введите определение: Машина, которая очень быстро считает
Слово добавлено!
Добавить или искать слово (a/l)? l
Введите слово: компьютер
Машина, которая очень быстро считает
Добавить или искать слово (a/l)? l
Введите слово: qwerty
Это слово пока отсутствует в словаре.
```

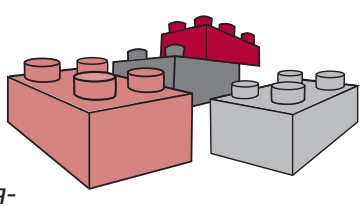
ГЛАВА 13

Функции

Размер и сложность наших программ все время возрастают. Чтобы их было проще писать и модифицировать, имеет смысл разбивать каждую программу на более мелкие фрагменты.

Возможны три способа. *Функции* являются строительным кирпичиками кода, которыми можно пользоваться снова и снова. *Объекты* описывают фрагменты программы как автономные блоки. *Модули* представляют собой отдельные файлы, содержащие части программы. Поговорим о функциях, а объектам и модулям посвящены следующие две главы. После этого знакомство с базовым инструментарием, необходимым для применения графики и звука, а также для создания игр, можно будет считать завершенным.

Функции как строительные кирпичики



Функция представляет собой фрагмент кода, выполняющий некие действия. Это кирпичик, который можно использовать для строительства программы. Одни кирпичики соединяются с другими, как в детском конструкторе. В Python функция создается, или *определяется*, при помощи ключевого слова `def`. После этого функцию можно использовать, или *вызвать*, по ее имени. Для начала рассмотрим простой пример.

Создание функции

В коде листинга 13.1 определяется функция, которая затем используется. Функция выводит на экран почтовый адрес.

Листинг 13.1. Создание и применение функции

```
def printMyAddress():
    print "Warren Sande"
    print "123 Main Street"
    print "Ottawa, Ontario, Canada"
    print "K2M 2E9"
    print
printMyAddress()
```

Определяем (создаем) функцию

Вызываем (используем) функцию

В строке 1 мы определяем функцию с помощью ключевого слова `def`. Мы указываем имя функции, затем ставим скобки `()` и двоеточие:

```
def printMyAddress():
```

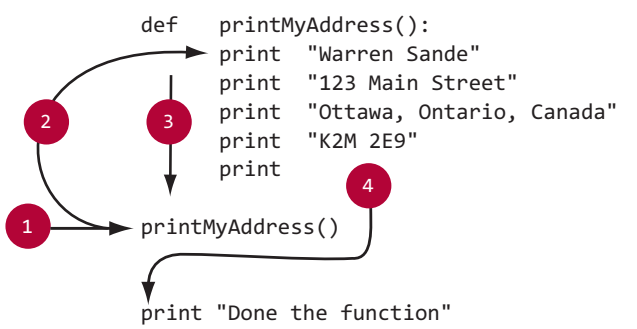
Зачем нужны скобки, вы скоро узнаете. Двоеточие же сообщает интерпретатору Python, что следом идет блок кода (как в случае с циклами `for` и `while` и инструкцией `if`).

После этого мы пишем код функции.

Последняя строка листинга 13.1 относится к основной программе: мы *вызываем* функцию, указывая ее имя и скобки. После этого программа начинает свою работу. Одна строка инициирует выполнение кода ранее определенной нами функции.

Когда основная программа вызывает функцию, функция помогает ей выполнить определенные действия.

Код блока `def` не имеет отношения к основной программе, поэтому после запуска программа пропускает эту часть и начинает работать с первой строки, расположенной вне блока `def`. Следующий рисунок показывает, что происходит при вызове функции. Я добавил в конец программы строку, которая после завершения работы функции выводит на экран соответствующее сообщение.



Рассмотрим показанные на рисунке этапы выполнения программы.

1. Начало. Здесь начинается основная программа.
2. После вызова функции мы переходим на первую строку ее кода.
3. Выполняется каждая строка кода функции.
4. После завершения функции мы продолжаем выполнение основной программы с прерванного места.

Вызов функции

Вызов функции означает запуск ее внутреннего кода. Если вы определите функцию, но ни разу ее не вызовете, ее код так и не будет запущен.

Для вызова функции достаточно указать ее имя, за которым следуют скобки. Иногда в скобках что-то содержится, иногда они пустые. Запустите программу из листинга 13.1 и посмотрите, что произойдет. Вы должны получить канадский адрес Уоррена Сэнд:

```
>>> ================== RESTART ==================
>>>
Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9
>>>
```

Совершенно аналогичный результат нам дал бы запуск вот такой намного более простой программы:

```
print "Warren Sande"
print "123 Main Street"
print "Ottawa, Ontario, Canada"
print "K2M 2E9"
print
```

Зачем же мы все усложнили и добавили в листинг 13.1 функцию?

Дело в том, что один раз определенную функцию можно использовать снова и снова, просто вызывая ее там, где это требуется. Например, для пятикратного вывода адреса на экран достаточно написать так:

```
printMyAddress()
printMyAddress()
printMyAddress()
printMyAddress()
printMyAddress()
```

Вот что получится в результате

```
Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9
Warren Sande
123 Main Street
```

```
Ottawa, Ontario, Canada
K2M 2E9
Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9
Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9
Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9
```

Вы могли бы сказать, что аналогичный результат можно получить при помощи цикла. Я знал, что это случится... В данном случае аналогичный результат действительно можно получить при помощи цикла. Но представьте, что вам нужно выводить адрес на экран в пяти разных местах программы. В этом случае цикл вам не поможет.

Еще один довод в пользу использования функций состоит в том, что функция может менять свое поведение при каждом следующем запуске. О том, как этого добиться, вы узнаете в следующем разделе.

Передача аргументов

Пришло время узнать, зачем нужны скобки: для аргументов!

Нет, Картер, компьютеры очень покладисты — они никогда не спорят. В программировании термин *аргумент* означает передаваемый в функцию фрагмент информации. Это называется *передачей* аргумента в функцию.

Представьте, что вам нужна функция, выводящая на печать адреса всех членов вашей семьи. Адрес во всех случаях будет один и тот же, а вот имена разные. В уже готовой программе имя «Warren Sande» запрограммировано в коде функции, но его можно превратить в переменную. И эта переменная будет передаваться в функцию в момент ее вызова.

Как это работает, проще всего продемонстрировать на примере. В листинге 13.2 функция вывода адреса была отредактирована таким образом, что имя адресата превратилось в аргумент. Аргументы, как и все другие переменные, имеют имена. Эту переменную я назвал `myName`.

После запуска функции переменная `myName` принимает значение переданного в функцию аргумента. Этот аргумент указывается в скобках в момент вызова функции.

В листинге 13.2 аргументу `myName` присваивается значение `"Carter Sande"`.

Листинг 13.2. Передача аргумента в функцию

```
def printMyAddress(myName):
    print myName
    print "123 Main Street"
    print "Ottawa, Ontario, Canada"
    print "K2M 2E9"
    print
printMyAddress("Carter Sande")
```

Передаем в функцию аргумент myName

Выводим на экран имя

Передаем строку "Carter Sande" в функцию в качестве аргумента, в результате переменная myName внутри функции будет иметь значение "Carter Sande"

После запуска кода из листинга 13.2 мы получим ровно то, что ожидали:

```
>>> ==================== RESTART ====================
>>>
Carter Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9
```

Это напоминает результат работы первой программы, в которой аргументы не применялись. Но теперь мы можем менять имя адресата при каждом вызове функции:

```
printMyAddress("Carter Sande")
printMyAddress("Warren Sande")
printMyAddress("Kyra Sande")
printMyAddress("Patricia Sande")
```

В результате на экран при каждом вызове функции будет выводиться разная информация. Имя начнет меняться, так как каждый раз в функцию *передается* новое имя:

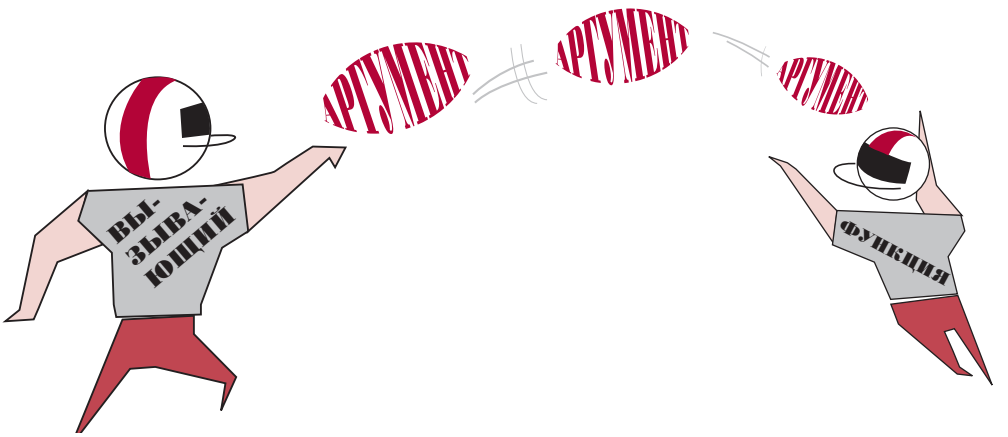
```
>>> ========================== RESTART ==========================
>>>
Carter Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9

Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9

Kyra Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9

Patricia Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9
```

Обратите внимание, что все переданные в функцию значения используются внутри нее и выводятся на экран как часть адреса.

Если мы хотим, чтобы при каждом запуске функции менялось несколько параметров, нам потребуется несколько аргументов. Именно об этом мы сейчас и поговорим.

Функции с несколькими аргументами

Функция из листинга 13.2 обладает одним аргументом. Но аргументов может быть и несколько. Более того, их количество ничем не ограничено. Рассмотрим пример функции с двумя аргументами, чтобы вы поняли, каким образом этого добиться. Затем вы сможете в своих программах добавлять в функции произвольное количество аргументов.



НОВЫЕ СЛОВА

Говоря о передаче элементов в функцию, порой употребляют и другой термин: *параметры*. Некоторые считают термины «аргумент» и «параметр» взаимозаменяемыми. То есть вы можете говорить: «Я передал в эту функцию два параметра» или «Я передал в эту функцию два аргумента».

Другие считают, что термин «аргумент» применим к процессу передачи (когда вы вызываете функцию), в то время как термин «параметр» следует использовать при описании процедуры приема (происходящей внутри функции).

Но в любом случае программисты поймут, что вы имеете в виду, какой бы из двух терминов вы в этом случае ни употребили.

Чтобы Картер получил возможность рассылать письма всем соседям с его улицы, нам требуется функция вывода адреса с двумя аргументами: одним для имени и вторым для номера дома (листинг 13.3).

Листинг 13.3. Функция с двумя аргументами

```
def printMyAddress(someName, houseNum):
    print someName
    print houseNum,
    print "Main Street"
    print "Ottawa, Ontario, Canada"
    print "K2M 2E9"
    print

printMyAddress("Carter Sande", "45")
printMyAddress("Jack Black", "64")
printMyAddress("Tom Green", "22")
printMyAddress("Todd White", "36")
```

Используем две переменные для двух аргументов

Выводятся обе переменные

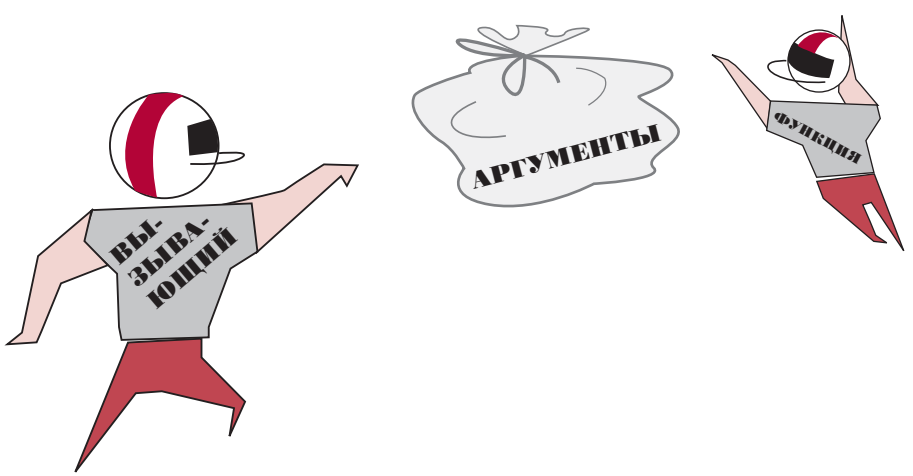
Благодаря запятой номер дома и улица выводятся в одну строку

Вызываем функцию, передавая в нее два параметра

Набор аргументов (или параметров) перечисляется через запятую, как в списках, что заставляет нас перейти к следующей теме….

Слишком много — это сколько?

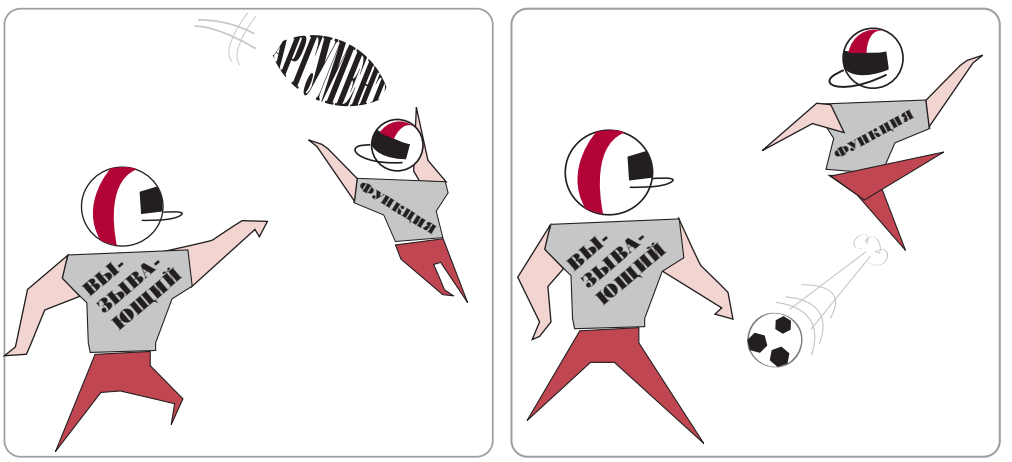
Ранее я отмечал, что вы можете передавать в функцию столько аргументов, сколько вам требуется. Это действительно так, но если ваша функция обладает более чем пятью-шестью аргументами, имеет смысл реализовать код другим способом. Например можно собрать все аргументы в список и передать его в функцию. В этом случае вы будете передавать одну переменную (список), содержащую целую кучу значений, что значительно упрощает чтение вашего кода.



Функции, возвращающие значения

Итак, мы знаем, что функция делает за нас какую-то работу. Но кроме этого функции могут нам кое-что давать.

Вы знаете, как передать в функцию информацию (аргументы), но функция может и сама возвращать информацию в вызывающий код. Эта информация называется *результатом*, или *возвращаемым значением*, функции.



Возврат значения

Заставить функцию возвращать значения позволяет ключевое слово `return`. Рассмотрим, как это выглядит на примере функции, рассчитывающей цену с учетом налога:

```
def calculateTax(price, tax_rate):
    taxTotal = price + (price * tax_rate)
    return taxTotal
```

Значение `taxTotal` возвращается в основную часть программы, которая вызвала функцию. Но куда именно отправляется возвращаемое значение? Оно передается вызвавшему функцию коду. Вот пример:

```
totalPrice = calculateTax(7.99, 0.06)
```

Функция `calculateTax` возвращает значение 8.4694, которое сохраняется в переменной `totalPrice`.

Вы можете заставить функцию вернуть значение в любом месте, где используются выражения. Это значение можно сохранить в переменной (как мы только что сделали), использовать его в другом выражении или вывести на экран, как в этом примере:

```
>>> print calculateTax(7.99, 0.06)
8.4694
>>> total = calculateTax(7.99, 0.06) + calculateTax(6.59, 0.08)
```

Более того, с возвращаемым значением можно вообще ничего не делать:

```
>>> calculateTax(7.49, 0.07)
```

В последнем примере функция запускается и считает итоговую цену, но результат нигде не используется.

Давайте напишем программу с возвращающей значение функцией. В листинге 13.4 возвращает значение функция `calculateTax()`. Мы передаем в нее цену без учета налога и налоговую ставку, а она возвращает нам цену с учетом налога. Это значение сохраняется в переменной. Поэтому если раньше мы ограничивались именем функции, то теперь нам понадобится переменная, знак присваивания (=) и только после этого имя функции. То есть мы присвоим переменной результат, возвращаемый функцией `calculateTax()`.

Листинг 13.4. Создание и использование функции, возвращающей значение

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)
    return total

my_price = float(raw_input ("Введите цену: "))

totalPrice = calculateTax(my_price, 0.06)
print "цена = ", my_price, " Итоговая цена = ", totalPrice
```

Функция рассчитывает налог и возвращает полную стоимость

Возвращаем результат в основную программу

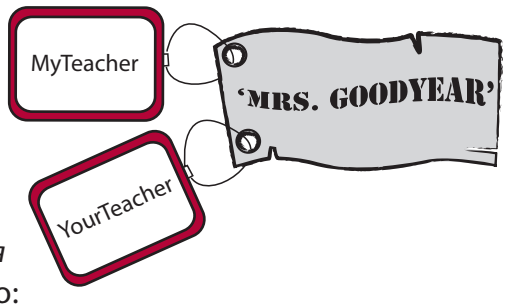
Вызываем функцию и сохраняем результат в переменной totalPrice

Наберите, сохраните и запустите программу из листинга 13.4. Обратите внимание, что налоговая ставка, равная 0.06 (что соответствует 6%-му налогу), жестко прописана в коде. Если же вам потребуется программа, позволяющая рассчитывать цену при разных налоговых ставках, нужно будет сделать так, чтобы пользователь вводил не только начальную цену, но и размер налоговой ставки.

Область видимости переменной

Возможно, вы обратили внимание, что переменные могут располагаться как вне функции, как переменная `totalPrice`, так и внутри нее, как переменная `total`. В данном случае мы имеем дело с двумя названиями одного объекта. Помните, как в главе 2 мы выполняли присваивание `YourTeacher = MyTeacher`?

В примере с функцией `calculateTax` переменные `totalPrice` и `total` представляют собой два ярлыка, присоединенные к одному элементу. Имена внутри функций соз-



даются только после ее запуска. До запуска и после его завершения они не существуют. В интерпретатор Python встроен механизм *управления памятью*, который выполняет все эти действия автоматически. Интерпретатор Python после запуска функции создает новое имя, которое *удаляется после того, как функция заканчивает свою работу*. Последнее крайне важно: после завершения функции все использовавшиеся внутри нее имена перестают существовать.

В процессе работы функции *внешние* имена как бы на время блокируются. Используются только имена, определенные внутри функции. Часть программы, в которой функционирует (или потенциально может функционировать) переменная, называется *областью видимости* (scope).

Локальные переменные

В листинге 13.4 переменные `price` и `total` используются только внутри функции. В этом случае говорят о том, что *областью видимости* переменных `price`, `total` и `tax_rate` является функция `calculateTax()`. Еще они называются *локальными переменными*. Итак, `price`, `total` и `tax_rate` — это локальные переменные функции `calculateTax()`.

Проиллюстрируем смысл этих переменных на практике. Добавим в листинг 13.4 строку, в которой попытаемся вывести на экран значение переменной `price` вне функции.

Листинг 13.5. Вывод локальной переменной

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)
    return total

my_price = float(raw_input ("Введите цену: "))

totalPrice = calculateTax(my_price, 0.06)
print "цена = ", my_price, " Итоговая цена = ", totalPrice
print price
```

Определяем функцию, вычисляющую размер налога и возвращающую итоговую цену

Вызываем функцию, сохраняем и выводим на экран результат ее работы

Пытаемся вывести цену на экран

После запуска этой программы вы получите сообщение об ошибке:

```
Traceback (most recent call last):
    File "C:/.../Listing_13-5.py", line 9, in <module>
        print price
NameError: name 'price' is not defined
```

Эта строка объясняет смысл ошибки

Последняя строка объясняет смысл происходящего: переменная `price` не определена вне функции `calculateTax()`. Она существует только во время работы функции. Поэтому при попытке вывести ее значение в коде, не относящемся к функции, вы получаете сообщение об ошибке.

Глобальные переменные

В отличие от *локальной* переменной `price`, переменные `my_price` и `totalPrice` в листинге 13.5 определены *не внутри* функции, а в основной части программы. Переменные с более широкой областью видимости называются *глобальными*. Если мы удлиним программу из листинга 13.5, переменными `my_price` и `totalPrice` можно будет пользоваться и в добавленном позднее фрагменте, причем они будут иметь те значения, которые им были присвоены ранее. Они все еще будут находиться в пределах *области видимости*. Так как ими можно пользоваться в любой части программы, они называются *глобальными переменными*.

Когда в листинге 13.5 в основной части программы мы попытались вывести значение переменной, определенной внутри функции, то получили сообщение об ошибке. Переменной не существовало; мы вышли *за пределы ее области видимости*. Как вы думаете, что произойдет, если мы попытаемся поступить ровно наоборот: вывести значение глобальной переменной внутри функции?

В листинге 13.6 мы попробуем вывести значение переменной `my_price` в коде функции `calculateTax()`. Запустите ее и посмотрите, что получится.

Листинг 13.6. Глобальная переменная внутри функции

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)
    print my_price
    return total

my_price = float(raw_input ("Введите цену: "))

totalPrice = calculateTax(my_price, 0.06)
print "цена = ", my_price, " Итоговая цена = ", totalPrice
```

Пытаемся вывести значение переменной my_price

Работает ли такая программа? Да! Но почему?

В разделе, посвященном локальным переменным, я упоминал, что интерпретатор Python через механизм управления памятью создает локальную переменную в момент запуска функции. Но диспетчер памяти умеет делать и другие вещи. Он позволяет вам использовать в функции имена переменных, определенных в основной части программы, при условии, что вы не пытаетесь их изменить.

То есть вы можете сделать так:

```
print my_price
```

Или так:

```
your_price = my_price
```

Это возможно, потому что ни одна из этих операций не меняет значение переменной `my_price`. Если же какая-либо часть функции пытается внести в переменную изменения, интерпретатор Python вместо нее создает локальную переменную. Например:

```
my_price = my_price + 10
```

В этом случае имя `my_price` получит новая локальная переменная, которую интерпретатор Python создает при запуске функции.

В листинге 13.6 выводилось значение *глобальной* переменной `my_price`, так как функция никак на нее не влияла. Программа в листинге 13.7 демонстрирует, как при попытке изменить глобальную переменную внутри функции вместо нее создается новая (локальная) переменная. Запустите ее и убедитесь сами.

Листинг 13.7. Изменение глобальной переменной внутри функции

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)

    my_price = 10000
    print "my_price (внутри функции) = ", my_price
    return total

my_price = float(raw_input ("Введит цену: "))

totalPrice = calculateTax(my_price, 0.06)
print "цена = ", my_price, " Итоговая цена = ", totalPrice
print "my_price (вне функции) = ", my_price
```

Меняем переменную my_price внутри функции

Выводим значение локальной версии переменной my_price

Выводим значение глобальной переменной my_price

Эти две переменные my_price находятся в разных фрагментах памяти

После запуска листинга 13.7 вы получите примерно такой результат:

```
>>> ========================== RESTART ==========================
>>>
Введите цену: 7.99
my_price (внутри функции) = 10000
цена = 7.99 Итоговая цена = 8.4694
my_price (вне функции) = 7.99
```

Выводим значение переменной my_price внутри функции

Выводим значение переменной my_price вне функции

Как видите, теперь у нас две разных переменных с именем `my_price` и с разными значениями. Во-первых, это локальная переменная в функции `calculateTax()`, которой мы присвоили значение 10 000. Во-вторых, это глобальная переменная, определенная в основной программе, которой было присвоено введенное пользователем значение 7.99.

Принудительное использование глобальной переменной

В предыдущем разделе вы видели, как при попытке изменить внутри функции значение *глобальной переменной* интерпретатор Python создает вместо нее новую *переменную — локальную*. Это сделано, чтобы исключить возможность случайного изменения глобальной переменной внутри функции.

Но бывают случаи, когда в подобных обстоятельствах нам *требуется* изменить глобальную переменную. Как это можно сделать?

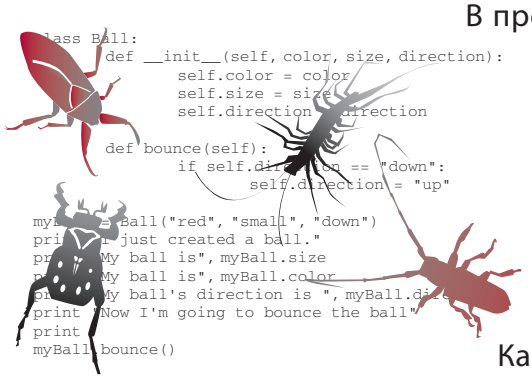
В Python есть ключевое слово `global`, позволяющее обойти запрет на изменение глобальной переменной. Вот как оно используется:

```
def calculateTax(price, tax_rate):
    global my_price
```

Сообщает интерпретатору Python, что нам нужно работать с глобальной переменной my_price

Ключевое слово `global` позволяет интерпретатору Python не создавать локальную переменную `my_price`, а воспользоваться глобальной переменной `my_price`. Если глобальной переменной с таким названием не существует, он ее создаст.

Пара слов об именовании переменных



В предыдущем разделе вы видели, что для локальных и глобальных переменных могут использоваться одни и те же имена. При необходимости интерпретатор Python автоматически создает новые локальные переменные, если это не запрещено в явном виде при помощи ключевого слова `global`. Но я настоятельно не рекомендую вам пользоваться одинаковыми именами.

Как можно заметить в приведенных примерах, различить, где локальная, а где глобальная переменная, достаточно сложно. Существование разных переменных с одинаковыми именами создает путаницу в коде. А путаница, как видите, практически всегда сопровождается багами (ошибками).

Поэтому я советую вам использовать разные имена для локальных и глобальных переменных. Это избавит вас от путаницы и предотвратит появление багов в программе.

Что мы узнали

В этой главе мы узнали:

- что такое функция;
- что такое аргументы (или параметры);
- как передать в функцию аргумент;
- как передать в функцию несколько аргументов;
- как заставить функцию вернуть значение в вызывающий код;
- что такое область видимости переменной и что такое локальные и глобальные переменные;
- как использовать глобальную переменную внутри функции.

Проверь себя

1. Какое ключевое слово используется для создания функции?
2. Как вызывается функция?
3. Как передать в функцию информацию (аргументы)?
4. Каково максимально возможное число аргументов функции?
5. Как заставить функцию вернуть информацию?
6. Что происходит с локальными переменными после того, как функция завершает свою работу?

Эксперименты

1. Напишите функцию, которая будет выводить ваше имя большими буквами, например:

```
  CCCC         A       RRRRR   TTTTTTT  EEEEEE  RRRRR
 C    C       A A      R    R     T     E       R    R
C            A   A     R    R     T     EEEE    R    R
C           AAAAAAA    RRRRR      T     E       RRRRR
 C    C   A        A   R    R     T     E       R    R
  CCCC   A          A  R     R    T     EEEEEE  R     R
```

   Напишите программу, которая будет несколько раз вызывать эту функцию.

2. Напишите функцию, которая будет выводить на экран имя, номера квартиры и дома, названия улицы и города, почтовый индекс и название страны.

   Подсказка: вам потребуется семь аргументов. Их можно передать как по отдельности, так и в виде списка.

3. Сделайте переменную `my_price` в листинге 13.7 глобальной и посмотрите, насколько будет отличаться результат работы программы.
4. Напишите функцию, которая будет считать количество мелких монеток достоинством в одну, пять, десять и пятьдесят копеек (как мы делали в разделе «Эксперименты» главы 5). Функция должна возвращать общую сумму. Затем напишите программу, которая будет вызывать эту функцию. Результат работы этой программы должен выглядеть примерно так:

```
монет по 1 копейке: 3
монет по 5 копеек: 6
монет по 10 копеек: 7
монет по 50 копеек: 2
Всего у нас 2,03 рубля
```

Объекты

В предыдущих главах рассматривались различные способы систематизации данных, программ и группировки элементов. Теперь вы знаете, что *списки* позволяют объединять переменные (данные), а для объединения фрагментов кода в доступный для многократного использования блок применяются *функции*.

Объекты развивают идею объединения элементов еще на один шаг. Они позволяют *собрать в одно целое функции и данные*. Это крайне полезная и применяемая во многих программах концепция. Более того, если внимательно присмотреться к элементам языка Python, практически каждый из них окажется объектом. В терминах программирования Python является *объектно-ориентированным* языком. Это означает, что в данном языке мы можем (и реализуется это очень легко) пользоваться объектами. Можно *обойтись* и без них, но объекты серьезно упрощают многие вещи.

В этой главе мы поговорим о том, что такое объекты, как они создаются и каким образом применяются. Позднее, когда мы займемся разработкой графики, вся эта информация нам очень пригодится.

Объекты в реальном мире

Что такое объект? Если этот вопрос задать в отрыве от программирования, может получиться примерно такой диалог:

Это хорошая отправная точка для определения объектов в Python. К примеру, мяч. Вы можете брать его в руки, бросать, бить по нему ногой, а некоторые мячи можно даже надувать. Все это называется *действиями* (actions). Еще можно описать мяч, сообщив его цвет, размер и вес. Это *атрибуты* мяча.

НОВЫЕ СЛОВА

Любой объект можно описать, перечислив его характеристики, или *атрибуты*. Одним из атрибутов мяча является его круглая форма. Другие примеры атрибутов — цвет, размер, вес и цена. Атрибуты еще принято называть *свойствами* (properties).

С объектами реального мира связаны:

- действия, которые вы можете с ними *проделывать*;
- *описывающие* их атрибуты, или свойства.

Аналогичным образом обстоят дела в программировании.

Объекты в Python

В Python характеристики, то есть «то, что мы знаем об объекте», называются *атрибутами* (attributes), а действия, которые мы можем проделывать с объектами, — *методами* (methods).

Если мы начнем создавать версию, или *модель*, мяча в Python, мяч превратится в объект, обладающий *атрибутами* и *методами*.

Атрибуты мяча (цвет, размер и вес) будут выглядеть так:

```
ball.color
ball.size
ball.weight
```

Это характеристики, при помощи которых вы можете *описать* мяч.

Методы (возможность ударить ногой, бросить и надуть) будут выглядеть следующим образом:

```
ball.kick()
ball.throw()
ball.inflate()
```

Это перечень действий, которые вы можете *проделывать* с мячом.

Что такое атрибуты?

К атрибутам относятся все сведения о мяче, которые вы знаете (или можете узнать). Атрибуты мяча представляют собой фрагменты данных — числа, строки и т. п. Звучит знакомо? Да, это именно переменные. Просто они находятся внутри объекта.

Их значение можно вывести на экран:

```
print ball.size
```

Им можно присваивать значения:

```
ball.color = 'green'
```

Их можно присваивать обычным переменным, не являющимся объектами:

```
myColor = ball.color
```

Кроме того, их можно присваивать атрибутам других объектов:

```
myBall.color = yourBall.color
```

Что такое методы?

Методами называют то, что можно *проделывать* с объектом. Это фрагменты кода, которые *вызываются* для выполнения неких действий. Звучит знакомо? Да, методы представляют собой всего лишь *функции*, помещенные внутрь объекта.

С методами можно делать то же самое, что и с обычными функциями, в том числе *передавать им аргументы* и заставлять их *возвращать значения*.

Объект = атрибуты + методы

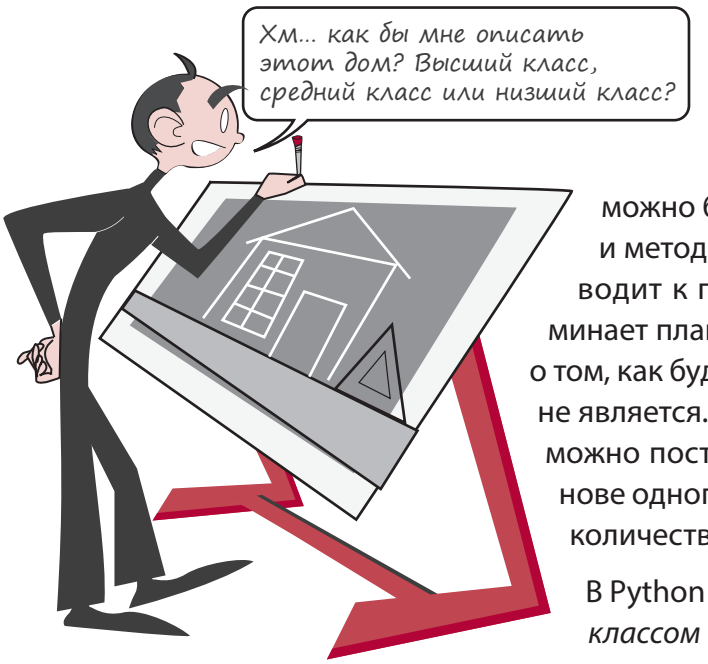
Итак, объекты являются способом объединения *атрибутов* и *методов* (характеристик, которые вы знаете, и действий, которые вы можете проделывать) внутри одного элемента. Атрибуты представляют собой информацию, а методы — действия.

Что такое точка?

Вы, вероятно, обратили внимание на точку между именем объекта и именем атрибута или метода в приведенных примерах. Это всего лишь обозначение, которое применяется в Python, когда нужно воспользоваться атрибутами или методами объекта: *объект.атрибут* или *объект.метод()*. Все очень просто. Такая запись называется *точечной нотацией* и фигурирует во многих языках программирования.

Теперь вы представляете, что такое объекты. Попробуем воспользоваться ими на практике!

Создание объектов



Создание объектов в Python происходит в два этапа.

В первую очередь нужно определить, как будет выглядеть объект и что с ним можно будет делать, то есть задать его атрибуты и методы. Но само по себе это описание не приводит к появлению объекта. Скорее, оно напоминает план дома. План дает точное представление о том, как будет выглядеть дом, но при этом домом он не является. Жить в плане нельзя. Но с его помощью можно построить реальный дом. Более того, на основе одного плана можно построить произвольное количество домов.

В Python описание, или план, объекта называется *классом* (class).

На втором этапе мы на основе класса создаем реальный объект. Этот объект называется *экземпляром* (instance) класса.

Посмотрим на пример создания *класса* и его *экземпляра*. Листинг 14.1 демонстрирует определение простого класса `Ball`.

Листинг 14.1. Создание простого класса Ball

```
class Ball:                                  # Эта строка говорит интерпретатору Python, что мы создаем класс

    def bounce(self):
        if self.direction == "вниз":         # Это метод
            self.direction = "вверх"
```

Листинг 14.1 содержит определение класса для мяча с единственным методом `bounce()`, благодаря которому мяч может отскакивать от пола. А как насчет атрибутов? В данном случае атрибуты классу не принадлежат — они принадлежат отдельным экземплярам. И поэтому у каждого экземпляра будет свой набор атрибутов.

Существует два способа задания атрибутов экземпляра. Мы рассмотрим оба.

Создание экземпляра объекта

Как уже упоминалось, определение класса объектом не является. Это всего лишь план. Давайте построим на его основе дом.

Вот как создается *экземпляр* класса `Ball`:

```
myBall = Ball()
```

Наш мяч пока лишен атрибутов, поэтому давайте их добавим:

```
myBall.direction = "вниз"
myBall.color = "зеленый"
myBall.size = "маленький"
```

Это один из способов задания атрибутов объекта. Второй рассматривается в следующем разделе.

Проверим работу метода `bounce()` на практике. Вот как им воспользоваться:

```
myBall.bounce()
```

Соберем все эти фрагменты в одну программу, добавив к ним инструкции `print`, чтобы видеть, что именно происходит (листинг 14.2).

Листинг 14.2. Использование класса Ball

```
class Ball:

    def bounce(self):
        if self.direction == "вниз":
            self.direction = "вверх"

myBall = Ball()
myBall.direction = "вниз"
myBall.color = "красный"
myBall.size = "маленький"

print "Я только что создал мяч."
print "Размер моего мяча", myBall.size
print "Цвет моего мяча", myBall.color
print "Мой мяч движется", myBall.direction
print "Сейчас я поменяю направление движения мяча"
print
myBall.bounce()
print "Теперь мяч движется", myBall.direction
```

Это наш класс, тот же самый, что и раньше

Создаем экземпляр нашего класса

Задаем атрибуты

Выводим значения атрибутов объекта

Вызываем метод

Вот результат работы программы из листинга 14.2:

```
>>> ========================= RESTART =========================
>>>
Я только что создал мяч.
Размер моего мяча маленький
Цвет моего мяча красный
Мой мяч движется вниз
Сейчас я поменяю направление движения мяча

Теперь мяч движется вверх
```

Заданные нами атрибуты

Меняем направление движения методом bounce()

Направление изменилось с «вниз» на «вверх»

Обратите внимание, что после вызова метода `bounce()` атрибут направления (`direction`) изменил свое значение с `вниз` на `вверх`, что, собственно, и должен делать этот метод.

Инициализация объекта

В момент создания объекта переменные `size`, `color` и `direction` не имели определенных значений. Они присваиваются *уже готовому* объекту. Но также существует возможность задать свойства объекта в процессе его создания. Это называется *инициализацией* объекта.

НОВЫЕ СЛОВА

Термин *инициализация* (initializing) означает «получение готового объекта с самого начала». Инициализируя элемент программного обеспечения, мы готовим его к применению, помещая в нужное нам состояние.

Одновременно с определением класса можно определить специальный метод `__init__()`, который будет запускаться при каждом создании экземпляра класса. Передавая в метод `__init__()` аргумент, вы получаете экземпляр с нужными вам свойствами (листинг 14.3).

Листинг 14.3. Добавление метода __init__()

```
class Ball:
    def __init__(self, color, size, direction):
        self.color = color
        self.size = size
        self.direction = direction

    def bounce(self):
        if self.direction == "вниз":
            self.direction = "вверх"
```

Это метод _init_(), с каждой стороны от init по два нижних подчеркивания

```
myBall = Ball("красный", "маленький", "вниз")
print "Я только что создал мяч."

print "Размер моего мяча", myBall.size
print "Цвет моего мяча", myBall.color
print "Мой мяч движется вниз ", myBall.direction
print "Сейчас я поменяю направление движения мяча"
print
myBall.bounce()
print "Теперь мяч движется", myBall.direction
```

Атрибуты при помощи аргумента передаются в метод __init__()

Запустив программу из листинга 14.3, вы получите тот же самый результат, что и при запуске программы из листинга 14.2. Но в листинге 14.3 атрибуты объекта задаются при помощи метода `__init__()`.

Спасибо за совет, Картер. В следующем разделе мы поговорим о том, что это за «волшебные» методы.

Метод __str__()

Объекты в Python обладают рядом «волшебных», как выражается Картер, методов. Разумеется, ничего волшебного в них нет! Это просто методы, которые интерпретатор Python автоматически добавляет к любому создаваемому классу. Программисты, пишущие на языке Python, обычно называют их *специальными методами*.

Вы уже видели метод `__init__()`, отвечающий за инициализацию объекта в процессе его создания. Этот метод встроен во все объекты. Если вы не добавите в определение класса ни одного метода, всю работу начнет делать встроенный метод, умеющий только создавать объекты.

Еще один специальный метод — `__str__()` — сообщает интерпретатору Python, что́ нужно выводить на экран после выполнения команды `print` для объекта. По умолчанию интерпретатор сообщает следующее:

- где определен объект (в приведенном Картером примере `__main__` означает основную часть программы);
- имя класса (`Ball`);
- место в памяти, где хранится экземпляр (`0x00BB83A0`).

Но если вы хотите вывести какую-то другую информацию, связанную с объектом, нужно определить собственный метод `__str__()`, который переопределит встроенный метод. Пример такого поведения рассмотрен в листинге 14.4.

Листинг 14.4. Использование метода __str__() для изменения выводимой на экран информации

```
class Ball:
    def __init__(self, color, size, direction):
        self.color = color
        self.size = size
        self.direction = direction

    def __str__(self):
        msg = "Привет, я " + self.size + " " + self.color + " мяч!"
        return msg
myBall = Ball("красный", "маленький", "вниз")
print myBall
```

Это метод __str__()

Вот что мы получим при запуске программы из листинга 14.4:

```
>>> ================ RESTART ================
>>>
Привет, я маленький красный мяч!
```

Это выглядит понятнее, чем `<__main__.Ball instance at 0x00BB83A0>`. Имена всех «волшебных» методов снабжаются двумя нижними подчеркиваниями с каждой стороны.

Что такое self

Вы могли обратить внимание на слово `self`, фигурировавшее в атрибутах классов и определениях методов. Примерно вот так:

```
def bounce(self):
```

Что означает это слово? Помните, мы говорили о том, что по плану можно построить произвольное количество домов? Кроме того, класс позволяет создать несколько экземпляров объекта:

```
cartersBall = Ball("красный", "маленький", "вниз")
warrensBall = Ball("зеленый", "средний", "вверх")
```

Когда мы вызываем для одного из этих экземпляров метод `warrensBall.bounce()`, ему нужно дать понять, о каком из экземпляров идет речь. Направление движения какого из мячей — `cartersBall` или `warrensBall` — следует поменять? Именно аргумент `self` сообщает методу, о каком объекте идет речь. Он называется *переменной экземпляра* (instance reference).

Но подождите! При вызове метода `warrensBall.bounce()` аргументы в скобках отсутствовали, а в самом методе появился аргумент `self`. Откуда он взялся, если мы ничего не передавали? Это еще одно «волшебство», которое интерпретатор Python делает с объектами. При вызове метода класса в него автоматически передается *переменная экземпляра* — информация о том, какой именно экземпляр имеется в виду.

Это все равно что написать вот так:

```
Ball.bounce(warrensBall)
```

В данном случае мы сообщили методу `bounce()`, направление движения какого мяча следует поменять. На самом деле представленная строка также будет работать, так как именно это интерпретатор Python делает в фоновом режиме в ответ на запись `warrensBall.bounce()`.

В главе 11 мы писали программу, создающую хот-доги. Сейчас мы определим для хот-догов отдельный класс, чтобы продемонстрировать работу с объектами.

Переменная `self` в Python не является ключевым словом. Просто все используют ее для обозначения переменной экземпляра. Это еще одно соглашение, облегчающее чтение кода. При желании вы можете использовать для переменной экземпляра собственную версию имени, но я рекомендую вам придерживаться общепринятых обозначений, чтобы избежать путаницы.

Пример класса

В нашем примере в состав хот-дога булочка будет входить всегда. (В противном случае получится слишком неаккуратно.) Снабдим хот-дог набором атрибутов и методов.

Вот его атрибуты:

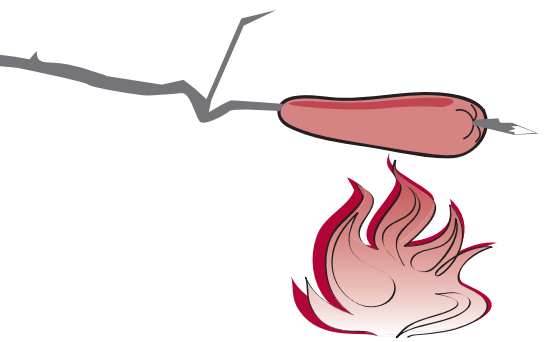
- `cooked_level` — число, показывающее, сколько времени готовилась сосиска. Значение 0–3 будет соответствовать сырому состоянию, больше 3 — средней про-

жарке, больше 5 — хорошо прожаренному, а выше 8 получится уголь! Изготовление хот-дога будет начинаться с сырого состояния;

- `cooked_string` — строка, описывающая степень прожарки сосиски;
- `condiments` — список компонентов хот-дога, таких как кетчуп, горчица и т. п.

Вот его методы:

- `cook()` — готовит сосиску определенное время, увеличивая степень ее прожарки;
- `add_condiment()` — добавляет в хот-дог компоненты;
- `__init__()` — создает экземпляр и задает его свойства, предлагаемые по умолчанию;
- `__str__()` — улучшает внешний вид выводимой на экран информации.



Первым делом нам требуется определить класс. Начнем с метода `__init__()`, задающего атрибуты хот-дога, предлагаемые по умолчанию:

```
class HotDog:
    def __init__(self):
        self.cooked_level = 0
        self.cooked_string = "Сырая"
        self.condiments = []
```

Начинаем с сырой сосиски. Теперь напишем метод приготовления сосиски:

```
def cook(self, time):
    self.cooked_level = self.cooked_level + time
        if self.cooked_level > 8:
            self.cooked_string = "Сгоревшая"
        elif self.cooked_level > 5:
            self.cooked_string = "Хорошо прожаренная"
        elif self.cooked_level > 3:
            self.cooked_string = "Средней прожарки"
        else:
            self.cooked_string = "Сырая"
```

Увеличиваем степень готовности в зависимости от сроков

Задаем строки для различных уровней готовности

Перед тем как приступить к остальным частям кода протестируем уже имеющийся. Первым делом мы создаем экземпляр хот-дога и проверяем его атрибуты:

```
myDog = HotDog()
print myDog.cooked_level
print myDog.cooked_string
print myDog.condiments
```

Объединим эти фрагменты кода в программу и запустим ее (листинг 14.5).

Листинг 14.5. Начало программы по изготовлению хот-догов

```
class HotDog:
    def __init__(self):
        self.cooked_level = 0
        self.cooked_string = "Сырая"
        self.condiments = []
    def cook(self, time):
    self.cooked_level = self.cooked_level + time
    if self.cooked_level > 8:
        self.cooked_string = "Сгоревшая"
    elif self.cooked_level > 5:
        self.cooked_string = "Хорошо прожаренная"
    elif self.cooked_level > 3:
        self.cooked_string = "Средней прожарки"
    else:
        self.cooked_string = "Сырая"
myDog = HotDog()
print myDog.cooked_level
print myDog.cooked_string
print myDog.condiments
```

Запустим код из листинга 14.5 и посмотрим, что получится:

```
>>>
0
Сырая
[]
```

Переменная cooked_level

Переменная cooked_string

Переменная condiments

Итак, мы видим, что атрибут `cooked_level` равен нулю, атрибут `cooked_string` — строке `"Сырая"`, а атрибуту `condiments` пока не присвоено никакого значения.

Проверим, как работает метод `cook()`. Для этого добавим в код листинга 14.5 такие строки:

```
print "Сейчас я буду готовить хот-дог"
myDog.cook(4)                                  Готовим сосиску 4 минуты
print myDog.cooked_level
print myDog.cooked_string                      Проверяем новые значения атрибутов
```

Снова запустите программу. Теперь результат ее работы будет выглядеть так:

```
>>>
0
Сырая                                          До приготовления
[]
Сейчас я буду готовить хот-дог
4
Средней прожарки                               После приготовления
```

Кажется, наш метод `cook()` работает. Переменная `cooked_level` изменилась с 0 на 4, поменялось и значение строки (от `"Сырая"` до `"Средней прожарки"`).

Попробуем добавить другие компоненты. Для этого нам потребуется еще один метод. Одновременно можно добавить функцию `__str__()`, упростив выводимую информацию. Измените свою программу в соответствии с листингом 14.6.

Листинг 14.6. Класс HotDog с методами cook(), add_condiments() и __str__()

```
class HotDog:                                                    Определяем класс
    def __init__(self):
        self.cooked_level = 0
        self.cooked_string = "Сырая"
        self.condiments = []
    def __str__(self):                                           Определяем новый метод __str__()
        msg = "сосиска"
        if len(self.condiments) > 0:
            msg = msg + " с "
        for i in self.condiments:
            msg = msg+i+", "
        msg = msg.strip(", ")
        msg = self.cooked_string + " " + msg + "."
        return msg
```

```
    def cook(self, time):
        self.cooked_level=self.cooked_level+time
        if self.cooked_level > 8:
            self.cooked_string = "Сгоревшая"
        elif self.cooked_level > 5:
            self.cooked_string = "Хорошо прожаренная"
        elif self.cooked_level > 3:
            self.cooked_string = "Средней прожарки"
        else:
            self.cooked_string = "Сырая"
    def addCondiment(self, condiment):
        self.condiments.append(condiment)

myDog = HotDog()
print myDog
print "Готовим сосиску 4 минуты..."
myDog.cook(4)
print myDog
print "Готовим сосиску еще 3 минуты..."
myDog.cook(3)
print myDog
print "Что произойдет, если я буду ее готовить еще 10 минут?"
myDog.cook(10)
print myDog
print "Сейчас я добавлю в хот-дог другие компоненты"
myDog.addCondiment("кетчуп")
myDog.addCondiment("горчица")
print myDog
```

Определяем класс

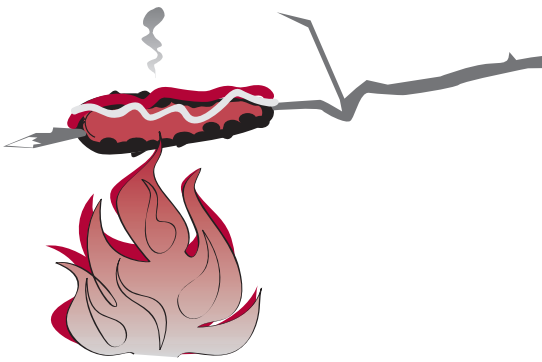
Определяем новый метод add_condiments()

Создаем экземпляр

Проверяем, все ли работает

Код этого листинга занимает много места, но я рекомендую вам его набрать. Часть его вы уже набирали для листинга 14.5. Ну а те, у кого устали пальцы или нет свободного времени, могут загрузить его из папки *\examples* на сайте нашей книги. Запустим программу, чтобы посмотреть, что получилось. Результат должен выглядеть примерно так:

```
>>> ============================== RESTART ==============================
>>>
Сырая сосиска.
Готовим сосиску 4 минуты...
Средней прожарки сосиска.
Готовим сосиску еще 3 минуты...
Хорошо прожаренная сосиска.
Что произойдет, если я буду готовить ее еще 10 минут?
Сгоревшая сосиска.
Сейчас я добавлю в хот-дог другие компоненты
Сгоревшая сосиска с кетчупом и горчицей.
```



Первая часть нашей программы создает класс. Вторая часть проверяет методы приготовления виртуальной сосиски и привносит в хот-дог дополнительные компоненты. Но судя по последним двум строкам, мы жарили сосиску слишком долго, напрасно потратив кетчуп и горчицу!

Скрываем данные

Возможно, вы уже осознали, что существует два способа просмотра и изменения данных (атрибутов) внутри объекта. Это можно сделать при непосредственном доступе к атрибуту, например:

```
myDog.cooked_level = 5
```

Можно также модифицировать его с помощью метода, например:

```
myDog.cook(5)
```

Если приготовление начинается с сырой сосиски (`cooked_level = 0`), эти две строки будут делать одно и то же. Они присваивают параметру `cooked_level` значение 5. Зачем нам это делать при помощи метода? Почему не прибегнуть к прямому присваиванию?

С моей точки зрения, по двум причинам:

- в случае непосредственного изменения атрибутов процесс приготовления сосиски разобьется по меньшей мере на две части: изменение переменной `cooked_level` и изменение переменной `cooked_string`. В качестве альтернативы выступает единственный вызов метода, который сделает все, что нам нужно;
- при непосредственном доступе к атрибутам никто не мешает нам написать, например, такую инструкцию:

```
cooked_level = cooked_level - 2
```

  В результате степень прожарки сосиски *понизится*. Но в реальной жизни подобное невозможно! Значит, эта строка просто не имеет смысла. Применение метода гарантирует, что значение переменной `cooked_level` будет только увеличиваться.

НОВЫЕ СЛОВА

В программировании ограничение доступа к данным объекта, при котором их изменение может осуществляться только через метод, называется *сокрытием данных* (data hiding). В Python отсутствуют средства принудительного сокрытия данных, но при желании вы можете писать код в соответствии с этим правилом.

Итак, вы знаете, что объекты обладают атрибутами и методами. Вы научились создавать объекты и инициализировать их при помощи специального метода `__init__()`. Еще вы познакомились со специальным методом `__str__()`, позволяющим выводить данные об объекте в более приемлемой форме.

Полиморфизм и наследование

А сейчас мы познакомимся, возможно, с самыми важными аспектами применения объектов: *полиморфизмом* (polymorphism) и *наследованием* (inheritance). Эти концепции делают объекты крайне полезными. Рассмотрим кратко, что они означают.

разные поведения одного метода

Говоря простыми словами, у нас может быть два (или более) метода с одинаковыми именами для разных классов. Эти методы могут вести себя по-разному в зависимости от того, к какому классу мы их применяем.

Представим, что мы пишем программу для занятий геометрией, которая вычисляет площадь треугольника и квадрата. Можно определить два класса:

```
class Triangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def getArea(self):
        area = self.width * self.height / 2.0
        return area

class Square:
    def __init__(self, size):
        self.size = size

    def getArea(self):
        area = self.size * self.size
        return area
```

Это класс Triangle (треугольник)

В обоих классах присутствует метод getArea()

Это класс Square (квадрат)

Как в классе `Triangle`, так и в классе `Square` присутствует метод `getArea()`. Поэтому, мы можем получить экземпляр каждого класса:

```
>>> myTriangle = Triangle(4, 5)
>>> mySquare = Square(7)
```

Далее при помощи единственного метода `getArea()` мы можем вычислить площадь любой из наших двух фигур:

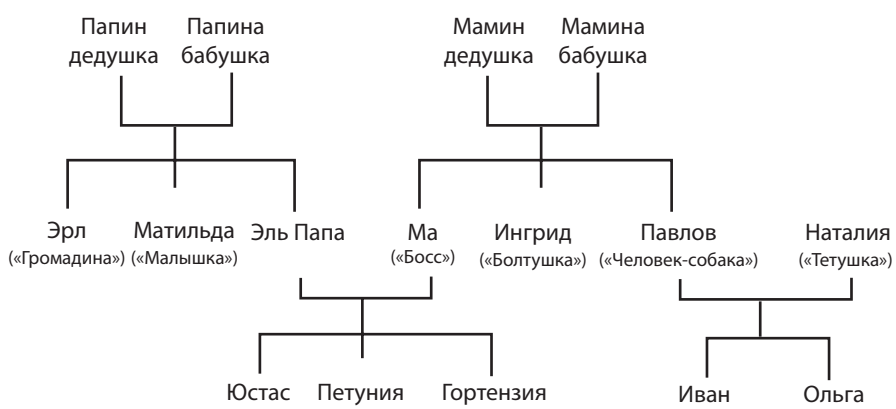
```
>>> myTriangle.getArea()
10.0
>>> mySquare.getArea()
49
```

Мы использовали для каждой из фигур метод `getArea()`, хотя для всех них он сработал по-своему. Это пример *полиморфизма*.

Учимся у родителей

В реальном мире люди могут *наследовать* от своих родителей и других родственников разное. Можно унаследовать такие особенности, как рыжие волосы, а можно деньги или недвижимость.

В объектно-ориентированном программировании классы могут наследовать атрибуты и методы от других классов. Это позволяет получать целые «семейства» классов, пользующихся общими атрибутами и методами, что избавляет от необходимости начинать все «с нуля», если нам требуется добавить в семью еще одного члена.



Класс, наследующий атрибуты или методы, называется *производным классом*, или *подклассом*. Покажем, что это такое, на примере.

Представьте, что вы пишите игру, по ходу которой игрок может собирать различные вещи, например еду, деньги или одежду. Можно определить класс `GameObject`. Он будет содержать такие атрибуты, как `name` (имя), например «монетка», «яблоко» или «шляпа», и такие методы, как `pickUp()` (добавление объекта в коллекцию игрока). При этом у всех игровых объектов должны быть одинаковые методы и атрибуты.

После этого создадим *подкласс* для монеток. Класс `Coin` станет *производным* от класса `GameObject`. Он будет *наследовать* атрибуты и методы класса `GameObject`, поэтому в него автоматически попадут атрибут `name` и метод `pickUp()`. Также классу `Coin`

понадобится атрибут `value` (достоинство монетки) и метод `spend()` (позволяет тратить монеты на покупку объектов).

Вот как может выглядеть код таких классов:

```
class GameObject:
    def __init__(self, name):
        self.name = name

    def pickUp(self, player):
        # сюда поместим код, добавляющий объекты
        # в коллекцию игрока

class Coin(GameObject):
    def __init__(self, value):
        GameObject.__init__(self, "монетка")
        self.value = value

    def spend(self, buyer, seller):
        # сюда поместим код, удаляющий монетку
        # из денег покупателя и
        # добавляющий ее к деньгам продавца
```

Определяем класс GameObject

Класс Coin — это подкласс класса GameObject

В методе __init__() наследуются начальные значения класса GameObject и добавляются предметы

Новый метод spend() для класса Coin

Просчитывайте свои действия

В последнем примере методы не содержат реального кода. Вместо него мы добавили туда комментарии, объясняющие назначение методов. Это способ планирования, позволяющий заранее наметить план дальнейших действий. Реальный код зависит от того, как развивается игра. Программисты часто так поступают, чтобы систематизировать мысли при создании сложного кода. «Пустые» функции и методы называются *кодовыми заглушками* (code stubs).

При попытке запустить предыдущий фрагмент кода вы получите сообщение об ошибке, так как определения функций отсутствуют.

Это так, Картер, но комментарии не считаются, они предназначены только для тебя, а не для компьютера.

В Python существует ключевое слово `pass`, которое используется, когда нужно сделать заглушку кода. Поэтому реальный код выглядел бы так:

```
class Game_object:
    def __init__(self, name):
        self.name = name

    def pickUp(self):
        pass
        # сюда поместим код, добавляющий объекты
        # в коллекцию игрока

class Coin(Game_object):
    def __init__(self, value):
        GameObject.__init__(self, "coin")
        self.value = value

    def spend(self, buyer, seller):
        pass
            # сюда поместим код, удаляющий монетку
            # из денег покупателя и
            # добавляющий ее к деньгам продавца
```

Более детальные примеры использования объектов, полиморфизма и наследования в этой главе мы рассматривать не будем. Их вы найдете в следующих главах. Лучше понять принцип работы с объектами вы сможете, воспользовавшись ими для создания реальных программ, например игр.

Что мы узнали

В этой главе мы изучили:

- чем являются объекты;
- атрибуты и методы;
- чем является класс;
- как создать экземпляр класса;
- специальные методы `__init__()` и `__str__()`;
- полиморфизм;
- наследование;
- кодовые заглушки.

Проверь себя

1. При помощи каких ключевых слов определяется новый тип объекта?
2. Что такое атрибуты?

3. Что такое методы?
4. В чем разница между классом и его экземпляром?
5. Какое имя обычно используется в методах для переменной экземпляра?
6. Что такое полиморфизм?
7. Что такое наследование?

Эксперименты

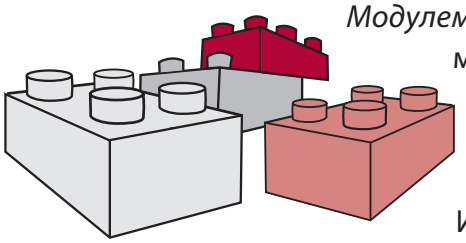
1. Напишите определение класса `BankAccount` для банковского счета. Оно должно содержать атрибуты для имени (строка), номера счета (строка или целое число) и состояния счета (число с плавающей точкой). С ним должны быть связаны методы вывода состояния счета на экран, добавления денег на счет и снятия их оттуда.
2. Создайте класс с именем `InterestAccount` для получения процентного дохода. Он должен быть подклассом класса `BankAccount` (наследуя его атрибуты и методы). Еще в нем должны присутствовать атрибут процентной ставки и метод ее добавления. Для простоты предположим, что метод `addInterest()` вызывается раз в год для начисления процентов по вкладу и обновления состояния счета.

ГЛАВА 15

Модули

Это последняя глава, в которой обсуждаются способы объединения элементов. Вы уже знаете, что такое *списки*, *функции* и *объекты*. Осталось поговорить о модулях, после чего в следующей главе вы воспользуетесь *модулем* Pygame для создания графики.

Что такое модуль



Модулем (module) называется часть, или фрагмент, какого-то элемента. Если элемент представлен в виде набора фрагментов или может быть легко разобран на фрагменты, его называют *модульным* (modular). Превосходной иллюстрацией модульной конструкции являются блоки для игры LEGO. Из массы различных фрагментов можно создать множество разнообразных конструкций.

В Python модулями называются небольшие фрагменты большой программы. Каждый модуль находится в отдельном файле на жестком диске вашего компьютера. Можно взять большую программу и разбить ее на несколько модулей, или файлов. А можно пойти другим путем — начать с маленького модуля и, добавляя фрагменты, создать большую программу.

Зачем нужны модули

Но зачем нам придумывать себе дополнительную головную боль, разбивая программу на части, если потом их все равно потребуется сложить вместе, чтобы программа начала функционировать? Почему не оставить все в одном большом файле?

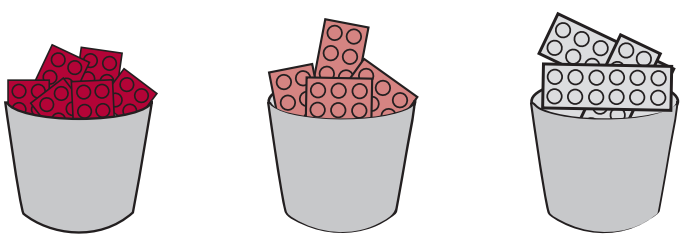
Для этого есть несколько причин:

- файлы становятся меньше, что упрощает поиск различных фрагментов кода;
- однажды созданный модуль можно использовать в разных программах. Вам не приходится писать код «с нуля», когда возникает необходимость вставить в программу те же самые функции;

- далеко не всегда нам требуется именно этот набор модулей. Модульность означает возможность комбинировать различные фрагменты для достижения разных результатов, аналогично тому, как из конструктора LEGO можно собирать разные объекты.

Корзины блоков

В главе 13, посвященной функциям, вы узнали, что функции напоминают строительные блоки. Модуль в этом случае рассматривается как корзина таких блоков. В одну корзину можно поместить произвольное количество блоков и пользоваться множеством разных корзин. Никто не мешает создать одну корзину для блоков квадратной формы, другую — для плоских, третью — для блоков неправильной формы. Именно так обычно поступают программисты, объединяя в модуль однотипные функции. Впрочем, в модуль можно объединить и функции, требующиеся для конкретной программы, собрав все строительные блоки в одну корзину.



Создание модуля

Давайте создадим модуль. Это будет Python-файл, представленный, например, в листинге 15.1. Введите код листинга в IDLE-редактор и сохраните его под именем *my_module.py*.

Листинг 15.1. Создание модуля

```
# это файл "my_module.py"
# мы собираемся использовать его в другой программе
def c_to_f(celsius):
    fahrenheit = celsius * 9.0 / 5 + 32
    return fahrenheit
```

Готово! Вы только что создали модуль! Он содержит единственную функцию `c_to_f()`, выполняющую преобразование температуры из градусов Цельсия в градусы Фаренгейта.

Теперь воспользуемся модулем *my_module.py* в другой программе.

Применение модуля

Если вы хотите воспользоваться заключенной в модуле функциональностью, первым делом следует объяснить интерпретатору Python, какой модуль вам нужен. Для вклю-

чения в программу сторонних модулей в Python существует ключевое слово `import`. Вот как оно используется:

```
import my_module
```

Давайте напишем программу, в которой будет фигурировать только что созданный нами модуль. Мы собираемся выполнить преобразование температуры при помощи функции `c_to_f()`.

Вы уже умеете пользоваться функциями и передавать в них параметры (или аргументы). Но в данном случае функция находится в отдельном файле, не связанном с основной программой, поэтому нам потребуется ключевое слово `import`. Представленная в листинге 15.2 программа использует созданный нами модуль *my_module.py*.

Листинг 15.2. Применение модуля

```
import my_module

celsius = float(raw_input ("Введите температуру в градусах Цельсия: "))
fahrenheit = c_to_f(celsius)
print "Это ", fahrenheit, " градусов Фаренгейта"
```

Модуль my_module содержит функцию c_to_f()

Откройте новое окно IDLE-редактора и введите программу. Сохраните ее под именем *modular.py* и запустите, чтобы посмотреть, что получится. Программу следует сохранить в той же самой папке, в которой находится файл *my_module.py*.

Работает ли программа? Вы должны увидеть примерно такой результат:

```
>>> ========================= RESTART ========================
>>>
Введите температуру в градусах Цельсия: 34

Traceback (most recent call last):
    File "C:/MyPythonPrograms/modular.py", line 4, in <module>
        fahrenheit = c_to_f(celsius)
NameError: name 'c_to_f' is not defined
```

Программа не работает! Что же случилось? Сообщение об ошибке говорит, что функция `c_to_f()` не определена. Но мы знаем, что она определена в модуле `my_module`, причем этот модуль мы *импортировали*.

Дело в том, что следует дать интерпретатору Python больше информации о функциях, которые определены в сторонних модулях. Решить проблему можно, к примеру, изменив следующую строку:

```
fahrenheit = c_to_f(celsius)
```

Она должна выглядеть так:

```
fahrenheit = my_module.c_to_f(celsius)
```

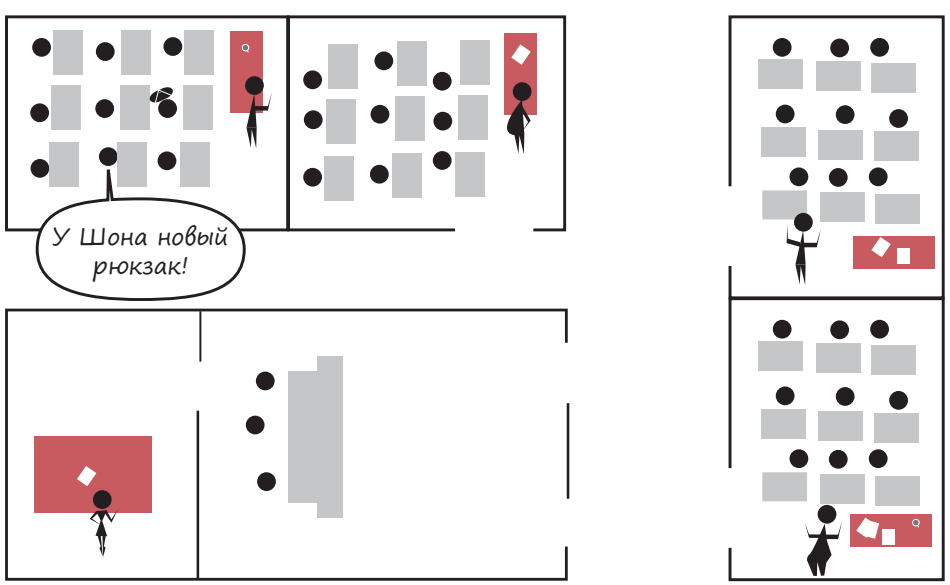
Здесь мы конкретно указали, что функция `c_to_f()` находится в модуле `my_module`. Измените таким образом свою программу и посмотрите, будет ли она работать.

Пространства имен

Картер говорит о вещах, относящихся к *пространствам имен* (namespaces). Это сложная тема, но без нее нельзя двигаться дальше, поэтому пришло время о ней поговорить.

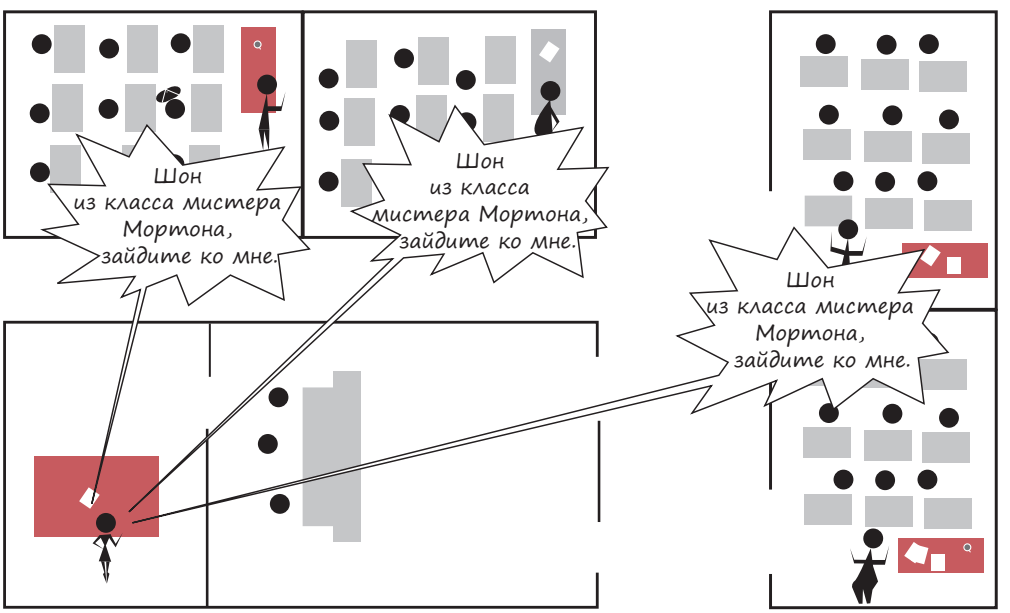
Что за пространство имен?

Представьте, что вы учитесь в классе мистера Мортона, и у вас есть одноклассник Шон. При этом в другом классе, которым руководит мистер Уилер, тоже есть ученик по имени Шон. Если у себя в классе вы скажете, что «у Шона новый рюкзак», все поймут (или, по крайней мере, предположат), о ком вы говорите. Если же вы имеете в виду другого Шона, придется уточнять «Шон из класса мистера Уилера» или «другой Шон».



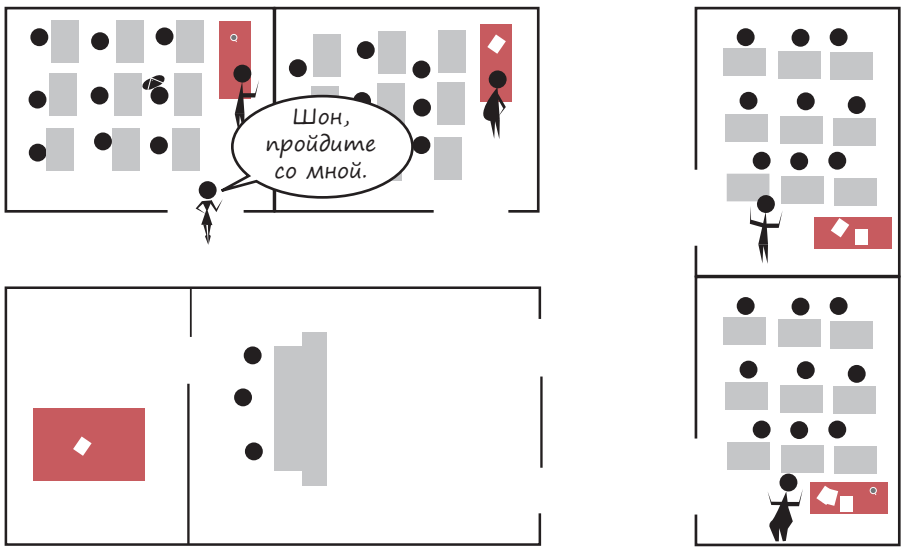
В вашем классе учится только один Шон, поэтому, когда вы называете его имя, одноклассники знают, о ком идет речь. Можно сказать, что в пространстве вашего класса существует всего одно имя «Шон». Ваш класс является вашим *пространством имен*, в котором существует всего один Шон, поэтому путаница исключена.

А теперь представим, что директор школы хочет вызывать Шона к себе по громкой связи. Если он скажет: «Шон, зайдите, пожалуйста, ко мне», у дверей его кабинета окажутся сразу оба Шона. Для директора, пользующегося широковещательной системой связи, пространством имен является вся школа. Это значит, что имя слышит не только один класс, а все, кто находится в школе. Поэтому следует более точно указывать, о каком Шоне идет речь. И директор говорит: «Шон из класса мистера Мортона, зайдите ко мне».



Нужного результата директор может достичь и другим способом. Достаточно зайти в класс и сказать: «Шон, пожалуйста, пройдите со мной». В этом случае его услышит только один Шон, именно тот, который нужен директору. В этом случае пространством имен будет всего один класс, а не вся школа.

В общем случае программисты называют небольшие пространства имен (такие, как ваш класс) *локальными*, а большие (такие, как вся школа) — *глобальными*.



ИМПОРТ ПРОСТРАНСТВ ИМЕН

Предположим, что в вашей школе имени Джона Янга нет ни одного ученика по имени Фред. Если директор по громкой связи попросит Фреда к нему зайти, никто не придет. Теперь предположим, что в соседней школе Стивена Лекока начался ремонт и один из классов на время перевели к вам. И в этом классе есть ученик по имени Фред. Но переведенный класс пока не подключен к громкой связи. Вызвав к себе Фреда, директор никого не дождется. Но после подключения широковещательной системы на этот вызов придет Фред из школы Стивена Лекока.

Присоединение широковещательной системы к фрагменту другой школы аналогично импорту модулей в Python. После этой операции вы получаете доступ ко всем именам модуля: всем переменным, всем функциям и всем объектам, имеющимся в модуле.

Импорт модуля означает то же самое, что и импорт пространства имен. Импортируя модуль, вы импортируете пространство имен.

Это можно делать двумя способами. Во-первых, так:

```
import StephenLeacock
```

В этом случае пространство имен StephenLeacock по-прежнему будет располагаться отдельно. У вас будет туда доступ, но перед тем как начать работу, вам придется в явном виде указать, какое именно пространство имен вам нужно. То есть директору потребуется выполнить вот такую операцию:

```
call_to_office(StephenLeacock.Fred)
```

То есть для вызова Фреда в свой кабинет ему по-прежнему нужно будет указывать как пространство имен (StephenLeacock), так и имя (Fred). Именно это мы делали несколько страниц назад в программе преобразования температуры. Чтоб заставить ее работать, мы написали:

```
fahrenheit = my_module.c_to_f(celsius)
```

Мы указали пространство имен (my_module) и только после этого имя функции (c_to_f).

Ключевое слово from

Во-вторых, пространство имен можно импортировать следующим образом:

```
from StephenLeacock import Fred
```

Если директор так поступит, имя Fred из пространства имен StephenLeacock станет частью его собственного пространства, и для вызова Фреда можно написать:

```
call_to_office(Fred)
```

Так как теперь Fred находится в пространстве имен директора, ему не нужно идти в пространство имен StephenLeacock для вызова ученика с именем Fred.

В этом примере директор импортировал в свое локальное пространство имен единственное имя Fred из пространства имен StephenLeacock. Для импорта всех остальных имен нужно выполнить вот такую операцию:

```
from StephenLeacock import *
```

Звездочка (*) в данном случае означает все имена. Но следует быть осторожным. Если в школе Стивена Лекока есть ученики с именами, совпадающими с именами учеников школы Джона Янга, начнется путаница.

Ох!

Возможно, пока концепция пространств имен кажется вам непонятной. Не волнуйтесь! Все станет ясно, когда в следующих главах мы перейдем к примерам. При каждом импорте модулей я буду объяснять вам, что именно происходит.

Стандартные модули

Итак, теперь вы умеете создавать и использовать модули. Означает ли это, что вам всегда придется писать их «с нуля»? Вовсе нет!

Интерпретатор Python поставляется с массой стандартных модулей, позволяющих в числе прочего выполнять такие операции, как поиск файлов, информирование о времени (или отсчет времени) и генерация случайных чисел. Иногда говорят, что Python обладает «встроенным арсеналом», подразумевая именно стандартные модули. Они известны как *стандартная Python-библиотека* (Python Standard Library).

Почему эти операции помещены в отдельные модули? На самом деле без этого можно было обойтись, но разработчики языка Python решили, что это повысит эффективность работы. В противном случае вам пришлось бы включать в каждую Python-программу все возможные функции. А так вы можете обойтись только теми, которые вам требуются.

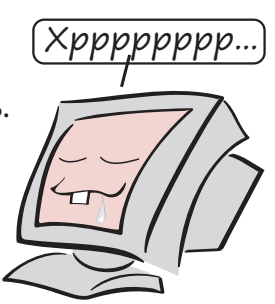
Разумеется, некоторые элементы (например инструкции `print`, `for` и `if-else`) относятся к базовым командам языка Python, и для них отдельные модули не требуются — они принадлежат основной части Python.

Если модуль с нужной вам функциональностью (например модуль, создающий графику в игре) отсутствует, можно загрузить дополнительные модули. Обычно эта операция является бесплатной! Мы добавили к книге несколько таких модулей, и если вы пользовались программой установки с нашего сайта, значит, они у вас есть. В противном случае ничто не мешает вам установить их самостоятельно.

Напоследок рассмотрим пару стандартных модулей.

time

Модуль `time` позволяет получать с системных часов вашего компьютера информацию о дате и времени. Кроме того, он дает возможность замедлить работу программы. (Иногда компьютер слишком быстро выполняет операции и его требуется притормозить.)



Задержку добавляет функция `sleep()` в модуле `time`. Фактически она заставляет программу остановиться и некоторое время ничего не делать. Программа как бы погружается в сон, именно поэтому функция называется `sleep()`. Длительность этого сна определяете вы.

Работу функции `sleep()` демонстрирует программа из листинга 15.3. Наберите ее, сохраните, запустите и посмотрите, что получится.

Листинг 15.3. Погружение программы в сон

```
import time
print "Как",
time.sleep(2)
print "твои",
time.sleep(2)
print "дела",
time.sleep(2)
print "сегодня?"
```

Обратите внимание на наличие префикса `time.` при вызове функции `sleep()`. Дело в том, что хотя мы и импортировали модуль `time`, частью пространства имен основной программы он не стал. Значит, каждый раз, чтобы воспользоваться функцией `sleep()`, нужно писать `time.sleep()`. А вот такой код работать не будет:

```
import time
sleep(5)
```

Причина в том, что функция `sleep()` в нашем пространстве имен отсутствует. Мы получим сообщение об ошибке:

```
NameError: name 'sleep' is not defined
```

Чтобы решить проблему, требуется импортировать модуль вот таким образом:

```
from time import sleep
```

В результате интерпретатор Python как бы получает сообщение «Найди переменную (или функцию, или объект) с именем `sleep` в модуле `time` и включи ее в мое пространство имен». После этого вы будете избавлены от необходимости добавлять перед именем функции `sleep()` префикс `time.` при работе с ней:

```
from time import sleep
print 'Привет, поговорим через 5 секунд...'
sleep(5)
print 'Привет еще раз'
```

Если вы хотите воспользоваться преимуществами импорта имен в локальное пространство имен (чтобы избавиться от необходимости каждый раз указывать имя модуля), но не знаете, какие имена в модуле вам нужны, воспользуйтесь звездочкой (*) и импортируйте все доступные имена:

```
from time import *
```

Однако с этой операцией следует быть крайне аккуратным. Если в вашей программе окажется такое же имя, как и в модуле `time`, возникнет конфликт версий. Поэтому лучше импортировать только то, что вам действительно нужно.

Помните программу обратного отсчета, которую мы писали в главе 8 (см. листинг 8.6)? Теперь вы знаете, зачем была нужна строка `time.sleep(1)`.

Случайные числа

Модуль `random` позволяет генерировать случайные числа. Это часто требуется в играх и симуляторах. Попробуем воспользоваться модулем `random` в интерактивном режиме:

```
>>> import random
>>> print random.randint(0, 100)
4
>>> print random.randint(0, 100)
72
```

Каждый раз при вызове функции `random.randint()` вы получаете новое случайное целое число. Так как в функцию были переданы аргументы 0 и 100, это число будет лежать в диапазоне от 0 до 100. Функцию `random.randint()` мы использовали в главе 1 для создания секретного числа в игре.

Для получения случайного десятичного числа пользуйтесь функцией `random.random()`. В этом случае в скобках не нужно ничего указывать, так как эта функция всегда возвращает число в диапазоне от 0 до 1:

```
>>> print random.random()
0.270985467261
>>> print random.random()
0.569236541309
```

Для получения случайного числа в диапазоне от 0 до 10 достаточно умножить результат на 10:

```
>>> print random.random() * 10
3.61204895736
>>> print random.random() * 10
8.10985427783
```

Что мы узнали

В этой главе мы получили информацию о том:

- что такое модуль;
- как создать модуль;

- как воспользоваться модулем в другой программе;
- что такое пространства имен;
- что подразумевается под локальными и глобальными пространствами имен и переменными;
- как импортировать имена из других модулей в ваше пространство имен.

Кроме того, мы познакомились с примерами стандартных Python-модулей.

Проверь себя

1. Каковы преимущества применения модулей?
2. Как создать модуль?
3. Какое ключевое слово позволяет воспользоваться Python-модулем?
4. Импортировать модуль — это то же самое, что импортировать _____.
5. Какими двумя способами можно импортировать модуль `time`, чтобы получить доступ ко всем его именам (то есть переменным, функциям и объектам)?

Эксперименты

1. Напишите модуль, содержащий функцию вывода вашего имени большими буквами, из раздела «Эксперименты» главы 13. Затем напишите программу, которая импортирует этот модуль и вызывает функцию.
2. Отредактируйте код листинга 15.2 таким образом, чтобы функция `c_to_f()` оказалась в пространстве имен основной программы. То есть измените ее так, чтобы появилась возможность написать:

```
fahrenheit = c_to_f(celsius)
```

   И подставить эту строку вместо строки:

```
fahrenheit = my_module.c_to_f(celsius)
```

3. Напишите короткую программу, генерирующую список из пяти случайных целых чисел от 1 до 20 и выведите их значения.
4. Напишите короткую программу, которая в течение 30 секунд будет каждые 3 секунды выводить значение случайного десятичного числа.

ГЛАВА 16

Графика

Вы уже много узнали о базовых элементах программирования: вводе и выводе данных, переменных, решениях, циклах, списках, функциях, объектах и модулях. Я надеюсь, что в процессе изучения всех этих вещей вы получали удовольствие! А теперь пришло время получения еще большего удовольствия от программирования и языка Python.

В этой главе вы научитесь рисовать на экране линии и формы, делать их цветными и даже анимировать. В дальнейшем это поможет нам при создании игр и других программ.

Наш помощник — модуль Pygame

Создавать на своем компьютере графику (и добавлять к ней звук) не так-то просто. Вам придется иметь дело с операционной системой, видеокартой и большим количеством низкоуровневого кода, о котором нам пока не хочется даже говорить. Поэтому для простоты решения мы воспользуемся Python-модулем, который называется Pygame.

Модуль Pygame позволяет создавать графику и другие вещи, необходимые для запуска игры на различных компьютерах и в разных операционных системах. При этом вам не нужно вникать в детали компьютерной архитектуры. Модуль Pygame является бесплатным, причем одна из его версий поставляется вместе с этой книгой. Если вы пользовались нашей программой установки, то этот модуль у вас уже есть. В противном случае ничто не мешает загрузить его с сайта ***www.pygame.org***.

Окно модуля Pygame

Первым делом нам нужно окно, в котором будет нарисована вся графика. Вот очень простая программа, которая нам его создаст (листинг 16.1).

Листинг 16.1. Создание Pygame-окна

```
import pygame
pygame.init()
screen = pygame.display.set_mode([640, 480])
```

Запустите эту программу. Что вы увидите? В зависимости от вашей операционной системы это может быть окно (с черным фоном), ненадолго появившееся на экране. Или окно, которое невозможно закрыть. Что же происходит?

Дело в том, что модуль Pygame предназначен для создания игр. А игры не работают сами по себе, им требуется взаимодействие с игроком. Поэтому в программах с модулем Pygame присутствует так называемый *цикл событий* (event loop), постоянно проверяющий, выполняет ли пользователь какие-либо действия, такие как движение мыши, нажатие клавиш или закрытие окна. В Pygame-программах цикл событий должен функционировать непрерывно. В нашей же первой программе этот цикл вообще не был запущен, поэтому она начала работать некорректно.

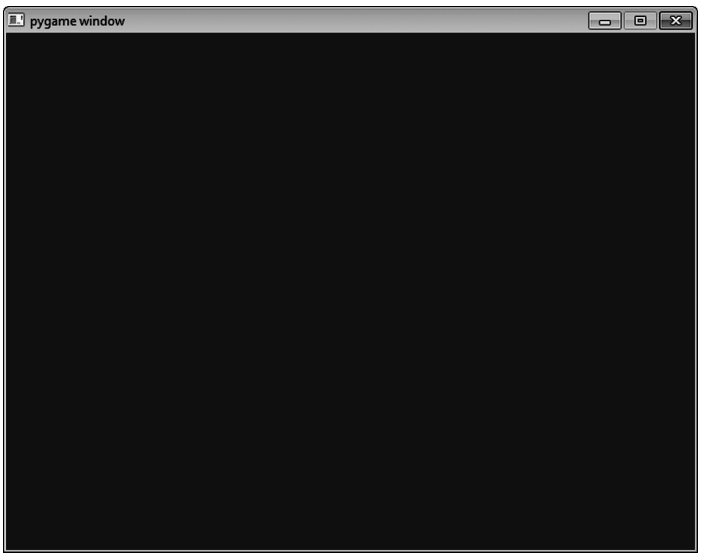
Реализуем цикл событий с помощью инструкции `while`. Он должен действовать все время, пока пользователь играет в игру. Так как в Pygame-программах меню, как правило, отсутствует, для завершения работы следует щелкнуть по кнопке закрытия в правом верхнем углу окна (в Windows) или в левом верхнем углу окна (в Mac OS). В операционных системах семейства Linux способ закрытия окна зависит от того, какой диспетчер окон и GUI-интерфейс вы используете. Впрочем, если вы работаете в Linux, скорее всего, вы уже знаете, что нужно сделать, чтобы закрыть окно.

Код из листинга 16.2 открывает и удерживает на экране Pygame-окно, пока пользователь не захочет его закрыть.

Листинг 16.2. Корректно работающее Pygame-окно

```
import pygame
pygame.init()
screen = pygame.display.set_mode([640, 480])
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
running = False
pygame.quit()
```

Запустите эту программу, и вы получите корректно работающее Pygame-окно, исчезающее, когда вы его закрываете.



Как именно работает код внутри этого цикла `while`? Там используется *цикл событий* Pygame-модуля. Подробно он будет рассмотрен в главе 18, когда речь пойдет о Pygame-событиях.

Рисование в окне

Итак, у нас есть Pygame-окно, которое остается открытым, пока мы не захотим его закрыть, после чего оно аккуратно закроется. Цифры [640, 480] в третьей строке листинга 16.2 задают размер нашего окна: 640 пикселов в ширину и 480 в высоту. Давайте в нем что-нибудь нарисуем. Измените вашу программу, чтобы она выглядела так, как показано в листинге 16.3.

Листинг 16.3. Рисование круга

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255,255,255])
pygame.draw.circle(screen, [255,0,0],[100,100], 30, 0)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
running = False
pygame.quit()
```

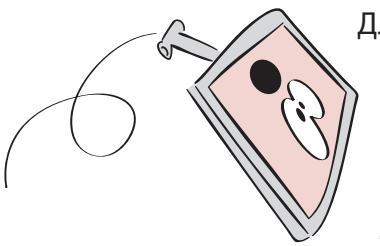
Заливаем окно белым цветом

Добавляем эти три строки

Опрокидываем ваш монитор... Шучу!

Рисуем круг

Что такое flip

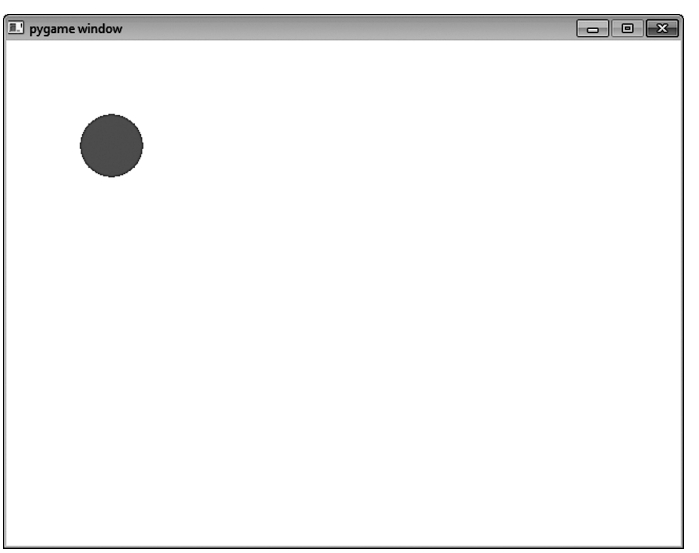


Для вывода объекта в Pygame-окне (у нас это происходит на поверхности `screen`, созданной в строке 3 листинга 16.3) строятся две его копии. Дело в том, что разрабатываемая анимация должна быть максимально быстрой и максимально гладкой. Поэтому вместо полного обновления экрана при внесении даже небольших изменений можно сразу выполнять все изменения, а потом быстро «переходить» на новую версию графического фрагмента. Именно это делает команда `pygame.display.flip()`. В результате изменения появляются не постепенно, а сразу, что позволяет избежать возникновения на экране кругов, нарисованных лишь наполовину (или инопланетян, или других объектов).

Две копии можно представить как текущий экран и «следующий» экран. Текущий экран — это то, что вы видите сейчас. «Следующий» экран вы увидите после перехода. Все изменения в рисунок вносятся на «следующем» экране, и вы видите их уже после перехода.

Как нарисовать круг

После запуска программы из листинга 16.3 в левом верхнем углу окна появится красный круг:



И это неудивительно, ведь функция `pygame.draw.circle()` рисует круги. Для этого вы должны сообщить ей следующие сведения:

- на какой *поверхности* следует рисовать круг (в данном случае это поверхность, определенная в строке 3 и носящая имя `screen`);
- какого *цвета* должен быть круг. (В данном случае это красный цвет, представленный как `[255, 0, 0]`);

- в каком *месте* его нужно рисовать (в данном случае это `[100, 100]`, что означает 100 пикселов вниз и 100 пикселов в сторону от левого верхнего угла);
- какого *размера* должен быть этот круг (в данном случае его радиус — расстояние от центра круга до внешней границы — равен 30 пикселам);
- *толщина линии* (если `width = 0`, круг полностью залит цветом, как в данном случае).

Теперь рассмотрим все эти параметры более подробно.

НОВЫЕ СЛОВА

Слово пиксел является сокращением от английского «picture element» — элемент изображения. Оно означает точку на экране или на картине. Если открыть любое изображение в программе для просмотра и сильно увеличить, вы увидите отдельные пикселы. Рисунок демонстрирует обычное фото и его сильно увеличенную версию, на которой видны пикселы.

Ого, у тебя отличное зрение! Маленькие линии на самом деле являются рядами пикселов. Экран обычного компьютера может состоять из 768 рядов пикселов, в каждом из которых 1024 пиксела. Мы говорим, что разрешение экрана составляет 1024 × 768. Некоторые мониторы содержат большее количество пикселов, некоторые меньшее.

ПОВЕРХНОСТИ В МОДУЛЕ PYGAME

Если в реальной жизни вас попросят нарисовать картину, первым делом вы спросите: «На чем ее рисовать?» В модуле Pygame *поверхность* (surface) — это то, на чем вы рисуете. *Поверхность отображения* (display surface) — то, что вы видите на экране. Именно она в листинге 16.3 получила имя `screen`. Но в Pygame-программе может быть множество поверхностей, причем существует возможность копировать изображения с одной поверхности на другую. Более того, сами поверхности можно поворачивать и масштабировать (делать больше или меньше).

Как я уже упоминал, существуют две копии поверхности отображения. Программисты говорят, что поверхность отображения имеет *двойную буферизацию* (double-buffered). Именно благодаря этому на экране не появляются наполовину нарисованные фигуры

и изображения. Круги, инопланетяне и другие объекты рисуются в буфере, на который затем «переходит» поверхность отображения, образуя полностью готовые рисунки.

Цвета в модуле pygame

В модуле Pygame используется распространенная цветовая модель, применяющаяся во многих других языках программирования и программах. Она называется RGB. Буквы R, G и B означают красный (red), зеленый (green) и синий (blue) цвета.

На уроках естествознания вы, наверное, проходили, что любой цвет получается смешением света трех *основных цветов*: красного, зеленого и синего. Аналогичным образом обстоят дела в программировании. Каждый цвет — красный, зеленый и синий — обозначается числом от 0 до 255. Цвета представляются в виде списка целых чисел, лежащих в диапазоне от 0 до 255. Если все числа равны 0, цвет отсутствует, что соответствует полной темноте, или черному цвету. Если все числа равны 255, вы получаете смесь самых ярких оттенков каждого цвета, что в результате дает белый цвет. Запись [255, 0, 0] означает чисто красный цвет без добавления зеленого и синего. Зеленому цвету соответствует запись [0, 255, 0], а синему — [0, 0, 255]. Наличие в списке трех одинаковых чисел, например [150, 150, 150], даст вам один из оттенков серого. Чем меньше числа, тем темнее будет этот оттенок.

Названия цветов

В модуль Pygame встроен список именованных цветов, которым можно пользоваться, если вам не нравится представление [R, G, B]. Список содержит свыше 600 названий цветов. Тут я их перечислять не буду, но если вы хотите с ними ознакомиться, найдите на жестком диске своего компьютера файл ***colordict.py*** и откройте его в текстовом редакторе.

Если вы предпочитаете пользоваться названиями цветов, в начало программы следует вставить вот такую строку:

```
from pygame.color import THECOLORS
```

Вот как выглядит процесс применения именованных цветов (из нашего примера с кругом):

```
pygame.draw.circle(screen, THECOLORS["red"],[100,100], 30, 0)
```

Если вы хотите немного поиграть и увидеть своими глазами, каким образом смешиваются красный, зеленый и синий цвета, запустите программу ***colormixer.py***, которую наша

программа установки поместила в папку *\examples*. Она даст вам возможность комбинировать основные цвета в любой пропорции и наблюдать, какой цвет получается в итоге.

Что происходит внутри

Почему 255? Диапазон от 0 до 255 дает нам 256 значений для каждого из основных цветов (красного, зеленого и синего). Что в этом числе такого особенного? Почему не 200, не 300 или не 500?

Две сотни и пятьдесят шесть — это количество цветов, которые можно получить из 8 разрядов. Это число всех возможных сочетаний восьми единиц и нулей. Восемь разрядов, или битов, называют байтом (byte), а байт — это минимальный фрагмент памяти, обладающий собственным адресом. Адрес указывает компьютеру, где именно в памяти располагаются определенные фрагменты данных.

Все это напоминает улицу с домами. Ваш дом и ваша квартира имеют адреса, а вот у вашей комнаты его уже нет. Квартира в этом случае считается минимальной «адресуемой единицей» в доме. А байт является минимальной «адресуемой единицей» в памяти компьютера.

Для каждого цвета можно было бы задействовать и больше чем 8 битов, но так как частью байта пользоваться не очень удобно, речь шла бы уже о 16 битах (2 байтах). Однако если учесть способность человеческого глаза различать цвета, оказывается, что для создания реалистичных цветов 8 битов хватит.

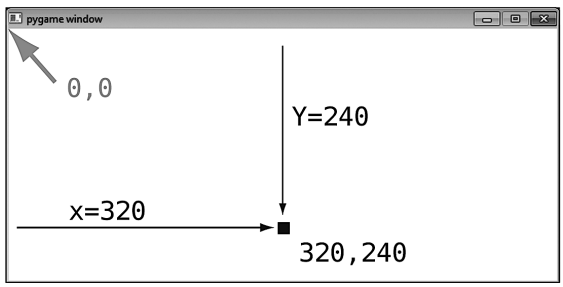
В итоге у нас есть три значения (красный, зеленый и синий), каждое по 8 битов, всего 24 бита. Поэтому данное представление называют «24-битным цветом». В нем используется 24 бита для каждого пиксела, по 8 для каждого основного цвета.

Местоположение — экранные координаты

При рисовании на поверхности некоего объекта следует указать, в какой именно точке он должен располагаться. Местоположение обозначается двумя числами: одно для оси *x* (направление по горизонтали), а второе для оси *y* (направление по вертикали). В модуле Pygame отсчет начинается от точки [0, 0], расположенной в левом верхнем углу окна.

В паре чисел [320, 240] первое указывает положение по горизонтали, отсчитываемое от левой стороны окна. Второе число определяет положение по вертикали и отсчитывается от верхней границы окна. В математике и в программировании для обозначения горизонтальной координаты часто используется буква *x*, а вертикальная координата обозначается буквой *y*.

Мы создали окно шириной 640 пикселов и высотой 480 пикселов. Чтобы круг оказался в центре этого окна, его нужно рисовать в точке с координатами [320, 240]. Это отступ в 320 пикселов от левой стороны окна и в 240 пикселов сверху.



Попробуем нарисовать круг в центре окна. Код программы для этого случая представлен в листинге 16.4.

Листинг 16.4. Размещение круга в центре окна

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
pygame.draw.circle(screen, [255,0,0],[320,240], 30, 0)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
running = False
pygame.quit()
```

Меняем этот параметр с [100, 100] на [320, 240]

Центром круга станет точка с координатами [320, 240]. Сравните результат работы программы из листинга 16.3 и этой программы и оцените разницу.

Размер фигур

При работе с функциями `draw` модуля Pygame следует указывать желаемый размер фигуры. В случае круга у нас будет всего один параметр — радиус. Если же мы хотим получить прямоугольник, потребуется указать его длину и ширину.

В модуле Pygame есть специальный вид объектов `rect` (сокращенное от слова *rectangle* — прямоугольник), позволяющий определять прямоугольные области. При этом указываются координаты левого верхнего угла, ширина и высота:

```
Rect(left, top, width, height)
```

Таким способом одновременно задаются расположение и размер. Например:

```
my_rect = Rect(250, 150, 300, 200)
```

Эта строка кода создает прямоугольник, левый верхний угол которого отстоит от левой стороны окна на 250 пикселов, а от его верхнего края — на 150 пикселов. Ширина прямоугольника составляет 300 пикселов, а высота — 200 пикселов. Попробуйте нарисовать такой прямоугольник и убедитесь сами.

Вставьте эту строку вместо строки 5 в листинг 16.4 и посмотрите, что получится:

```
pygame.draw.rect(screen, [255,0,0], [250, 150, 300, 200], 0)
```

Цвет прямоугольника — [255,0,0]

Местоположение и размер прямоугольника — [250, 150, 300, 200]

Ширина линии (или заливка) — 0

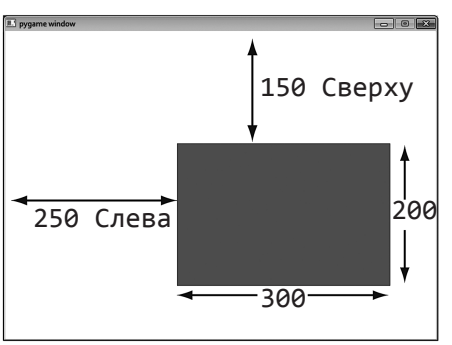
Расположение и размер прямоугольника можно представить как обычный список (или кортеж) чисел или как объект `Rect` модуля Pygame. Поэтому предыдущую строку можно заменить на вот такие две строки:

```
my_list = [250, 150, 300, 200]
pygame.draw.rect(screen, [255,0,0], my_list, 0)
```

Или так:

```
my_rect = pygame.Rect(250, 150, 300, 200)
pygame.draw.rect(screen, [255,0,0], my_rect, 0)
```

Вот как будет выглядеть полученный нами прямоугольник. Я добавил размеры, чтобы стало понятно, что обозначают конкретные числа:



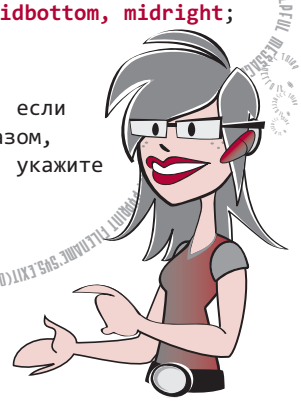
Обратите внимание, что в функцию `pygame.draw.rect` передают только четыре аргумента. Дело в том, что объект `rect` объединяет положение и размер в один аргумент. У функции `pygame.draw.circle` положение и размер обозначаются двумя разными параметрами, поэтом в нее мы передаем пять аргументов.

ДУМАЙ КАК ПРОГРАММИСТ

При создании прямоугольника с помощью функции **Rect(left, top, width, height)** можно пользоваться и другими атрибутами, отвечающими за перемещение и выравнивание нашего объекта **Rect**:

- четыре ребра: **top, left, bottom, right**;
- четыре угла: **topleft, bottomleft, topright, bottomright**;
- середина каждой стороны: **midtop, midleft, midbottom, midright**;
- центр: **center, centerx, centery**;
- размеры: **size, width, height**.

Они добавлены просто для удобства. К примеру, если вам нужно передвинуть прямоугольник таким образом, чтобы его центр оказался в определенной точке, укажите координаты центра, чтобы не высчитывать новое положение левого верхнего угла.



ТОЛЩИНА ЛИНИИ

Последний параметр, который нужно указывать при рисовании фигур, представляет собой толщину линии. В рассмотренных ранее примерах этот параметр был равен нулю, что обеспечивало равномерную заливку круга красным цветом. При увеличении этого параметра создается контур фигуры.

Попробуйте изменить толщину линии на 2:

Делаем его равным 2

```
pygame.draw.rect(screen, [255,0,0], [250, 150, 300, 200], 2)
```

Запустите новую версию кода и посмотрите, что получится. Проверьте работу программы с другими значениями этого параметра.

СОВРЕМЕННОЕ ИСКУССТВО?

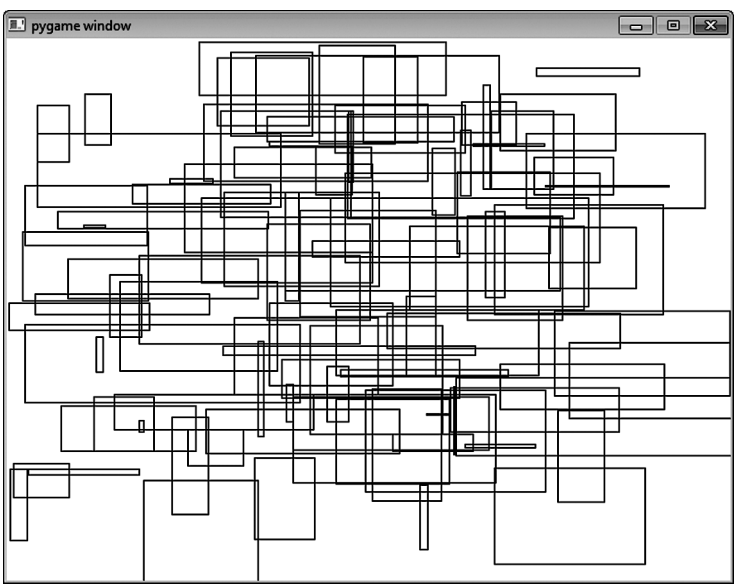
Хотите получить сгенерированную компьютером картину? Просто для удовольствия запустите код из листинга 16.5. Можно взять листинг 16.4 и отредактировать его или же набрать код с самого начала.

Листинг 16.5. Рисование картины с помощью функции draw.rect

```
import pygame, sys, random
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
for i in range (100):
```

```
    width = random.randint(0, 250)
    height = random.randint(0, 100)
    top = random.randint(0, 400)
    left = random.randint(0, 500)
pygame.draw.rect(screen, [0,0,0], [left, top, width, height], 1)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Запустите код и посмотрите, что получится. Результат должен выглядеть примерно так:



Понимаете, как работает эта программа? Она рисует 100 прямоугольников произвольного размера и в произвольных местах. Чтобы сделать картину еще более «художественной», добавьте цвет и сделайте произвольной толщину линий, как в листинге 16.6.

Листинг 16.6. Современное искусство в цвете

```
import pygame, sys, random
from pygame.color import THECOLORS
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
```

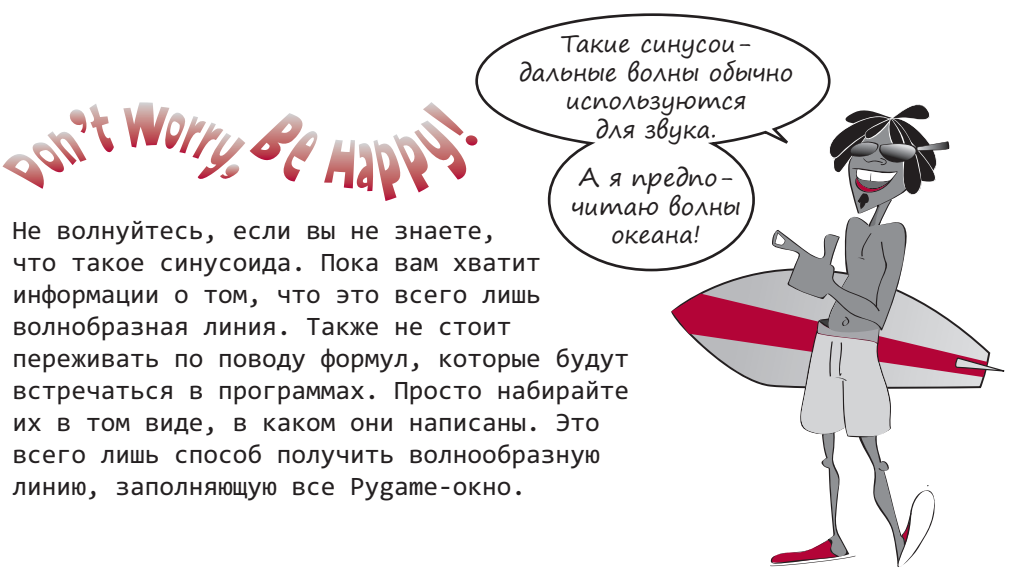
```
for i in range (100):
    width = random.randint(0, 250); height = random.randint(0, 100)
    top = random.randint(0, 400); left = random.randint(0, 500)
    color_name = random.choice(THECOLORS.keys())
    color = THECOLORS[color_name]
    line_width = random.randint(1, 3)
    pygame.draw.rect(screen, color, [left, top, width, height], line_width)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

О назначении этой строки пока не думайте

При каждом запуске эта программа будет давать вам другой результат. Если вы вдруг получите действительно красивую картинку, дайте ей звучное название, например «Глас Машины», и попробуйте продать в художественную галерею!

Отдельные пикселы

Иногда нарисовать требуется не круг или прямоугольник, а отдельные точки, или пикселы. Допустим, вы пишете математическую программу и хотите вывести на экран синусоиду.



Не волнуйтесь, если вы не знаете, что такое синусоида. Пока вам хватит информации о том, что это всего лишь волнобразная линия. Также не стоит переживать по поводу формул, которые будут встречаться в программах. Просто набирайте их в том виде, в каком они написаны. Это всего лишь способ получить волнообразную линию, заполняющую все Pygame-окно.

Так как метода `pygame.draw.sinewave()` не существует, синусоиду придется рисовать самостоятельно из отдельных точек. Нужный результат получается, к примеру, при

рисовании маленьких кругов или прямоугольников размером в один или два пиксела. Листинг 16.7 демонстрирует это с помощью прямоугольников.

Листинг 16.7. Рисование кривых из множества маленьких прямоугольников

```
import pygame, sys
import math
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
for x in range(0, 640):
    y = int(math.sin(x/640.0 * 4 * math.pi) * 200 + 240)
    pygame.draw.rect(screen, [0,0,0],[x, y, 1, 1], 1)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

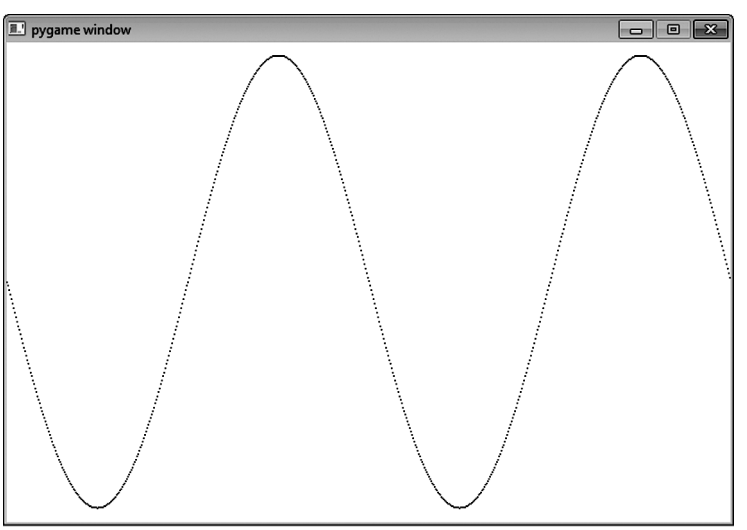
Импортируем математические функции, в том числе sin()

Циклически смещаемся слева направо от x = 0 до 639

Вычисляем координату y каждой точки

Рисуем точку в виде маленького прямоугольника

Вот что получается при запуске такой программы:



Для изображения каждой точки мы использовали прямоугольник шириной в 1 пиксел и высотой в 1 пиксел. Обратите внимание, что толщина в данном случае равна 1, а не 0. При нулевой толщине линии на экране ничего не появится, так как это привело бы к отсутствию области, которую можно было бы залить цветом.

Соединяем точки

Присмотревшись, можно обнаружить, что наша синусоида не непрерывна, так как отдельные точки не соединены между собой. Это связано с ее крутизной — ведь сдвигаясь на один пиксел вправо, мы одновременно должны сдвинуться на три пиксела вверх (или вниз). А так как мы рисуем не линию, а отдельные точки, расстояние между ними оказывается ничем не заполненным.

Соединим точки, нарисовав между ними линию. В модуле Pygame есть не только метод, позволяющий рисовать линию сразу, но и метод, рисующий линии между группами точек. Он называется `pygame.draw.lines()` и содержит пять параметров:

- поверхность `surface`, на которой вы будете рисовать;
- цвет линии `color`;
- параметр замкнутости `closed` замкнет линию, соединив последнюю точку с первой. В нашем случае это не требуется, поэтому он будет иметь значение `False`;
- список соединяемых точек `list`;
- толщина линии `width`.

В нашем примере с синусоидой метод `pygame.draw.lines()` будет выглядеть так:

```
pygame.draw.lines(screen, [0,0,0],False, plotPoints, 1)
```

Теперь вместо рисования каждой точки внутри цикла `for` мы создадим список точек, которые соединим методом `draw.lines()`. Остается один раз вызвать метод `draw.lines()` за пределами цикла `for`. Программа целиком представлена в листинге 16.8.

Листинг 16.8. Хорошо соединенная синусоида

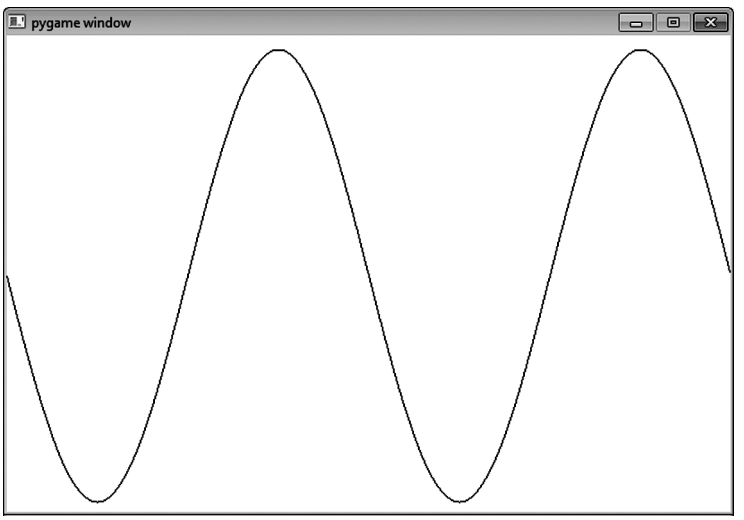
```
import pygame, sys
import math
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
plotPoints = []
for x in range(0, 640):
    y = int(math.sin(x/640.0 * 4 * math.pi) * 200 + 240)
    plotPoints.append([x, y])
pygame.draw.lines(screen, [0,0,0],False, plotPoints, 1)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Вычисляем координату y каждой точки

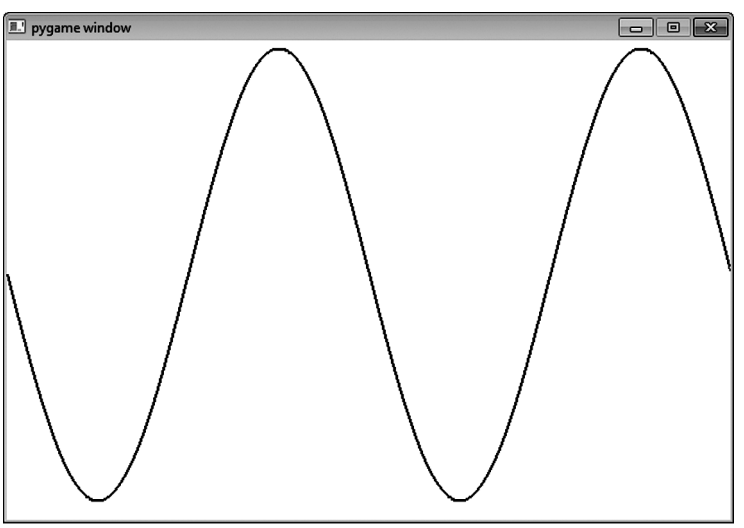
Добавляем каждую точку в список

Рисуем кривую целиком с помощью функции draw.lines()

После запуска программы мы получим вот такую картинку:



Как видите, теперь зазоры между точками отсутствуют. Кривая будет выглядеть еще лучше, если мы увеличим толщину линии до 2:



Еще раз соединяем точки

Помните, в детстве вы решали головоломки, в которых требовалось соединять точки? Вот версия такой головоломки от Pygame.

Программа из листинга 16.9 при помощи функции `draw.lines()` и списка точек создает форму. Чтобы увидеть, какая картинка там скрыта, наберите программу. На этот раз вы не сможете облегчить себе жизнь! В папке *\examples* нужного файла нет — если вы хотите увидеть таинственную картинку, вам придется набрать код вручную. Впрочем,

ввод всех этих цифр — крайне утомительная процедура, поэтому список `dots` можно взять из папки *\examples* или с сайта книги.

Листинг 16.9. Тайная картинка, получаемая соединением точек

```
import pygame, sys
pygame.init()

dots = [[221, 432], [225, 331], [133, 342], [141, 310],
        [51, 230], [74, 217], [58, 153], [114, 164],
        [123, 135], [176, 190], [159, 77], [193, 93],
        [230, 28], [267, 93], [301, 77], [284, 190],
        [327, 135], [336, 164], [402, 153], [386, 217],
        [409, 230], [319, 310], [327, 342], [233, 331],
        [237, 432]]

screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
pygame.draw.lines(screen, [255,0,0],True, dots, 2)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

На этот раз параметр closed = True

Рисование по точкам

Ненадолго вернемся к рисованию по точкам. Вряд ли имеет смысл рисовать небольшой круг или прямоугольник, когда требуется поменять цвет одного пиксела. Поэтому в таких случаях следует пользоваться уже не функциями рисования, а методом `Surface.set_at()`, дающим доступ к отдельным пикселам поверхности. Достаточно указать нужный цвет и пиксел, который вы хотите раскрасить:

```
screen.set_at([x, y], [0, 0, 0])
```

Если применить эту строку в примере с синусоидой (поставив ее вместо строки 8 в листинге 16.7), вы получите такой же результат, как при использовании прямоугольников шириной в один пиксел. Метод `Surface.get_at()` позволяет проверить текущий цвет пиксела. Достаточно вот таким способом передать координаты:

```
pixel_color = screen.get_at([320, 240])
```

В данном примере `screen` — это имя поверхности.

Изображения

Рисование фигур, линий и отдельных пикселов является лишь одним из вариантов создания графики. Но иногда возникает необходимость воспользоваться картинкой, взятой в другом месте. Это может быть цифровое фото, графический фрагмент, загруженный из Интернета или созданный вами в графическом редакторе. Простейшим способом работы с такими картинками в модуле Pygame являются функции `image`.

Рассмотрим пример: выведем на экран картинку, которая, если вы пользовались прилагаемой к книге программой установки, должна быть на жестком диске вашего компьютера. Программа создала папку *images*, вложенную в папку *\examples*. Нам понадобится находящийся в ней файл *beach_ball.png*. Соответственно, в операционной системе Windows вы найдете его по адресу *c:\Program Files\helloworld\examples\images\beach_ball.png*.

Если вы не пользовались нашей программой установки, загрузить файл *beach_ball.png* можно с сайта книги *www.manning.com/books/hello-world-second-edition*.

Файл *beach_ball.png* следует скопировать в папку, в которой вы сохраняете примеры программ в процессе чтения книги. В этом случае интерпретатор Python после запуска программы без труда его обнаружит. Скопировав файл *beach_ball.png* в корректную папку, наберите программу из листинга 16.10 и запустите ее.

Листинг 16.10. Изображение пляжного мяча в Pygame-окне

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load("beach_ball.png")
screen.blit(my_ball, [50, 50])
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Это единственные пока незнакомые вам строки

После запуска программы в левом верхнем углу Pygame-окна вы увидите изображение пляжного мяча:

В листинге 16.10 вам пока незнакомы строки 5 и 6. Все остальное вы уже видели в листингах 16.3–16.9. Код функции `draw` из предыдущих примеров мы заменили кодом, который загружает изображение с диска и выводит его на экран.

В строке 5 функция `pygame.image.load()` загружает изображение с диска и создает объект `my_ball`. Этот объект представляет собой поверхность. (О поверхностях речь шла несколько страниц назад.) Но нам она пока не видна, так как существует только в памяти. Единственной видимой нам поверхностью является поверхность отображения `screen`. (Ее мы создали в строке 3.) Строка 6 копирует поверхность `my_ball` на поверхность `screen`. Затем функция `display.flip()` уже знакомым нам способом делает ее видимой.

Не волнуйся, Картер. Скоро он начнет двигаться! В строке 6 листинга 16.10 вы могли заметить необычную вещь: `screen.blit()`. Что означает слово `blit`?

НОВЫЕ СЛОВА

В графических программах часто приходится копировать пикселы с места на место (например копировать из переменной на поверхность или с одной поверхности на другую). В программировании эта операция копирования битового массива носит специальное название *блитирования* (blitting). Мы *блитируем* (blit) изображение (или часть изображения, или просто группу пикселов) с одного места на другое. Фактически это всего лишь еще один из вариантов обозначения процедуры копирования. Просто слово «блитировать» сразу дает понять, что копируем мы пикселы, а не что-либо другое.

В модуле Pygame пикселы копируются с одной *поверхности* на другую. В рассматриваемом примере мы скопировали пикселы с поверхности `my_ball` на поверхность `screen`.

В строке 6 листинга 16.10 изображение пляжного мяча было скопировано в координату (50, 50). Это означает, что в результате мяч сдвинулся на 50 пикселов от левого края и на 50 пикселов от верхней границы окна. В случае поверхности `surface` или прямоугольника `rect` это соответствует положению верхнего левого угла картинки. То есть левый край нашего мяча на 50 пикселов отстоит от левой границы окна, а его верхняя точка сдвинута на 50 пикселов вниз от верхнего края.

Заставь его двигаться!

Теперь, когда графический фрагмент уже находится в Pygame-окне, пришло время привести его в движение. Да, мы действительно собираемся создать анимацию! Компьютерная анимация является всего лишь перемещением изображений (групп пикселов) с одного места на другое. Заставим наш мяч двигаться.

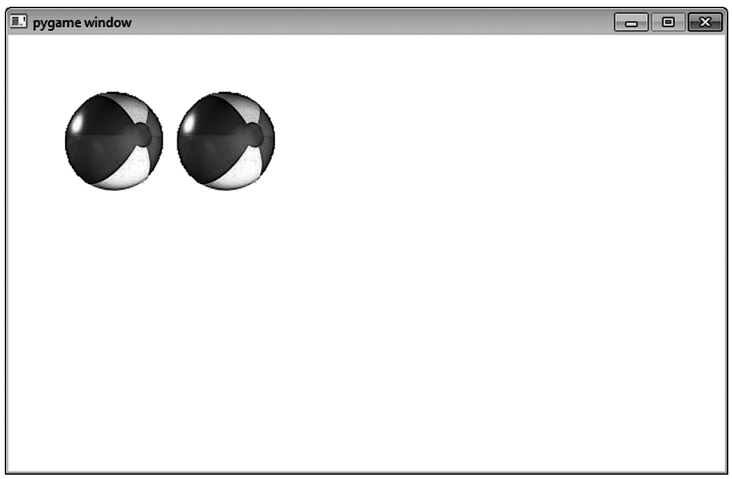
Для этого нам нужно поменять его местоположение. Для начала сдвинем его в сторону. Чтобы убедиться, что мы видим движение, переместим мяч на 100 пикселов вправо. За движение влево-вправо (по горизонтали) отвечает первое число в паре координат, задающих положение объекта. Поэтому для смещения вправо на 100 пикселов нам нужно увеличить первую координату на 100. Выполним эти процедуры с задержкой, чтобы убедиться, что анимация действительно имеет место.

Отредактируйте программу из листинга 16.10 в соответствии с листингом 16.11. (Нужно добавить строки 8, 9 и 10 перед циклом `while`.)

Листинг 16.11. Пытаемся двигать мяч

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
screen.blit(my_ball,[50, 50])
pygame.display.flip()
pygame.time.delay(2000)                    |
screen.blit(my_ball,[150, 50])             |  Это три новых строки
pygame.display.flip()                      |
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Запустите программу и посмотрите, что получилось. Сдвинулся ли мяч? В принципе да. Вы должны увидеть два мяча:



Первый располагается в исходной точке, а второй через пару секунд появляется чуть правее. Да, мы сдвинули мяч вправо, но забыли одну вещь. Мы забыли удалить первый мяч!

Анимация

В компьютерной графике анимация реализуется в два этапа.

1. Вы рисуете объект на новом месте.
2. Вы удаляете объект со старого места.

Первую часть вы уже видели. Мы нарисовали мяч в новом месте. Теперь нужно удалить исходное изображение. Но что в данном случае подразумевается под «удалением»?

Удаление изображений

Рисунок, нарисованный на бумаге или на школьной доске, стереть легко — возьмите ластик или тряпку. А как быть с картиной? Представьте, что вы нарисовали голубое небо, а потом пририсовали птицу. Как ее «стереть»? В этом случае единственное, что вы можете, — это нарисовать на ее месте небо.

Компьютерная графика напоминает, скорее, картину. Чтобы что-то «стереть», нужно «нарисовать нечто поверх». Но что это должно быть? Картина с небом имеет голубой цвет, поэтому птицу нужно зарисовать голубым. На нашем же рисунке белый фон, поэтому исходное изображение мяча мы будем зарисовывать белым.

Попробуем это сделать. Отредактируйте программу из листинга 16.11 в соответствии с листингом 16.12. Вам нужно добавить всего одну новую строчку.

Листинг 16.12. Снова пытаемся двигать мяч

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
screen.blit(my_ball,[50, 50])
pygame.display.flip()
pygame.time.delay(2000)
screen.blit(my_ball, [150, 50])
pygame.draw.rect(screen, [255,255,255], [50, 50, 90, 90], 0)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Эта строка «стирает» первый мяч

Мы добавили строку 10, рисующую поверх первого мяча белый прямоугольник. Картинка с мячом имеет 90 пикселов в ширину и 90 в высоту, именно такой размер должен иметь наш прямоугольник. После запуска программы из листинга 16.12 вы увидите, как мяч перемещается в другую точку.

А что под рисунком?

Нарисовать поверх объекта белый фон (или синее небо) просто. А что делать, если птица была нарисована на небе с облаками? Или на фоне деревьев? Чтобы ее стереть, придется рисовать облака или деревья. В данном случае важно то, что вы должны отслеживать, какой фон находится «под» вашим рисунком, так как после его перемещения нужно будет вернуть первоначальное состояние фона.

В примере с мячом все реализуется очень легко, так как мяч находится на белом фоне. Но все сильно усложняется, если поместить мяч, к примеру, на пляж. Рисовать придется уже не белый прямоугольник, а корректный фрагмент фоновой картинки. Еще можно перерисовать всю сцену, поместив мяч в новое положение.

Сглаживание анимации

Итак, мы заставили наш мяч мгновенно переместиться в сторону! Попробуем сделать это движение более реалистичным. Анимируемые объекты обычно смещают небольшими шажками, чтобы добиться плавного движения. Поэтому смоделируем постепенное смещение нашего мяча.

При этом мы не просто уменьшим шаг смещения, а добавим в программу цикл, в процессе работы которого мяч будет двигаться (так как мы хотим добиться множества маленьких сдвигов). Начните с листинга 16.12 и отредактируйте его код в соответствии с листингом 16.13.

Листинг 16.13. Плавное смещение мяча

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
screen.blit(my_ball,[x, y])
pygame.display.flip()
for looper in range (1, 100):
    pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + 5
    screen.blit(my_ball, [x, y])
    pygame.display.flip()

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Добавляем эти строчки

Используем имена x и y (вместо чисел)

Начинаем цикл for

Значение параметра time.delay изменилось с 2000 на 20

После запуска этой программы вы увидите, как мяч смещается из исходной точки к правому краю окна.

Непрерывное движение

Пока что наш мяч смещается к правому краю окна и там останавливается. Попробуем сделать его движение безостановочным.

Что произойдет, если мы просто продолжим увеличивать координату *x*? Мяч будет сдвигаться все дальше и дальше вправо. Но при значении `x = 640` наше окно (поверхность отображения) заканчивается, а значит, мяч просто исчезнет. Попробуйте изменить цикл `for` в строке 10 листинга 16.13 вот таким образом:

```
for looper in range (1, 200):
```

Теперь, когда цикл работает в два раза дольше, мяч начинает исчезать у правого края окна! Предотвратить это можно двумя способами.

- Заставить мяч *отскакивать* от края окна.
- Заставить мяч снова *появляться в начальной точке*.

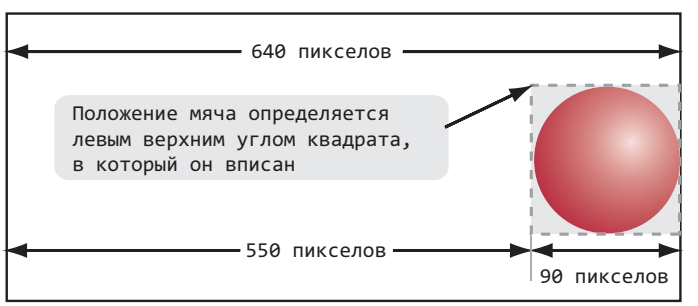
Посмотрим, каким способом реализуются оба варианта.

Отскок мяча

Чтобы мяч *отскакивал* от края окна, нужно заставить его в этой точке поменять направление движения на противоположное. Добавив это поведение для правого и левого краев окна, мы получим непрерывно двигающийся мяч.

С левой стороны все просто, так как достаточно удостовериться, что мяч дошел до координаты 0 (или до какого-то меньшего числа).

А вот во второй точке нужно проверять, коснулась ли правого края окна правая сторона мяча. При условии, что положение мяча задается его левой стороной (левым верхним углом вставленного в программу графического фрагмента). Значит, нам нужно будет вычесть из координаты края окна размер мяча.



Двигаясь к правому краю окна, мяч должен отскакивать (менять направление движения) в момент достижения точки с координатой 550. Внесем в код следующие изменения:

- заставим мяч двигаться бесконечно (пока открыто Pygame-окно). Так как у нас уже есть цикл `while`, работающий, пока окно не закроется, переместим в него код, отвечающий за вывод мяча на экран (это цикл `while` в конце программы);
- вместо того чтобы постоянно добавлять к координате мяча значение 5, создадим новую переменную `speed`, задающую скорость перемещения мяча на каждой итерации. Заодно заставим мяч двигаться быстрее, присвоив этой переменной значение 10.

Новая версия кода нашей программы представлена в листинге 16.14.

Листинг 16.14. Отскакивающий от границ окна мяч

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
```

```
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
x_speed = 10

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
        running = False

    pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + x_speed
    if x > screen.get_width() - 90 or x < 0:
        x_speed = - x_speed
    screen.blit(my_ball, [x, y])
    pygame.display.flip()
pygame.quit()
```

Это переменная скорости speed

Помещаем код вывода мяча на экран внутрь этого цикла while

Когда мяч достигает края окна...

...меняем направление его движения, сменив знак скорости

За отскоки мяча от краев окна отвечают строки 19 и 20. В строке 19 (`if x > screen.get_width() - 90 or x < 0:`) мы определяем, достиг ли мяч края окна, и в случае положительного результата проверки в строке 20 меняем направление его движения (`x_speed = - x_speed`).

Запустите программу и посмотрите, как она работает.

Двухмерный отскок

Пока наш мяч может перемещаться только влево или вправо, то есть его движение является одномерным. Добавим к этому горизонтальному движению одновременное смещение вверх и вниз. Для этого в программу нужно внести изменения в соответствие с листингом 16.15.

Листинг 16.15. Движение мяча в двухмерном пространстве

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
```

```
x_speed = 10
y_speed = 10
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + x_speed
    y = y + y_speed
    if x > screen.get_width() - 90 or x < 0:
        x_speed = - x_speed
    if y > screen.get_height() - 90 or y < 0:
        y_speed = -y_speed
    screen.blit(my_ball, [x, y])
    pygame.display.flip()
pygame.quit()
```

Добавляем скорость смещения по координате y (смещение по вертикали)

Добавляем скорость смещения по координате y (смещение по вертикали)

Заставляем мяч отскакивать от верхней и нижней границ окна

К уже имеющейся у нас программе мы добавили строки 9 (`y_speed = 10`), 18 (`y = y + y_speed`), 21 (`if y > screen.get_height() - 90 or y < 0:`) и 22 (`y_speed = -y_speed`). Запустите новую версию кода и посмотрите, как она работает!

Замедлить скорость движения мяча можно двумя способами:

- уменьшить переменные скорости (`x_speed` и `y_speed`). В результате за каждый шаг анимации мяч будет проходить меньшее расстояние, а плавность его движения повысится;
- увеличить задержку. В листинге 16.15 этот параметр имел значение 20. Он измеряется в миллисекундах, то есть в тысячных долях секунды. И на каждой итерации цикла программа приостанавливалась на 0.02 секунды. При увеличении этого параметра движение замедлится. Если же мы уменьшим его, скорость перемещения возрастет.

Попробуйте поменять значения скорости и задержки и посмотрите, как это влияет на нашу анимацию.

Сквозной перенос мяча

Рассмотрим второй вариант бесконечного движения мяча. Вместо отскоков от границ окна мы заставим его выполнить *сквозной перенос*. Это означает, что исчезнувший справа мяч будет возникать слева.

Для простоты оставим движение только по горизонтали. Код новой версии программы представлен в листинге 16.6.

Листинг 16.16. Движение мяча со сквозным переносом

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
x_speed = 5
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + x_speed
    if x > screen.get_width():
        x = 0
    screen.blit(my_ball, [x, y])
    pygame.display.flip()
pygame.quit()
```

Если мяч достиг правого края...

...начнем его движение с левой стороны

В строках 17 (`if x > screen.get_width():`) и 18 (`x = 0`) мы определяли момент достижения мячом правого края окна и возвращали его в исходную точку, то есть совершали сквозной перенос влево.

Обратите внимание, что мяч, достигший правого края, моментально «перепрыгивает» в точку с координатой [0, 50]. Это выглядит естественнее постепенного ухода за границу окна. В строке 18 измените (`x = 0`) на `x = -90` и посмотрите, как после этого будет двигаться мяч.

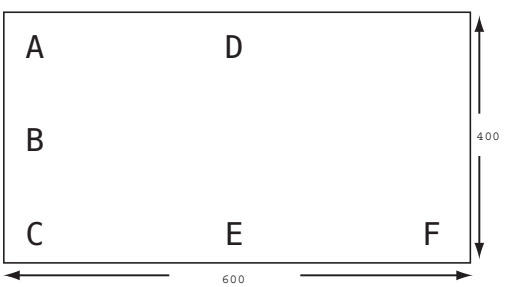
Что мы узнали

Ого! Это была насыщенная глава! В ней мы научились:

- работать с модулем Pygame;
- создавать графическое окно и рисовать в нем фигуры;
- задавать цвета объектов;
- копировать изображения в графическом окне;
- анимировать изображения, в том числе «стирать» их после перемещения на новое место;
- моделировать мяч, «отскакивающий» от границ окна;
- моделировать сквозной перенос мяча.

Проверь себя

1. Какому цвету соответствует RGB-значение [255, 255, 255]?
2. Какому цвету соответствует RGB-значение [0, 255, 0]?
3. Какой Pygame-метод позволяет рисовать прямоугольники?
4. Каким Pygame-методом вы воспользуетесь, чтобы соединить линиями группу точек?
5. Что такое пиксел?
6. Где в Pygame-окне находится точка с координатами [0, 0]?
7. Какая буква в показанном на рисунке Pygame-окне шириной 600 и высотой 400 пикселов имеет координаты [50, 200]?



8. Какая буква на рисунке имеет координаты [300, 50]?
9. Какой Pygame-метод применяется для копирования на поверхность (например на поверхность отображения) графических фрагментов?
10. Назовите два основных этапа процесса «движения», или анимации, изображения?

Эксперименты

1. Мы рассматривали рисование кругов и прямоугольников. Но модуль Pygame позволяет создавать и другие объекты, в том числе линии, дуги, эллипсы и многоугольники. Попробуйте самостоятельно нарисовать все эти фигуры.

   Узнать названия методов можно в документации к модулю Pygame по адресу *www.pygame.org/docs/ref/draw.html*. Если вы не имеете доступа в Интернет, документацию можно найти на жестком диске вашего компьютера (она добавляется в момент установки модуля Pygame), но это не всегда просто. Ищите файл *pygame_draw.html*.

   Кроме того, можно воспользоваться справочной системой языка Python (о которой мы говорили в конце главы 6). Но она не имеет интерактивной оболочки, поэтому запустите IDLE и наберите следующие команды:

```
>>> import pygame
>>> help()
help> pygame.draw
```

Вы получите список методов рисования различных фигур с краткими пояснениями.

2. Вставьте в любую из программ с мячом другое изображение. Можете взять его из папки *\examples\images*, загрузить из Интернета или даже нарисовать собственноручно. Можно использовать даже фрагмент цифровой фотографии.
3. В листинге 16.15 или 16.16 поиграйте со значениями параметров `x_speed` и `y_speed`, заставляя мяч ускоряться и замедляться в различных направлениях.
4. Отредактируйте листинг 16.15, заставив мяч «отскочить» от невидимой стены или пола, расположенных на некотором расстоянии от края окна.
5. В листингах 16.5–16.9 (это программы с современным искусством, синусоидой и таинственной картинкой) переместите строку `pygame.display.flip` внутрь цикла `while`. Для этого достаточно добавить к ней отступ в четыре пробела. После этой строки также внутри цикла `while` добавьте задержку и посмотрите, что получится:

```
    pygame.time.delay(30)
```

ГЛАВА 17

Спрайты и обнаружение столкновений

В этой главе мы продолжим использовать модуль Pygame для анимирования объектов. Вы познакомитесь со *спрайтами* (sprites), которые помогают отслеживать множество двигающихся по экрану картинок. Еще вы узнаете, как определить, когда два изображения перекрываются или ударяются друг о друга, так же как, к примеру, мяч ударяется о баскетбольное кольцо или космический корабль натыкается на астероид.

Спрайты

В предыдущей главе вы убедились, что простая на вид анимация на самом деле совсем не проста. А уж когда по экрану приходится перемещать большое число объектов, зачастую крайне сложно следить за тем, что находится «под» каждым изображением, чтобы перерисовать фон после того, как объект сместится в сторону. Пляжный мяч из первого примера располагался на белом фоне, что сильно упрощало нашу задачу. Но только представьте, что в качестве фона выступает графический фрагмент.

НОВЫЕ СЛОВА

Спрайт — это группа пикселов, которые двигаются и выводятся на экран как единое целое — своего рода графический объект.

Процитирую введение в модуль Sprite из справочного руководства по Pygame Пита Шиннерса, которое можно найти по адресу www.pygame.org/docs/tut/SpriteIntro.html:

> Термин «спрайт» достался нам в наследство от прежних компьютеров и игровых автоматов, которые были не в состоянии достаточно быстро рисовать и стирать графические фрагменты, чтобы происходящее было похоже на игру. Поэтому за обработку объектов, которые нужно было анимировать быстро, отвечало специальное аппаратное обеспечение. Эти объекты назывались «спрайтами» и на них налагался ряд ограничений, зато их можно было очень быстро рисовать и обновлять. В наши дни компьютеры в состоянии справиться с анимацией спрайтов, не передавая эту задачу аппаратной части. Но термин «спрайт» до сих пор используют для обозначения анимируемых объектов в двухмерных играх.

К счастью, нам в этой ситуации приходит на помощь библиотека Pygame. Отдельные изображения или их части, перемещающиеся в разных направлениях, называются *спрайтами*, и в Pygame для работы с ними существует специальный модуль. Он значительно облегчает задачу перемещения графических объектов.

В предыдущей главе мы заставили мяч отскакивать от границ экрана. Почему бы не увеличить количество таких мячей? Разумеется, для каждого из них можно написать свой код, но мы упростим себе жизнь и воспользуемся Pygame-модулем `sprite`.

Что такое спрайт?

Спрайт можно представить как маленький графический фрагмент, своего рода графический объект, который перемещается по экрану и взаимодействует с другими графическими объектами. Большинство спрайтов обладает двумя базовыми свойствами:

- `image` — выводимое спрайтом изображение;
- `rect` — прямоугольная область, заключающая в себя спрайт.

В качестве изображения может выступать рисунок, созданный при помощи Pygame-функций (их примеры вы видели в предыдущей главе) или взятый из файла.

Класс Sprite

Модуль `sprite` библиотеки Pygame предоставляет вам базовый класс `Sprite`. (Помните, пару глав назад мы говорили про объекты и классы?) С базовыми классами обычно не работают напрямую. В данном случае на основе класса `pygame.sprite.Sprite` создается подкласс. Назовем его `MyBallClass`. Вот как будет выглядеть код в этом случае:

```
class MyBallClass(pygame.sprite.Sprite):
    def __init__(self, image_file, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
```

Инициализируем спрайт

Загружаем файл с изображением

Формируем прямоугольник, определяющий границы изображения

Задаем исходное положение мяча

Последнюю строку кода имеет смысл рассмотреть поподробнее. Переменная `location` представляет собой координаты [*x*, *y*], то есть список из двух элементов. Так как он находится слева от знака равенства, его элементам (`x` и `y`) можно присвоить два значения. В данном случае мы присвоили атрибуты `left` и `top` прямоугольнику, в который заключен наш спрайт.

Теперь, когда у нас есть класс `MyBallClass`, создим пару его экземпляров. (Напоминаем, что определение класса является всего лишь чертежом, нам же нужно построить на его основе несколько домов.) Для этого опять потребуется уже знакомый по предыдущей главе код создания Pygame-окна. Кроме того, на экране должны появиться дополни-

тельные мячи, распределенные по строкам и столбцам. Это реализуется при помощи вложенного цикла:

```
img_file = "beach_ball.png"
balls = []
for row in range (0, 3):
    for column in range (0, 3):
        location = [column * 180 + 10, row * 180 + 10]
        ball = MyBallClass(img_file, location)
        balls.append(ball)
```

На каждой итерации цикла координаты места меняются

В указанной точке создаем мяч

Из мячей составляем список

Кроме того, нам нужно *блитировать* мячи на поверхность отображения. (Помните это смешное слово? Мы упоминали его в предыдущей главе.)

```
for ball in balls:
    screen.blit(ball.image, ball.rect)
pygame.display.flip()
```

А теперь соберем все фрагменты воедино и получим программу из листинга 17.1.

Листинг 17.1. Вывод на экран множества мячей при помощи спрайтов

```
import sys, pygame
class MyBallClass(pygame.sprite.Sprite):
    def __init__(self, image_file, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location

size = width, height = 640, 480
screen = pygame.display.set_mode(size)
screen.fill([255, 255, 255])
img_file = "beach_ball.png"
balls = []
for row in range (0, 3):
    for column in range (0, 3):
        location = [column * 180 + 10, row * 180 + 10]
        ball = MyBallClass(img_file, location)
        balls.append(ball)

for ball in balls:
    screen.blit(ball.image, ball.rect)
pygame.display.flip()
```

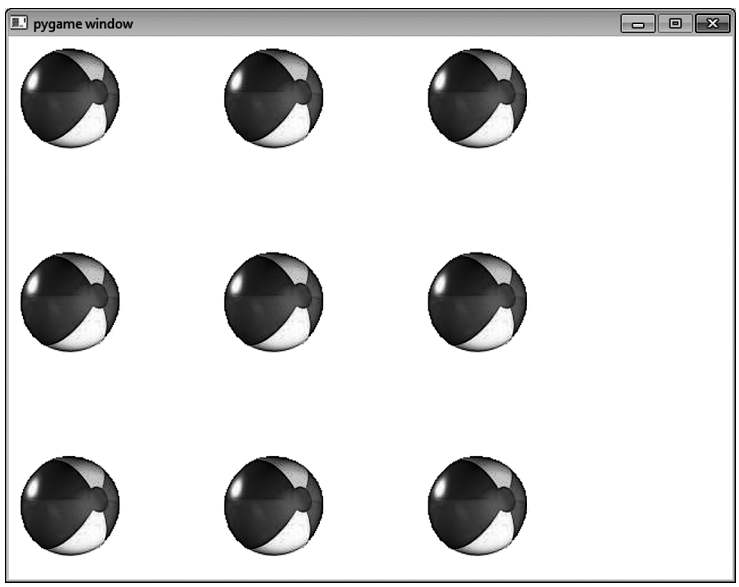
Определяем подкласс ball

Задаем размер окна

Добавляем мячи в список

```
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

После запуска этой программы в Pygame-окне появится девять мячей:



Через минуту мы приведем их в движение.

Обратили внимание на небольшое изменение в строках 10 и 11, задающих размер Pygame-окна? У нас была такая строка:

```
screen = pygame.display.set_mode([640,480])
```

Теперь мы ее заменили:

```
size = width, height = 640, 480
screen = pygame.display.set_mode(size)
```

Этот код не только задает размер окна, как раньше, но и определяет две переменные `width` и `height`, которые потребуются нам позднее. В данном случае примечательно то, что всего одной инструкцией мы создали список `size`, состоящий из двух элементов, попутно определив две целочисленные переменные `width` и `height`. Кроме того, список в данном случае фигурирует без квадратных скобок, но интерпретатор Python воспринимает это нормально.

Я попытался показать вам, что одни и те же вещи порой реализуются в Python по-разному. При этом нельзя сказать, что один способ реализации лучше другого (ведь они оба работают). Разумеется, вы должны следовать синтаксису языка Python, но пространство для самовыражения тоже есть. Попросив 10 программистов написать одинаковую программу, вы можете получить десять разных фрагментов кода.

Метод move()

Мы формируем мячи как экземпляры класса `MyBallClass`, поэтому для их перемещения имеет смысл воспользоваться методом класса. Вот как создать метод `move()` класса:

```
def move(self):
    self.rect = self.rect.move(self.speed)
    if self.rect.left < 0 or self.rect.right > width:
        self.speed[0] = -self.speed[0]

    if self.rect.top < 0 or self.rect.bottom > height:
        self.speed[1] = -self.speed[1]
```

Если происходит столкновение с правой и левой сторонами окна, меняем знак скорости по координате x

Если происходит столкновение с верхней и нижней границами окна, меняем знак скорости по координате y

Спрайты (точнее, связанные с ними прямоугольники) обладают встроенным методом `move()`. В этот метод передается параметр `speed`, информирующий, насколько далеко (то есть как быстро) следует перемещать объект. Так как мы собираемся создавать двухмерную графику, параметр `speed` будет представлен списком из двух чисел — скоростями по координатам ***x*** и ***y***. Кроме того, требуется проверка столкновений мяча с границами окна, за счет которых мяч «скачет» по экрану.

Отредактируем определение класса `MyBallClass`, добавив туда свойство `speed` и метод `move()`:

```
class MyBallClass(pygame.sprite.Sprite):
    def __init__(self, image_file, location, speed):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > width:
            self.speed[0] = -self.speed[0]

        if self.rect.top < 0 or self.rect.bottom > height:
            self.speed[1] = -self.speed[1]
```

Добавляем аргумент speed

Добавляем эту строку, создающую для мяча атрибут speed

Добавляем метод перемещения мяча

Обратите внимание, как изменилась строка 2 (`def __init__(self, image_file, location, speed):`). Кроме того, появились строка 7 (`self.speed = speed`) и новый метод `move()` в строках с 9 по 15.

Теперь при создании экземпляров мяча наряду с файлами изображения и начальными координатами нужно будет указывать скорость перемещения каждого экземпляра:

```
speed = [2, 2]
ball = MyBallClass(img_file, location, speed)
```

Предыдущий фрагмент кода создает мячи, движущиеся с одинаковой скоростью (в одном направлении), но лучше будет, если мы внесем в их перемещения некоторое разнообразие. Воспользуемся для задания скорости функцией `random.choice()`:

```
from random import *
speed = [choice([-2, 2]), choice([-2, 2])]
```

В результате для скорости по обеим координатам будет выбираться значение –2 или 2.

Полностью код программы представлен в листинге 17.2.

Листинг 17.2. Перемещение мячей по экрану при помощи спрайтов

```
import sys, pygame
from random import *

class MyBallClass(pygame.sprite.Sprite):
    def __init__(self, image_file, location, speed):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > width:
            self.speed[0] = -self.speed[0]

        if self.rect.top < 0 or self.rect.bottom > height:
            self.speed[1] = -self.speed[1]

size = width, height = 640, 480
screen = pygame.display.set_mode(size)
screen.fill([255, 255, 255])
```

Определение класса мячей

```
img_file = "beach_ball.png"
balls = []                                    Создаем список для
                                              отслеживания мячей
for row in range (0, 3):
    for column in range (0, 3):
        location = [column * 180 + 10, row * 180 + 10]
        speed = [choice([-2, 2]), choice([-2, 2])]
        ball = MyBallClass(img_file, location, speed)
        balls.append(ball)
running = True                                Добавляем в список каждый
                                              созданный мяч
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    pygame.time.delay(20)
    screen.fill([255, 255, 255])
    for ball in balls:
        ball.move()                           Перерисовываем экран
        screen.blit(ball.image, ball.rect)
    pygame.display.flip()
pygame.quit()
```

Для отслеживания мячей в программе используется список. В строке 32 (`balls.append(ball)`) каждый только что созданный мяч добавляется в этот список.

Последние пять строк кода отвечают за перерисовку экрана. Мы слегка смошенничали и вместо «стирания» (рисования поверх) каждого мяча по отдельности просто заливаем фон белым цветом и заново рисуем на нем все мячи.

Поэкспериментируйте с этим кодом, меняя количество мячей, их скорость, способ перемещения и «отскока» и другие параметры. Вы заметите, что мячи перемещаются по экрану и отскакивают от границ окна, но пока не отскакивают друг от друга!

Бум! Обнаружение столкновений

В большинстве компьютерных игр важную роль играет информация о соударении спрайтов. К примеру, нужно знать момент, когда шар для боулинга сбивает кегли или когда боеприпас попадает в космический корабль.

Может показаться, что если вы имеете представление о положении и размере каждого спрайта, ничто не мешает написать код, проверяющий положение и размер всех спрайтов и определяющий момент пересечения. Однако создатели библиотеки Pygame уже сделали эту работу за нас. В библиотеку Pygame встроен механизм *распознавания столкновений* (collision detection).

НОВЫЕ СЛОВА

Распознавание столкновений означает получение информации о соприкосновении или перекрывании двух спрайтов. Когда два движущихся объекта натыкаются друг на друга, это называется столкновением.

Библиотека Pygame позволяет также *группировать* спрайты. Например, в модели игры в боулинг кегли можно поместить в одну группу, а шар — в другую.

Группирование идет рука об руку с распознаванием столкновений. В примере с игрой в боулинг нам важен момент соприкосновения шара с любой кеглей, поэтому проверяться будет столкновение спрайта, представляющего шар, с любым спрайтом из группы кеглей. Также можно распознавать столкновения внутри группы (столкновения кеглей друг с другом).

Рассмотрим пример. Начнем с отскакивающих от стен мячей, но для простоты будем рассматривать всего четыре мяча вместо девяти. А вместо списка мячей, которым мы пользовались в последнем примере, прибегнем к Pygame-классу `group`.

Кроме того, мы немного почистим код, поместив связанную с анимацией мяча часть кода (последние несколько строк из листинга 17.2) в функцию с именем `animate()`. Именно там будет располагаться код распознавания столкновений. После столкновения мячи будут менять направление движения (листинг 17.3).

Листинг 17.3. Распознавание столкновений с использование группы спрайтов вместо списка

```
import sys, pygame
from random import *

class MyBallClass(pygame.sprite.Sprite):
    def __init__(self, image_file, location, speed):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > width:
            self.speed[0] = -self.speed[0]
        if self.rect.top < 0 or self.rect.bottom > height:
            self.speed[1] = -self.speed[1]
```

Определение класса мячей

```
def animate(group):
    screen.fill([255,255,255])
    for ball in group:
        group.remove(ball)
        if pygame.sprite.spritecollide(ball, group, False):
            ball.speed[0] = -ball.speed[0]
            ball.speed[1] = -ball.speed[1]

        group.add(ball)
        ball.move()
        screen.blit(ball.image, ball.rect)
    pygame.display.flip()
    pygame.time.delay(20)
size = width, height = 640, 480
screen = pygame.display.set_mode(size)
screen.fill([255, 255, 255])
img_file = "beach_ball.png"
group = pygame.sprite.Group()
for row in range (0, 2):
    for column in range (0, 2):
        location = [column * 180 + 10, row * 180 + 10]
        speed = [choice([-2, 2]), choice([-2, 2])]
        ball = MyBallClass(img_file, location, speed)
        group.add(ball)

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    animate(group)
pygame.quit()
```

Удаляем спрайт из группы

Новая функция animate()

Проверка столкновения спрайта и группы

Возвращаем мяч в группу

Здесь начинается основная программа

Создаем группу спрайтов

На этот раз создаем только четыре мяча

Добавляем каждый мяч в группу

Вызываем функцию animate(), передавая в нее группу

Интереснее всего в данном случае способ распознавания столкновений. Pygame-модуль `sprite` обладает функцией `spritecollide()`, которая определяет столкновение одного спрайта с любым спрайтом из группы. Проверка столкновения спрайтов *внутри* группы происходит в три этапа.

1. Удаляем спрайт из группы.
2. Проверяем, не сталкивается ли этот спрайт с остальными спрайтами группы.
3. Возвращаем спрайт в группу.

Именно это происходит в цикле `for` в строках с 21 по 29 (в средней части функции `animate()`). Без предварительного удаления спрайта из группы функция `spritecollide()`

обнаружит, что он сталкивается сам с собой. На первый взгляд это может показаться странным, но через некоторое время вы найдете такой подход разумным.

Запустите программу и посмотрите, что получится. Обратили внимание на странное поведение? Я заметил две вещи:

- столкновение мячей сопровождается «заминкой», или двойным ударом;
- иногда мяч застревает у края окна и некоторое время висит там.

Увидеть это будет проще, если вы увеличите шаг анимации. Для этого увеличьте скорость с 2 до 5, а задержку между шагами — с 20 до 50.

Почему это происходит? Это связано с нашей реализацией функции `animate()`. Обратите внимание, что мы двигаем один мяч, проверяем, не сталкивается ли он с чем-нибудь, затем двигаем другой мяч и проверяем его на столкновения, и т. д. Наверное, лучше сначала завершить все перемещения и только потом проверять столкновения.

Итак, нам нужно взять строку 28 (`ball.move()`) и сделать для нее отдельный цикл:

```
def animate(group):
    screen.fill([255,255,255])
    for ball in group:
        ball.move()
    for ball in group:
        group.remove(ball)

        if pygame.sprite.spritecollide(ball, group, False):
            ball.speed[0] = -ball.speed[0]
            ball.speed[1] = -ball.speed[1]

    group.add(ball)

    screen.blit(ball.image, ball.rect)
pygame.display.flip()
pygame.time.delay(20)
```

Сначала перемещаем все мячи

Затем распознаем столкновения и реализуем отскоки

Запустите программу, и вы увидите, что она стала работать лучше.

Можно поиграть с кодом, редактируя такие параметры, как скорость (число `time.delay()`), количество мячей, их исходное положение, хаотичность их перемещений и т. п.

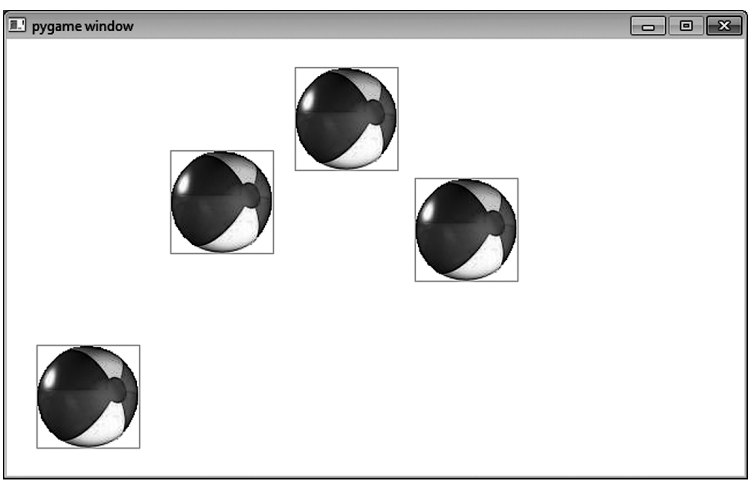
Точность обнаружения столкновений

Наверное, вы обратили внимание, что при «столкновении» мячи далеко не всегда соприкасаются полностью. Дело в том, что функция `spritecollide()` распознает столкновения не по форме мячей, а по размерам контейнера `rect`, в который они помещены.

Для наглядности нарисуйте вокруг мяча прямоугольник и используйте в модели эти новые изображения. Можете взять уже подготовленный мною рисунок:

```
img_file = "b_ball_rect.png"
```

На экране это будет выглядеть примерно так:



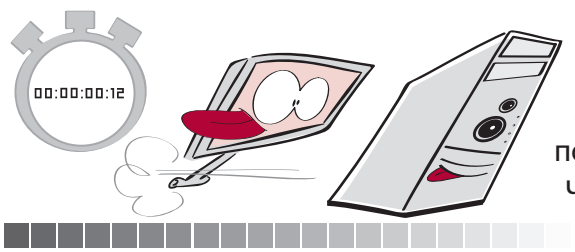
Чтобы заставить мячи отскакивать при соприкосновении их краев (а не краев прямоугольных контейнеров), нам потребуется *распознавание столкновений с точностью до пиксела*. Этого функция `spritecollide()` делать не умеет, *распознавая столкновения по ограничивающим прямоугольникам*.

Рассмотрим, в чем состоит разница. Во втором случае столкновением считается соприкосновение любых точек прямоугольников. Если же мы распознаем столкновения с точностью до пиксела, засчитывается только соприкосновение самих шаров:

Распознавание столкновений с точностью до пиксела более реалистично. (Ведь вокруг *настоящего* мяча нет никаких невидимых прямоугольников.) Но на практике его реализовать сложнее.

Для большинства вещей, которые делаются с помощью библиотеки Pygame, хватает распознавания по ограничивающим прямоугольникам. Распознавание с точностью до пиксела требует большего количества кода и замедляет игру, поэтому прибегать к нему стоит только в случае, когда без него невозможно обойтись. Реализуется оно с помощью специальных модулей, которые можно найти на сайте библиотеки Pygame (на момент написания данной книги были доступны по крайней мере два из них). Либо поищите подходящие модули в Интернете.

Отсчет времени



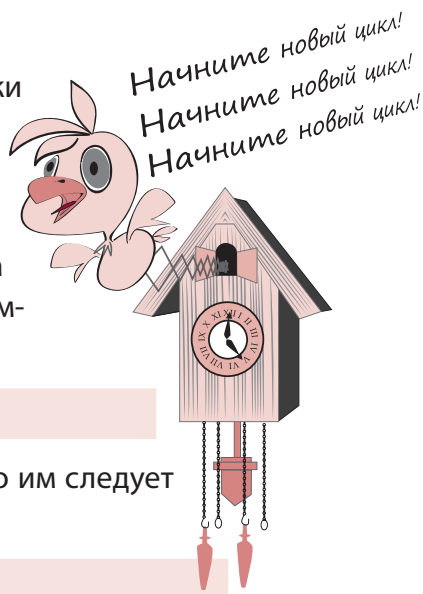
До этого момента мы контролировали скорость нашей анимации при помощи функции `time.delay()`. Но это не самый лучший способ, так как он не позволяет определить продолжительность анимации. Часть времени занимает выполнение кода цикла (неизвестное время), затем возникает задержка (известное время). В итоге получается, что часть времени нам известна, а часть нет. Чтобы определить частоту запуска анимационного цикла, нужно знать общее время его работы, которое представляет собой сумму времени выполнения кода и времени задержки. Время анимации обычно считается в миллисекундах, то есть в тысячных долях секунды. Миллисекунды обозначаются аббревиатурой мс, поэтому 25 миллисекунд мы запишем как 25 мс.

Предположим, что в нашем примере время выполнения кода составляет 15 мс. За это время будет выполнен код цикла `while`, время задержки — `time.delay()` — сюда не входит. Но его мы и так знаем, так как в строке `time.delay(20)` сделали его равным 20 мс. В итоге общее время одного шага анимации составит 20 мс + 15 мс = 35 мс, а в одной секунде 1000 мс. Если один шаг анимации занимает 35 мс, разделив 1000 мс на 35 мс, мы получим 28.57. Это означает, что за одну секунду выполняется примерно 29 шагов анимации. Шаг анимации в компьютерной графике называется *кадром* (frame), и обсуждая, насколько быстро обновляется графика, программисты говорят о *частоте кадров* (frame rate) и *количестве кадров в секунду* (frames per second). В нашем примере частота кадров составляет примерно 29 кадров в секунду, или 29 к/с.

Проблема в том, что по-настоящему контролировать время выполнения кода мы не можем. При добавлении или удалении части кода оно меняется. Более того, даже при неизменном коде достаточно поменять количество спрайтов (например игровые объ-

екты могут появляться и исчезать), и время их прорисовки изменится. Кроме того, скорость выполнения кода зависит от компьютера, на котором он запускается. Вместо 15 мс это может занять 10 или 20 мс. Поэтому хотелось бы найти более предсказуемый способ управления частотой кадров. К счастью, Pygame-модуль `time` предоставляет нам нужные инструменты в классе `Clock`.

Функция pygame.time.Clock()



Функция `pygame.time.Clock()` вместо добавления задержки к каждой итерации цикла контролирует частоту итераций. Она напоминает таймер, который при срабатывании говорит: «Начните новый цикл! Начните новый цикл!..»

Перед началом работы с ним нужно создать экземпляр объекта `Clock`. Эта процедура ничем не отличается от создания экземпляра любого другого класса:

```
clock = pygame.time.Clock()
```

Затем в теле основного цикла вы указываете часам, как часто им следует «тикать», то есть с какой частотой будет запускаться наш цикл:

```
clock.tick(60)
```

Передаваемое функции `clock.tick()` значение не является количеством миллисекунд. Скорее, оно указывает, сколько раз в секунду должен запускаться цикл. Например, предполагается, что данный цикл будет работать 60 раз в секунду. Я пишу «предполагается», потому что скорость выполнения цикла зависит от способности компьютера к обработке кода. 60 кадров в секунду означают, что один цикл будет длиться 1000 / 60 = 16.66 мс (примерно 17 мс). Если выполнение включенного в цикл кода займет более 17 мс, к моменту, когда функция `clock` подаст сигнал о начале нового цикла, старый еще не будет закончен.

По сути это означает ограничение, налагаемое на частоту кадров при запуске вашей графики. Ограничение зависит от сложности графики, размера окна и скорости выполнения программы. Есть программы, которые на одном компьютере будут воспроизводиться с частотой 90 кадров в секунду, в то время как на более старых и медленных компьютерах этот показатель упадет до 10 кадров в секунду.

Pygame-программы с графикой средней сложности большинство современных компьютеров без проблем воспроизводит с частотой от 20 до 30 кадров в секунду. Поэтому, если вы хотите, чтобы ваша программа воспроизводилась с примерно одинаковой скоростью, выбирайте частоту кадров от 20 до 30 в секунду или меньше. Этого хватает

для имитации плавного движения. В приводимых примерах мы с этого момента будем использовать значение `clock.tick(30)`.

Проверка частоты кадров

Узнать, с какой частотой воспроизводится анимация, позволяет функция `clock.get_fps()`. Разумеется, если мы зададим частоту 30 кадров в секунду, программа практически всегда будет воспроизводиться именно с этой частотой (при условии, что на это хватает мощности компьютера). Чтобы определить максимальную частоту воспроизведения на конкретной машине, сделайте параметр `clock.tick` очень большим (например 200 к/с). После чего запустите программу и проверьте реальную частоту кадров с помощью функции `clock.get_fps()`. (Пример будет рассмотрен чуть позже.)

Масштабирование частоты кадров

Если вы хотите гарантировать одинаковую скорость воспроизведения анимации на всех компьютерах, можно сделать хитрый ход. Зная скорость, с которой должна воспроизводиться анимация, и скорость, с которой она на самом деле воспроизводится, можно отрегулировать, или *отмасштабировать*, этот параметр в соответствии со скоростью работы машины.

Например, предположим, вы указали `clock.tick(30)`, что соответствует частоте воспроизведения 30 кадров в секунду. Если применив функцию `clock.get_fps()`, вы обнаружили, что анимация воспроизводится с частотой всего 20 кадров в секунду, значит, объекты на экране двигаются медленнее, чем нужно. При показе меньшего количества кадров в секунду заставить объекты двигаться с корректной скоростью можно, увеличив расстояние, проходимое ими в каждом кадре. С движущимися объектами связана переменная (или атрибут) `speed`, определяющий смещение в каждом кадре. Значит, на более медленных компьютерах нужно увеличивать атрибут `speed`.

Как сильно его нужно увеличить? Это определяется соотношением желаемой и фактической частот кадров. Если скорость объекта равна 10, желаемая частота составляет 30 кадров в секунду, а программа осуществляет воспроизведение с частотой 20 кадров в секунду, нужно сделать так:

```
object_speed = current_speed * (desired fps / actual fps)
object_speed = 10 * (30 / 20)
object_speed = 15
```

Значит, объект, который перемещался в кадре на 10 пикселов, для компенсации замедленного воспроизведения следует переместить на 15 пикселов.

В листинге 17.4 представлена программа с мячами, в которую добавлены обсуждавшиеся в последних разделах класс `Clock` и функция `get_fps()`.

Листинг 17.4. Класс Clock и функция get_fps() в программе с мячами

```
import sys, pygame
from random import *
class MyBallClass(pygame.sprite.Sprite):
    def __init__(self, image_file, location, speed):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > width:
            self.speed[0] = -self.speed[0]
        if self.rect.top < 0 or self.rect.bottom > height:
            self.speed[1] = -self.speed[1]
```

Определение класса ball

Функция time.delay() больше не используется

```
def animate(group):
    screen.fill([255,255,255])
    for ball in group:
        ball.move()
    for ball in group:
        group.remove(ball)
        if pygame.sprite.spritecollide(ball, group, False):
            ball.speed[0] = -ball.speed[0]
            ball.speed[1] = -ball.speed[1]
        group.add(ball)
        screen.blit(ball.image, ball.rect)
    pygame.display.flip()
```

Функция animate

```
size = width, height = 640, 480
screen = pygame.display.set_mode(size)
screen.fill([255, 255, 255])
img_file = "beach_ball.png"
clock = pygame.time.Clock()
group = pygame.sprite.Group()
for row in range (0, 2):
    for column in range (0, 2):
        location = [column * 180 + 10, row * 180 + 10]
        speed = [choice([-4, 4]), choice([-4, 4])]
        ball = MyBallClass(img_file, location, speed)
        group.add(ball) #добавляем мяч в группу
```

Создаем экземпляр класса Clock

Инициализируем все и рисуем мячи

```
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
            frame_rate = clock.get_fps()
            print "frame rate = ", frame_rate
    animate(group)
    clock.tick(30)
pygame.quit()
```

Здесь начинается основной цикл while

Проверяем частоту кадров

Теперь частоту кадров (ограниченную скоростью работы компьютера) контролирует функция clock.tick

Этот материал дает основные представления о библиотеке Pygame и спрайтах. В следующей главе, создавая при помощи Pygame настоящую игру, вы познакомитесь с добавлением текста (для вывода на экран заработанных игроком очков) и вводом с помощью мыши и клавиатуры.

Что мы узнали

В этой главе мы познакомились с:

- Pygame-спрайтами и их применением для управления множеством движущихся изображений;
- группами спрайтов;
- обнаружением столкновений;
- функцией `pygame.clock` и понятием частоты кадров.

Проверь себя

1. Что такое обнаружение столкновений по ограничивающим прямоугольникам?
2. Что такое обнаружение столкновений с точностью до пиксела и чем оно отличается от обнаружения столкновений по ограничивающим прямоугольникам?
3. Назовите два способа слежения за группой спрайтовых объектов.
4. Назовите два способа управления скоростью анимации.
5. Почему применение функции `pygame.clock` дает более точные результаты, чем `pygame.time.delay()`?
6. Как определить частоту воспроизведения компьютером вашей анимации?

Эксперименты

Если вы набирали код всех программ из этой главы вручную, значит, вы хорошо потрудились. А всем, кто ленился, я рекомендую это проделать. Обещаю, вы кое-чему при этом научитесь!

ГЛАВА 18

События

До сих пор ввод данных в наши программы осуществлялся крайне просто. Пользователь вводил строки через функцию `raw_input()` или данные поставлялись модулем EasyGui (с ним мы познакомились в главе 6). Кроме того, я показал вам, как с помощью мыши закрыть Pygame-окно, но механизм этого действия не объяснялся.

В этой главе вы познакомитесь с дополнительным инструментом ввода — *событиями* (events). При этом мы детально рассмотрим, что делает код закрытия Pygame-окна и как он функционирует. Также вы узнаете, каким образом ввести данные при помощи мыши и научите свои программы немедленно реагировать на нажатие клавиш, не дожидаясь, пока пользователь нажмет клавишу ***Enter***.

События

На вопрос «Что такое событие?» вы, скорее всего, ответите: «То, что происходит». Это определение работает не только в реальном мире, но и в программировании. У программ зачастую возникает необходимость отреагировать на «происходящее», в частности на:

- перемещение указателя мыши или щелчок мыши;
- нажатие клавиш;
- завершение определенного временно́го интервала.

Большинство написанных нами программ от начала до конца работали по совершенно предсказуемому алгоритму, отвлекаясь разве что на циклы или условия. Но существует и другой класс программ, которые называют *событийно-управляемыми* (event-driven), и они работают немного по-другому. Обычно они не выполняют никаких действий, ожидая, пока не случится определенное *событие*. А после этого начинают процедуру его *обработки*.

Хорошим примером такой программы является операционная система Windows (или любой другой GUI-интерфейс). После включения компьютера вы сидите и ждете, пока она загрузится. При этом программы не запускаются, указатель по экрану не двигается. А перемещения указателя или щелчки кнопки мыши ведут к определенным последствиям. Например может появиться стартовое меню.

Цикл событий

Чтобы событийно-управляемые программы могли «увидеть» происходящее событие, они должны его «ждать». Программа постоянно сканирует ту часть компьютерной памяти, которая обычно подает сигнал о возникновении события. Во время ее работы это происходит снова и снова. В главе 8 вы уже познакомились с программами, которые раз за разом выполняют одни и те же действия — в них используются *циклы*. Наступление события также отслеживается при помощи цикла. Он называется *циклом событий* (event loop).

Все Pygame-программы, которые мы писали в последних двух главах, заканчивались циклом `while`. Было отмечено, что этот цикл функционирует, пока работает программа. Именно он является циклом событий. (Итак, на место встал первый фрагмент мозаики, представляющей принцип работы кода выхода.)

Очередь событий

Все эти события наступают после перемещения указателя мыши, щелчка кнопки мыши или нажатия клавиши. Что происходит потом? Я уже отмечал, что цикл событий постоянно сканирует определенный фрагмент памяти компьютера. Фрагмент памяти, в котором хранятся события, называется *очередью событий*.

НОВЫЕ СЛОВА

Всем нам доводилось стоять в очередях. В программировании *очередью* обычно называют список элементов, поступающих в список в определенном порядке или подлежащих использованию в определенном порядке.

Очередь событий представляет собой список всех совершившихся событий, расположенных в порядке их возникновения.

Обработчики событий

Программа или игра с GUI-интерфейсом должна знать, когда пользователь нажимает клавишу или двигает указатель. Все эти нажатия, щелчки и перемещения указателя мыши относятся к *событиям*, и программе нужно иметь информацию о том, что ей в этих случаях делать. Она должна их *обработать*. Часть программы, реагирующая на определенное событие, называется *обработчиком события*.

Обрабатываются не все события. При перемещении указателя по рабочему столу возникают сотни событий, ведь цикл событий выполняется быстро. Каждую долю секунды при смещении указателя мыши хотя бы на миллиметр в сторону генерируется новое событие. Но ваша программа может не обращать внимания на перемещения указателя. Значение могут иметь только щелчки в определенных местах. В этом случае программа будет игнорировать события `mouseMove`, обращая внимание только на события `mouseClick`.

Событийно-управляемые программы имеют обработчики только нужных событий. Если движением корабля в игре игрок управляет при помощи клавиш со стрелками, нужно

написать обработчик события keyDown. Если же движение корабля контролируется при помощи мыши, потребуется уже обработчик события mouseMove.

Для начала рассмотрим события, которые могут потребоваться в ваших программах. Мы снова воспользуемся библиотекой Pygame, поэтому все события, о которых пойдет речь, будут принадлежать очереди Pygame-событий. С другими Python-модулями связаны другие группы событий. Например в главе 20 вы познакомитесь с модулем PyQ. Его набор событий отличается от Pygame-событий. Впрочем, во всех случаях (и даже в большинстве языков программирования) способ обработки событий примерно один и тот же. О полном совпадении в разных системах, разумеется, говорить не приходится, но сходств куда больше, чем различий.

События клавиатуры

Рассмотрим пример события клавиатуры. Предположим, что после нажатия клавиши должно быть выполнено некое действие. В Pygame-модуле за это отвечает событие KEYDOWN. Проиллюстрируем его применение на примере мяча из листинга 16.15, в котором объект движется в стороны, отскакивая от границ окна. Но перед тем как добавить события, обновим программу с учетом ранее изученного материала:

- добавим спрайты;
- воспользуемся функцией clock.tick() вместо функции time.delay().

Первым делом нужно поместить мяч в отдельный класс, который будет обладать методами __init__() и move(). Мы создадим экземпляр этого класса и в основном цикле while воспользуемся функцией clock.tick(30). Код после внесения в него указанных изменений представлен в листинге 18.1.

Листинг 18.1. Программа движения мяча после добавления спрайтов и функции Clock.tick()

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
background = pygame.Surface(screen.get_size())
background.fill([255, 255, 255])
clock = pygame.time.Clock()

class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed
```

Класс Ball с методом move()

```
    def move(self):
        if self.rect.left <= screen.get_rect().left or \
                self.rect.right >= screen.get_rect().right:
            self.speed[0] = - self.speed[0]
        newpos = self.rect.move(self.speed)
        self.rect = newpos

my_ball = Ball('beach_ball.png', [10,0], [20, 20])
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    clock.tick(30)
    screen.blit(background, (0, 0))
    my_ball.move()
    screen.blit(my_ball.image, my_ball.rect)
    pygame.display.flip()
pygame.quit()
```

Класс Ball с методом move()

Создаем экземпляр мяча

Скорость, местоположение

Это часы

Перерисовываем все

Обратите внимание, что на этот раз мы по-другому «стирали» мяч после его перемещения. Как вы уже знаете, удалить спрайт перед тем, как рисовать его на новом месте, можно двумя способами: нарисовать фон там, где изначально располагался спрайт, и в каждом кадре рисовать весь фон заново, фактически начиная все «с чистого листа». В данном случае был выбран второй вариант. Но вместо использования на каждой итерации цикла функции `screen.fill()` мы создали поверхность `background` и залили ее белым цветом. Затем на каждой итерации цикла мы *блитировали* эту поверхность на поверхность отображения `screen`. В результате мы получили тот же самый результат, но немножко другим способом.

События клавиш

Добавим обработчик события, который заставит мяч при нажатии клавиши ↓ двигаться вниз, а при нажатии клавиши ↑ двигаться вверх. Библиотека Pygame состоит из набора модулей. В это главе мы будем работать с модулем `pygame.event`.

У нас уже есть цикл Pygame-событий (это цикл `while`). Он ждет особого события, которое называется `QUIT`:

```
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
```

Метод `pygame.event.get()` получает список всех событий в очереди. Цикл `for` последовательно просматривает все события списка и, обнаружив событие `QUIT`, присваивает переменной `running` значение `False`. В результате цикл `while` и работа программы завершаются. Итак, мы выяснили, как работает код завершения работы программы при щелчке по кнопке закрытия Pygame-окна.

Но в данном случае я хочу познакомить вас с еще одним типом событий. Так как нам нужно поймать момент нажатия клавиши, потребуется событие `KEYDOWN`. Это будет примерно такой код:

```
if event.type == pygame.KEYDOWN
```

Так как одна инструкция `if` у нас уже есть, достаточно добавить еще одно условие `elif`, как мы это делали в главе 7:

```
while running:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            # какой-то код
```

Это новая часть кода, распознающая нажатие клавиши

Что это за «какой-то код», фигурирующий после кода распознавания нажатия клавиши? Как уже отмечалось, при нажатии клавиши ↑ мяч должен двигаться вверх, а при нажатии клавиши ↓ — вниз. Поэтому можно написать так:

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_UP:
                my_ball.rect.top = my_ball.rect.top - 10
            elif event.key == pygame.K_DOWN:
                my_ball.rect.top = my_ball.rect.top + 10
```

Поднимаем мяч вверх на 10 пикселов

Опускаем мяч вниз на 10 пикселов

Именами `K_UP` и `K_DOWN` Pygame обозначает клавиши ↑ и ↓. После внесения изменений программа должна выглядеть так, как показано в листинге 18.2.

Листинг 18.2. Мяч, передвигаемый при помощи клавиш со стрелками

```
import pygame, sys
pygame.init()
```

Инициализируем все

```
screen = pygame.display.set_mode([640,480])
background = pygame.Surface(screen.get_size())
background.fill([255, 255, 255])
clock = pygame.time.Clock()

class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        if self.rect.left <= screen.get_rect().left or \
                self.rect.right >= screen.get_rect().right:
            self.speed[0] = - self.speed[0]
        newpos = self.rect.move(self.speed)
        self.rect = newpos

my_ball = Ball('beach_ball.png', [10,0], [20, 20])
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_UP:
                my_ball.rect.top = my_ball.rect.top - 10
            elif event.key == pygame.K_DOWN:
                my_ball.rect.top = my_ball.rect.top + 10

    clock.tick(30)
    screen.blit(background, (0, 0))
    my_ball.move()
    screen.blit(my_ball.image, my_ball.rect)
    pygame.display.flip()
pygame.quit()
```

Инициализируем все

Определение класса Ball с методом move()

Создаем экземпляр мяча

Проверяем нажатие клавиши и перемещаем мяч вверх или вниз

Перерисовываем всю сцену

Запустите программу из листинга 18.2 и начните нажимать клавиши со стрелками. Корректно ли она работает?

ПОВТОР НАЖАТИЯ КЛАВИШИ

Возможно, вы обратили внимание, что при нажатии и удержании клавиши со стрелкой мяч все равно перемещается всего на один шаг вверх или вниз. Дело в том, что мы не

объяснили программе, как реагировать на *удержание* нажатой клавиши. В ответ на действие пользователя генерируется одно событие KEYDOWN, хотя в случае удержания клавиши Pygame-модуль позволяет генерировать множественные события KEYDOWN. Эта операция называется *повтором нажатия клавиши* (key repeat). Вы указываете, сколько времени должно пройти до момента повтора и как часто следует повторять операцию. Все параметры задаются в миллисекундах (сотых долях секунды). Вот как это выглядит:

```
delay = 100
interval = 50
pygame.key.set_repeat(delay, interval)
```

Параметр `delay` указывает модулю Pygame, через какой промежуток времени следует повторить действие, а параметр `interval` задает темп повтора, то есть промежуток между отдельными событиями KEYDOWN.

Добавьте этот фрагмент кода в листинг 18.2 (после строки `pygame.init`, но перед циклом `while`) и посмотрите, как изменится поведение программы.

Имена событий и клавиш

Отслеживая нажатия клавиш ↑ и ↓, мы использовали тип события KEYDOWN и имена клавиш K_UP и K_DOWN. Какие еще типы событий нам доступны? Как называются остальные клавиши? Так как клавиш очень много, я не буду приводить здесь подробный список. Но вы можете найти его на сайте Pygame.

Список событий располагается в соответствующем разделе документации по Pygame:

www.pygame.org/docs/ref/event.html

Список имен клавиш находится здесь:

www.pygame.org/docs/ref/key.html

Вот наиболее распространенные события, которыми мы будем пользоваться:

- QUIT;
- KEYDOWN;
- KEYUP;
- MOUSEMOTION;
- MOUSEBUTTONUP;
- MOUSEBUTTONDOWN.

В библиотеке Pygame имена присвоены каждой клавише. Вы уже знаете, что клавиши ↑ и ↓ называются K_UP и K_DOWN. В дальнейшем вы познакомитесь и с другими именами, при этом все они будут начинаться с буквы K_, за которой следует имя клавиши, например:

- K_a, K_b и т. д. (для клавиш с буквами);
- K_SPACE;
- K_ESCAPE
- и т. п.

События мыши

Вы только что узнали, как инициировать события клавиатуры и использовать их для управления какими-то операциями. Мы создали мяч, который можно двигать вверх и вниз при помощи клавиш со стрелками. Теперь попробуем сделать так, чтобы мячом можно было управлять при помощи мыши. Это позволит вам познакомиться со способами обработки событий мыши и применения информации о положении указателя мыши.

Чаще всего используются следующие три события мыши:

- MOUSEBUTTONUP;
- MOUSEBUTTONDOWN;
- MOUSEMOTION.

Проще всего заставить мяч следовать за перемещениями указателя в Pygame-окне. Для этого нам потребуется атрибут rect.center. Именно с его помощью центр мяча привязывается к положению указателя.

Заменим в коде while код, распознающий события клавиатуры, кодом, распознающим события мыши:

```
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEMOTION:
            my_ball.rect.center = event.pos
```

Распознаем перемещение указателя и двигаем мяч

В данном случае все даже проще, чем в предыдущем примере. Отредактируйте программу из листинга 18.2 и запустите ее. Переменная event.pos определяет положение (координаты *x* и *y*) указателя. Именно в это положение мы перемещаем центр мяча. Обратите внимание, что перемещение мяча продолжается все время, пока мы двигаем мышью — именно столько продолжается событие MOUSEMOVE. Изменение параметра rect.center одновременно меняет положение по обеим координатам. Поэтому теперь движение мяча не ограничено вертикальным направлением. В отсутствие событий мыши — из-за того, что мышь перестала двигаться или указатель вышел за границы Pygame-окна, — мяч продолжит отскоки от границ окна.

Теперь сделаем так, чтобы на движение мяча влияло *только* перемещение указателя при нажатой кнопке мыши. Это действие называется *перетаскиванием* (dragging). Поскольку специального типа события для перетаскивания (вроде MOUSEDRAG) не существует, для получения нужного эффекта воспользуемся теми событиями, которые у нас есть.

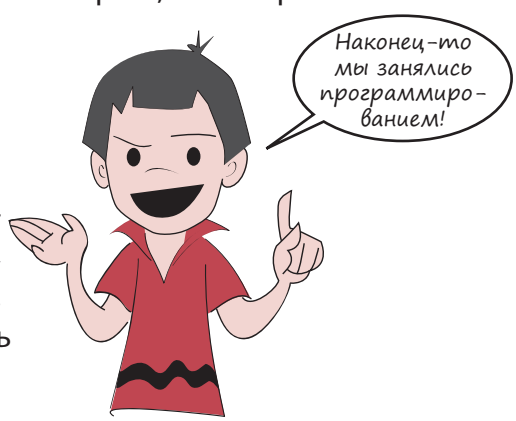
Какими признаками отличается перетаскивание? Это перемещение указателя при нажатой и удерживаемой кнопке мыши. Нажатие кнопки мыши соответствует событию MOUSEBUTTONDOWN, а в момент ее отпускания возникает событие MOUSEBUTTONUP. То есть нам нужно проследить за состоянием кнопки. Для этого потребуется дополнительная переменная; назовем ее held_down. Вот как в этом случае мог бы выглядеть код:

```
held_down = False
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            held_down = True
        elif event.type == pygame.MOUSEBUTTONUP:
            held_down = False
        elif event.type == pygame.MOUSEMOTION:
            if held_down:
                my_ball.rect.center = event.pos
```

Определяем, нажата ли кнопка мыши

Выполняется при перетаскивании

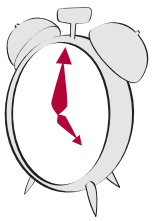
Наличие перетаскивания (перемещения указателя при нажатой кнопке мыши) распознается в последнем блоке elif. Внесите это изменение в цикл while ранее отредактированной версии листинга 18.2. Запустите программу, чтобы посмотреть, как она работает.



Ну уж нет, Картер, программированием мы занимаемся с самой первой главы! Но теперь, когда в дело пошли графика, спрайты и мышь, процесс стал куда более захватывающим. Я обещал тебе, что так будет. Просто нужно было выполнять мои рекомендации и на первом этапе изучить основы.

События таймера

Итак, вы уже познакомились с событиями клавиатуры и мыши. Но существует еще одна разновидность событий, особенно полезная в играх и моделировании — события


таймера. Таймер генерирует событие через определенные интервалы времени, совсем как ваш будильник. Запрограммированный и оставленный будильник каждый день звонит в одно и то же время.

Pygame-таймеры могут быть запрограммированы на любой интервал. После завершения этого интервала возникает событие, которое может быть выявлено циклом событий. К какому типу оно принадлежит? В данном случае речь идет о так называемом *пользовательском событии* (user event).

В Pygame существует набор предустановленных типов событий. Они имеют нумерацию, начинающуюся с 0, кроме того, всем им для простоты запоминания присвоены имена. С некоторыми из них, например `MOUSEBUTTONDOWN` и `KEYDOWN`, вы уже познакомились. В Pygame есть место и для событий, *определяемых пользователями*. Под конкретные действия они заранее не зарезервированы, поэтому вы можете делать с ними все, что вам заблагорассудится. Например применить их при работе с таймерами.

Установка таймера в Pygame происходит при помощи функции `set_timer()`:

```
pygame.time.set_timer(EVENT_NUMBER, interval)
```

Здесь `EVENT_NUMBER` — номер события, а `interval` — частота (в миллисекундах), с которой таймер будет генерировать событие.

Как определить значение переменной `EVENT_NUMBER`? В данном случае нам требуется значение, которое в библиотеке Pygame пока не зарезервировано под что-то другое. Pygame-модуль позволяет узнать список зарезервированных номеров. Введите этот код в интерактивном режиме:

```
>>> import pygame
>>> pygame.USEREVENT
24
```

Код сообщил нам, что в Pygame заняты номера от 0 до 23, поэтому первым, доступным для пользователей номером события будет 24. Значит, нужно выбрать число 24 или больше. Насколько большое число можно взять? Снова спросим об этом у Pygame-модуля:

```
>>> pygame.NUMEVENTS
32
```

Значение `NUMEVENTS` сообщает, что максимальное количество типов Pygame-событий равно 32 (от 0 до 31). То есть нужно выбрать число 24 или больше, но не больше 32. Поэтому настроим наш таймер таким образом:

```
pygame.time.set_timer(24, 1000)
```

Но если по каким-то причинам значение параметра USEREVENT изменится, наш код перестанет работать. Поэтому лучше записать его так:

```
pygame.time.set_timer(pygame.USEREVENT, 1000)
```

Для задания следующего пользовательского события мы задействуем уже номер USEREVENT + 1 и т. д. Значение 1000 в этом примере означает 1000 миллисекунд, то есть одну секунду. Это означает, что наш таймер будет срабатывать один раз в секунду. Вставим его в программу с движущимся мячом.

Как и раньше, это событие требуется для перемещения мяча вверх или вниз. Но так как в этом случае пользователь никак не будет влиять на движение объекта, заставим мяч отскакивать как от верхней и нижней, так и от боковых сторон окна. Полностью код программы на основе отредактированного листинга 18.2 показан в листинге 18.3.

Листинг 18.3. Перемещение мяча вверх и вниз при помощи события таймера

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
background = pygame.Surface(screen.get_size())
background.fill([255, 255, 255])

clock = pygame.time.Clock()
class Ball(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed
    def move(self):
        if self.rect.left <= screen.get_rect().left or \
                self.rect.right >= screen.get_rect().right:
            self.speed[0] = - self.speed[0]
        newpos = self.rect.move(self.speed)
        self.rect = newpos

my_ball = Ball('beach_ball.png', [10,0], [20, 20])
pygame.time.set_timer(pygame.USEREVENT, 1000)
direction = 1
running = True
```

Инициализируем все

Определение класса Ball

❶ *Эта строка продолжается*

Создаем экземпляры класса Ball

Создаем таймер: 1000 мс = 1 секунда

```
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.USEREVENT:
            my_ball.rect.centery = my_ball.rect.centery + (30*direction)
            if my_ball.rect.top <= 0 or \
            my_ball.rect.bottom >= screen.get_rect().bottom:
                direction = -direction
    clock.tick(30)
    screen.blit(background, (0, 0))
    my_ball.move()
    screen.blit(my_ball.image, my_ball.rect)
    pygame.display.flip()
pygame.quit()
```

Обработчик события для таймера

❶ *Эта строка продолжается*

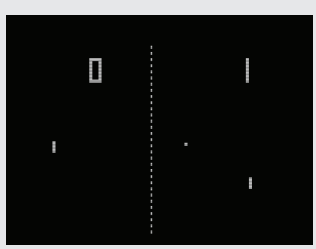
Перерисовываем все

Напоминаем, что символ \ отвечает за перенос строки ❶. С его помощью можно записать в две строки то, что в обычном виде должно было располагаться на одной. (Но ни в коем случае не ставьте пробел после символа \, иначе перенос не сработает.)

Сохранив и запустив программу из листинга 18.3, вы увидите, что мяч движется в разные стороны (от одного края окна к другому), раз в секунду смещаясь на 30 пикселов вверх или вниз. Движение вверх и вниз порождается событием таймера.

В СТАРЫЕ ДОБРЫЕ ВРЕМЕНА

Игра Pong — одна из первых аркадных видеоигр, в которую можно было играть дома. Изначально это была не программа, а набор микросхем! Ведь бытовых компьютеров в те древние времена еще не существовало. Игра подключалась к телевизору, а управление «ракеткой» осуществлялось при помощи специального пульта. Вот как выглядел игровой процесс на экране телевизора:



Малоизвестный факт:
Бабушка была не только мастером игры в Pong, но и чемпионом мира по пинг-понгу!

Время еще одной игры — PyPong

В этом разделе мы объединим ряд изученных нами тем, включая спрайты, распознавание столкновений и события, и создадим простую игру, напоминающую Pong.

Начнем с простой версии для одного игрока. Нам потребуются:

- шарик, который будет перебрасываться из стороны в сторону;
- ракетка, которой мы будем отбивать шарик;
- способ управления ракеткой;
- способ подсчета очков и их вывода на экран;
- способ слежения за числом «жизней» — количеством сыгранных партий.

В процессе создания программы мы будем действовать последовательно.

Шарик

Мяч, который мы до этого момента использовали, для игры в Pong несколько великоват. Нам нужно что-нибудь поменьше. Мы с Картером придумали вот такую необычную версию теннисного шарика.

Любой бы испугался, если бы его начали лупить ракеткой, перебрасывая туда-сюда!

В этой игре мы воспользуемся спрайтами, значит, первым делом следует создать спрайт для нашего шарика и получить его экземпляр. Для этой цели нам послужит класс `Ball` с методами `__init__()` и `move()`:

```
class MyBallClass(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > width:
            self.speed[0] = -self.speed[0]

        if self.rect.top <= 0 :
            self.speed[1] = -self.speed[1]
```

Шарик отскакивает от боковых сторон окна

Шарик отскакивает от верхней границы окна

Создавая экземпляр шарика, мы указываем, какой рисунок будет использоваться, с какой скоростью он должен перемещаться и в каком положении находиться в начальный момент:

```
myBall = MyBallClass('wackyball.bmp', ball_speed, [50, 50])
```

Кроме того, шарик следует добавить в группу, чтобы получить возможность распознавать его столкновения с ракеткой. Создадим группу, одновременно добавив в нее шарик:

```
ballGroup = pygame.sprite.Group(myBall)
```

РАКЕТКА

В качестве ракетки, в соответствии с принятыми в Pong традициями, используется обычный прямоугольник. У нас белый фон, поэтому прямоугольник сделаем черным. Снова создадим класс спрайтов и получим его экземпляр:

```
class MyPaddleClass(pygame.sprite.Sprite):
    def __init__(self, location):
        pygame.sprite.Sprite.__init__(self)
        image_surface = pygame.surface.Surface([100, 20])
        image_surface.fill([0,0,0])
        self.image = image_surface.convert()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location

paddle = MyPaddleClass([270, 400])
```

Создаем поверхность ракетки

Заливаем поверхность черным цветом

Преобразуем поверхность в изображение

Обратите внимание, что картинка не загружается из файла, а создается — прямоугольник черного цвета. Но так как спрайту нужен атрибут `image`, этот объект преобразуется в картинку методом `Surface.convert()`.

Ракетка может двигаться только влево или вправо. Перемещения вверх и вниз для нее невозможны. Сделаем так, чтобы координата *x* ракетки (отвечающая за движение слева направо) следовала за указателем мыши. Это даст пользователю возможность управлять ракеткой мышью. Так как данная операция выполняется непосредственно в цикле событий, отдельный метод `move()` ракетке не потребуется.

УПРАВЛЕНИЕ РАКЕТКОЙ

Как я уже упомянул, управление ракеткой будет осуществляться при помощи мыши. Мы воспользуемся событием `MOUSEMOTION`, то есть ракетка будет перемещаться в Pygame-окне вслед за указателем. Так как Pygame-модуль «видит» мышь только внутри окна, перемещения ракетки автоматически ограничатся его пределами. Сделаем так, чтобы за указателем следовал центр ракетки.

Вот как будет выглядеть соответствующий код:

```
elif event.type == pygame.MOUSEMOTION:
    paddle.rect.centerx = event.pos[0]
```

Здесь `event.pos` — это список значений [*x*, *y*] положения мыши. Соответственно, значение `event.pos[0]` даст нам координату *x* мыши в момент любого перемещения. Конечно, в момент, когда мышь достигнет левого или правого края окна, ракетка наполовину окажется вне его пределов, но в этом нет ничего страшного.

Осталось добавить только код *распознавания столкновений* между шариком и ракеткой. Именно так имитируется «удар» ракеткой по шарику. В момент столкновения мы просто меняем знак скорости по координате *y* (падающий вниз шарик, ударившись о ракетку, отскакивает от нее и начинает движение вверх).

Вот как будет выглядеть код:

```
if pygame.sprite.spritecollide(paddle, ballGroup, False):
    myBall.speed[1] = -myBall.speed[1]
```

Не забываем перерисовать сцену на каждой итерации цикла. Сложив все вместе, получим основу нашей игры, напоминающей Pong. Полностью код представлен в листинге 18.4.

Листинг 18.4. Первая версия игры PyPong

```
import pygame, sys
from pygame.locals import *

class MyBallClass(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > screen.get_width():
            self.speed[0] = -self.speed[0]

        if self.rect.top <= 0 :
            self.speed[1] = -self.speed[1]
```

Определение класса для шарика

Перемещаем шарик (вызывая его отскок от верхней и боковых границ окна)

```
class MyPaddleClass(pygame.sprite.Sprite):
    def __init__(self, location = [0,0]):
        pygame.sprite.Sprite.__init__(self)
        image_surface = pygame.surface.Surface([100, 20])
        image_surface.fill([0,0,0])
        self.image = image_surface.convert()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location

pygame.init()
screen = pygame.display.set_mode([640,480])
clock = pygame.time.Clock()
ball_speed = [10, 5]
myBall = MyBallClass('wackyball.bmp', ball_speed, [50, 50])
ballGroup = pygame.sprite.Group(myBall)
paddle = MyPaddleClass([270, 400])

running = True
while running:
    clock.tick(30)
    screen.fill([255, 255, 255])
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False
        elif event.type == pygame.MOUSEMOTION:
            paddle.rect.centerx = event.pos[0]

    if pygame.sprite.spritecollide(paddle, ballGroup, False):
        myBall.speed[1] = -myBall.speed[1]
    myBall.move()
    screen.blit(myBall.image, myBall.rect)
    screen.blit(paddle.image, paddle.rect)
    pygame.display.flip()
pygame.quit()
```

Определение класса для ракетки

Инициализируем Pygame-модуль, часы, шарик и ракетку

Начало основного цикла while

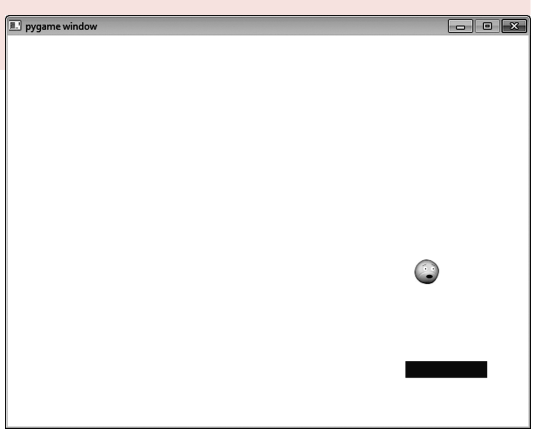
Двигаем ракетку вслед за движением мыши

Проверяем, ударился ли шарик о ракетку

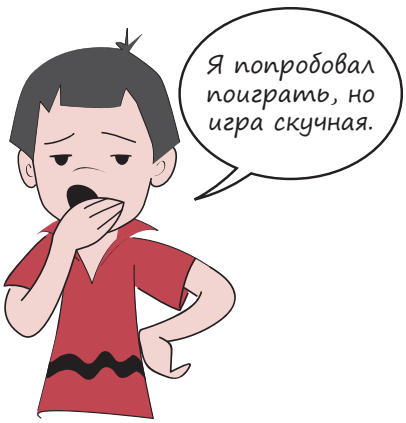
Двигаем шарик

Перерисовываем все

Вот как должно выглядеть окно программы после ее запуска:



Да, пока это не самая увлекательная игра, но ведь это наш первый опыт создания игр с помощью библиотеки Pygame. Давайка, Картер, наделим ее дополнительными возможностями.



ПОДСЧЕТ И ВЫВОД ОЧКОВ

Нам нужно проследить за двумя вещами: количеством жизней и числом набранных очков. Для простоты мы будем начислять одно очко за каждое касание шариком верхней границы окна. И дадим игроку три жизни.

Еще нам потребуется способ вывода очков на экран. В библиотеке Pygame за это отвечает модуль `font`. Вот как им пользоваться:

- создайте объект `font`, сообщив Pygame-модулю гарнитуру и размер шрифта;
- *визуализируйте* текст, передав строку объекту `font`, который возвращает новую *поверхность* с написанным на ней текстом;
- скопируйте эту поверхность на поверхность отображения.

В нашем случае в роли строки выступят набранные игроком очки (но сначала потребуется преобразовать их из типа `int` в тип `string`).

НОВЫЕ СЛОВА

В компьютерной графике *визуализировать* (render) означает представить некую информацию в удобном для наблюдения виде.

В листинг 18.4 непосредственно перед циклом событий (после строки `paddle = MyPaddleClass([270, 400])`) нужно будет поместить вот такой код:

```
score_font = pygame.font.Font(None, 50)
score_surf = score_font.render(str(score), 1, (0, 0, 0))
score_pos = [10, 10]
```

Создаем объект font

Визуализируем текст на поверхности score_surf

Задаем положение текста

Переменная `None` в первой строке должна сообщить Pygame-модулю, какой именно шрифт мы хотим использовать. Значение `None` заставляет Pygame-модуль задействовать шрифт, установленный по умолчанию.

Затем внутри цикла событий нам потребуется вот такая строка:

```
screen.blit(score_surf, score_pos)
```

Копируем поверхность с количеством очков в указанную позицию

Это позволит обновлять текст с количеством очков на каждой итерации цикла.

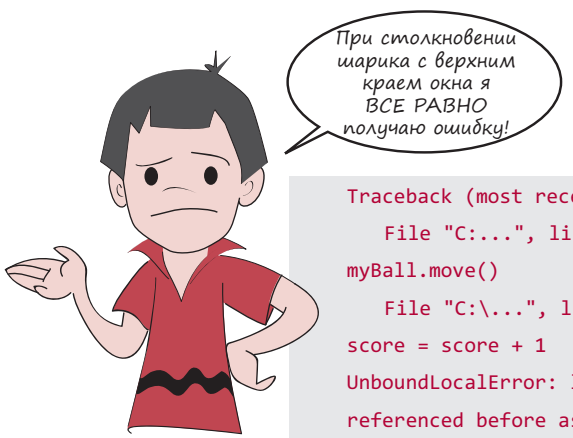
Разумеется, Картер, ведь мы же еще не создали переменную `score`. (Я только собирался этим заняться.) Добавь такую строчку перед кодом, отвечающим за создание объекта `font`:

```
score = 0
```

Как же теперь нам отслеживать количество очков... Мы уже определили момент столкновения шарика с верхней границей окна в его методе `move()`, чтобы смоделировать отскок. Поэтому добавим пару строк:

```
            if self.rect.top <= 0 :
    self.speed[1] = -self.speed[1]
    score = score + 1
    score_surf = score_font.render(str(score), 1, (0, 0, 0))
```

Две новые строки




```
Traceback (most recent call last):
   File "C:...", line 59, in <module>
myBall.move()
   File "C:\...", line 24, in move
score = score + 1
UnboundLocalError: local variable 'score'
referenced before assignment
```

Ой! Мы совсем забыли про *пространства имен*. Помните это длинное объяснение из главы 15? Теперь вы видите реальный пример применения данной концепции. У нас есть переменная `score`, ею мы пытаемся воспользоваться внутри метода `move()` класса `Ball`. Класс ищет локальную переменную `score`, которой попросту не существует. А на самом деле нам нужна уже созданная нами глобальная переменная. Это и следует объяснить методу `move()` следующим образом:

```
def move(self):
    global score
```

Еще нужно сделать глобальными переменные `score_font` (шрифт, которым пишется сумма очков) и `score_surf` (поверхность, содержащая визуализированный текст), так как они обновляются в методе `move()`. Поэтому код должен выглядеть следующим образом:

```
def move(self):
    global score, score_font, score_surf
```

А вот *теперь* все должно работать! Попробуйте, и вы увидите свои очки в левом верхнем углу окна. Они будут увеличиваться при каждом касании шарика верхней границы окна.

Количество жизней

Теперь нужно добавить в игру механизм учета количества жизней. Пока что пропущенный шарик просто падал на пол и пропадал из виду. Мы же хотим дать игроку три попытки, поэтому создадим переменную `lives` и присвоим ей значение 3:

```
lives = 3
```

После того как пропущенный игроком шарик упадет на пол, нужно вычесть из переменной `lives` единицу, подождать пару секунд и возобновить игру с новым шариком:

```
if myBall.rect.top >= screen.get_rect().bottom:
    lives = lives - 1
    pygame.time.delay(2000)
    myBall.rect.topleft = [50, 50]
```

Этот код следует поместить внутрь цикла `while`. Кстати, атрибут `myBall.rect` для шарика и метод `get_rect()` для поверхности `screen` мы использовали по следующим причинам:

- `myBall` — это спрайт, а у спрайтов есть атрибут `rect`;
- `screen` — это поверхность, а значит, атрибут `rect` у нее отсутствует. Найти прямоугольник, включающий в себя поверхность, позволяет функция `get_rect()`.

Если вы внесете эти изменения и запустите программу, то увидите, что игрок теперь обладает тремя жизнями.

Счетчик жизней

Большинство игр с ограниченным количеством жизней у игрока дает возможность видеть оставшееся число жизней. Добавим такую возможность и в нашу игру.

Проще всего сделать количество шариков равным количеству оставшихся жизней. Эту информацию можно поместить в правый верхний угол окна. Вот формула, по которой внутри цикла `for` будет создаваться счетчик жизней:

```
for i in range (lives):
    width = screen.get_rect().width
    screen.blit(myBall.image, [width - 40 * i, 20])
```

Этот код следует поместить внутрь основного цикла `while` непосредственно перед циклом событий (после строки `screen.blit(score_text, textpos)`).

Конец игры

Нам осталось добавить сообщение об окончании игры, появляющееся после того, как игрок уронит последний шарик. Создадим пару объектов `font`, которые будут включать в себя это сообщение и окончательный счет игрока, визуализируем их (создав поверхности с текстом на них) и скопируем эти поверхности на поверхность отображения.

Кроме того, нужно, чтобы после последнего раунда игры движение шарика останавливалось. Для этого создадим переменную `done`, которая будет сообщать о завершении игры. Вот как будет выглядеть наш код (его следует поместить в основной цикл `while`):

```
if myBall.rect.top >= screen.get_rect().bottom:
    lives = lives - 1                          # Вычитаем одну жизнь при падении шарика на пол
    if lives == 0:
        final_text1 = "Игра окончена"
        final_text2 = "Ваш счет: " + str(score)
        ft1_font = pygame.font.Font(None, 70)
        ft1_surf = ft1_font.render(final_text1, 1, (0, 0, 0))
        ft2_font = pygame.font.Font(None, 50)
        ft2_surf = ft2_font.render(final_text2, 1, (0, 0, 0))
        screen.blit(ft1_surf, [screen.get_width()/2 - \      # Располагаем текст по центру окна
                    ft1_surf.get_width()/2, 100])            # Символы переноса строки
        screen.blit(ft2_surf, [screen.get_width()/2 - \
                    ft2_surf.get_width()/2, 200])
        pygame.display.flip()
        done = True
    else: #ждем 2 секунды и подаем очередной шарик
        pygame.time.delay(2000)
        myBall.rect.topleft = [(screen.get_rect().width) - 40*lives, 20]
```

Соберем все вместе и получим для окончательной версии игры PyPong листинг 18.5.

Листинг 18.5. Окончательная версия игры PyPong

```
import pygame, sys

class MyBallClass(pygame.sprite.Sprite):                 # Определяем класс для шарика
    def __init__(self, image_file, speed, location):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed
```

```
    def move(self):
        global score, score_surf, score_font
        self.rect = self.rect.move(self.speed)
        if self.rect.left < 0 or self.rect.right > screen.get_width():
            self.speed[0] = -self.speed[0]

        if self.rect.top <= 0 :
            self.speed[1] = -self.speed[1]
            score = score + 1
            score_surf = score_font.render(str(score), 1, (0, 0, 0))

class MyPaddleClass(pygame.sprite.Sprite):
    def __init__(self, location = [0,0]):
        pygame.sprite.Sprite.__init__(self)
        image_surface = pygame.surface.Surface([100, 20])
        image_surface.fill([0,0,0])
        self.image = image_surface.convert()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location

pygame.init()
screen = pygame.display.set_mode([640,480])
clock = pygame.time.Clock()
myBall = MyBallClass('wackyball.bmp', [10,5], [50, 50])
ballGroup = pygame.sprite.Group(myBall)
paddle = MyPaddleClass([270, 400])
lives = 3
score = 0
score_font = pygame.font.Font(None, 50)
score_surf = score_font.render(str(score), 1, (0, 0, 0))
score_pos = [10, 10]
done = False
running = True
while running:
    clock.tick(30)
    screen.fill([255, 255, 255])
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEMOTION:
            paddle.rect.centerx = event.pos[0]
    if pygame.sprite.spritecollide(paddle, ballGroup, False):
        myBall.speed[1] = -myBall.speed[1]
    myBall.move()
```

Определяем класс для шарика

Определяем класс для ракетки

Инициализируем все

Создаем объект для шрифта

Запускаем основную часть программы (цикл while)

Обнаруживаем перемещения мыши, двигающие ракетку

Обнаруживаем столкновения шарика с ракеткой

Двигаем шарик

```
    if not done:
        screen.blit(myBall.image, myBall.rect)
        screen.blit(paddle.image, paddle.rect)
        screen.blit(score_surf, score_pos)
        for i in range (lives):
            width = screen.get_width()
            screen.blit(myBall.image, [width - 40 * i, 20])
        pygame.display.flip()
    if myBall.rect.top >= screen.get_rect().bottom:
        lives = lives - 1
        if lives == 0:
            final_text1 = "Игра окончена"
            final_text2 = "Ваш счет: " + str(score)
            ft1_font = pygame.font.Font(None, 70)
            ft1_surf = ft1_font.render(final_text1, 1, (0, 0, 0))
            ft2_font = pygame.font.Font(None, 50)
            ft2_surf = ft2_font.render(final_text2, 1, (0, 0, 0))
            screen.blit(ft1_surf, [screen.get_width()/2 - \
                        ft1_surf.get_width()/2, 100])
            screen.blit(ft2_surf, [screen.get_width()/2 - \
                        ft2_surf.get_width()/2, 200])
            pygame.display.flip()
            done = True
        else:
            pygame.time.delay(2000)
            myBall.rect.topleft = [50, 50]
pygame.quit()
```

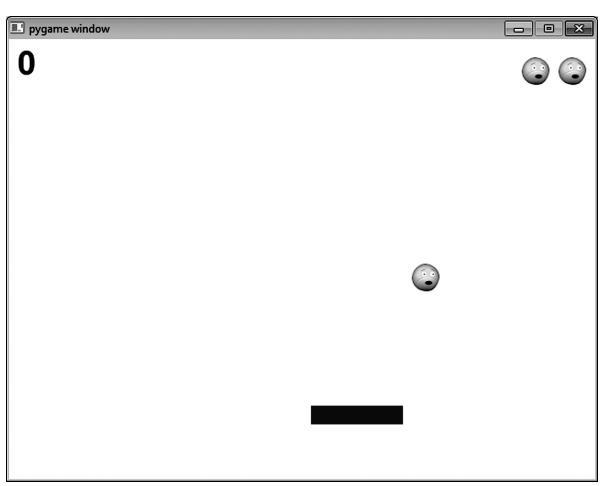
Перерисовываем все

Уменьшаем счетчик жизней при падении шарика на пол

Создаем и выводим на экран текст с окончательным счетом

Начинаем новый раунд после 2-секундной задержки

После запуска кода из листинга 18.5 вы получите примерно такой результат:



Возможно, при наборе кода вы обратили внимание, что он состоит из примерно 75 строк (плюс пустые строки). Это самая большая программа из всех вами созданных. Она содержит множество элементов, хотя после запуска выглядит совсем простой.

В следующей главе вы познакомитесь со звуками из библиотеки Pygame и добавите к нашей игре PyPong звуковое сопровождение.

Что мы узнали

В этой главе мы познакомились с:

- событиями;
- циклом событий библиотеки Pygame;
- обработкой событий;
- событиями клавиатуры;
- событиями мыши;
- событиями таймера (и типами пользовательских событий);
- атрибутом `pygame.font` (предназначенным для добавления текста в Pygame-программы).

Также мы собрали все эти концепции воедино и написали игру!

Проверь себя

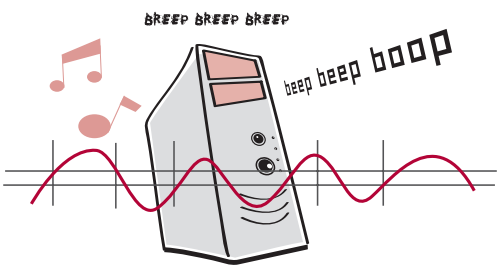
1. На какие два вида событий может реагировать программа?
2. Как называется фрагмент кода, реагирующий на событие?
3. Как называется тип события, который в Pygame возникает при нажатии клавиши?
4. Какой атрибут события `MOUSEMOVE` сообщает, в каком месте окна располагается указатель мыши?
5. Как определить следующий доступный номер Pygame-события (например если вам нужно добавить пользовательское событие)?
6. Как создать таймер для генерации Pygame-событий таймера?
7. Какой вид объектов используется для вывода текста в Pygame-окне?
8. Назовите три этапа вывода текста в Pygame-окне?

Эксперименты

1. Вы не заметили ничего странного в момент, когда вместо верхней поверхности ракетки шарик ударяется о ее боковую поверхность? Он некоторое время как бы прыгает по ракетке по направлению к ее середине. Как объяснить, почему так происходит? Может, имеет смысл убрать этот эффект из программы? Попробуйте сделать это самостоятельно, не заглядывая в ответы.
2. Перепишите программу (см. листинг 18.4 или 18.5), сделав отскоки шарика более хаотичными. Можете изменить способ отскока от ракетки или от границ окна, случайным образом выбирать величину скорости или внести любое другое изменение, пришедшее вам в голову. (С функциями `random.randint()` и `random.random()` вы познакомились в главе 15, а значит, вы уже умеете генерировать как целые, так и десятичные случайные числа.)

ГЛАВА 19

Звуки



В предыдущей главе вашими руками была создана первая графическая игра PyPong, на примере которой мы рассмотрели процесс применения графики, спрайтов, столкновений, анимации и событий. Теперь пришло время добавить в мозаику еще один фрагмент — звук. Он используется во всех видеоиграх и во множестве прочих программ, делая работу с программным обеспечением более интересной и приятной.

Звук может являться как вводимой, так и выводимой информацией. В первом случае требуется подсоединить к компьютеру микрофон или другой источник звука, и программа будет его записывать или выполнять другие действия (например отправит через Интернет). Но намного чаще встречается второй случай, и именно его мы будем рассматривать в этой главе. Вы узнаете о способах воспроизведения музыки и звуковых эффектов и добавите их в свои программы, в частности в игру PyPong.

Модуль mixer

Добавление звуковой дорожки относится к операциям, с которыми, как и с графикой, может возникнуть сложность из-за различий в программном и аппаратном обеспечении, отвечающем за воспроизведение звука. Поэтому для простоты мы снова обратимся к библиотеке Pygame.

Модуль для работы со звуком в библиотеке Pygame называется `pygame.mixer`. В обычном мире устройство для сведения разных звуковых сигналов воедино называется *микшером* (mixer), и именно его название было взято в качестве имени Pygame-модуля.

Возникновение звуков

Программы могут создавать звуки двумя основными способами. Во-первых, звуки можно *синтезировать*, то есть создавать «с нуля», генерируя звуковые волны различной длины

и громкости. Во-вторых, можно *воспроизвести* записанный звук. Это может быть звуковой фрагмент с компакт-диска, звуковой файл в формате MP3 или звук любого другого типа.

В этой книге будет рассматриваться только процесс воспроизведения звуков. Создание собственной музыки — слишком большая тема, которой мы касаться не будем. При желании можете ознакомиться с программами генерации звуков самостоятельно.

Воспроизведение звука

Процесс воспроизведения заключается в том, что вы берете файл, находящийся на жестком диске вашего компьютера (или на компакт-диске, или где-то в Сети), и превращаете его в звуки, которые можно услышать через колонки или наушники. Существует множество типов звуковых файлов. Вот наиболее распространенные из них:

- *Wav-файлы* — файлы с расширением ***.wav***, например ***hello.wav***;
- *MP3-файлы* — файлы с расширением ***.mp3***, например ***mySong.mp3***;
- *WMA-файлы* (WMA расшифровывается как Windows Media Audio) — файлы с расширением ***.wma***, например ***someSong.wma***;
- *Vorbis-*, или *ogg-файлы* — файлы с расширением ***.ogg***, например ***yourSong.ogg***.

В этой главе мы будем работать с wav- и mp3-файлами. Все они находятся в папке ***\sounds***, куда была установлена программа HelloWorld. К примеру, на компьютере с операционной системой Windows это будет папка ***c:\Program Files\HelloWorld\examples\sounds***.

Существует два способа вставки в программу звуковых файлов. Можно скопировать его в папку, где уже находится наша программа. Именно здесь интерпретатор Python будет его искать, поэтому достаточно указать имя файла:

```
sound_file = "my_sound.wav"
```

Если же звуковой файл находится в другой папке, следует указать полный маршрут доступа к нему:

```
sound_file = "c:\Program Files\HelloWorld\sounds\my_sound.wav"
```

В приведенных в книге примерах предполагается, что звуковые файлы скопированы в те папки, в которых сохранены соответствующие программы. Именно поэтому в коде будет фигурировать только имя файла, а не путь доступа к нему. Если вы не копировали звуковые файлы в папки с соответствующими программами, укажите вместо имени файла его полный адрес.

Начало работы с модулем pygame.mixer

Для воспроизведения звука нужно *инициализировать* модуль `pygame.mixer`. Помните, что такое *инициализация*? Это подготовка элемента к использованию.

В случае модуля `pygame.mixer` она выполняется очень просто. После инициализации этого Pygame-модуля достаточно добавить такую строку:

```
pygame.mixer.init()
```

В итоге в начале программы, в которой задействована библиотека Pygame, должен оказаться вот такой код:

```
import pygame
pygame.init()
pygame.mixer.init()
```

Теперь все готово к воспроизведению звука. В наших программах будут использоваться звуки двух типов. Во-первых, звуковые эффекты из клипов. Они, как правило, короткие и хранятся в wav-файлах. Для поддержания звуков этого типа в модуле `pygame.mixer` применяется объект `Sound`:

```
splat = pygame.mixer.Sound("splat.wav")
splat.play()
```

Во-вторых, мы будем использовать музыку. Она чаще всего хранится в mp3-, wma- или ogg-файлах. Для их воспроизведения внутри Pygame-модуля `mixer` применяется другой модуль — `music`. Вот пример его использования:

```
pygame.mixer.music.load("bg_music.mp3")
pygame.mixer.music.play()
```

Этот код проигрывает песню (или другое содержимое музыкального файла) один раз.

Давайте попробуем воспроизвести какой-нибудь звук. Например: «Плюх!».

Нам нужен цикл `while`, который будет отвечать за работу Pygame-программы. В данном случае мы не будем рисовать графику, тем не менее для работы любой Pygame-программы требуется окно. Кроме того, в ряде систем инициализация модуля `mixer` занимает некоторое время. И слишком рано начав воспроизведение звука, вы рискуете услышать его только частично или не услышать совсем. Поэтому мы подождем, пока модуль

mixer не будет готов к работе. Для этого нам потребуется примерно такой код, который представлен в листинге 19.1.

Листинг 19.1. Добавление звука к Pygame-программе

```
import pygame, sys
pygame.init()
pygame.mixer.init()

screen = pygame.display.set_mode([640,480])
pygame.time.delay(1000)

splat = pygame.mixer.Sound("splat.wav")
splat.play()

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Инициализируем библиотеку Pygame и модуль mixer

Создаем Pygame-окно

Ждем одну секунду, пока инициализируется модуль mixer

Создаем звуковой объект

Воспроизводим звук

Обычный цикл Pygame-событий

Запустите программу, чтобы посмотреть, как она работает.

Теперь попробуем воспользоваться модулем mixer.music для воспроизведения музыки. Для этого в листинге 19.1 нужно изменить всего пару строк (листинг 19.2).

Листинг 19.2. Воспроизведение музыки

```
import pygame, sys
pygame.init()
pygame.mixer.init()

screen = pygame.display.set_mode([640,480])
pygame.time.delay(1000)

pygame.mixer.music.load("bg_music.mp3")
pygame.mixer.music.play()

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

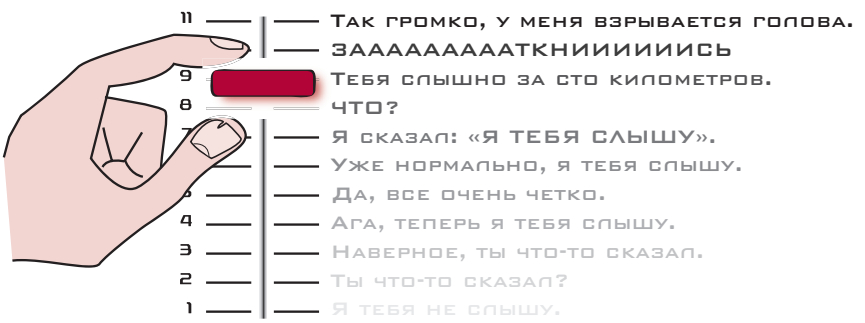
Эти две строки были изменены

Запустите программу и убедитесь, что музыка действительно играет.

Не знаю, как вам, но мне она кажется слишком громкой. Пришлось уменьшить громкость звука в динамиках. Думаю, нам нужно обсудить способы контроля громкости в программах.

Управление громкостью

Для управления громкостью звука в операционной системе существуют специальные регуляторы. В Windows он обозначен значком в виде рупора в области уведомлений панели задач. Именно здесь осуществляется управление громкостью всех издаваемых компьютером звуков. Кроме того, можно воспользоваться ручкой регулировки громкости на самих динамиках.



А еще можно управлять уровнем звука, который Pygame-модуль отправляет на звуковую карту вашего компьютера.

И особенно хорошо то, что каждым звуком можно управлять по отдельности. При желании ничто не мешает вам сделать музыку потише, а «Плюх!» погромче.

Для музыки в этом случае применяется метод `pygame.mixer.music.set_volume()`. Если же речь идет о звуках, нужно воспользоваться методом `set_volume()` для каждого из звуковых объектов. В первом примере звуковой объект носил имя `splat`, поэтому мы писали `splat.set_volume()`. Громкость звука обозначается числом с плавающей точкой в диапазоне от 0 до 1, к примеру 0.5 означает 50 %, или половину максимальной громкости.

Теперь давайте добавим музыку и звук в одну программу. Пусть у нас играет песенка, после завершения которой должен раздаться «Плюх!». Кроме того, нужно будет слегка уменьшить громкость. Уровень громкости музыки сделаем равным 30 %, в то время как громкость «Плюха» пусть составит 50 %. Код показан в листинге 19.3.

Листинг 19.3. Музыка и звук с регулировкой громкости

```
import pygame, sys
pygame.init()
pygame.mixer.init()

screen = pygame.display.set_mode([640,480])
pygame.time.delay(1000)

pygame.mixer.music.load("bg_music.mp3")
pygame.mixer.music.set_volume(0.30)
pygame.mixer.music.play()
splat = pygame.mixer.Sound("splat.wav")
splat.set_volume(0.50)
splat.play()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
```

Регулируем громкость музыки

Регулируем громкость звукового эффекта

Запустите программу и посмотрите, как она работает. Картер заметил, что программа, начав воспроизведение музыки, сразу же переходит к воспроизведению имитации всплеска. Это связано с тем, что музыка часто используется в качестве фона, а в этом случае программа не ждет завершения воспроизведения одного файла, перед тем как запустить следующий звуковой фрагмент. Посмотрим, как можно исправить ситуацию, добившись нужной нам последовательности звуков.

ФОНОВАЯ МУЗЫКА

Фоновой называется музыка, звучащая во время игрового процесса. Поэтому после ее запуска Pygame-модуль должен быть готов к следующим операциям, например к перемещению спрайтов или проверке клавиатурного ввода. Окончания песни он ждать не должен.

Но как быть в ситуации, когда вам нужно дождаться завершения музыкального фрагмента? Например, чтобы после этого начать другую музыку или воспроизвести какой-то звук (собственно, как мы и хотим сделать). Как узнать, что музыка закончилась? Это может сообщить вам Pygame-модуль `mixer.music`: достаточно спросить у него, занят ли он воспроизведением музыки. Давайте попробуем так сделать.

Для этого нам потребуется функция `get_busy()`. Если воспроизведение еще не закончено, она возвращает значение `True`, в противном случае — значение `False`. На этот раз заставим программу проиграть музыкальный фрагмент, затем воспроизведем звуковой эффект, а потом автоматически завершим работу (листинг 19.4).

Листинг 19.4. Ожидание окончания песни

```
import pygame, sys
pygame.init()
pygame.mixer.init()

screen = pygame.display.set_mode([640,480])
pygame.time.delay(1000)

pygame.mixer.music.load("bg_music.mp3")
pygame.mixer.music.set_volume(0.3)
pygame.mixer.music.play()
splat = pygame.mixer.Sound("splat.wav")
splat.set_volume(0.5)
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    if not pygame.mixer.music.get_busy():
        splat.play()
        pygame.time.delay(1000)
        running = False
pygame.quit()
```

Проверяем, завершилось ли воспроизведение музыкального фрагмента

Секунду ждем завершения «всплеска»

Этот код один раз проигрывает музыку, затем воспроизводит звуковой эффект, после чего программа завершает свою работу.

Зацикленная музыка

Музыка, которую вы собираетесь использовать в качестве фоновой, должна звучать, пока работает программа. Модуль `music` легко позволяет это сделать. Можно проиграть музыкальный фрагмент нужное число раз:

```
pygame.mixer.music.play(3)
```

В данном случае песня прозвучит три раза.

А можно просто зациклить проигрывание файла, передав специальный аргумент –1:

```
pygame.mixer.music.play(-1)
```

В этом случае музыкальный фрагмент будет циклически повторяться все время, пока работает Pygame-программа. (Это может быть не только –1, но и любое другое отрицательное число.)

В сопроводительной документации к Pygame-модулю написано, что команда `pygame.mixer.music.play(3)` воспроизводит музыкальный файл четыре раза (первый раз плюс три повторения). Но на самом деле музыкальный фрагмент прозвучит всего три раза.

Добавление звуков к игре PyPong

Теперь, когда вы знаете, как добавить в программу звук, пришла пора переписать нашу игру PyPong. Для начала добавим звук столкновения ракетки с шариком. Этот момент нам уже известен, так как ранее распознавание столкновений использовалось для изменения направления движения шарика после соприкосновения с ракеткой. Помните код из листинга 18.5?

```
if pygame.sprite.spritecollide(paddle, ballGroup, False):
    myBall.speed[1] = -myBall.speed[1]
```

Осталось добавить сюда код воспроизведения звука. Кроме того, в начало программы следует поместить строку `pygame.mixer.init()` и создать готовый к использованию звуковой объект:

```
hit = pygame.mixer.Sound("hit_paddle.wav")
```

Еще имеет смысл позаботиться о не слишком большой громкости звука:

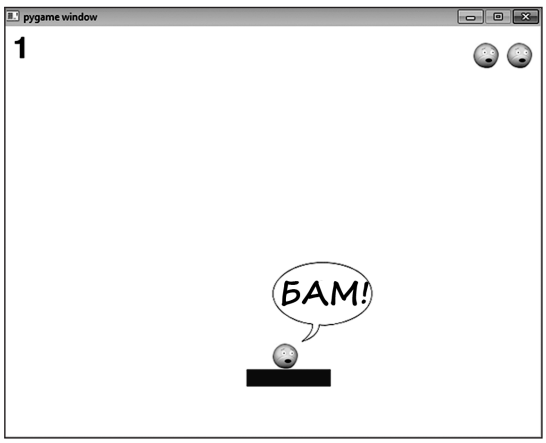
```
hit.set_volume(0.4)
```

А звук должен прозвучать после столкновения шарика с ракеткой:

```
if pygame.sprite.spritecollide(paddle, ballGroup, False):
    myBall.speed[1] = -myBall.speed[1]
    hit.play()
```

Воспроизводим звук

Добавим все это в программу PyPong из листинга 18.5. Удостоверьтесь, что файл *hit_paddle.wav* скопирован в папку с этой программой. Теперь все столкновения шарика с ракеткой будут сопровождаться звуком.



Дополнительные звуки

Теперь, когда при столкновении шарика с ракеткой раздается звук `hit`, добавим в игру дополнительные звуки. Они должны сопровождать:

- столкновение шарика со стеной;
- столкновение шарика с потолком, при котором игрок зарабатывает очко;
- падение пропущенного игроком шарика на пол;
- возобновление игры после промаха игрока;
- завершение игры.

Первым делом нужно создать объект для каждого из перечисленных случаев. Соответствующий код может располагаться в любом месте после строки `pygame.mixer.init()`, но до цикла `while`:

```
hit_wall = pygame.mixer.Sound("hit_wall.wav")
hit_wall.set_volume(0.4)
get_point = pygame.mixer.Sound("get_point.wav")
get_point.set_volume(0.2)
splat = pygame.mixer.Sound("splat.wav")
splat.set_volume(0.6)
new_life = pygame.mixer.Sound("new_life.wav")
new_life.set_volume(0.5)
bye = pygame.mixer.Sound("game_over.wav")
bye.set_volume(0.6)
```

Уровень громкости я в каждом случае подбирал на слух. Вы можете менять этот параметр по своему усмотрению. Не забывайте копировать звуковые файлы в папку с кодом программы. Все файлы можно взять из папки *\examples\sounds* или с сайта книги.

Теперь к местам возникновения звуков нужно применить метод `play()`. Звук `hit_wall` должен возникать при ударе шарика о боковую сторону окна. Эта ситуация распознается в методе `move()` шарика, после чего мы меняли знак *x*-скорости (заставляя шарик «отскакивать» от стен). В исходном листинге 18.5 это была строка 14:

```
if self.rect.left < 0 or self.rect.right > screen.get_width():
```

Значит, после изменения направления движения можно воспроизвести звук. Код будет выглядеть следующим образом:

```
if self.rect.left < 0 or self.rect.right > screen.get_width():
    self.speed[0] = -self.speed[0]
    hit_wall.play()
```

Воспроизводим звук при ударе шарика о боковую стенку

Аналогично можно поступить со звуком `get_point`. Чуть ниже в методе `move()` шарика распознается соударение с верхней границей окна. В этот момент мы моделируем отскок вниз и добавляем очко к счету игрока. Теперь мы добавим сюда звук. Вот как будет выглядеть новый вариант кода:

```
if self.rect.top <= 0 :
    self.speed[1] = -self.speed[1]
    points = points + 1
    score_text = font.render(str(points), 1, (0, 0, 0))
    get_point.play()
```

Воспроизводим звук получения дополнительного очка

Вставьте эти фрагменты в программу и посмотрите, как она работает.

Далее нужно озвучить момент пропуска шарика, после которого у игрока сгорает одна жизнь.

Это событие распознается в основном цикле `while`, в строке 63 исходного листинга 18.5 (`if myBall.rect.top >= screen.get_rect().bottom:`). Остается только добавить вот такую строку:

```
if myBall.rect.top >= screen.get_rect().bottom:
    splat.play()
    # утрата жизни при падении шарика на пол
    lives = lives - 1
```

Воспроизводим звук падения шарика на пол и сгорания жизни

Еще можно добавить звук начала новой партии. Это происходит в последних трех строках листинга 18.5, в блоке `else`. В данном случае слегка разнесем во времени звуковой эффект и начало новой партии:

```
else:
    pygame.time.delay(1000)
    new_life.play()
    myBall.rect.topleft = [50, 50]
    screen.blit(myBall.image, myBall.rect)
    pygame.display.flip()
    pygame.time.delay(1000)
```

Вместо двухсекундного ожидания (как это было в исходной программе) мы подождем секунду (1000 миллисекунд), воспроизведем звук, подождем еще одну секунду и только затем начнем новую партию. Запустите программу и посмотрите, как это выглядит на практике.

Осталось добавить всего один звуковой эффект, возникающий в момент завершения игры. Этот момент наступает в строке 65 листинга 18.5 (`if lives == 0:`). Добавим сюда строку со звуком `bye`:

```
if lives == 0:
    bye.play()
```

Запустите программу и посмотрите, как она работает.

Ой! Мы кое о чем забыли. Ведь код звуков `bye` и `splat` находится в основном цикле `while`, который работает, пока не закроется Pygame-окно. Нужно сделать так, чтобы эти звуки проигрывались только один раз.

В этом случае можно воспользоваться переменной `done`, сообщающей об окончании игры. Вот отредактированный вариант кода:

```
if myBall.rect.top >= screen.get_rect().bottom:
    if not done:
        splat.play()
    lives = lives - 1
    if lives == 0:
        if not done:
            bye.play()
```

Гарантируем однократное воспроизведение звука

Запустите программу и убедитесь в корректности ее работы.

Странно… Надо подумать. Переменная done сообщает нам об окончании игры, благодаря этому мы определяем момент воспроизведения финального звукового эффекта и вывода количества набранных очков. А что в это время делает шарик?

Несмотря на падение на пол, шарик продолжает двигаться! Ничто не препятствует его движению вниз, а значит, координата *y* будет увеличиваться. Шарик находится «под» нижним краем экрана, поэтому мы не можем его увидеть, но мы его по-прежнему слышим! Он продолжает движение, периодически «отскакивая» от краев окна, когда его координата *x* становится слишком большой или маленькой. Все это происходит в методе `move()`, который работает до тех пор, пока не завершится цикл `while`.

Как поступить в подобной ситуации? Существуют разные варианты действий.

- Можно прекратить движение шарика, в момент завершения игры присвоив его скорости значение [0,0].
- Можно проверить положение шарика, и если выяснится, что он оказался под нижней границей окна, остановить звук `hit_wall`.
- Можно проверить переменную `done` и остановить звук `hit_wall`, если игра уже закончена.

Все эти варианты работают, но я выбрал второй. Вы можете сделать другой выбор. Даю вам возможность самостоятельно отредактировать код и решить проблему.

Добавление музыки в игру PyPong

Игра почти готова, осталось только добавить музыку. Нужно загрузить музыкальный файл, задать громкость и начать воспроизведение. Музыка должна звучать на протяжении всей игры, поэтому воспользуется значением –1:

```
pygame.mixer.music.load("bg_music.mp3")
pygame.mixer.music.set_volume(0.3)
pygame.mixer.music.play(-1)
```

Этот фрагмент кода можно поместить в любое место перед основным циклом `while`. Он инициирует воспроизведение музыки. Теперь нужно, чтобы в конце игры музыка останавливалась. Для этого воспользуемся методом `fadeout()` модуля `pygame.mixer.music`. Он постепенно уменьшает громкость звука до нуля. Достаточно указать, сколько времени должен занять этот процесс:

```
pygame.mixer.music.fadeout(2000)
```

Аргумент 2000 миллисекунд составляет 2 секунды. Эту строку нужно добавить туда, где уже находится строка `done = True`. (Неважно, выше или ниже.)

Теперь в программу добавлены как звуковые эффекты, так и фоновая музыка. Запустите ее и послушайте, как все звучит! На случай если вы захотите посмотреть на программу в целом, я добавил свою окончательную версию кода в листинг 19.5. Проследите за тем, чтобы файл *wackyball.bmp*, как и звуковые файлы, был скопирован в папку с программой.

Листинг 19.5. Игра PyPong со звуковыми эффектами и фоновой музыкой

```
import pygame, sys

class MyBallClass(pygame.sprite.Sprite):
    def __init__(self, image_file, speed, location = [0,0]):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image_file)
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
        self.speed = speed

    def move(self):
    global points, score_text
    self.rect = self.rect.move(self.speed)
    if self.rect.left < 0 or self.rect.right > screen.get_width():
        self.speed[0] = -self.speed[0]
        if self.rect.top < screen.get_height():
            hit_wall.play()

    if self.rect.top <= 0 :
        self.speed[1] = -self.speed[1]
        points = points + 1
        score_text = font.render(str(points), 1, (0, 0, 0))
        get_point.play()

class MyPaddleClass(pygame.sprite.Sprite):
    def __init__(self, location = [0,0]):
        pygame.sprite.Sprite.__init__(self)
        image_surface = pygame.surface.Surface([100, 20])
        image_surface.fill([0,0,0])
        self.image = image_surface.convert()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location
pygame.init()
pygame.mixer.init()
```

Воспроизводим звук при ударе шарика о стену

Воспроизводим звук при ударе шарика о потолок (игрок зарабатывает очко)

Инициализируем звуковой Pygame-модуль

```
pygame.mixer.music.load("bg_music.mp3")
pygame.mixer.music.set_volume(0.3)
pygame.mixer.music.play(-1)
hit = pygame.mixer.Sound("hit_paddle.wav")
hit.set_volume(0.4)
new_life = pygame.mixer.Sound("new_life.wav")
new_life.set_volume(0.5)
splat = pygame.mixer.Sound("splat.wav")
splat.set_volume(0.6)
hit_wall = pygame.mixer.Sound("hit_wall.wav")
hit_wall.set_volume(0.4)

get_point = pygame.mixer.Sound("get_point.wav")
get_point.set_volume(0.2)
bye = pygame.mixer.Sound("game_over.wav")
bye.set_volume(0.6)
screen = pygame.display.set_mode([640,480])
clock = pygame.time.Clock()
myBall = MyBallClass('wackyball.bmp', [12,6], [50, 50])
ballGroup = pygame.sprite.Group(myBall)
paddle = MyPaddleClass([270, 400])
lives = 3
points = 0

font = pygame.font.Font(None, 50)
score_text = font.render(str(points), 1, (0, 0, 0))
textpos = [10, 10]
done = False

running = True
while running:
    clock.tick(30)
    screen.fill([255, 255, 255])
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEMOTION:
            paddle.rect.centerx = event.pos[0]

    if pygame.sprite.spritecollide(paddle, ballGroup, False):
        hit.play()
        myBall.speed[1] = -myBall.speed[1]
    myBall.move()

    if not done:
        screen.blit(myBall.image, myBall.rect)
```

Загружаем музыкальный файл

Задаем громкость музыки

Начинаем воспроизведение музыки и повторяем ее до бесконечности

Создаем звуковой объект, загружаем звуки и задаем уровень громкости каждого из них

Воспроизводим звук при столкновении шарика с ракеткой

```
        screen.blit(paddle.image, paddle.rect)
        screen.blit(score_text, textpos)
        for i in range (lives):
            width = screen.get_width()
            screen.blit(myBall.image, [width - 40 * i, 20])
        pygame.display.flip()

    if myBall.rect.top >= screen.get_rect().bottom:
        if not done:
            splat.play()
        lives = lives - 1
        if lives <= 0:
            if not done:
                pygame.time.delay(1000)
                bye.play()
            final_text1 = "Игра окончена"
            final_text2 = "Ваш счет: " + str(points)
            ft1_font = pygame.font.Font(None, 70)
            ft1_surf = font.render(final_text1, 1, (0, 0, 0))
            ft2_font = pygame.font.Font(None, 50)

            ft2_surf = font.render(final_text2, 1, (0, 0, 0))
            screen.blit(ft1_surf, [screen.get_width()/2 - \
                           ft1_surf.get_width()/2, 100])
            screen.blit(ft2_surf, [screen.get_width()/2 - \
                           ft2_surf.get_width()/2, 200])
            pygame.display.flip()
            done = True
            pygame.mixer.music.fadeout(2000)
        else:
            pygame.time.delay(1000)
            new_life.play()
            myBall.rect.topleft = [50, 50]
            screen.blit(myBall.image, myBall.rect)
            pygame.display.flip()
            pygame.time.delay(1000)
pygame.quit()
```

Воспроизводим звук, когда игрок теряет жизнь

Ждем одну секунду и проигрываем завершающий звук

Постепенно уменьшаем громкость музыки до нуля

Воспроизводим звук в начале новой партии

Это очень длинная программа! (Примерно 100 строк, включая пустые.) Ее можно было сделать короче, но это затруднило бы чтение и понимание. В этой главе мы постепенно добавляли к программе небольшие фрагменты, что избавило вас от необходимости набирать сразу весь код. Если вы читаете книгу с самого начала, то без труда поймете, зачем нужна каждая часть программы и как отдельные части соединяются друг с другом. На всякий случай я поместил весь текст этой программы в папку *\examples* и на сайт книги.

В следующей главе вам предстоит написать графическую программу совсем другого типа: с кнопками, меню и тому подобным — всем тем, что называется графическим интерфейсом пользователя (GUI).

Что мы узнали

В этой главе мы научились:

- добавлять в программу звуки;
- воспроизводить звуковые клипы (обычно это wav-файлы);
- воспроизводить музыкальные файлы (обычно это mp3-файлы);
- определять момент завершения воспроизведения;
- управлять громкостью звуковых эффектов и музыки;
- выполнять циклическое воспроизведение музыкального фрагмента;
- постепенно уменьшать громкость воспроизведения до нуля.

Проверь себя

1. Назовите три типа файлов, используемых для хранения звуковой информации.
2. Какой модуль библиотеки Pygame используется для воспроизведения музыки?
3. Как задать громкость воспроизведения звукового Pygame-объекта?
4. Как задать громкость воспроизведения фоновой музыки?
5. Как сделать так, чтобы громкость воспроизведения уменьшилась до нуля?

Эксперименты

Добавьте звуки в игру с угадыванием цифр из главы 1. Эта игра была создана в текстовом режиме, поэтому вам нужно будет добавить к ней Pygame-окно, как это было сделано в приведенном в данной главе примере. Вы можете воспользоваться следующими звуками из папки *\examples\sounds* (или с сайта):

- *Ahoy.wav*;
- *TooLow.wav*;
- *TooHigh.wav*;
- *WhatsYerGuess.wav*;
- *AvastGotIt.wav*;
- *NoMore.wav*.

Или вы захотите записать собственные варианты звуков? Это можно сделать, например, с помощью программы Sound Recorder, встроенной в операционную систему Windows. Также с сайта *audacity.sourceforge.net/* загружается бесплатная программа Audacity (доступная для разных операционных систем).

ГЛАВА 20

И снова GUI-интерфейсы

В главе 6 мы уже разработали несколько простых графических интерфейсов пользователя (GUI), применив для моделирования диалоговых окон модуль EasyGui. Но для полноценного графического интерфейса одних диалоговых окон мало. Через GUI-интерфейс запускается большая часть современных программ. В этой главе мы познакомимся с модулем PyQt, обеспечивающим большую гибкость в выборе внешнего вида элементов.

Модуль PyQt может помочь нам в создании GUI-интерфейсов. Знакомство с ним мы начнем с создания новой версии программы преобразования температуры.

Модуль PyQt

Первым делом убедитесь, что у вас установлен модуль PyQt. Если вы пользовались прилагаемой к книге программой установки, значит, модуль PyQt у вас уже есть. В противном случае загрузите его с сайта *www.riverbankcomputing.com/software/pyqt/download*. Выберите подходящий вариант с учетом своей операционной системы и установленной у вас версии Python (если вы пользовались нашей программой установки, то это версия 2.7.3). Лично я использую PyQt 4.1.

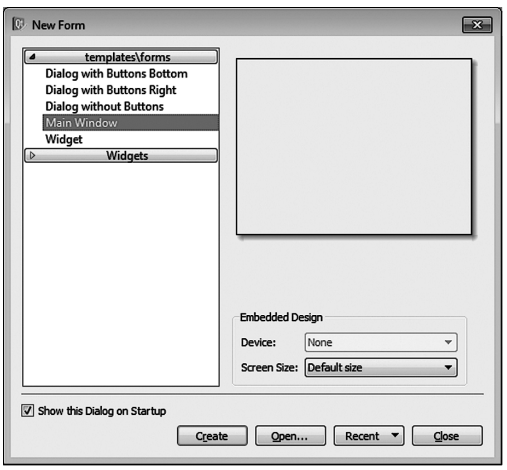
Процедуру разработки GUI можно разбить на две основные части. Нужно создать сам интерфейс пользователя (то есть «UI»), а затем написать код, который заставит его функционировать нужным вам образом. Первая часть сводится к размещению в окне таких элементов, как кнопки, текстовые поля, переключатели и т. п. Затем пишется код, определяющий, что случится при щелчке по кнопке, вводе текста, установке переключателя или выборе варианта в раскрывающемся списке.

У модуля PyQt существует версия 5, сильно отличающаяся от версии 4. Если вы не пользовались нашей программой установки, убедитесь, что у вас выбрана версия PyQt 4.

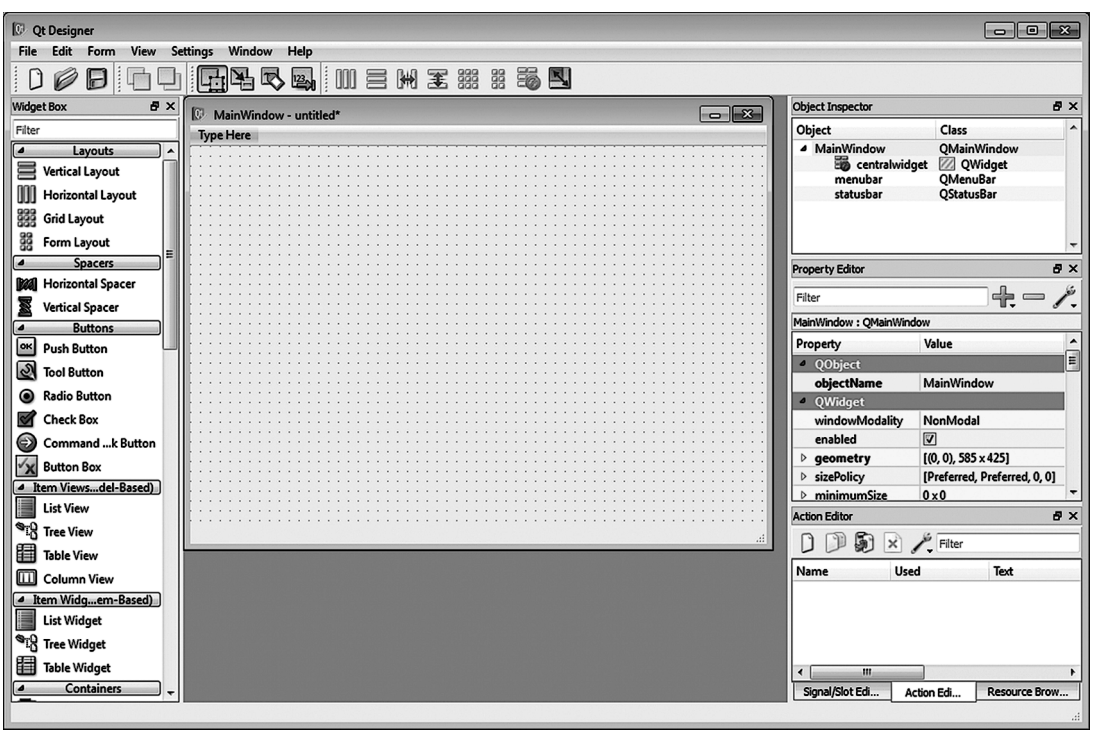
При работе с модулем Qt для создания UI применяется так называемый Qt-дизайнер. Давайте посмотрим, как он работает.

Qt-дизайнер

Установка модуля PyQt сопровождается установкой программы, называемой Qt-дизайнером. Найдите ее значок (например в стартовом меню операционной системы Windows) и запустите. Откроется окно, в центре которого вы увидите диалоговое окно ***New Form*** (Новая форма):



Формой (form) в программировании называется GUI-окно. Так как нам нужно создать новое GUI-окно, выделите вариант ***Main Window*** (Главное окно) и щелкните по кнопке ***Create*** (Создать). И давайте рассмотрим остальные части окна Qt-дизайнера.



Слева располагается панель виджетов с перечнем графических элементов, которые могут потребоваться вам при создании GUI. Они сгруппированы в пять категорий.

НОВЫЕ СЛОВА

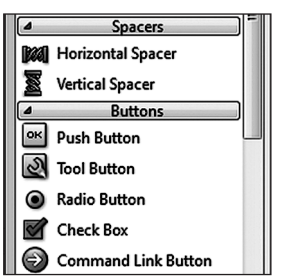
Отдельные кнопки, флажки и прочее в GUI называют *виджетами* (widgets). Впрочем, иногда можно встретить термины *компоненты* и *элементы управления*.

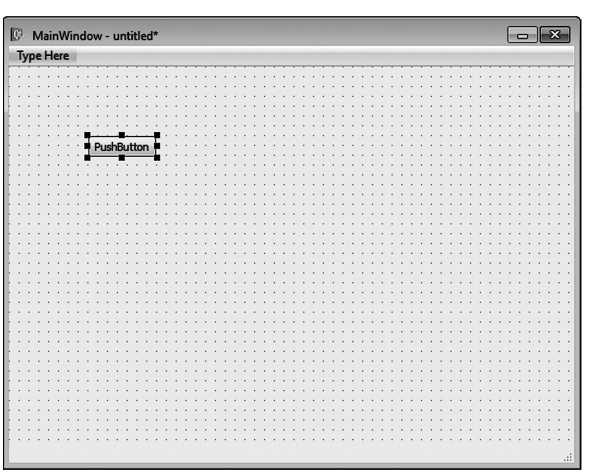
Справа мы видим панели инспектора объектов и редактора свойств. Именно здесь можно посмотреть и изменить свойства отдельных виджетов. Под ними находится еще одно окно, назначение которого зависит от выбранной снизу вкладки. Это может быть окно редактора сигналов/слотов, редактора действий и обозревателя ресурсов.

В середине окна находится новая создаваемая вами форма. На ее верхней панели написано *Main Window – untitled* (Главное окно — без имени), так как пока мы не присвоили ей никакого имени. Именно сюда мы будем переносить виджеты в процессе создания нового UI-интерфейса. На компьютерах Mac для перехода к данному представлению нужно выбрать в меню команду *Qt Designer* ▸ *Preferences* (Qt-дизайнер ▸ Предпочтения) и изменить режим интерфейса пользователя с *Multiple Top-Level Windows* (Несколько окон верхнего уровня) на *Docked Window* (Фиксируемое окно). Без этого все панели будут свободно перемещаться в окне программы.

ДОБАВЛЕНИЕ КНОПКИ

Добавим к нашему GUI-интерфейсу кнопку. С левой стороны окна Qt-дизайнера найдите раздел *Buttons* (Кнопки), а в нем — виджет *Push Button* (Обычная кнопка).

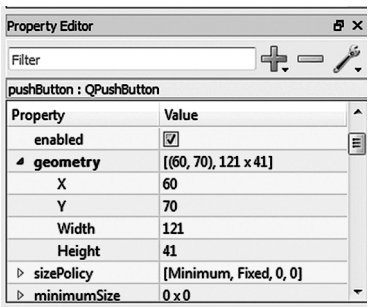




Перетащите этот виджет на форму и оставьте в произвольном месте. Теперь у нас есть кнопка с меткой *PushButton*.

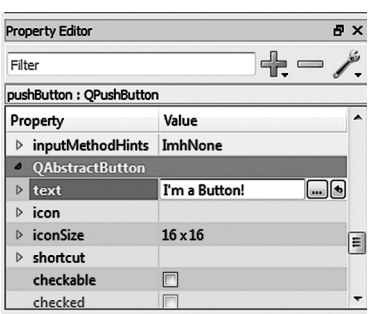
Обратите внимание на расположенную справа панель редактора свойств. Если выделение с кнопки не снято (вокруг нее находится синяя рамка), ее свойства будут видны в редакторе. Вы увидите, что кнопка называется *PushButton*. Воспользовавшись вертикальной полосой прокрутки для просмотра остальных свойств, вы узнаете ширину и высоту кнопки, ее координаты *x* и *y*, а также многое другое.

Редактирование кнопки



Поменять размер и положение кнопки можно двумя способами: при помощи мыши и через редактор свойств. Для перемещения кнопки с помощью мыши нажмите кнопку мыши в произвольном месте и перетащите ее на новое место. Чтобы изменить ее размер, нажмите кнопку мыши на любом из синих квадратиков, расположенных по периметру кнопки (они называются *манипуляторами*), и перетащите ребро или угол. Для изменения размера в редакторе свойств щелкните по маленькому треугольнику рядом с названием ***geometry***, и вы увидите поля для задания свойств ***X***, ***Y***, ***Width*** и ***Height***. После этого останется ввести в эти поля нужные вам значения.

Текст на кнопке также можно отредактировать. В настоящее время он совпадает с именем кнопки. Найдите в редакторе свойств строку ***text*** и введите в расположенное рядом поле текст ***I'm a Button!*** (Я кнопка!). Кроме того, двойной щелчок по самой кнопке активирует режим редактирования текста, и вы можете ввести новое название прямо там.



Теперь на кнопке в форме есть надпись ***I'm a Button!***. Но при этом имя виджета (свойство ***objectName***) осталось без изменений — ***PushButton***. Именно его вы будете использовать в коде, когда с кнопкой потребуется выполнить какую-нибудь операцию.

Сохранение GUI-интерфейса

Давайте сохраним результат наших усилий. В модуле PyQt описание GUI-интерфейса сохраняется в файле с расширением ***.ui***. Этот файл содержит всю информацию об окне, меню и виджетах. Именно эта информация отображается в окне свойств в правой части окна Qt-дизайнера, и нам нужно сохранить ее в файле, которым при запуске будет пользоваться PyQt-программа.

Для сохранения UI выберите в меню команду ***File*** ▸ ***Save As*** (Файл ▸ Сохранить как) и укажите имя файла. Давайте назовем его ***MyFirstGui***. Обратите внимание, что нам автоматически предлагается расширение ***.ui***. То есть наш интерфейс будет сохранен в файле ***MyFirstGui.ui***. По умолчанию конструктор помещает сохраняемые файлы в собственную папку, но вы можете выбрать любое другое место. Найдите папку, где уже хранятся ваши Python-программы, и щелкните по кнопке ***Save*** (Сохранить).

Сохраненный файл можно посмотреть в любом текстовом редакторе, в том числе в IDLE. Открыв его, вы увидите примерно следующее:

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
  <class>MainWindow</class>
  <widget class="QMainWindow" name="MainWindow">
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>576</width>
        <height>425</height>
      </rect>
    </property>
    <property name="windowTitle">
    <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralwidget">
    <widget class="QPushButton" name="pushButton">
      <property name="geometry">
        <rect>
          <x>60</x>
          <y>70</y>
          <width>121</width>
          <height>41</height>
        </rect>
      </property>
      <property name="toolTip">
        <string/>
      </property>
      <property name="text">
          <string>I'm a Button!</string>
        </property>
      </widget>
    </widget>
    <widget class="QMenuBar" name="menubar">
      <property name="geometry">
        <rect>
          <x>0</x>
          <y>0</y>
          <width>576</width>
          <height>21</height>
        </rect>
      </property>
    </widget>
    <widget class="QStatusBar" name="statusbar"/>
  </widget>
```

Определяем окно (фон)

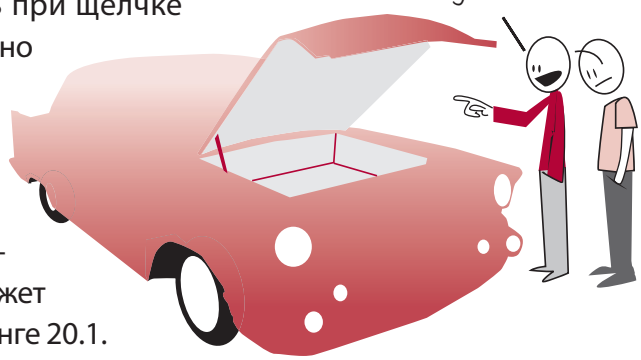
Определяем кнопку

```
  <resources/>
  <connections/>
</ui>
```

На первый взгляд код выглядит непонятным, но присмотревшись, вы обнаружите раздел, описывающий свойства окна, а также раздел, описывающий свойства кнопки. Есть и другие, пока незнакомые нам разделы, например описание меню и строки состояния.

Заставим наш GUI-интерфейс работать

У нас есть базовый GUI-интерфейс — окно с кнопкой. Но пока он совершенно бесполезен. Мы не написали код, объясняющий программе, что делать при щелчке пользователя по кнопке. Это все равно что построить автомобиль с колесами и кузовом, но без двигателя. Выглядит прекрасно, но не может сдвинуться с места. Нам нужен код, который послужит двигателем программы. Для PyQt-программы он может выглядеть так, как показано в листинге 20.1.



Листинг 20.1. Минимально необходимый для PyQt-программы код

```
import sys
from PyQt4 import QtCore, QtGui, uic

form_class = uic.loadUiType("MyFirstGui.ui")[0]

class MyWindowClass(QtGui.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)

app = QtGui.QApplication(sys.argv)
myWindow = MyWindowClass()
myWindow.show()
app.exec_()
```

Импортируем необходимые библиотеки PyQt

❶ Загружаем созданный нами в конструкторе UI-интерфейс

Определяем класс для главного окна

PyQt-объект, запускающий цикл событий

Создаем экземпляр класса для окна

Запускаем программу и выводим GUI-окно на экран

Если вам интересно, что это за [0] в конце строки ❶, помеченной единичкой, читайте врезку.

В конце строки, загружающей UI-интерфейс, есть символы [0], так как метод `uic.loadUiType()` возвращает список из двух элементов: `form_class` и `base_class`. Для наших целей требуется только первый из них — `form_class`, представленный в списке как `item[0]`.

Как это часто бывает в Python, все элементы в PyQt-коде являются объектами. Каждое окно — объект, определяемый ключевым словом `class`. Эта и все наши следующие PyQt-программы будут содержать классы, наследующие от класса `QMainWindow`. В листинге 20.1 мы присвоили этому классу имя `MyWindowClass` (в строке 6), но можно было назвать его и по-другому. Напоминаем, что определение класса — не более чем план дома. А нам еще нужно построить сам дом — создать экземпляр этого класса. Именно это делается в строке `myWindow = MyWindowClass()` в нижней части программы. Здесь `myWindow` — это экземпляр класса `MyWindowClass`.

Введите этот код в IDLE- или SPE-редактор и сохраните под именем *MyFirstGui.py*:

- основной код — *MyFirstGui.py*;
- UI-файл — *MyFirstGui.ui*.

Эти два файла нужно сохранить в одной папке, чтобы основная программа в начале своей работы могла найти и загрузить UI-файл.

Теперь можно запустить программу в IDLE. Откроется окно, в котором вы можете щелкнуть по кнопке. Но ничего не произойдет. Мы заставили программу работать, но код для кнопки пока отсутствует. Закройте программу, щелкнув по системной кнопке закрытия в строке заголовка.

Добавим какую-нибудь простую функциональность. Пусть кнопка после щелчка по ней смещается в другое место. Для создания листинга 20.2 добавьте к листингу 20.1 строки с 10 по 17.

Листинг 20.2. Обработчик события для кнопки из листинга 20.1

```
import sys
from PyQt4 import QtCore, QtGui, uic

form_class = uic.loadUiType("MyFirstGui.ui")[0]

class MyWindowClass(QtGui.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.pushButton.clicked.connect(self.button_clicked)

    def button_clicked(self):
        x = self.pushButton.x()
```

Связываем обработчик с событием

Добавьте эти строки, чтобы кнопка начала смещаться после щелчка

Обработчик события

```
        y = self.pushButton.y()
        x += 50
        y += 50
        self.pushButton.move(x, y)

app = QtGui.QApplication(sys.argv)
myWindow = MyWindowClass()
myWindow.show()
app.exec_()
```

Перемещаем кнопку после щелчка по ней

Добавьте эти строки, чтобы кнопка начала смещаться после щелчка

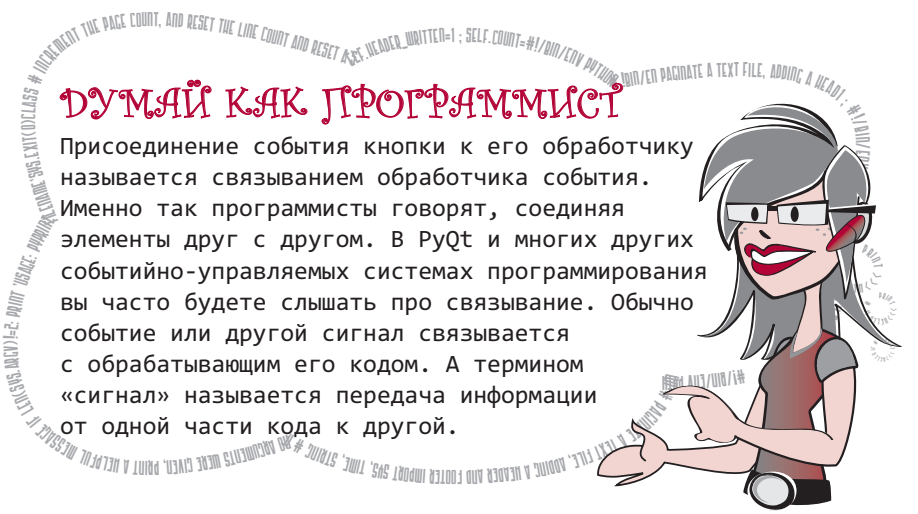
Не забудьте, что содержимое блока `def` должно быть сдвинуто относительно инструкции `class` на четыре пробела вправо, как показано в листинге. Это нужно сделать потому, что все компоненты находятся внутри окна или являются его частями. Именно поэтому код обработчика событий кнопки попадает в определение класса.

Запустите программу и посмотрите, что получится. Подробно этот код рассматривается в следующем разделе.

Результат действия обработчиков

В Pygame-программах, рассмотренных в предыдущих главах, вы познакомились с *обработчиками событий* и узнали, каким образом они позволяют отслеживать *события* мыши и клавиатуры. Эта информация пригодится вам и при работе с модулем PyQt.

В классе `MyWindowClass` мы определили обработчики событий для окна. Так как именно в этом окне находится кнопка, в класс `MyWindowClass` попадает и обработчик ее событий.



Думай как программист

Присоединение события кнопки к его обработчику называется связыванием обработчика события. Именно так программисты говорят, соединяя элементы друг с другом. В PyQt и многих других событийно-управляемых системах программирования вы часто будете слышать про связывание. Обычно событие или другой сигнал связывается с обрабатывающим его кодом. А термином «сигнал» называется передача информации от одной части кода к другой.

Первым делом нужно, чтобы главное окно узнало о наличии обработчика события для конкретного виджета. В листинге 20.2 это происходит в строке 10:

```
self.pushButton.clicked.connect(self.button_clicked)
```

В данном случае мы связываем событие (`self.pushButton.clicked`) с его обработчиком (`self.button_clicked`). Определение обработчика событий `button_clicked` начинается в строке 12. Для кнопки возможны такие события, как щелчок по ней (`clicked`), нажатие (`pressed`) и отпускание (`released`).

Что такое self

Обработчик события `button_clicked()` обладает параметром `self`. Как уже упоминалось в главе 14 при первом разговоре про объекты, параметр `self` относится к вызвавшему метод экземпляру. В данном случае все события порождаются фоном или основным окном, а значит, именно этот объект будет вызывать обработчик события. То есть сейчас параметр `self` относится к основному окну. Могло показаться, что он относится к компоненту, по которому выполняется щелчок, но на самом деле это не так; параметр связан с окном, в котором содержится данный компонент.

Смещение кнопки

Как нам сослаться на кнопку, с которой нужно что-то сделать? Модуль PyQt следит за всеми добавленными в окно виджетами. Мы знаем, что параметр `self` относится к окну, а виджет называется `pushButton`, значит, для доступа к виджету можно написать `self.pushButton`.

В приведенном в листинге 20.2 примере мы заставляли кнопку смещаться после каждого щелчка по ней. Положение кнопки в окне задается свойством `geometry`, состоящим из свойств `x`, `y`, `width` и `height`. Их можно редактировать двумя способами.

Во-первых, существует метод `setGeometry()`, меняющий геометрические свойства. Во-вторых, можно воспользоваться методом `move()`, меняющим только координаты объекта и никак не затрагивающим свойства `width` и `height` (именно так мы действовали в листинге 20.2). Координата *x* равна расстоянию от левого края окна, а координата *y* — расстоянию от его верхней кромки. Таким образом, координата верхнего левого угла окна равна [0, 0] (как и в Pygame).

Запустив программу, вы обнаружите, что после нескольких щелчков кнопка в правом нижнем углу исчезнет. При желании можно увеличить размер окна (перетащив его кромку или угол) и обнаружить кнопку. Завершить работу с программой можно, закрыв окно щелчком по системной кнопке закрытия в строке заголовка (или способом, принятым в используемой вами операционной системе).

Обратите внимание, что в отличие от ситуации в Pygame-окне, нам не нужно думать о «стирании» кнопки со старого места и рисовании ее в новом. Вы ее просто двигаете. Все операции удаления и перерисовки берет на себя модуль PyQt.

Более функциональные GUI-интерфейсы

Наш первый GUI-интерфейс был призван лишь продемонстрировать принцип создания таких интерфейсов при помощи модуля PyQt, но практической ценности он не имеет. Впрочем, остаток этой главы и главу 22 мы посвятим паре проектов, которые дадут вам более полное представление о возможностях модуля PyQt.

Во-первых, сейчас мы напишем PyQt-версию программы преобразования температуры. Во-вторых, в главе 22 модуль PyQt поможет нам создать GUI-интерфейс для игры Виселица. Еще чуть позже мы воспользуемся модулем PyQt для создания виртуального домашнего любимца.

Программа TempGUI

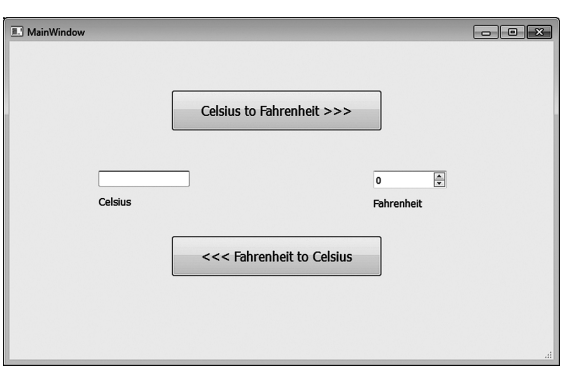
В разделе «Эксперименты» в главе 3 вы создали первую версию программы преобразования температуры. В главе 5 к ней была добавлена возможность пользовательского ввода, что избавило нас от жестко запрограммированной в коде температуры, позволив варьировать этот параметр. В главе 6 с помощью модуля EasyGui мы научились получать данные от пользователя и выводить результат работы программы на экран. Теперь воспользуемся модулем PyQt для создания графической версии программы преобразования температуры.

Компоненты программы TempGUI

Наш GUI-интерфейс будет простым, так как нам нужны только:

- место для ввода температуры (в градусах Цельсия или Фаренгейта);
- кнопки, активирующие процедуру преобразования;
- метки, дающие пользователю понять, где что находится.

В образовательных целях используем для ввода градусов Цельсия и Фаренгейта разные виджеты. В реальности никто так делать не будет (так как это сбивает с толку пользователей), но в данном случае нам важно проиллюстрировать принципы работы.



Окончательная версия GUI будет выглядеть примерно так:

Скорее всего, вы в состоянии самостоятельно справиться с этой задачей, так как Qt-дизайнер обладает более чем понятным интерфейсом. Но на всякий случай я все-таки пошагово все объясню. К тому же в этом случае в вашей версии программы гарантированно окажутся те же самые имена, что и в моей, а значит, впоследствии вам будет проще работать с кодом.

Не старайтесь полностью выполнить выравнивание компонентов и придать им именно такой вид, как на рисунке. Важно соблюсти только базовый принцип компоновки.

Новый GUI-интерфейс

Первым делом нам нужно создать новый PyQt-проект. Когда вы закроете открытый в данный момент UI-интерфейс (MyFirstGui), конструктор снова откроет окно ***New Form*** (Новая форма). Убедитесь, что выбран тип формы ***Main Window*** (Главное окно) и щелкните по кнопке ***Create*** (Создать).

Начнем добавлять виджеты: полем для ввода температуры в Цельсиях послужит виджет ***Line Edit*** (Строковый редактор), а для ввода температуры в Фаренгейтах возьмем ***Spin Box*** (Поле со счетчиком). В качестве меток под обоими полями возьмите виджеты ***Label*** (Метка) и добавьте два компонента ***Push Button*** (Обычная кнопка). Для поиска всех компонентов используйте полосу прокрутки, расположенную с левой стороны панели виджетов. Вот процесс создания GUI.

1. На панели виджетов найдите строку ***Push Button*** и перетащите ее на форму, чтобы получить кнопку. Затем выполните следующие действия:
   - придайте кнопке нужный размер, перетащив манипуляторы или указав численное значение свойств в разделе ***geometry*** (как мы это делали в программе MyFirstGui);
   - присвойте свойству ***objectName*** значение ***btnFtoC***;
   - присвойте свойству ***text*** значение ***<<< Fahrenheit to Celsius***;
   - присвойте параметру ***font size*** значение 12. Если в редакторе свойств вы щелкнете по кнопке с тремя точками  , расположенной справа от поля ***font***, откроется диалоговое окно ***Font*** (Шрифт), напоминающее то, с которым вам, скорее всего, приходилось работать в текстовых редакторах.
2. Перетащите в окно еще один виджет ***Push Button***, поместите его над первой кнопкой, придайте ему нужный размер и отредактируйте следующие свойства:
   - присвойте свойству ***objectName*** значение ***btnCtoF***;
   - присвойте свойству ***text*** значение ***Celsius to Fahrenheit >>>***;
   - сделайте размер шрифта равным 12.

3. Перетащите в окно виджет *Line Edit* и поместите слева от кнопок, присвоив свойству *objectName* значение *editCel*.
4. Перетащите в окно виджет *Spin Box* и поместите его справа от кнопок, присвоив свойству *objectName* значение *spinFahr*.
5. Перетащите в окно виджет *Label* и поместите его под виджетом *Line Edit*:
   - присвойте свойству *text* значение *Celsius*;
   - сделайте размер шрифта равным 10.
6. Перетащите в окно еще один виджет *Label*, поместив его под виджетом *Spin Box*:
   - присвойте свойству *text* значение *Fahrenheit*;
   - сделайте размер шрифта равным 10.

Итак, теперь все фрагменты GUI (виджеты, они же компоненты, они же элементы управления) на своих местах, мы присвоили им нужные имена и метки. Сохраните файл как *tempconv.ui*, выбрав в окне Qt-дизайнера команду *File* ▸ *Save As* (Файл ▸ Сохранить как). Не забудьте выбрать папку, предназначенную для сохранения Python-программ.

Теперь откройте в IDLE новый файл и введите в него базовый PyQt-код (можете скопировать его из нашей первой программы):

```
import sys
from PyQt4 import QtCore, QtGui, uic

form_class = uic.loadUiType("tempconv.ui")[0]

class MyWindowClass(QtGui.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)

app = QtGui.QApplication(sys.argv)
myWindow = MyWindowClass()
myWindow.show()
app.exec_()
```

Преобразование из Цельсия в Фаренгейты

Первым делом заставим работать функцию, преобразующую градусы Цельсия в градусы Фаренгейта. Вот формула, по которой осуществляется преобразование:

$$fahr = cel \times 9.0 / 5 + 32.$$

Температуру в градусах Цельсия программа должна узнавать от виджета *Line Edit*, который называется `editCel`, и после расчетов помещать результат в виджет *Spin Box*

с именем spinFahr. Все эти операции должны совершаться после щелчка пользователя по кнопке *Celsius to Fahrenheit* >>>, то есть код следует поместить в обработчик события этой кнопки.

Первым делом свяжем событие clicked с его обработчиком:

```
self.btn_CtoF.clicked.connect(self.btn_CtoF_clicked)
```

Этот код следует поместить в метод __init__() класса MyWindow, как это было сделано в нашей первой программе.

Теперь определим обработчик события. Для получения значения от поля *Celsius* (Градусы Цельсия), то есть виджета *Line Edit* с именем editCel, воспользуемся функцией self.editCel.text(). Она возвращает строку, которую нам требуется преобразовать в десятичное число:

```
cel = float(self.editCel.text())
```

После чего останется выполнить преобразование:

```
fahr = cel * 9.0 / 5 + 32
```

Полученное значение нужно поместить в поле *Fahrenheit* (Градусы Фаренгейта), представляющее собой виджет *Spin Box* с именем spinFahr. Но есть одна трудность: счетчики отображают только целые значения. То есть перед передачей полученного результата в поле счетчика его следует преобразовать в тип int. Выводимое счетчиком число задается его свойством value, то есть наш код будет выглядеть так:

```
self.spinFahr.setValue(int(fahr))
```

К результату расчетов мы добавим 0.5, чтобы функция int(), преобразующая десятичные числа в целые, произвела округление *до ближайшего целого*, а не просто *отбросила дробную часть*. Соединив все, получим:

```
def btn_CtoF_clicked(self):
    cel = float(self.editCel.text())
    fahr = cel * 9.0 / 5 + 32
    self.spinFahr.setValue(int(fahr + 0.5))
app = QtGui.QApplication(sys.argv)
myWindow = MyWindowClass()
myWindow.show()
app.exec_()
```

Получаем значение в градусах Цельсия

Преобразуем в градусы Фаренгейта

Округляем и помещаем в счетчик Fahrenheit

Преобразование градусов Фаренгейта в градусы Цельсия

Код обратного преобразования (из градусов Фаренгейта в градусы Цельсия) выглядит почти так же. Вот формула пересчета:

$$cel = (fahr - 32) * 5.0 / 9.$$

Ее следует поместить в обработчик события кнопки *<<< Fahrenheit to Celsius*. Обработчик связывается с кнопкой в методе `__init__()` окна.

```
self.btn_FtoC.clicked.connect(self.btn_FtoC_clicked)
```

Затем обработчик событий должен узнать температуру в градусах Фаренгейта из счетчика:

```
fahr = self.spinFahr.value()
```

В данном случае мы сразу получаем целое число, поэтому операция приведения типов нам не требуется. Можно сразу применить формулу:

```
cel = (fahr - 32) * 5.0 / 9
```

После чего остается преобразовать результат в строку и поместить его в поле *Celsius*:

```
self.editCel.setText(str(cel))
```

Соединив все части воедино, получим листинг 20.3.

Листинг 20.3. Программа преобразования температуры

```
import sys
from PyQt4 import QtCore, QtGui, uic

form_class = uic.loadUiType("tempconv.ui")[0]
class MyWindowClass(QtGui.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.btn_CtoF.clicked.connect(self.btn_CtoF_clicked)
        self.btn_FtoC.clicked.connect(self.btn_FtoC_clicked)

    def btn_CtoF_clicked(self):
        cel = float(self.editCel.text())
        fahr = cel * 9 / 5.0 + 32
        self.spinFahr.setValue(int(fahr + 0.5))

    def btn_FtoC_clicked(self):
        fahr = self.spinFahr.value()
```

Загружаем определение UI-интерфейса

Связываем обработчик события с кнопкой

Обработчик события для кнопки Celsius to Fahrenheit

Обработчик события для кнопки Fahrenheit to Celsius

```
        cel = (fahr - 32) * 5 / 9.0
        self.editCel.setText(str(cel))

app = QtGui.QApplication(sys.argv)
myWindow = MyWindowClass(None)
myWindow.show()
app.exec_()
```

Обработчик события для кнопки Fahrenheit to Celsius

Сохраните программу под именем ***TempGui.py***. Затем запустите и посмотрите, как функционирует написанный нами интерфейс.

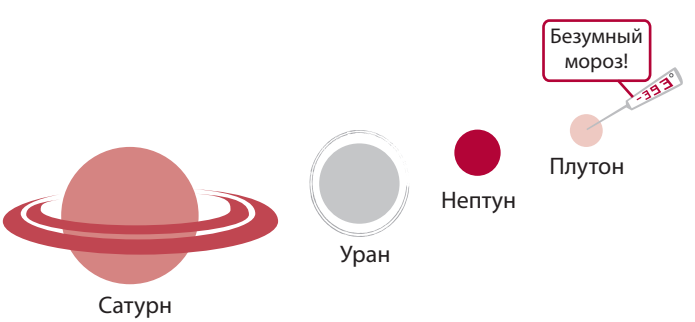
НЕБОЛЬШОЕ УСОВЕРШЕНСТВОВАНИЕ

Запустив программу, вы обнаружите, что результат преобразования градусов Фаренгейта в градусы Цельсия имеет большое количество знаков после десятичной точки; возможно, часть разрядов просто обрезается текстовым полем. Существует способ устранения этого недостатка, который называется *форматированием вывода*. Этой темы мы пока не касались, поэтому вы можете либо перейти к главе 21 и прочитать там полное объяснение, либо просто набрать предлагаемый мною код. Замените последнюю строку обработчика событий `btn_FtoC_clicked` этими двумя строками:

```
cel_text = '%.2f' % cel
self.editCel.setText(cel_text)
```

Теперь результат пересчета имеет два знака после десятичной точки.

Хмммм… Возможно, следует заняться отладкой. Вдруг пользователь захочет выполнить преобразование температур в Антарктиде? Или на Плутоне?



Поиск ошибки

Мы уже упоминали, что понять происходящее во время работы программы позволяет вывод значений некоторых переменных. Попробуем этот метод на практике.

Так как проблема возникла в процессе преобразования в градусы Фаренгейта, используем это как отправную точку. После последней строки в обработчике события `btn_CtoF_clicked` в листинге 20.3 добавьте:

```
print 'cel = ', cel, ' fahr = ', fahr
```

Теперь щелчок по кнопке *Celsius to Fahrenheit* >>> будет приводить к выводу в IDLE-окне значений переменных `cel` и `fahr`. Попробуйте несколько различных значений переменной `cel` и посмотрите, что получится. Я получил вот такой результат:

```
>>> ============================ RESTART ============================
>>>
cel = 50.0 fahr = 122.0
cel = 0.0 fahr = 32.0
cel = -10.0 fahr = 14.0
cel = -50.0 fahr = -58.0
```

Кажется, значение переменной `fahr` рассчитывается корректно. Почему же тогда в поле *Fahrenheit* не отображаются числа меньше 0 (или больше 99, если уж на то пошло!)?

Вернитесь в окно Qt-дизайнера и выделите виджет `spinFahr`, который у нас показывает температуру в градусах Фаренгейта. Обратите внимание на его свойства в редакторе свойств. Видите свойства *minimum* и *maximum* (они находятся в самом низу списка)? Какие там значения? Теперь понимаете, где источник проблемы?

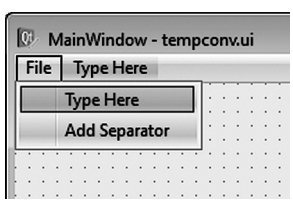
А что в меню?

GUI-интерфейс нашей программы преобразования температур снабжен кнопками, щелчок по которым инициирует пересчет. Однако во многих программах функциональность реализуется через *меню*. При этом порой в меню попадают команды, уже реализованные при помощи кнопок. Зачем могут понадобиться два способа получения одного и того же результата?

Прежде всего некоторые пользователи предпочитают работу с меню щелчкам по кнопкам. В сложных программах с множеством функциональных возможностей интерфейс без меню был бы перегружен кнопками. Кроме того, с меню можно работать при помощи клавиатуры, и некоторые считают, что так намного быстрее, чем переходить от клавиатуры к мыши и обратно.

Поэтому добавим к нашей программе меню, предоставив пользователям альтернативный способ преобразования температур. Заодно добавим в меню команду *File* ▸ *Exit* (Файл ▸ Выход), имеющуюся практически в любой программе.

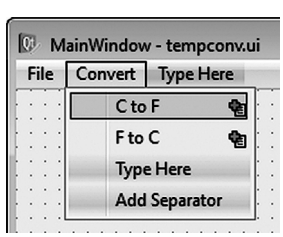
Модуль PyQt предоставляет инструмент создания и редактирования меню. В верхнем левом углу формы, предлагаемой конструктором, находится метка *Type Here* (Пишите здесь). Именно отсюда начинается работа над меню. В большинстве программ меню начинается с пункта *File* (Файл), поэтому давайте придерживаться общепринятой практики. Щелкните по строке *Type Here*, введите *File* и нажмите клавишу *Enter*. Появится пункт меню *File* и пространство для ввода новых пунктов сбоку и снизу, как показано на рисунке.



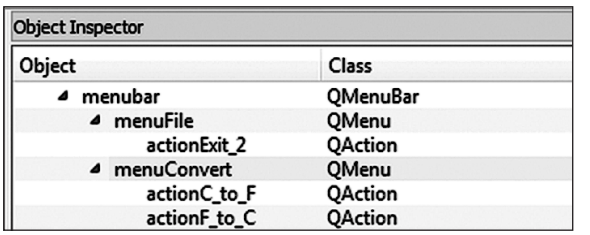
ДОБАВЛЕНИЕ ПУНКТА МЕНЮ

Добавим в меню *File* пункт *Exit* (Выход). В поле *Type Here* под пунктом *File* введите *Exit* и нажмите клавишу *Enter*.

Теперь создадим меню для преобразования температур (на случай, если потребитель не захочет пользоваться кнопками). В поле *Type Here* справа от пункта *File* введите *Convert* (Преобразовать), а под ним создайте два пункта меню *C to F* и *F to C*. В готовом виде это будет выглядеть так.



В окне инспектора объектов, расположенном в правом верхнем углу конструктора, вы увидите примерно такую картину:



Там находятся меню *File* и *Convert* вместе с их пунктами *Exit*, *C to F* и *F to C*. В терминологии модуля PyQt пункты меню являются экземплярами класса `QAction`. Так и должно быть, ведь нам нужно, чтобы за выбором пунктов меню следовали некие *действия* (actions).

Сохраните отредактированный файл под именем *tempconv_menu.ui*.

На компьютерах Mac нужно также убрать свойство `nativeMenuBar` объекта `menubar` (на последнем месте в списке редактора свойств). В противном случае наше меню вступит в конфликт с основным Python-меню на рабочем столе, и из PyQt-приложения исчезнет меню *File*.

Теперь, когда у нас есть пункты меню (или действия), следует связать их события с обработчиками. Для пунктов меню *C to F* и *F to C* обработчики уже написаны — это те же обработчики, которые должны активизироваться при щелчках по кнопкам. Нужно сделать так, чтобы выбор пункта меню давал тот же результат. Для этого необходимо связать события выбора пунктов меню с теми же самыми обработчиками.

В случае с пунктом меню (действием) нужно обрабатывать не *щелчок*, а *триггер*. Пункт, связываемый с обработчиком события, называется `action_CtoF`. А привязывается он к обработчику щелчка по кнопке `btn_CtoF_clicked`. Вот как будет выглядеть выполняющий данное связывание код:

```
self.action_CtoF.triggered.connect(self.btn_CtoF_clicked)
```

Аналогичным образом следует поступить с пунктом меню *F to C*.

Еще нужно создать обработчик события для пункта меню *Exit* и связать его с соответствующим событием. Назовем обработчик `menuExit_selected`, а так будет выглядеть связывающий его код:

```
self.actionExit.triggered.connect(self.menuExit_selected)
```

Реальный обработчик события для пункта меню *Exit* представляет собой одну строку в теле программы, отвечающую за закрытие окна:

```
def menuExit_selected(self):
    self.close(
```

Напоследок изменим имя загружаемого UI-файла (в третьей строке), подставив финальный сохраненный вариант *tempconv_menu.ui*.

Код после всех этих изменений представлен в листинге 20.4.

Листинг 20.4. Программа преобразования температуры после добавления меню

```
import sys
from PyQt4 import QtCore, QtGui, uic

form_class = uic.loadUiType("tempconv_menu.ui")[0]

class MyWindowClass(QtGui.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.btn_CtoF.clicked.connect(self.btn_CtoF_clicked)
        self.btn_FtoC.clicked.connect(self.btn_FtoC_clicked)
        self.action_CtoF.triggered.connect(self.btn_CtoF_clicked)
        self.action_FtoC.triggered.connect(self.btn_FtoC_clicked)
        self.actionExit.triggered.connect(self.menuExit_selected)

    def btn_CtoF_clicked(self):
        cel = float(self.editCel.text())
        fahr = cel * 9 / 5.0 + 32
        self.spinFahr.setValue(int(fahr + 0.5))

    def btn_FtoC_clicked(self):
        fahr = self.spinFahr.value()
        cel = (fahr - 32) * 5 / 9.0
        self.editCel.setText(str(cel))

    def menuExit_selected(self):
        self.close()

app = QtGui.QApplication(sys.argv)
myWindow = MyWindowClass(None)
myWindow.show()
app.exec_()
```

Загружаем UI-файл с меню

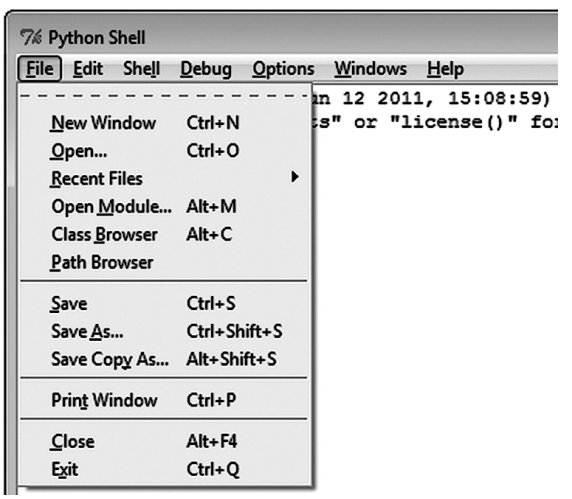
Связываем меню Convert с обработчиками событий

Связываем пункт меню Exit с обработчиком событий

Обработчик событий нового элемента меню Exit

Горячие клавиши

Мы уже упоминали, что некоторые пользователи предпочитают работать с меню, потому что при этом можно ограничиться клавиатурой, не трогая мышь. Пока что нашим меню можно пользоваться только с помощью мыши, потому что мы не предоставили средства для работы с клавиатурой. Но сейчас мы исправим это упущение, добавив горячие клавиши.



Горячие клавиши (также называемые *клавиатурными комбинациями*) позволяют выбирать пункты меню при помощи клавиатуры. В операционных системах Windows и Linux система меню активируется клавишей *Alt*. При нажатии этой клавиши в каждом пункте меню выделяется определенная буква (обычно она подчеркнута). Если нажать такую букву, активируется выделенное меню. Соответственно, для входа в меню *File* нужно нажать *Alt+F*. То есть вы, удерживая нажатой клавишу *Alt*, должны нажать *F*. После этого вы увидите пункты меню *File* и список горячих клавиш для каждого из них. Попробуйте проделать это с IDLE-окном.

Чтобы открыть новое окно, нажмите клавиши *Alt+F+N* (удерживая клавишу *Alt*, нажмите *F*, а затем — *N*).

Запрограммируем горячие клавиши для GUI-интерфейса, выполняющего преобразование температур. Поместите перед буквой, которую вы хотите сделать горячей клавишей, символ &. Для меню (например *File*) это задается в свойстве *Title*, а для пункта меню (например *Exit*) — в свойстве *Text*. Для меню *File* в качестве горячей принято выбирать клавишу *F*; для меню *Exit* — клавишу *X*. То есть *File* превратится в *&File*, а *Exit* — в *E&xit*.

Нужно выбрать, какие клавиши будут использоваться для пунктов меню *Convert*. Давайте возьмем *C* для меню *Convert*, *C* — для преобразования градусов Цельсия в градусы Фаренгейта и *F* — для преобразования градусов Фаренгейта в градусы Цельсия. Соответственно, нужно написать *&Convert*, *&C to F* и *&F to C*, что даст нам два сочетания

клавиш *Alt*+*C*+*C* (преобразование градусов Цельсия в градусы Фаренгейта) и *Alt*+*C*+*F* (преобразование градусов Фаренгейта в градусы Цельсия).

После определения горячих клавиш в конструкторе больше ничего делать не нужно. О подчеркивании клавиш и обработке клавиатурного ввода позаботятся модуль PyQt и операционная система. Просто сохраните файл с UI-интерфейсом. Можно присвоить ему другое имя, например *tempconv_menu_hotkeys.ui*. В этом случае не забудьте отредактировать третью строку в листинге 20.4, чтобы программа загружала нужный UI-файл:

```
form_class = uic.loadUiType("tempconv_menu_hotkeys.ui")[0]
```

Вообще-то нет. Так как все компьютеры Mac изначально поставлялись с мышью (или сенсорной панелью), в операционной системе Mac OS предполагается, что для работы с меню пользователи будут применять мышь. Поэтому клавиатурных комбинаций для пунктов меню в Mac OS попросту не существует. Горячие клавиши есть для многих функциональных возможностей, часть из них даже соответствует пунктам меню. Но полностью управлять меню с клавиатуры, как это делается в Windows, невозможно.

На этом мы заканчиваем работу над GUI-интерфейсом программы преобразования температуры. В главе 22 модуль PyQt поможет нам написать игру Виселица.

Что мы узнали

В этой главе мы познакомились с:

- модулем PyQt;
- программой Qt-дизайнер;
- виджетами — кнопками, текстовыми полями и другими элементами, из которых состоит GUI-интерфейс;
- обработчиками событий, заставляющими ваши компоненты выполнять некие действия;
- пунктами меню и клавиатурными комбинациями.

Проверь себя

1. Назовите три термина, обозначающих такие вещи, как кнопки, текстовые поля и прочие элементы, составляющие GUI-интерфейс.
2. Как называется клавиша, нажимаемая вместе с клавишей *Alt* для получения доступа к меню?
3. С каким расширением следует сохранять файлы Qt-дизайнера?
4. Назовите пять типов виджетов, которые можно вставить в GUI-интерфейс при помощи модуля PyQt.
5. Чтобы виджет (например кнопка) что-то делал, у него обязательно должен быть _____.
6. Какой спецсимвол задает горячие клавиши в меню?
7. Содержимое счетчика в модуле PyQt всегда _____.

Эксперименты

1. В главе 1 мы писали текстовую программу для угадывания чисел, а в главе 6 для нее был создан простой GUI-интерфейс. Попробуйте написать GUI-интерфейс для этой игры с помощью модуля PyQt.
2. Вы поняли, почему счетчик не показывал значения ниже 0? (Эту ошибку Картер обнаружил в листинге 20.2.) Скорректируйте свойства счетчика таким образом, чтобы проблема больше не возникала. Убедитесь, что вы скорректировали обе границы диапазона, и счетчик может показывать очень высокие температуры наравне с очень низкими. (Вдруг пользователь вашей программы захочет пересчитать температуру на Меркурии и Венере, а не только на Плутоне!)

ГЛАВА 21

Форматирование вывода и строки

Давным-давно, в главе 1, вы познакомились с инструкцией `print`. Это была самая первая команда языка Python, которой вы воспользовались. Кроме того, вы узнали (в главе 5), что поставленная после инструкции `print` запятая заставляет интерпретатор Python выводить следующие данные на той же строке. (По крайней мере в Python 2. В Python 3 это не так.) Вы открывали командную строку для функции `raw_input()`, пока не узнали, что приглашение командной строки можно поместить внутрь этой функции.

В этой главе мы поговорим о *форматировании вывода* (print formatting) — способах придания выводимым данным нужного нам вида. При этом рассматриваются следующие темы:

- создание новой строки (когда это оправдано);
- растяжение элементов по горизонтали (и выравнивание их в столбцах);
- вывод переменных в середине строк;
- форматирование целых и десятичных чисел, а также чисел в экспоненциальной форме, и задание количества знаков после десятичной точки.

Заодно вы познакомитесь со встроенными в Python методами работы со строками. Они поддерживают:

- разбиение строк на более мелкие фрагменты;
- объединение строк;
- поиск строки;
- поиск внутри строки;
- удаление части строки;
- изменение регистра (верхнего и нижнего).

Все это пригодится нам в текстовых (без GUI-интерфейса) программах, а часть из них будет полезна даже при создании GUI-интерфейсов и игр. В Python существуют и другие варианты форматирования вывода, но мы остановимся только на наиболее распространенных, охватывающих 99 % ваших программ.

Новая строка

Вы уже много раз сталкивались с инструкцией `print`. Что произойдет, если вы воспользуетесь ею больше одного раза? Запустите эту короткую программу:

```
print "Эй"
print "Привет"
```

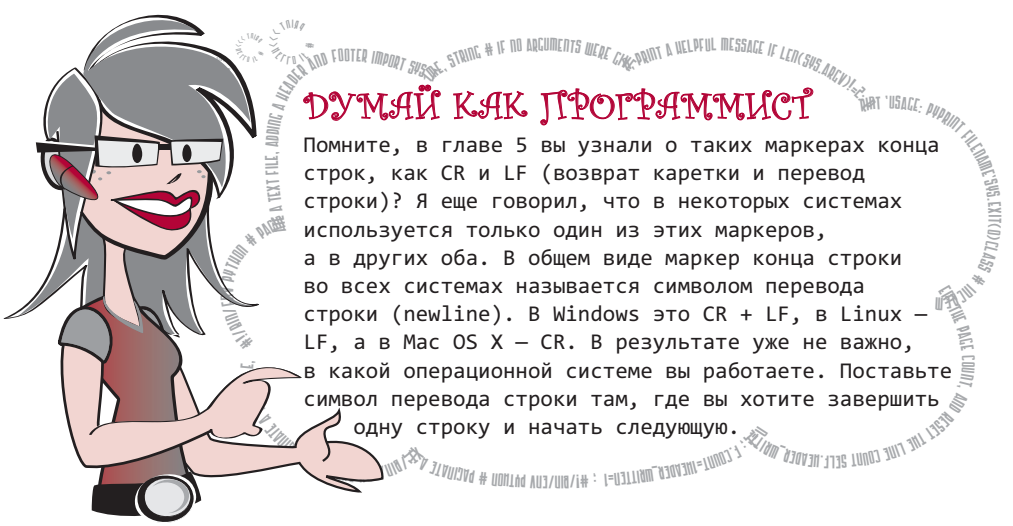
Вы получите вот такой результат:

```
>>> ========== RESTART ==========
>>>
Эй
Привет
```

Почему два слова выводятся на разных строках? Почему результат работы программы выглядит не так:

```
ЭйПривет
```

Без дополнительных команд интерпретатор Python начинает с новой строки каждую операцию вывода, реализуемую с помощью инструкции `print`. После `Эй` он смещается на строку вниз и возвращается в начальный столбец, чтобы вывести на экран слово `Привет`. Между двумя словами интерпретатор Python вставляет символ новой строки. Аналогичный символ появляется при нажатии клавиши *Enter* в текстовом редакторе.



Думай как программист

Помните, в главе 5 вы узнали о таких маркерах конца строк, как CR и LF (возврат каретки и перевод строки)? Я еще говорил, что в некоторых системах используется только один из этих маркеров, а в других оба. В общем виде маркер конца строки во всех системах называется символом перевода строки (newline). В Windows это CR + LF, в Linux — LF, а в Mac OS X — CR. В результате уже не важно, в какой операционной системе вы работаете. Поставьте символ перевода строки там, где вы хотите завершить одну строку и начать следующую.

Инструкция print и запятая

Инструкция `print` автоматически вставляет символ перевода строки после выводимой информации. Как этого можно избежать? Нужно поставить запятую (как мы это делали в главе 5):

```
print 'Эй',
print 'Привет'
>>> ================== RESTART ==================
>>>
Эй Привет
```

(Еще раз напомню: в Python 3 этот подход не работает.) Обратите внимание, что на этот раз между словами `Эй` и `Привет` стоит пробел. Когда вы ставите запятую, чтобы не дать интерпретатору Python выполнить перенос строки, он добавляет пробел.

Для вывода двух строк без пробела применяется уже знакомая вам операция *конкатенации*:

```
print 'Эй' + 'Привет'
>>> ================== RESTART ==================
>>>
ЭйПривет
```

Еще раз напомню, что конкатенация, *по сути*, является сложением строк, просто слово «сложение» обычно относят к операции с числами.

Добавление символа перевода строки

Что делать, если вы хотите вставить символ перевода строки вручную? Например, чтобы вставить пустую строку между словами `Эй` и `Привет`. Проще всего это реализуется при помощи дополнительной инструкции `print`:

```
print "Эй"
print
print "Привет"
```

Запустив эту программу, вы получите:

```
>>> ================== RESTART ==================
>>>
Эй

Привет
```

Управляющие символы

Перевод строки можно выполнить и другим способом. В Python присутствуют специальные управляющие символы, меняющие вид выводимых строк. Все они начинаются с обратной косой черты (\). Символ перевода строки выглядит как \n. Выполните этот код в интерактивном режиме:

```
>>> print "Привет мир"
Привет мир
>>> print "Привет \nмир"
Привет
мир
```

Вставка символа \n привела к тому, что слова Привет и мир были выведены на отдельных строках, так как между ними появился символ перевода строки.

Табуляция

Вы только что узнали о способах управления межстрочным интервалом (добавление символов перевода строк или запятых, запрещающих перевод). Теперь мы поговорим об изменении горизонтальных интервалов с помощью *табуляции*.

Табуляция применяется для выравнивания элементов в столбцах. Чтобы понять, как она работает, представьте, что все строки на экране разделены на блоки одного размера. Пусть ширина одного блока равняется восьми символам. Вставка символа табуляции переносит в начало следующего блока.

Лучше всего проиллюстрировать это на практике. Символ табуляции выглядит как \t. Запустите эту строку в интерактивном режиме:

```
>>> print 'ABC\tXYZ'
ABC     XYZ
```

Как видите, между группами символов XYZ и ABC вставлены пробелы. Причем символы XYZ отстоят от начала строки ровно на восемь символов. Это связано с размером блока. Принято говорить, что символ табуляции переводит нас на восемь символов вправо.

Посмотрите на несколько примеров применения команды print, показывающих, как табуляция сдвигает строки.

Можно представить себе экран (или каждую строку) как набор блоков по восемь символов

```
>>> print 'ABC\tXYZ'
ABC     XYZ

>>> print 'ABCDE\tXYZ'
ABCDE   XYZ

>>> print 'ABCDEF\tXYZ'
ABCDEF  XYZ

>>> print 'ABCDEFG\tXYZ'
ABCDEFG XYZ

>>> print 'ABCDEFGHI\tXYZ'
ABCDEFGHI       XYZ
```

в каждом. Обратите внимание, что при увеличении группы символов ABC группа XYZ остается на том же месте. Символ \t указывает интерпретатору Python, что начинать строку XYZ нужно со следующего шага табуляции, то есть в следующем доступном блоке. При этом если длина последовательности ABC выходит за границы первого блока, интерпретатор Python смещает группу XYZ еще на один шаг табуляции.

Табуляция позволяет распределять элементы по столбцам, сохраняя их выравнивание. Давайте воспользуемся этим свойством для вывода таблицы квадратов и кубов последовательности чисел. Откройте в IDLE новое окно и введите короткую программу из листинга 21.1. Сохраните и запустите ее. Я назвал свою программу *squbes.py*, что является сокращением от «squares and cubes» (квадраты и кубы).

Листинг 21.1. Программа вывода квадратов и кубов чисел

```
print "Number \tSquare \tCube"
for i in range (1, 11):
    print i, '\t', i**2, '\t', i**3
```

После запуска программы вы получите аккуратно выровненный результат:

```
>>> ====================== RESTART ======================
>>>
Number   Square  Cube
1        1       1
2        4       8
3        9       27
4        16      64
5        25      125
6        36      216
7        49      343
8        64      512
9        81      729
10       100     1000
```

Вывод обратной косой черты

Обратная косая черта (\) является частью управляющих символов. Но что делать, если мы захотим вывести этот символ на экран? Существует ли возможность объяснить это интерпретатору Python? Да, такая возможность существует. Для этого достаточно поставить рядом две обратные косые черты:

```
>>> print 'эй\\привет'
эй\привет
```

Первый символ \ сообщает интерпретатору Python, что следом идет нечто особенное, а второй показывает, что этим особенным является символ \.

Новые слова

Установка рядом двух обратных косых черт с целью вывести одну из них на экран называется *экранированием* (escaping) символа. Говорят, что первая черта *экранирует* вторую, в результате вторая обратная косая черта воспринимается интерпретатором как обычный, а не как специальный символ.

Вставка переменных в строки

До сих пор для вставки переменных в строки мы поступали так:

```
name = 'Уоррен Сэнд'
print 'Меня зовут', name, 'и я написал эту книгу.'
```

Запустив этот код, получим строку:

```
Меня зовут Уоррен Сэнд и я написал эту книгу.
```

Но существует и другой способ вставки переменных в строки, позволяющий лучше контролировать вид конечного результата, особенно чисел. Это *форматирующие строки* (format strings), содержащие символ процента (%). Предположим, нам нужно вставить строковую переменную в середину вывода, реализуемого инструкцией print, как это было сделано в предыдущем примере. С форматирующей строкой это будет выглядеть следующим образом:

```
name = 'Уоррен Сэнд'
print 'Меня зовут %s и я написал эту книгу' % name
```

Символ % встречается в двух местах. Прежде всего в середине строки, куда мы хотим вставить переменную, и затем после строки, сообщая интерпретатору Python, что следом идет предназначенная для вставки переменная.

Запись %s означает вставку строковой переменной. Если нам нужно вставить целое число, мы напишем %i; для вставки десятичного числа используется запись %f.

Вот еще пара примеров:

```
age = 13
print 'Мне %i лет.' % age
```

Вот что получится после запуска этого кода:

```
Мне 13 лет.
```

Другой пример:

```
average = 75.6
print 'Среднее математического теста составляет %f процентов.' % average
```

Запустив код, получим:

```
Среднее математического теста составляет 75.600000 процентов.
```

Именно сочетания символов `%s`, `%f` и `%i` называются *форматирующими строками* и определяют вид выводимой переменной.

Существуют и другие элементы, добавляемые к форматирующим строкам для придания выводимым данным нужного вида. Есть даже строки, позволяющие получить запись в экспоненциальной форме. (Помните, мы говорили о ней в главе 3?) Их мы рассмотрим в следующих разделах.

Форматирование чисел

При выводе чисел хотелось бы управлять их видом. Для этого нужно знать:

- количество знаков после запятой;
- обычная или экспоненциальная запись;
- наличие ведущих или конечных нулей;
- наличие знака + или –.

Форматирующие строки позволяют делать как это, так и многое другое! Вот два варианта вывода информации программой, сообщающей прогноз погоды:

```
Сегодня макс.: 23.45672132, мин 15.4985756
Сегодня макс: 23, мин: 15
```

В каком виде вы бы предпочли получать информацию? В большинстве программ корректный вид чисел крайне важен.

Рассмотрим пример. Скажем, нужно вывести десятичное число с двумя знаками после десятичной точки. Запустите следующий код в интерактивном режиме:

```
>>> dec_number = 12.3456
>>> print 'Сегодня %.2f градуса.' % dec_number
Сегодня 12.35 градуса
```

В середине инструкции `print` находится форматирующая строка. Но на этот раз мы записали не `%f`, а `%.2f`. Это заставило интерпретатор Python вывести два знака после

десятичной точки в числе с плавающей точкой. (Обратите внимание, что интерпретатор Python округлил число до двух цифр после точки, а не просто отбросил лишние разряды.)

Второй знак %, стоящий после строки, сообщает интерпретатору Python, что следом идет информация, которую нужно показать. Вид выводимого числа задается форматирующей строкой. Для наглядности приведем еще несколько примеров.

У тебя хорошая память, Картер! Знак % действительно соответствует оператору, вычисляющему остаток от целочисленного деления, с которым мы познакомились в главе 3, но кроме этого он задает форматирующие строки. Точное назначение этого знака интерпретатор Python определяет по его внешнему виду.

Целые числа: %d или %i

Для вывода целых чисел используйте форматирующую строку %d или %i (Я не знаю, почему в данном случае наличествуют два варианта, но вы можете пользоваться любым по своему выбору.):

```
>>> number = 12.67
>>> print '%i' % number
12
```

Обратите внимание, что на этот раз число не округляется. От него просто *отбрасывается дробная часть*. В случае округления результат был бы равен 13, а не 12. При форматировании целых чисел у них отбрасывается дробная часть, в то время как числа с плавающей точкой при форматировании округляются.

В данном случае следует обратить внимание на три вещи.

- В строке может не быть никакого другого текста — форматирующая строка может использоваться и сама по себе.
- Несмотря на то, что исходное число было вещественным, выводится оно как целое. Форматирующие строки позволяют проделывать такой трюк.
- Интерпретатор Python отбросил дробную часть числа. Но в отличие от функции `int()`, с которой вы познакомились в главе 4, форматирующая строка не создает нового значения. Она всего лишь меняет вид выводимого на экран числа.

В данном случае мы отформатировали число 12,67 как целое, и интерпретатор Python вывел значение 12. Но значение переменной `number` при этом не изменилось. Убедитесь в этом сами:

```
>>> print number
12.67
```

Значение переменной number осталось тем же. Мы просто показали его по-другому при помощи форматирующей строки.

Числа с плавающей точкой: %F или %f

Форматирующая строка для вещественных чисел включает в себя букву f в верхнем (%F) или нижнем (%f) регистре:

```
>>> number = 12.3456
>>> print '%f' % number
12.345600
```

Запись %f округляет число до шести цифр после десятичной точки. Добавив перед f .n, где n — любое целое число, вы оставите n знаков после точки:

```
>>> print '%.2F' % number
12.35
```

Как видите, число 12.3456 было округлено до двух знаков после точки: 12.35.

Если указанное вами количество знаков после точки превысит реально существующее, интерпретатор Python заполнит свободные места нулями:

```
>>> print '%.8f' % number
12.34560000
```

В данном случае число имеет всего четыре знака после точки, а мы попросили вывести восемь, поэтому результат вывода содержит четыре нуля.

В случае отрицательного числа строка %f всегда будет приводить к выводу знака -. Если нужно, чтобы знак присутствовал всегда, даже у положительных чисел, поставьте после знака % знак + (это позволяет выровнять списки, состоящие из положительных и отрицательных чисел):

```
>>> print '%+f' % number
+12.345600
```

Если вам нужно выровнять список из положительных и отрицательных чисел, не выводя перед первыми знак +, поставьте сразу после знака % вместо знака + пробел:

```
>>> number2 = -98.76
>>> print '% .2f' % number2
-98.76
>>> print '% .2f' % number
 12.35
```

Обратите внимание на пробел перед числом 12 в результатах вывода. Благодаря ему числа 12 и 98 выровнены по вертикали, несмотря на то, что перед одним из них есть знак, а перед другим нет.

Экспоненциальная запись: %E или %e

Когда в главе 3 речь зашла об экспоненциальной записи, я пообещал показать, как вывести любое число в такой форме. Наконец настал момент выполнить обещание:

```
>>> number = 12.3456
>>> print '%e' % number
1.234560e+01
```

Форматирующая строка `%e` используется для вывода чисел в экспоненциальном представлении. По умолчанию всегда выводятся шесть знаков после десятичной точки.

Уменьшить или увеличить количество знаков после точки можно, вставив `.n` после знака `%`, как мы делали с вещественными числами:

```
>>> number = 12.3456
>>> print '%.3e' % number
1.235e+01
>>> print '%.8e' % number
1.23456000e+01
```

Запись `%.3e` округляет до трех знаков после точки, а написав `%.8e`, вы добавите к числу несколько нулей, заполняющих свободные разряды.

Буква `e` может принадлежать как к верхнему, так и к нижнему регистрам, это влияет только на регистр выводимой на экран буквы:

```
>>> print '%E' % number
1.234560E+01
```

Автоматический выбор формата: %G или %g

Если вы хотите, чтобы интерпретатор Python сам выбрал, в каком виде ему выводить число, в десятичной или экспоненциальной форме, воспользуйтесь форматирующей

строкой %g. И снова, если вы хотите получить в итоговой записи E в верхнем регистре, именно в этом регистре должна быть ваша форматирующая строка:

```
>>> number1 = 12.3
>>> number2 = 456712345.6
>>> print '%g' % number1
12.3
>>> print '%g' % number2
4.56712e+08
```

Надеюсь, вы заметили, что интерпретатор Python автоматически выбрал экспоненциальную запись для большого числа и обычную десятичную запись для небольшого значения?

Вывод знака процента

Возможно, вы уже задаете себе вопрос, каким же образом выводится фигурирующий в форматирующей строке специальный символ процента (%)?

Прежде всего иногда интерпретатор Python сам в состоянии разобраться, когда знак % начинает форматирующую строку, а когда он является частью текста. Запустите этот код:

```
>>> print 'Я на 90% решил задания по математике!'
Я на 90% решил задания по математике!
```

В данном случае второй знак % отсутствует, кроме того, нет форматируемой переменной, поэтому интерпретатор Python полагает, что знак % является частью строки.

Но если при выводе вы используете форматирующую строку и хотите, чтобы результат вывода содержал знак процента, укажите его дважды. Аналогичным образом осуществлялся вывод обратной косой черты. Говорят, что первый знак процента *экранирует* второй, как это было в ранее приведенном примере с косыми чертами:

```
>>> math = 75.4
>>> print 'Я решил %.1f%% заданий по математике' % math
Я решил 75.4% заданий по математике
```

Первый знак % начинает форматирующую строку. Два подряд знака %% сообщают интерпретатору Python, что вы хотите отобразить символ %. После строки стоит третий знак %, указывающий, что следом идет предназначенная для вывода переменная.

Несколько форматирующих строк

Как быть, если вы хотите поместить в одной инструкции `print` несколько форматирующих строк? Вот как это делается:

```
>>> math = 75.4
>>> science = 82.1
>>> print 'Я решил %.1f заданий по математике и %.1f по физике' % (math,
science)
```

В инструкцию `print` можно поместить произвольное количество форматирующих строк, после чего остается указать кортеж подлежащих выводу переменных. Напоминаю, что кортеж отличается от списка круглыми скобками взамен квадратных и невозможностью редактирования. В данном случае интерпретатор Python проявляет разборчивость — можно воспользоваться кортежем, но нельзя списком. Единственным исключением является случай форматирования одной переменной; при этом кортеж не нужен. (Именно этот случай рассматривался в большинстве примеров.) Количество форматирующих строк (внутри кавычек) должно совпасть с количеством переменных (вне кавычек), в противном случае вы увидите сообщение об ошибке.

Сохранение форматированных чисел

Иногда отформатированное число предназначено не для вывода, а для сохранения в виде строки и дальнейшего использования. Тогда достаточно назначить его переменной:

```
>>> my_string = '%.2f' % 12.3456
>>> print my_string
12.35
>>> print 'Ответ', my_string
Ответ 12.35
```

Мы не стали сразу выводить отформатированное число на экран, а сохранили его в переменной `my_string`. Затем эту переменную скомбинировали с текстом и вывели полученное предложение.

Сохранение отформатированных чисел в виде строк часто требуется в GUI-интерфейсах и других графических программах, например в играх. Как только вы получите переменную с отформатированной строкой, ее можно отобразить, как вам будет угодно: в текстовом поле, на кнопке, в диалоговом окне или на игровом экране.

Новый способ форматирования

Изученный вами синтаксис форматирующих строк работает во всех версиях Python. Но в Python 2.6 и более поздних версиях поддерживается также альтернативный способ форматирования. Так как при выполнении упражнений из данной книги используется версия Python 2.7, имеет смысл рассмотреть этот альтернативный способ. Вы уже могли видеть его в примерах кода, а значит, представляете, о чем идет речь. Впоследствии вы

сможете выбрать, какой из вариантов синтаксиса — старый или новый — вам больше нравится использовать для форматирования строк.

Метод format()

Python-строки (в версии 2.6 и более поздних версиях) снабжены методом `format()`. Он функционирует аналогично форматирующим строкам `%`, с которыми вы уже познакомились. Спецификаторы формата — `f`, `g`, `e` и другие — в обоих случаях одни и те же, просто немного по-разному применяются. Лучше всего рассмотреть это на примере.

Старый способ:

```
print 'Я решил %.1f заданий по математике, %.1f заданий по физике' %
(math, science)
```

Новый способ:

```
print 'Я решил {0:.1f} заданий по математике, {1:.1f} зданий по физике'.
format(math, science)
```

Во втором случае спецификатор формата не предваряется знаком `%`, а помещается в фигурные скобки. Цифры 0 и 1 указывают интерпретатору Python, какую переменную из кортежа следует форматировать.

Напоминаю, что отсчет в Python начинается с нуля, поэтому первый элемент кортежа (переменная `math`) имеет индекс 0, а вторая (переменная `science`) — индекс 1. После чего остается написать `.1f`, как это делалось в старой версии.

Вот и все, что нужно сделать. Отформатированную строку можно сохранить в переменной, как и в случае с форматированием при помощи символа `%`:

```
distance = 149597870700
myString = 'От Солнца до Земли {0:.4e} метра'.format(distance)
```

Так как в данном случае символ `%` для форматирования строки не применяется, никаких специальных действий для его вывода на экран выполнять не нужно:

```
>>> print 'Я сделал {0:.1f}% заданий по математике'.format(math)
Я сделал 87% заданий по математике
```

Программисты, пишущие на языке Python, предпочитают пользоваться методом `format()`, особенно в Python 3. Но вы можете действовать в соответствии с собственными предпочтениями. Во всех примерах из этой книги форматирование осуществлялось при помощи символа `%`.

Операции с текстовыми строками

При первом знакомстве со строками (в главе 2) вы узнали, что две строки можно объединить при помощи знака +:

```
>>> print 'кошка' + 'собака'
кошкасобака
```

Пришло время познакомиться и с другими операциями со строками.

На самом деле строки в Python являются *объектами* (как видите, объектом является все…) и обладают собственными *методами* для выполнения таких действий, как поиск, разбиение и объединение. Они называются *строковыми методами*. В эту группу входит и метод `format()`, с которым вы познакомились в предыдущем разделе.

Разбиение строк

Иногда возникает необходимость разбить длинную строку на ряд фрагментов. Обычно это делается в определенной точке, в которой находится какой-то символ. Так, данные, хранящиеся в текстовом файле, часто отделяются запятыми. Представьте список имен:

```
>>> name_string = 'Sam,Brad,Alex,Cameron,Toby,Gwen,Jenn,Connor'
```

Допустим, эту строку нужно превратить в список, каждому элементу которого будет соответствовать только одно имя. Значит, строку нужно разбить в точках, где располагаются запятые. В Python эта операция выполняется методом `split()`. Вот как он работает:

```
>>> names = name_string.split(',')
```

Вы указываете символ, который будет использоваться как маркер разбиения, а метод возвращает вам список, сформированный из разбитой на части исходной строки. Если вывести на экран результат работы метода, мы получим список отдельных имен:

```
>>> print names
['Sam','Brad','Alex','Cameron','Toby','Gwen','Jenn','Connor']
>>> for name in names:
print name
Sam
Brad
Alex
Cameron
Toby
Gwen
Jenn
Connor
```

Маркером разбиения могут стать и несколько символов. К примеру, выбрав на эту роль имя `'Toby,'`, мы получим вот такой список:

```
>>> parts = name_string.split('Toby,')
>>> print parts
['Sam,Brad,Alex,Cameron', 'Gwen,Jenn,Connor']
>>> for part in parts:
print part
Sam,Brad,Alex,Cameron
Gwen,Jenn,Connor
```

На этот раз строка разбивается на две части: имена, расположенные справа от имени `'Toby,'`, и имена, находящиеся слева от него. Обратите внимание, что имя `'Toby,'` в списке отсутствует. Дело в том, что маркер разбиения отбрасывается.

Можно вообще не указывать маркер разбиения. В этом случае интерпретатор Python разобьет строку по любым *разделителям*:

```
>>> names = name_string.split()
```

К разделителям относятся пробелы, символы табуляции и символы перевода строк.

Соединение строк

Вы только что узнали, как разбить строку на фрагменты. А как насчет обратной операции — соединения двух и более строк с целью получения одной большой строки? В главе 2 вы узнали, что соединить строки можно при помощи оператора `+`. Эта операция напоминает сложение двух строк и называется *конкатенацией*.

Но существует и альтернативный инструмент — функция `join()`. Достаточно указать ей, какие строки вы хотите объединить и какие символы следует вставить между ними. По сути она является антонимом метода `split()`. Вот пример ее применения в интерактивном режиме:

```
>>> word_list = ['Меня', 'зовут', 'Уоррен']
>>> long_string = ' '.join(word_list)
>>> long_string
'Меня зовут Уоррен'
```

Признаю, что это выглядит несколько странно. Предназначенные для вставки между соединяемыми строками символы указываются *перед* функцией `join()`. В данном случае между словами нужно было вставить пробелы, поэтому мы написали `' '.join()`. Это отличается от того, что ожидают увидеть большинство пользователей, но именно так работает метод `join()` в Python. А следующий пример заставляет меня лаять по-собачьи:

```
>>> long_string = ' ГАВ ГАВ '.join(word_list)
>>> long_string
'Меня ГАВ ГАВ зовут ГАВ ГАВ Уоррен'
```

Другими словами, строка, указываемая перед функцией `join()`, служит как бы *клеем*, соединяющим фрагменты строк воедино.

Поиск строк

Предположим, вы пишете для своей мамы программу, которая будет показывать рецепты блюд в GUI-интерфейсе. Список ингредиентов хранится отдельно от рецептов. Предположим, рецепт выглядит так:

```
Шоколадный торт
Ингредиенты:
2 яйца
75 г муки
1 ч. л. пекарского порошка
500 г шоколада
Рецепт:
Разогрейте духовку до 180 градусов
Смешайте все ингредиенты
Выпекайте 30 минут
```

Предположим также, что все строки рецепта представляют собой *список*, то есть каждая из них является отдельным элементом этого списка. Как же найти «Рецепт»? В Python для этой цели есть специальные методы.

Метод `startswith()` позволяет узнать, начинается ли строка с определенного символа или символов. Проще показать его работу на примере в интерактивном режиме:

```
>>> name = "Frankenstein"
>>> name.startswith('F')
True
>>> name.startswith("Frank")
True
>>> name.startswith("Flop")
False
```

Имя Frankenstein начинается с буквы *F*, поэтому первая попытка применения метода дала результат `True`. Еще имя Frankenstein начинается с буквосочетания *Frank*, поэтому во втором случае применения метода мы тоже получили `True`. Но буквосочетание *Flop* в начале этого имени *отсутствует*, поэтому здесь метод дал результат `False`.

Так как метод `startswith()` возвращает значения `True` и `False`, его можно использовать для сравнения в инструкции `if`:

```
>>> if name.startswith("Frank"):
    print "Можно я буду звать тебя Фрэнк?"
```

Существует аналогичный метод `endswith()`, проверяющий, заканчивается ли строка определенным символом или символами:

```
>>> name = "Frankenstein"
>>> name.endswith('n')
True
>>> name.endswith('stein')
True
>>> name.endswith('stone')
False
```

Теперь вернемся к поставленной задаче. Для поиска строки «Рецепт» можно использовать такой код:

```
i = 0
while not lines[i].startswith("Рецепт"):
    i = i + 1
```

Цикл работает, пока не будет обнаружена строка «Рецепт». Напоминаю, что в записи `lines[i]` *индекс* `i` используется для маркировки строк. Поэтому отсчет начинается с `lines[0]` (первая строка), затем идет `lines[1]` (вторая строка) и т. д. В момент завершения цикла `while` индекс `i` укажет номер строки, начинающейся словом «Рецепт», которую мы, собственно, и ищем.

Поиск в строке

Методы `startswith()` и `endswith()` ищут сочетания букв в начале и в конце строки. Но как быть, если нам нужно найти символы в середине строки?

Предположим, у нас есть набор адресов:

```
657 Maple Lane
47 Birch Street
95 Maple Drive
```

И мы хотим найти все адреса на улице «Maple». Ни один из элементов списка не начинается и не заканчивается словом «Maple», при этом два из них содержат это слово. Как же их найти?

На самом деле ответ на этот вопрос вы уже знаете. Когда мы рассматривали списки (в главе 12), вы видели, что принадлежность элемента к списку можно проверить вот так:

```
if someItem in my_list:
    print "Найдено!"
```

Для проверки вхождения элемента в список мы пользовались ключевым словом `in`. Но оно применимо и к строкам. Ведь строка, по сути, представляет собой список символов, а, значит, можно написать:

```
>>> addr1 = '657 Maple Lane'
>>> if 'Maple' in addr1:
        print "Этот адрес содержит улицу 'Maple'."
```

Ключевое слово `in` сообщает нам, *присутствует ли* в проверяемой строке нужная подстрока. Но ее местоположение нам не указывается. Для его поиска требуется уже метод `index()`. Как и в случае списков, метод `index()` сообщает, в каком месте большой строки начинается подстрока. Вот пример:

НОВЫЕ СЛОВА

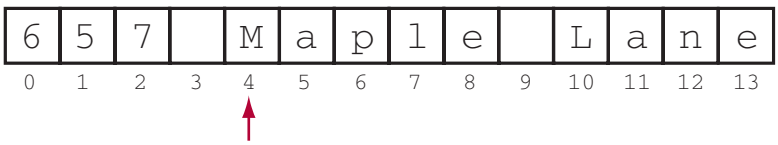
При поиске маленькой строки внутри большой, например при поиске слова «Maple» в адресе «657 Maple Lane», маленькая строка называется *подстрокой*.

```
>>> addr1 = '657 Maple Lane'
>>> if 'Maple' in addr1:
        position = addr1.index('Maple')
        print "обнаружено слово 'Maple' в позиции", position
```

Запустив этот код, вы получите следующий результат:

```
Обнаружено слово 'Maple' в позиции 4
```

Слово `Maple` начинается с позиции 4 строки `"657 Maple Lane"`. Как и в случае списков, индексы (или позиции) символов в строке начинаются с нуля, поэтому буква `M` находится в позиции 4.



Обратите внимание, что перед применением метода `index()` мы проверили наличие подстроки `"Maple"` *внутри* большой строки. Дело в том, что при отсутствии в строке искомого объекта метод `index()` выдает ошибку. Проверка при помощи ключевого слова `in` гарантирует отсутствие ошибки. Аналогично мы поступали со списками в главе 12.

Удаление части строки

Иногда необходимо удалить, или *обрезать* (strip off), часть строки. Обычно отбрасывается нечто в конце, например символ новой строки или лишние пробелы. В Python эту работу выполняет метод `strip()`. Достаточно сообщить ему, от какого фрагмента вы хотите избавиться:

```
>>> name = 'Warren Sande'
>>> short_name = name.strip('de')
>>> short_name
'Warren San'
```

В данном случае из моей фамилии были убраны буквы `de`. Если бы этих букв не было, со строкой бы ничего не произошло:

```
>>> name = 'Bart Simpson'
>>> short_name = name.strip('de')
>>> short_name
'Bart Simpson'
```

Можно не указывать методу `strip()`, что именно вы хотите отсечь. В этом случае он уберет все разделители. Мы уже говорили, что этим термином обозначаются все пробелы, символы табуляции и перевода строк. Чтобы избавиться от лишних пробелов в конце, достаточно написать:

```
>>> name = "Warren Sande     "
>>>> short_name = name.strip()
>>> short_name
'Warren Sande'
```

Видите лишние пробелы после моего имени?

Теперь лишние пробелы после моего имени удалены. При этом мне не понадобилось отдельно сообщать методу `strip()`, сколько именно пробелов подлежит удалению. Он убрал все лишние разделители, обнаруженные в конце строки.

Изменение регистра

Покажу еще два метода. Они позволяют менять буквы строки со строчных на прописные, и наоборот. Иногда нужно сравнить две строки, например «Hello» и «hello», чтобы понять, совпадают ли в них буквы. Так что сделаем все буквы в строках строчными.

Для этой цели используется метод `lower()`. Запустите в интерактивном режиме код:

```
>>> string1 = "Hello"
>>> string2 = string1.lower()
>>> print string2
hello
```

Существует также парный метод `upper()`:

```
>>> string3 = string1.upper()
>>> print string3
HELLO
```

Можно получить копии строк с буквами в нижнем (или верхнем) регистре и сравнить их, чтобы понять, совпадают ли они друг с другом.

Что мы узнали

В этой главе мы научились:

- регулировать межстрочный интервал (добавляя или удаляя символы перевода строк);
- регулировать горизонтальный интервал при помощи табуляции;
- отображать числа в различном виде при помощи форматирующих строк;
- осуществлять форматирование двумя способами: при помощи символа `%` и метода `format()`;
- разбивать строки методом `split()` и соединять их методом `join()`;
- осуществлять поиск в строках методами `startswith()`, `endswith()` и `index()`, а также при помощи ключевого слова `in`;
- удалять символы, расположенные в конце строки, методом `strip()`;
- менять регистр букв в строке на верхний или нижний методами `upper()` и `lower()`.

Проверь себя

1. Пусть имеются две инструкции `print`:

```
print "И как же"
print "тебя зовут?"
```

   Как вывести все в одну строку?

2. Как при выводе добавить пустые строки?
3. Каким образом осуществляется выравнивание выводимого текста по вертикали?
4. Какая форматирующая строка позволяет вывести число в экспоненциальном представлении?

Эксперименты

1. Напишите программу, которая будет спрашивать у пользователя его имя, возраст и любимый цвет и выводить полученную информацию в одну строку. Результат работы такой программы должен выглядеть так:

```
>>> ======================== RESTART ========================
>>>
Как тебя зовут? Петр
Сколько тебе лет? 12
Какой твой любимый цвет? зеленый
Тебя зовут Петр тебе 12 лет твой любимый цвет зеленый
```

2. Помните программу, выводящую таблицу умножения, которую мы писали в главе 8 (см. листинг 8.5)? Напишите новую версию с применением знаков табуляции для выравнивания чисел в столбцах.
3. Напишите программу, вычисляющую все дроби с дробной частью 8 (то есть 1/8, 2/8, 3/8, … до 8/8) и отображающую их с точностью до трех знаков после десятичной точки.

ГЛАВА 22

Файловый ввод и вывод

Вы задумывались, каким образом ваша любимая компьютерная игра запоминает максимальное количество набранных очков даже после выключения компьютера? Или как браузер запоминает ваши любимые сайты? В этой главе вы найдете ответ.

Уже неоднократно упоминалось, что каждая программа «стоит на трех китах»: вводе, обработке и выводе данных. До этого момента ввод данных осуществлялся в основном самим пользователем с помощью клавиатуры или мыши. А вывод шел непосредственно на экран (или в колонки в случае звука). Но иногда возникает необходимость воспользоваться данными из других источников. Зачастую программам требуется информация, не вводимая в процессе их работы, а хранящаяся в каком-то месте. Скажем, данные могут быть взяты из файла на жестком диске вашего компьютера.

Например игре Виселица потребуется список слов, из которых будет выбираться загаданное слово. Этот список должен где-то храниться, скорее всего, в файле, поставляемом вместе с программой. И программа должна будет открыть это файл, прочитать список и выбрать оттуда слово.

Все сказанное верно и для процедуры вывода. Порой результат работы программы требуется сохранить. Все задействованные в программе переменные являются временными, их значения теряются после прекращения работы. И если вы хотите сохранить некую информацию для последующего применения, нужно использовать более надежное хранилище, например жесткий диск компьютера. Так, список лучших результатов игры нужно держать в файле. В этом случае при следующем запуске программа сможет прочитать эти данные и вывести их на экран.

В этой главе вы узнаете, как открыть, прочитать и записать файл.

Что такое файл

Перед тем как рассмотреть выполняемые с файлами операции — открытие, чтение и запись, — нужно понять, что представляет собой файл.

Говорят, что компьютеры хранят информацию в *двоичном* формате, в котором из всех цифр используются только единицы и нули. Каждая единица или ноль называется *битом*,

а группа из восьми битов составляет *байт*. Файл представляет собой набор байтов, имеющий имя и хранящийся на жестком диске компьютера, компакт-диске, DVD-диске, флэш-диске или любом другом накопителе.

Файлы хранят данные разных типов. Это может быть текст, рисунок, музыка, компьютерная программа, список телефонных номеров и т. п. Все что хранится на жестком диске вашего компьютера, хранится в виде файлов. Из одного или нескольких файлов состоят все программы. Операционная система вашего компьютера (например Windows, Mac OS X или Linux) включает в себя множество файлов, необходимых для ее запуска. Файлы имеют следующие свойства:

- имя;
- тип, определяющий хранимые в файле данные (графику, музыку, текст);
- местоположение (где именно хранится файл);
- размер (количество байтов в файле).

Имена файлов

В большинстве операционных систем (в том числе в Windows) часть имени файла информирует о типе хранящихся в нем данных. Каждое имя файла содержит хотя бы одну точку. Часть имени, находящаяся после точки, и указывает на тип файла. Эта часть называется *расширением* (extension).

Вот несколько примеров:

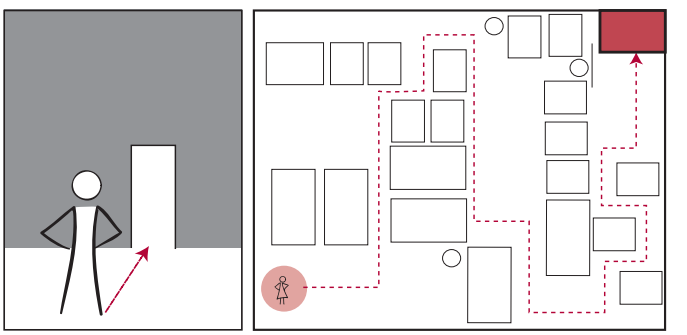
- у файла ***my_letter.txt*** расширение ***.txt***, что является сокращением от слова «text» и указывает на наличие внутри текстовой информации;
- у файла ***my_song.mp3*** расширение ***.mp3***, которое говорит о формате MP3, относящемся к звуковым файлам;
- у файла ***my_program.exe*** расширение ***.exe***, что является сокращением от слова «executable» (исполняемый). Такое расширение имеют исполняемые программы;
- у файла ***my_cool_game.py*** расширение ***.py***, что обычно соответствует Python-программам.

В операционной системе Mac OS X файлы программ имеют расширение *.app*, что является сокращением от слова «application» (приложение). Приложение — это то же самое, что и программа.

Вы можете присваивать файлам любые имена и давать любые расширения. Можно создать текстовый файл (например в Блокноте) и назвать его, к примеру, ***my_notes.mp3***. Это *не* превратит его в звуковой файл. Внутри все равно останется текст, так что на

самом деле это будет текстовый файл. Просто вы присвоите ему расширение, *присущее* звуковым файлам, что, скорее всего, станет причиной путаницы. Поэтому при выборе имен имеет смысл придерживаться общепринятых расширений, совпадающих с типом хранящихся в файле данных.

Местоположение файла

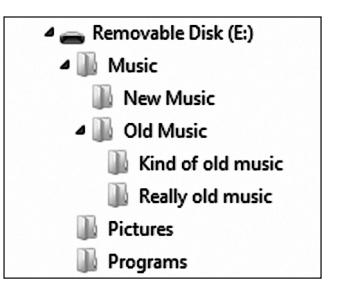


До этого момента мы работали с файлами, хранящимися в одной папке с самой программой. Нам не приходилось думать о том, как найти эти файлы в будущем.

Можно привести такую аналогию. Когда вы находитесь в собственной комнате, вам не приходится ломать голову над тем, как найти шкаф. Он стоит вон там. Но если вы переместитесь в другую комнату, другой дом или вообще в другой город, доступ к шкафу значительно усложнится!

Все файлы *где-то* хранятся, поэтому в дополнение к имени каждый файл имеет местоположение. Жесткие диски и прочие накопители разбиты на *папки*, или *каталоги*. Папка и каталоги — это два названия одной и той же вещи, служащей для группирования файлов. Способ расположения папок друг относительно друга и их связь между собой называется *структурой папок*.

В операционной системе Windows каждому накопителю присваивается буква, например ***C*** для жесткого диска и ***E*** для флэш-диска. В операционных системах Mac OS X и Linux все накопители имеют имя (например ***hda*** или ***FLASH DRIVE***). Каждый накопитель можно поделить на множество папок, к примеру ***Music*** (музыка), ***Pictures*** (изображения), ***Programs*** (программы). В Windows Explorer это будет выглядеть так:



В папки могут быть вложены другие папки, а те, в свою очередь, содержат еще папки, и т. д. Вот пример папок трех уровней вложенности:

Первый уровень — *Music*. На следующем уровне находятся папки *New Music* (Новая музыка) и *Old Music* (Старая музыка), а еще ниже располагаются папки *Kind of old music* (Жанры старой музыки) и *Really old music* (Реально старая музыка).

НОВЫЕ СЛОВА

Папки, вложенные в другие папки, называются *подпапками*. Если вы предпочитаете пользоваться термином «каталоги», то они будут называться *подкаталогами*.

В процессе поиска файла или папки в программе Windows Explorer (или в любом другом диспетчере файлов) папки напоминают ветви дерева. Корнем является само дерево, например *C:* или *E:*. Каждая папка верхнего уровня подобна толстой ветке. Вложенные в них папки напоминают более мелкие ветки, и т. д.

Однако для доступа к файлу из программы такая модель не совсем подходит. Программа не умеет щелкать по папкам и рассматривать ветви в поисках отдельных файлов. Ей требуется более простой способ. К счастью, древовидную структуру можно представить и по-другому. В адресной строке Windows Explorer при переходе в различные папки можно увидеть такую запись:

```
E:\Music\Old Music\Really old music\my_song.mp3
```

Это так называемый *путь* (path). Он описывает местоположение файла в структуре папок. Этот конкретный путь расшифровывается следующим образом.

1. Начинаем поиск с диска *E:*.
2. Выполняем переход в папку *Music*.
3. В папке *Music* открываем подпапку *Old Music*.
4. В папке *Old Music* открываем подпапку *Really old music*.
5. В папке *Really old music* находим файл *my_song.mp3*.

Подобный путь позволяет найти любой файл на вашем компьютере. Именно его программы используют для поиска и открытия файлов. Вот пример:

```
image_file = "c:/program files/HelloWorld/examples/beachball.png"
```

Вы всегда можете получить доступ к файлу, воспользовавшись полным путем к нему. Полный путь включает имя файла и имена всех папок, вплоть до корня (имени диска, например *C:*). В приведенном примере мы имеем полный путь к файлу.

Прямая или обратная косая черта?

Крайне важно корректно пользоваться косыми чертами (\ и /). В операционной системе Windows полное имя файла может записываться как с прямой (/), так и с обратной косой чертой (\), но написав в программе Python `c:\test_results.txt`, вы получите проблему из-за записи `\t`. Помните, в главе 21 мы рассматривали специальные символы, отвечающие за форматирование вывода? Запись `\t` означает табуляцию. Поэтому в маршрутах доступа к файлам символа \ лучше избегать. Интерпретатор Python (и Windows) сочтут запись `\t` символом табуляции, а не частью имени файла. Поэтому лучше пользоваться символом /.

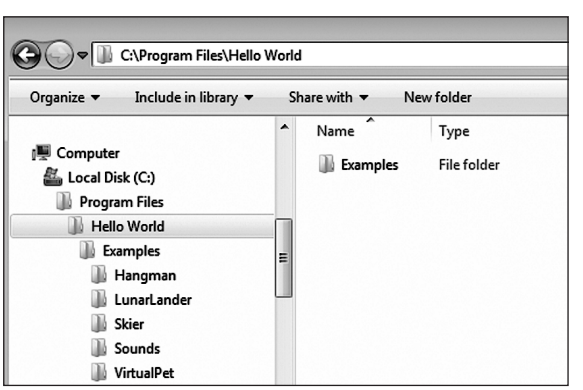
Альтернативой является применение двух обратных косых черт:

```
image_file "c:\\program files\\HelloWorld\\images\\beachball.png"
```

Напоминаю, что для вывода символа \ перед ним нужно поместить еще одну обратную косую черту. В именах файлов это тоже работает. Но я бы порекомендовал вам пользоваться обычной косой чертой (/).

Иногда полный путь к файлу не требуется. В следующем разделе мы поговорим о способах поиска файла в ситуации, когда половина пути к нему уже пройдена.

Где я нахожусь?



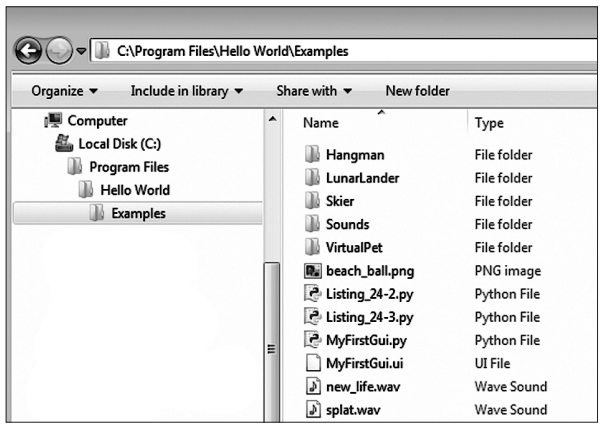
В большинстве операционных систем (в том числе в Windows) существует понятие *рабочей папки* (working directory), иногда называемой *текущей рабочей папкой* (current working directory). Так называется папка, в которой вы в данный момент находитесь.

Представьте, что вы начали с корня (*C:*) и сместились по ветке *Program Files* до папки *Hello World*. Тогда ваше текущее местоположение, или *рабочая папка*, — *C:/Program Files/Hello World*.

Теперь для доступа к файлу *beachball.png* нужно перейти по ветке *Examples*. То есть вы проделываете путь */Examples/beachball.png*. Так как половина пути в нужном направлении уже пройдена, для попадания в нужную точку остается пройти только остаток пути.

Помните, в главе 19 при работе со звуком мы открывали звуковой файл *splat.wav* и другие? Путь при этом мы не указывали. Это стало возможным, поскольку я попросил

вас перед началом работы скопировать звуковые файлы в папку с программой. В Windows Explorer результат ваших действий будет выглядеть так:



Обратите внимание, что Python-файлы (с расширением *.py*) находятся в одной папке со звуковыми файлами (с расширением *.wav*). При запуске Python-программы ее *рабочей папкой* будет та, в которую сохранен файл с расширением *.py*.

Если вы сохраните программу в папке *e:/programs*, после запуска программы ее *рабочей папкой* будет *e:/programs*. Положите в эту папку звуковые файлы, и тогда программе понадобятся только их имена. Указывать маршрут к ним не потребуется, ведь файлы уже там. Поэтому можно написать так:

```
my_sound = pygame.mixer.Sound("splat.wav")
```

Обратите внимание, что вам не нужно использовать полный путь к звуковому файлу (то есть *e:/programs/splat.wav*). Вы указываете только имя этого файла, так как он находится в одной папке с открывающей его программой.

Хватит о путях!

Полная информация о папках, путях, рабочих каталогах и т. п. очень велика, и детальное объяснение может растянуться на много страниц. Но эта книга посвящена не операционным системам, местоположениям файлов и путям, а программированию, поэтому если вы не смогли разобраться в данной теме самостоятельно, попросите родителей, учителя или кого-то, кто разбирается в компьютерах, помочь вам.

Во всех примерах в книге нужные файлы находятся в одной папке с использующей их программой, поэтому вам не нужно беспокоиться о путях и полных именах файлов.

Открытие файла

Перед тем как открыть файл, нужно понять, что вы с ним собираетесь делать.

- Если файл будет использоваться как *источник данных* (его содержимое просматривается, но не редактируется), он открывается для *чтения*.
- Если вы *создаете* новый файл или *заменяете* существующий файл новым, файл открывается для *записи*.
- Если вы *добавляете* данные в существующий файл, он открывается для *добавления*.

Открыв файл, вы с точки зрения интерпретатора Python создаете *файловый объект*. (Помните, я говорил вам, что для интерпретатора Python многие вещи являются объектами?) Объект создается с помощью функции `open()` с именем файла в качестве аргумента:

```
my_file = open('my_filename.txt','r')
```

Имя файла представляет собой *строку*, поэтому его нужно заключить в кавычки. Строка `'r'` указывает, что файл открыт для чтения. Более подробно эта тема рассматривается в следующем разделе.

Важно понимать разницу между *файловым объектом* и *именем файла*. *Файловый объект* является средством доступа к файлу, применяемому программой. А *имя файла* — это средство обращения к файлу на диске, применяемое операционной системой Windows (или Linux, или Mac OS X).

В реальной жизни дела обстоят точно так же. В разных обстоятельствах одного и того же человека называют по-разному. Если учителя зовут Фред Уисли, ученики, скорее всего, зовут его мистер Уисли. Его друзья, скорее всего, зовут его Фред, а его учетная запись на компьютере называется fweasley. У файла есть имя, применяемое операционной системой для сохранения его на диске (имя файла), и имя, которым пользуется программа при работе с этим файлом (файловый объект).

Эти два имени — имя объекта и имя файла — могут не совпадать. Объект можно назвать как угодно. Например, если у вас есть текстовый файл с заметками, который называется *notes.txt*, можно написать так:

```
notes = open('notes.txt', 'r')
```

Файловый объект *Имя файла*

Или так:

```
some_crazy_stuff = open("notes.txt", 'r')
```

Файловый объект *Имя файла*

После того как вы открыли файл и создали файловый объект, имя файла уже не требуется. Все действия с файлом в программе выполняются через его файловый объект.

Чтение файла

Итак, при помощи функции `open()` вы открываете файл и создаете файловый объект. Это одна из встроенных Python-функций. Чтобы открыть файл для чтения, используется второй аргумент `'r'`:

```
my_file = open('notes.txt', 'r')
```

Попытавшись открыть для чтения несуществующий файл, вы получите сообщение об ошибке. (В конце концов, невозможно прочитать то, чего нет, не так ли?)

В Python есть еще пара встроенных функций для получения информации из открытого файла. За чтение строк текста отвечает метод `readlines()`:

```
lines = my_file.readlines()
```

Он прочитает файл целиком и создаст список, элементами которого будут строки текста. Предположим, файл *notes.txt* содержит перечень на сегодня:

```
Wash the car (Помыть машину)
Make my bed (Застелить постель)
Collect allowance (Сделать заначку)
```

Создать подобный файл можно, к примеру, в программе Блокнот или в другом текстовом редакторе. Присвойте ему имя *notes.txt* и сохраните в папку для Python-программ. Закройте Блокнот.

Открыть файл с помощью Python-программы и прочитать его позволит небольшой код из листинга 22.1.

Листинг 22.1. Открытие файла и чтение из него

```
my_file = open('notes.txt', 'r')
lines = my_file.readlines()
print lines
```

Результат запуска этой программы будет выглядеть так (в зависимости от содержимого вашего файла):

```
>>>====================== RESTART ========================
>>>
['Wash the car\n', 'Make my bed\n', 'Collect allowance']
```

Строки текста были считаны из файла и помещены в список `lines`. Каждый элемент списка представляет собой одну строку из файла. Обратите внимание на символы `\n` в конце первых двух строк. Это символы *перевода строк*, разделяющие строки в файле. Это аналог нажатия клавиши *Enter*, которое мы выполняли при создании файла. Если после третьей строки тоже нажать *Enter*, символы `\n` появятся и после третьего элемента списка.

В программу из листинга 22.1 осталось добавить одну вещь. Закончив работу с файлом, мы должны его закрыть:

```
my_file.close()
```

Понимаешь, Картер, если этим файлом захочет воспользоваться другая программа, тот факт, что мы его не закрыли, может помешать доступу к нему. Поэтому в общем случае лучше закрывать файлы, работа с которыми уже завершена.

Теперь, когда файл оказался в нашей программе в виде набора строк, с ним можно делать все что угодно. Этот список ничем не отличается от прочих Python-списков. Его можно просматривать в цикле, сортировать, добавлять элементы, удалять элементы и т. п. Строки ничем не отличаются от любых других строк. Их можно выводить на экран, преобразовывать в типы `int` и `float` (если они содержат числа), использовать как метки в GUI и выполнять все прочие доступные для строк операции.

Построчное чтение

Метод `readlines()` читает все строки, пока файл не закончится. Для чтения по одной строке за раз используется метод `readline()`:

```
first_line = my_file.readline()
```

В данном случае будет прочитана только первая строка файла. При повторном использовании метода `readline()` в рамках той же программы интерпретатор Python запоминает предшествующее положение. Соответственно, при втором вызове этого метода будет прочитана вторая строка файла. Пример его применения показан в листинге 22.2.

Листинг 22.2. Многократное применение метода readline()

```
my_file = open('notes.txt', 'r')
first_line = my_file.readline()
second_line = my_file.readline()
print "первая строка = ", first_line
print "вторая строка = ", second_line
my_file.close()
```

Вот как будет выглядеть результат работы этой программы:

```
>>>================== RESTART ====================
>>>
первая строка = Помыть машину
вторая строка = Застелить постель
```

Метод `readline()` читает по одной строке за раз, поэтому ему нет нужды помещать результат в список. Каждое применение метода `readline()` дает вам одну строку.

Возвращение к началу

Если после многократного применения метода `readline()` возникает необходимость вернуться к началу файла, воспользуйтесь методом `seek()`:

```
first_line = my_file.readline()
second_line = my_file.readline()
my_file.seek(0)
first_line_again = my_file.readline()
```

Метод `seek()` заставляет интерпретатор Python перейти к указанному вами месту в файле. Число в скобках обозначает количество байтов от начала файла. Тот есть параметр 0 возвращает вас в самое начало.

Текстовые и бинарные файлы

Приведенные примеры открытия файлов и чтения оттуда предполагали *наличие в файле текста*. Но напоминаю, что текст является всего лишь одним из типов данных, которые можно сохранить в файле. Все остальные виды файлов программисты относят к категории, которая называется *двоичными*, или *бинарными*, *файлами*.

Существует два основных типа файлов, которые вы можете открыть.

- *Текстовые файлы* содержат внутри текст с буквами, цифрами, знаками препинания и специальными символами, например символами перевода строки.
- *Двоичные файлы* не содержат внутри текста. Внутри может находиться графика, музыка, какие-то данные, но разбиение на строки там отсутствует, так как отсутствуют символы переноса строк.

То есть вы не можете применять методы `readline()` и `readlines()` к двоичным файлам. Попытавшись прочитать «строку» из файла с расширением *.wav*, вы получите неизвестно что. Скорее всего, это будет какая-то такая абракадабра:

```
>>> f = open('splat.wav', 'r')
>>> print f.readline()
RIFFö▲ WAVEfmt ► ☺ ☺ "V "V datap▲
ÇÇÇÇÇÇÇüÇÇÇÇÇÇÇÇÇÇÇÇÇÇÇ⌂ÇÇ⌂⌂⌂⌂⌂Ç⌂⌂⌂⌂Ç
ÇÇÇ⌂⌂ÇÇÇÇÇÇÇÇÇÇÇÇÇÇÇÇÇÇ⌂Ç⌂ÇÇÇÇüÇÇÇÇÇÇÇÇÇÇÇÇ⌂ÇÇÇÇÇüüÇÇÇ⌂ÇÇ⌂ÇÇÇÇ⌂⌂ÇÇüéééÇzvvy{|Çâ
    çïê}trv~ut|⌂yrqrtxÇîÖ₧æäàütvçÆÄ|mlfWR]jnmpxüêÅ ºƒâràó"¼Ö}`ORj⌂{hZZgwàëy{
    äæá⌐¿ÿéåÿèmWLISjÇàzrvÇüytv~üÇ}yrifjt}äêèêëÄöÉémSCFZlrtyéïö¥ñ¬½⌐ñ¢ÆÄìÅôòÆ
    ÄÅæ|åÜ⌐¬ÿüpd\UME@;99:>EJMW]YTZfuçòƒ⌐▒║┴──├────┴╕⌐¢ôë~{|{yxzzuiZNGHLSbs⌂~
    wrnf\TPQU]`jvàæÉ⌂osÇïôæä}üàë₧┼
```

В начале wav-файла мы видим нечто, напоминающее текст, но потом начинается полная бессмыслица. Все дело в том, что wav-файлы не содержат текста. В них содержится музыка. А методы `readline()` и `readlines()` работают только с текстовыми файлами.

В большинстве случаев, когда нам требуется бинарный файл, для его загрузки будет применяться Pygame-модуль или другой модуль, как это было в главе 19:

```
pygame.mixer.music.load('bg_music.mp3')
```

В данном случае об открытии файла и чтении двоичных данных (здесь это музыка) заботится Pygame-модуль.

Тема обработки двоичных файлов выходит за рамки данной книги. Но на всякий случай расскажу вам, что такой файл можно открыть, добавив к переменной, задающей режим открытия, букву *b*:

```
my_music_file = open('bg_music.mp3', 'rb')
```

Сочетание символов `'rb'` означает, что файл открывается для *чтения в двоичном режиме*.

Итак, теперь вы знаете, как добавить в программу содержащиеся в файле данные, то есть выполнить операцию *чтения* из файла. Настало время поговорить о том, как вставить в файл информацию из программы. Эта операция называется *записью* в файл.

Запись в файл

Если вы хотите сохранить результаты работы программы надолго, можно записать их на бумаге. Но в этом случае не совсем понятно, зачем вам нужен компьютер!

Гораздо лучше сохранить информацию на жестком диске, чтобы после завершения работы программы, более того, после выключения компьютера, ваши данные все равно не пропали и были доступны для дальнейшего использования. Вы уже проделывали эту операцию много раз. Каждый раз при сохранении школьного реферата, картинки, песни или Python-программы вы помещали их на жесткий диск своего компьютера.

Как я уже упоминал, поместить информацию в файл можно двумя способами.

- *Запись* — создание нового файла с перезаписью существующего.
- *Добавление* — вставка в существующий файл с сохранением уже имеющегося содержимого.

Но как для записи, так и для добавления файл сначала требуется открыть. Для этого применяется уже знакомая вам функция `open()`, но с измененным вторым параметром.

- Для чтения файл открывается в режиме `'r'`:

```
my_file = open('new_notes.txt', 'r')
```

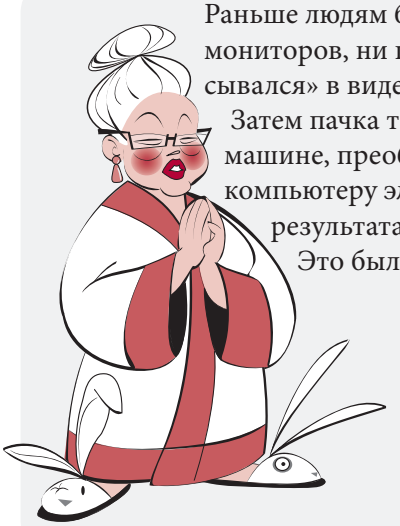
- Для записи файл открывается в режиме `'w'`:

```
my_file = open('new_notes.txt', 'w')
```

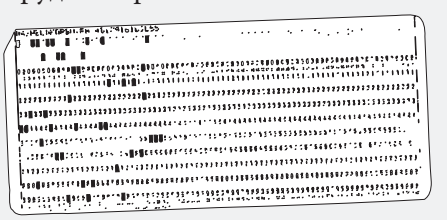
- Для добавления файл открывается в режиме `'a'`:

```
my_file = open('notes.txt', 'a')
```

В СТАРЫЕ ДОБРЫЕ ВРЕМЕНА



Раньше людям была доступна только бумага! Не было ни мониторов, ни принтеров, ни даже клавиатуры. Код «записывался» в виде отверстий в специальных карточках. Затем пачка таких карточек скармливалась большой машине, преобразовывавшей отверстия в понятные компьютеру электрические сигналы. Иногда получение результата занимало несколько дней. Это были трудные времена!



Древняя перфокарта

Указывать параметр `'a'` нужно только для имеющихся на диске файлов, иначе вы получите ошибку. Ведь операция *добавления* возможна только в уже существующий файл.

Картер снова прав!

Параметр `'w'`, задающий режим *записи*, дает нам две возможности.

- Если файл уже существует, его содержимое будет потеряно и замещено новой информацией.
- Если файла не существует, будет создан новый файл с указанным именем, и вся новая информация будет сохранена в нем.

Давайте рассмотрим несколько примеров.

Добавление в файл

Добавим что-нибудь в ранее созданный файл *notes.txt*. Например строку `"Потратить заначку"`. Внимательно рассмотрев пример с методом `readlines()`, вы могли заметить, что в конце последней строки отсутствует символ переноса `\n`. Значит, его нужно добавить и после этого вставить новую строку (листинг 22.3). Для записи строк в файл служит метод `write()`.

Листинг 22.3. Режим добавления

```
todo_list = open('notes.txt', 'a')
todo_list.write('\nПотратить заначку')
todo_list.close()
```

Открываем файл в режиме добавления

Добавляем в конец строку

Закрываем файл

Когда мы рассматривали процесс чтения, я упомянул, что после завершения работы файл следует закрыть. Но куда важнее воспользоваться методом `close()` после *завершения записи*. Дело в том, что внесенные в файл изменения не сохранятся, пока вы его не закроете.

После запуска программы из листинга 22.3 откройте файл *notes.txt* в Блокноте (или любом другом текстовом редакторе) и посмотрите, как поменялось его содержимое. Не забудьте затем закрыть Блокнот.

Запись в файл

Теперь попробуем выполнить запись в файл в режиме записи. Для этого откроем файл, отсутствующий на жестком диске вашего компьютера. Наберите текст листинга 22.4 и запустите его.

Листинг 22.4. Запись в новый файл

```
new_file = open("my_new_notes.txt", 'w')
new_file.write("Съесть ужин\n")
new_file.write("Поиграть в футбол\n")
new_file.write("Пойти спать")
new_file.close()
```

Как узнать, что программа сработала корректно? Проверьте папку, в которой вы сохранили программу из листинга 22.4. Там вы найдете файл *my_new_notes.txt*. Откройте его в Блокноте, и вы увидите такой текст:

```
Съесть ужин
Поиграть в футбол
Пойти спать
```

При помощи программы вы создали текстовый файл и сохранили в нем текст. Этот текст теперь находится на жестком диске и останется там навсегда, по крайней мере на то время, пока существует диск, если вы его не удалите. Итак, вы узнали способ навсегда сохранять данные из ваших программ. Теперь ваши программы могут надолго оставить свой след в истории (или хотя бы на жестком диске вашего компьютера). Все, что должно быть сохранено после того как программа прекратит свою работу, а компьютер будет выключен, следует поместить в файл.

Посмотрим, что произойдет при вставке чего-либо в режиме записи в уже существующий файл. Помните наш файл *notes.txt*? После выполнения кода из листинга 22.3 он выглядит так:

```
Помыть машину
Застелить постель
Сделать заначку
Потратить заначку
```

Откроем его в режиме записи и выполним запись. Для этого нам потребуется код из листинга 22.5.

Листинг 22.5. Запись в существующий файл

```
the_file = open('notes.txt', 'w')
the_file.write("Проснуться\n")
the_file.write("Посмотреть мультик")
the_file.close()
```

Запустите этот код и откройте файл *notes.txt* в Блокноте. Вы увидите такой текст:

```
Проснуться
Посмотреть мультик
```

Ранее содержавшаяся в файле *notes.txt* информация исчезла. Ее заменили новые данные из представленной в листинге 22.5 программы.

Запись в файл командой print

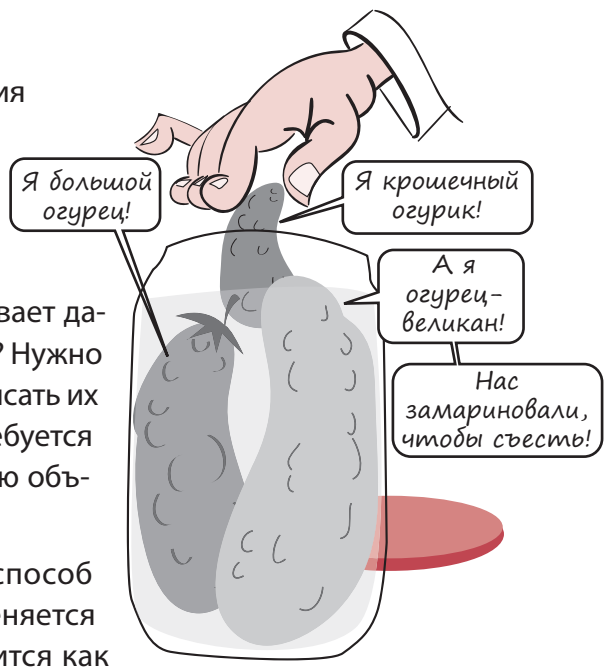
В предыдущем разделе запись в файл осуществлялась при помощи функции `write()`. Но для этой цели прекрасно подходит команда `print`. Разумеется, и в этом случае нам первым делом нужно будет открыть файл для записи:

```
my_file = open("new_file.txt", 'w')
print >> my_file, "Привет, сосед!"
my_file.close()
```

Два символа > (иногда называемые *кавычками «ёлочками»*) сообщают инструкции `print`, что результат ее работы следует отправить не на экран, а в файл. Эта операция называется *переадресацией* (redirecting) результата.

Иногда удобнее пользоваться именно командой `print`, так как она автоматически преобразует числа в строки и т. п. Впрочем, вы можете самостоятельно выбирать, каким способом осуществлять запись в текстовый файл — командой `print` или методом `write()`.

Модуль pickle



В первой части главы речь шла об операциях чтения и записи с текстовыми файлами. Но это лишь один из способов хранения данных на жестком диске. А что делать, если требуется сохранить список или объект?

Список иногда может состоять из строк, но так бывает далеко не всегда. А как быть с сохранением объектов? Нужно преобразовать все свойства объекта в строки и записать их в текстовый файл. Но тогда вам в дальнейшем потребуется совершить обратную операцию по восстановлению объекта, что может оказаться непростой задачей.

К счастью, в Python существует более простой способ сохранения списков и объектов. Для этого применяется модуль `pickle`. Это забавное название переводится как «маринованный огурец». Но давайте вдумаемся. Маринование — это способ сохранения пищи для использования в будущем. В Python вы «маринуете» свои данные, чтобы получить возможность сохранить их на диске и использовать в дальнейшем. Все вполне логично!

Сериализация

Представьте, что у вас есть список различных элементов:

```
my_list = ['Fred', 73, 'Hello there', 81.9876e-13]
```

Для применения команды `pickle` первым делом следует импортировать одноименный модуль:

```
import pickle
```

Затем, чтобы «замариновать» что-нибудь, например список, используйте функцию `dump()`. Это английское слово означает «сброс». Имя функции легко запомнить, если

представить, как огурцы кладут в банку. Функция `dump()` работает с файловыми объектами, и вы уже знаете, каким способом они создаются:

```
pickle_file = open('my_pickled_list.pkl', 'w')
```

Мы открываем файл для *записи* при помощи символа `'w'`, так как нам нужно кое-что там *сохранить*. Имя и расширение файла можно выбрать по своему желанию. Я указал расширение *.pkl* (сокращение от «pickle»).

Теперь остается преобразовать объект в поток байтов (эта процедура называется *сериализацией*) и сохранить их в файле:

```
pickle.dump(my_list, pickle_file)
```

Целиком процесс выглядит так, как показано в листинге 20.6.

Листинг 20.6. Сохранение списка в файле при помощи модуля pickle

```
import pickle
my_list = ['Fred', 73, 'Hello there', 81.9876e-13]
pickle_file = open('my_pickled_list.pkl', 'w')
pickle.dump(my_list, pickle_file)
pickle_file.close()
```

Метод подходит для сохранения в файле всех структур данных. Но как извлечь их назад?

Десериализация

В реальной жизни маринованные овощи невозможно превратить в свежие. Маринование — операция необратимая. Но в Python вы можете не только «сохранить» свои данные с помощью сериализации, но и выполнить обратную операцию — *десериализацию* — и вернуть данным первоначальный вид.

Десериализация выполняется с помощью функции `load()`. Ей передается файловый объект с сериализованными данными, а она возвращает их в исходное состояние. Попробуем это на практике. После запуска программы из листинга 22.6 в папке с этой программой должен был появиться файл *my_pickled_list.pkl*. Запустите программу из листинга 22.7 и посмотрите, получите ли вы назад свой список.

Листинг 22.7. Десериализация с помощью функции load()

```
import pickle
pickle_file = open('my_pickled_list.pkl', 'r')
recovered_list = pickle.load(pickle_file)
pickle_file.close()

print recovered_list
```

Вы должны получить такой результат:

```
['Fred', 73, 'Hello there', 8.1987599999999997e-012]
```

Кажется, десериализация работает! Мы получили из маринованных элементов свежие. Немного другой вид приобрело число в экспоненциальной форме, но оно осталось тем же, по крайней мере до 16-го знака после десятичной точки. Различие возникло из-за *ошибок округления*, о которых мы говорили в главе 4.

В следующем разделе мы воспользуемся тем, что узнали о вводе и выводе файлов, и напишем новую игру.

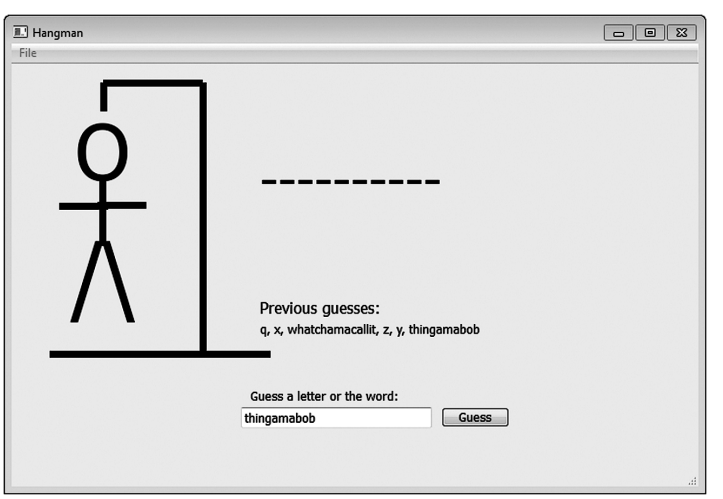
И снова время играть — Виселица

Как в главе, посвященной файлам, оказалась игра? Дело в том, что особый интерес в игре Виселица представляет длинный список слов, из которого выбирается слово, предназначенное для угадывания. Проще всего считать слова из файла. Кроме того, мы воспользуемся модулем PyQt, чтобы продемонстрировать, что библиотека Pygame — далеко не единственное средство создания графических программ.

Я не буду вдаваться в детали функционирования этой программы, как это делалось раньше. К настоящему моменту вы должны уже самостоятельно понимать, как работает большая часть фрагментов кода. Чтобы помочь вам в этом, я напишу ряд комментариев.

GUI-интерфейс игры Виселица

Вот как выглядит GUI-интерфейс игры Виселица:



На рисунке мы видим повешенного целиком, хотя в начале игры он скрыт. Детали изображения открываются при неверно предложенных игроком буквах.

Как только вы увидите повешенного целиком, игра будет окончена!

Игрок предлагает букву, а программа проверяет ее наличие в загаданном слове. В случае положительного результата проверки буква открывается. В центральной части окна под заголовком *Previous guesses* (Предыдущие догадки) игрок видит все предложенные им варианты. Кроме того, можно попытаться сразу угадать слово.

Итак, вот схема работы нашей программы. Вначале производятся следующие действия:

- загружается список английских слов из файла;
- с конца каждой строки удаляется символ перевода строки;
- скрываются все части человечка;
- из списка случайным образом выбирается английское слово;
- на экран выводятся черточки, число которых равно числу букв в слове.

При щелчке по кнопке *Guess* (Угадайте) программа выполняет следующие действия:

- проверяет, присутствует ли предложенный вариант в слове;
- в случае буквы:
  - проверяет наличие ее в слове;
  - если игрок угадал, все черточки, соответствующие угаданной букве, заменяются этой буквой;
  - если игрок не угадал, открывается часть повешенного;
  - предложенная буква добавляется в список *Previous guesses*;
  - проверяется, угадал ли игрок слово (открыв все буквы).
- в случае слова:
  - поверяется правильность догадки игрока;
  - в случае положительного результата проверки появляется диалоговое окно *You won!* (Вы выиграли!), и игра начинается заново.
- в случае проигрыша появляется диалоговое окно *You lost* (Вы проиграли) и показывается загаданное слово.

Доступ к словам из списка

Так как эта глава посвящена файлам, рассмотрим часть программы, отвечающую за извлечение слов из списка. Вот как выглядит код этой операции:

```
f = open("words.txt", 'r')
self.lines = f.readlines()
for line in self.lines:
    line.strip()
f.close()
```

Удаляем из каждой строки символ перевода строки

Файл *words.txt* представляет собой обычный текстовый файл, чтение из которого можно выполнить при помощи функции `readlines()`. Затем останется выбрать из списка слово при помощи функции `random.choice()`:

```
self.currentword = random.choice(self.lines)
```

Рисование человечка

Существует несколько способов слежения за уже открытыми и стоящими в очереди на открытие фрагментами рисунка. Я выбрал вариант с циклом:

```
def wrong(self):
        self.pieces_shown += 1
        for i in range(self.pieces_shown):
            self.pieces[i].setHidden(False)
        if self.pieces_shown == len(self.pieces):
            message = "You lose. The word was " + self.currentword
            QtGui.QMessageBox.warning(self,"Hangman",message)
            self.new_game()
```

Объект `self.pieces_shown` применяется для слежения за количеством открытых фрагментов рисунка. Как только рисунок покажется целиком, появится диалоговое окно с сообщением о проигрыше и с правильным вариантом слова.

Проверка предлагаемых букв

Сложнее всего реализуется проверка наличия в слове предложенной игроком буквы. Трудность связана с тем, что эта буква может встречаться в слове более одного раза. К примеру, если загадано слово *lever*, а игрок предложил букву *e*, потребуется открыть вторую и четвертую буквы.

Здесь нам на помощь приходят две функции. Во-первых, функция `find_letters()` найдет все вхождения буквы в слово и вернет данные об их местоположении. Например в случае с буквой *e* и словом *lever* она вернет список `[1, 3]`, ведь буква в строке появляется в позициях 1 и 3. (Напоминаю, что отсчет индексов начинается с 0.) Вот код этой функции:

```
def find_letters(letter, a_string):
    locations = []
    start = 0
    while a_string.find(letter, start, len(a_string)) != -1:
        location = a_string.find(letter, start, len(a_string))
        locations.append(location)
        start = location + 1
    return locations
```

Заменяем черточки буквой

Заменяем черточки буквой

Затем функция `replace_letters()` получает от функции `find_letters()` список и заменяет черточки в указанных местах угаданной буквой. В нашем примере вместо `-----` мы получим `-e-e-`. Игрок увидит, где именно находятся угаданные им буквы, при этом остальная часть слова будет представлена в виде набора черточек. Вот код этой функции:

```
def replace_letters(string, locations, letter):
    new_string = ''
    for i in range (0, len(string)):
        if i in locations:
            new_string = new_string + letter
        else:
            new_string = new_string + string[i]
    return new_string
```

И после этого, едва игрок предложит букву, мы воспользуемся только что определенными функциями `find_letters()` и `replace_letters()`:

```
if len(guess) == 1:
        if guess in self.currentword:
            locations = find_letters(guess, self.currentword)
            self.word.setText(replace_letters(str(self.word.text()),
                                              locations,guess))
            if str(self.word.text()) == self.currentword:
                self.win()
        else:
            self.wrong()
```

Проверяем наличие буквы в слове

Проверяем, где встречается буква

Мы угадываем одну букву?

Заменяем черточки буквой

Проверяем, все ли буквы открыты (что означает выигрыш игрока)

Программа состоит из 95 строк, включая пустые строки, добавленные для лучшей читабельности (см. листинг 22.8 с набором примечаний). Если вы пользовались прилагаемой к книге программой установки, то код программы вы найдете в папке *\Examples\Hangman*. Еще его можно загрузить с сайта. Там же находятся файлы *hangman.py*, *hangman.ui* и *words.txt*. Если вы работаете с компьютером Mac, следует открыть файл *hangman.ui* в Qt Designer и отключить свойство `nativeMenuBar` объекта `menubar`.

Листинг 22.8. Программа hangman.py

```
import sys
from PyQt4 import QtCore, QtGui, uic
import random

form_class = uic.loadUiType("hangman.ui")[0]

def find_letters(letter, a_string):
    locations = []
    start = 0
    while a_string.find(letter, start, len(a_string)) != -1:
        location = a_string.find(letter, start, len(a_string))
        locations.append(location)
        start = location + 1
    return locations
```

Ищем буквы

```
def replace_letters(string, locations, letter):
    new_string = ''
    for i in range (0, len(string)):
        if i in locations:
            new_string = new_string + letter
        else:
            new_string = new_string + string[i]
    return new_string

def dashes(word):
    letters = "abcdefghijklmnopqrstuvwxyz"
    new_string = ''
    for i in word:
        if i in letters:
            new_string += "-"
        else:
            new_string += i
    return new_string

class MyWidget(QtGui.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.btn_guess.clicked.connect(self.btn_guess_clicked)
        self.actionExit.triggered.connect(self.menuExit_selected)
        self.pieces = [self.head, self.body, self.leftarm, self.leftleg,
                       self.rightarm, self.rightleg]
        self.gallows = [self.line1, self.line2, self.line3, self.line4]
        self.pieces_shown = 0
        self.currentword = ""
        f=open("words.txt", 'r')
        self.lines = f.readlines()
        f.close()
        self.new_game()

    def new_game(self):
        self.guesses.setText("")
        self.currentword = random.choice(self.lines)
        self.currentword = self.currentword.strip()
        for i in self.pieces:
            i.setFrameShadow(QtGui.QFrame.Plain)
            i.setHidden(True)
        for i in self.gallows:
            i.setFrameShadow(QtGui.QFrame.Plain)
        self.word.setText(dashes(self.currentword))
        self.pieces_shown = 0
```

Заменяем черточку буквой, если игрок угадал ее правильно

❶ Заменяем буквы черточками в начале программы

Связываем обработчики событий

Части человечка

Части виселицы

Получаем список слов

Случайным образом выбираем слово из списка

Скрываем человечка

Вызываем функцию, заменяющую буквы черточками

```
    def btn_guess_clicked(self):
        guess = str(self.guessBox.text())
        if str(self.guesses.text()) != "":
            self.guesses.setText(str(self.guesses.text())+", "+guess)
        else:
            self.guesses.setText(guess)
        if len(guess) == 1:
            if guess in self.currentword:
                locations = find_letters(guess, self.currentword)
                self.word.setText(replace_letters(str(self.word.text()),
                                                  locations,guess))
                if str(self.word.text()) == self.currentword:
                    self.win()
            else:
                self.wrong()
        else:
            if guess == self.currentword:
                self.win()
            else:
                self.wrong()
        self.guessBox.setText("")

    def win(self):
        QtGui.QMessageBox.information(self,"Hangman","You win!")
        self.new_game()

    def wrong(self):
        self.pieces_shown += 1
        for i in range(self.pieces_shown):
            self.pieces[i].setHidden(False)
        if self.pieces_shown == len(self.pieces):
            message = "You lost. The word was " + self.currentword
            QtGui.QMessageBox.warning(self,"Hangman", message)
            self.new_game()

    def menuExit_selected(self):
        self.close()

app = QtGui.QApplication(sys.argv)
myapp = MyWidget(None)
myapp.show()
app.exec_()
```

Предлагаем букву

Позволяем пользователю предложить букву или слово

Предлагаем слово

Выводим диалоговое окно при выигрыше игрока

Открываем следующий фрагмент человечка

Игрок проиграл

Предложена неверная буква

Для простоты в программе используются только строчные буквы. Все слова в списке написаны строчными буквами, поэтому игрок должен вводить именно их.

Функция `dashes()` ❶ в начале новой игры заменяет буквы черточками. Но это не касается знаков препинания, например апострофов. Поэтому, скажем, вместо слова *doesn't* игрок увидит запись `- - - - - ' -`.

Я рекомендую вам написать подобную программу самостоятельно. Создать GUI-интерфейс можно в Qt-дизайнере. Ваш вариант может и отличаться от представленной мной версии. Просто не забывайте, что имена компонентов в файле с расширением *.ui* должны совпадать с используемыми в коде.

Наберите этот код, если можете. Запустите программу и посмотрите, как она работает. Если у вас возникнут какие-либо дополнительные идеи, попробуйте их реализовать! Развлекайтесь, играйте, экспериментируйте. Именно эти аспекты программирования приносят наибольшее внутреннее удовлетворение и могут многому вас научить.

Что мы узнали

В этой главе мы узнали:

- что такое файл;
- как открыть и закрыть файл;
- различные способы открытия файлов: для чтения, для записи, для добавления;
- различные средства записи в файл: функция `write()` и инструкция `print >>`;
- как при помощи модуля `pickle` сохранять в файле списки, объекты и другие структуры данных;
- многое о папках (также называемых каталогами), местоположении файлов и путях доступа к ним.

Еще мы написали игру Виселица, в которой данные из файла использовались для получения списка слов.

Проверь себя

1. Объект, которым интерпретатор Python пользуется для работы с файлами, называется _____.
2. Как создать файловый объект?
3. В чем разница между файловым объектом и именем файла?
4. Что нужно сделать с файлом, завершив чтение или запись?
5. Что произойдет, если открыть файл в режиме добавления и сделать в него запись?
6. Что произойдет, если открыть файл в режиме записи и записать туда что-нибудь?
7. Как начать чтение файла сначала, если часть файла уже прочитана?
8. Какая функция модуля `pickle` сохраняет Python-объекты в файле?

9. Какой метод модуля `pickle` выполняет десериализацию объекта — берет его из файла и превращает в Python-переменную?

Эксперименты

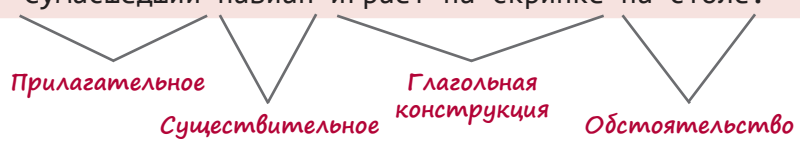
1. Напишите программу, генерирующую нелепые фразы. Каждая фраза должна состоять по меньшей мере из четырех частей:

   _____ _____ _____ _____.

   (прилагательное) (существительное) (глагольная конструкция) (обстоятельство)

   Например:

```
"Сумасшедший павиан играет на скрипке на столе."
```



   Программа должна генерировать предложения, случайным образом выбирая прилагательное, существительное, глагольную конструкцию и обстоятельство. Слова должны храниться в файлах, которые вы можете создать в Блокноте. Проще всего реализовать такую программу, создав для каждой из четырех групп собственный файл со словами, но вы можете поступить и по-другому. Вот варианты слов, хотя я уверен, что вы в состоянии придумать и свой словарный набор:

   - прилагательные: сумасшедший, глупый, робкий, тупой, злой, ленивый, упрямый, фиолетовый;
   - существительные: павиан, слон, велосипедист, учитель, автор, хоккеист;
   - глагольные конструкции: играет на скрипке, танцует джигу, расчесывает волосы, хлопает ушами;
   - обстоятельства: на столе, в гастрономе, в душе, после завтрака, в окно.

   Вот еще один пример того, что может получиться:

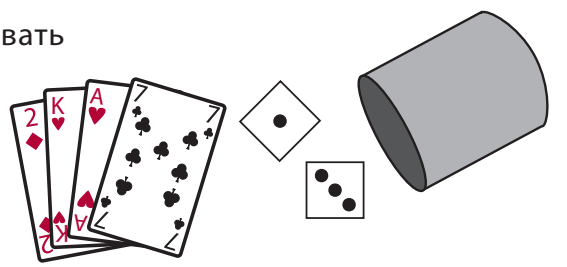
```
Ленивый автор танцует джигу в душе.
```

2. Напишите программу, которая просит пользователя ввести имя, возраст, любимый цвет и любимую еду. Сделайте так, чтобы все эти данные сохранялись в текстовом файле, каждый пункт на отдельной строчке.
3. Повторите задание 2, воспользовавшись для сохранения в файле модулем `pickle`. (Подсказка: это просто, если поместить данные в список.)

ГЛАВА 23

Непредсказуемость игры

В игре интереснее всего то, что вы никогда заранее не знаете, что произойдет. Игры непредсказуемы. И именно эта непредсказуемость делает их интересными.



Как вы уже знаете, компьютеры умеют моделировать случайное поведение. В программе угадывания чисел (в главе 1) случайное целое число, которое должен был угадать пользователь, мы генерировали при помощи модуля `random`. Этот же модуль мы использовали для выбора слов в программе, генерирующей нелепые фразы. Ее вы должны были написать в разделе «Эксперименты» предыдущей главы.

Еще компьютеры умеют имитировать броски игральных костей и тасование колоды карт. Это дает нам возможность создавать компьютерные игры с картами и костями (и другими объектами, ведущими себя случайным образом). К примеру, практически каждый из нас раскладывал пасьянс Косынка в операционной системе Windows. Карты в этой игре случайным образом тасуются перед каждым раундом. Еще популярна игра в нарды, в которой используется пара костей.

В этой главе вы узнаете, как с помощью модуля `random` смоделировать кости и колоду карт для различных игр. Еще мы рассмотрим процесс генерирования случайных событий и понятие *вероятности*, означающее возможность наступления определенного события.

Что такое непредсказуемость

Перед тем как мы начнем говорить о создании программ с хаотическим поведением, нужно понять, что же представляет собой хаос.

Рассмотрим, к примеру, монетку. Если ее подбросить и дать упасть, она приземлится орлом или решкой вверх. Шансы выпадения обеих сторон одинаковы. Иногда вверху оказывается орел, иногда решка.

Вы никогда не знаете, как монета упадет в следующий раз. Так как результат падения монеты непредсказуем, говорят, что монета падает *случайным* образом. Это пример *случайного события*.

Если бросать монетку много раз, количество выпавших орлов и решек будет одинаковым. Но в этом нельзя быть уверенным. Подбросив монетку 4 раза, можно получить два орла и две решки. А можно три орла и одну решку, или даже четыре решки (или орла) подряд. При 100 бросках можно получить 50 орлов. А может быть 20, 44, 67 или даже 100 орлов подряд! Это практически невероятное событие, но в принципе оно возможно.

Дело в том, что каждое событие является случайным. При большом количестве бросков можно обнаружить некую закономерность, но при каждом отдельном броске шанс выпадения орла равен шансу выпадения решки. Другими словами, монетка не обладает *последействием*. Поэтому даже если выпало 99 орлов подряд, и вы думаете, что 100 орлов подряд получить практически невозможно, шанс выпадения орла при следующем броске все равно составляет 50 %. Именно такое поведение называют *случайным*.

Случайное событие — это событие с двумя и более возможными исходами, точный результат которого непредсказуем. В качестве результата может выступать порядок следования карт в перетасованной колоде, количество точек на верхней грани кости или рисунок, который вы увидите после приземления монетки.

Бросаем кости

Практически все мы играли в игры с костями. В Монополии, покере на костях, настольных играх, нардах и ряде других игр именно кости применяются для генерирования случайного события.

Пару инструментов предоставляет нам модуль `random`. Во-первых, это функция `randint()`, выбирающая случайные целые числа. Так как количество точек на гранях кости выражается целым числом (1, 2, 3, 4, 5 и 6), бросок кости имитируется вот так:

```
import random
die_1 = random.randint(1, 6)
```

Это дает нам число от 1 до 6, при этом вероятность выпадения каждого из чисел одинакова, как и в случае с реальными костями.

Во-вторых, мы можем создать список с вероятными результатами и выбирать из него варианты при помощи функции `choice()`. Вот как это делается:

```
import random
sides = [1, 2, 3, 4, 5, 6]
die_1 = random.choice(sides)
```

Результат работы такого кода будет таким же, как и в предыдущем примере. Функция `choice()` случайным образом выбирает элемент из списка. А в данном случае список стоит из чисел от 1 до 6.

несколько костей

Как быть, если нужно смоделировать бросок двух костей? Узнать сумму выпавших очков позволяет следующий код:

```
two_dice = random.randint(2, 12)
```

В конце концов, сумма очков на двух костях попадает в интервал от 2 до 12, не так ли? На самом деле и да, и нет. Вы *получите* случайное число в интервале от 2 до 12, но способом, отличным от сложения двух случайных цифр от 1 до 6. Эта строка кода моделирует бросок одной большой кости с одиннадцатью сторонами, а не двух костей с шестью сторонами. В чем разница? Все дело в *вероятностях*. Разницу проще всего понять на примере.

Нам потребуются цикл и список. Цикл будет моделировать броски костей, а список следить, сколько раз выпадет та или иная сумма. Начнем с одной кости с одиннадцатью сторонами (листинг 23.1).

Листинг 23.1. 1000 бросков кости с одиннадцатью сторонами

```
import random

totals = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
for i in range(1000):
    dice_total = random.randint(2, 12)
    totals[dice_total] += 1

for i in range (2, 13):
    print "сумма", i, "выпала", totals[i], "раз"
```

❶ *В списке 13 элементов с индексом от 0 до 12*

❷ *Добавляем единицу к этой общей сумме*

Список ❶ включает в себя индексы от 0 до 12, но первые два мы отбросим, так как сумма со значением 0 или 1 просто не может выпасть. ❷ Получив результат для конкретного элемента, мы прибавим к нему единицу. Если сумма равна 7, добавим единицу к `totals[7]`. В результате элемент `totals[2]` будет показывать количество выпавших двоек, элемент `totals[3]` — количество выпавших троек, и т. д.

Запустив код, мы получим примерно такой результат:

```
сумма 2 выпала 95 раз
сумма 3 выпала 81 раз
сумма 4 выпала 85 раз
сумма 5 выпала 86 раз
```

```
сумма 6 выпала 100 раз
сумма 7 выпала 85 раз
сумма 8 выпала 94 раз
сумма 9 выпала 98 раз
сумма 10 выпала 93 раз
сумма 11 выпала 84 раз
сумма 12 выпала 99 раз
```

Все суммы выпадали примерно одинаковое число раз, от 80 до 100. Полученные значения не совпадают, ведь мы имеем дело со случайными числами, но они близки друг к другу и нет ощущения, что одни варианты выпадают чаще остальных. Запустите программу несколько раз и убедитесь. Можно увеличить количество циклов до 10 000 или до 100 000.

Теперь попробуем проделать такой трюк для двух костей с шестью гранями (листинг 23.2).

Листинг 23.2. 1000 бросков двух костей с шестью сторонами

```
import random

totals = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
for i in range(1000):
    die_1 = random.randint(1, 6)
    die_2 = random.randint(1, 6)
    dice_total = die_1 + die_2
    totals[dice_total] += 1

for i in range (2, 13):
    print "сумма", i, "выпала", totals[i], "раз"
```

Запустив код из листинга 23.2, мы получим примерно такой результат:

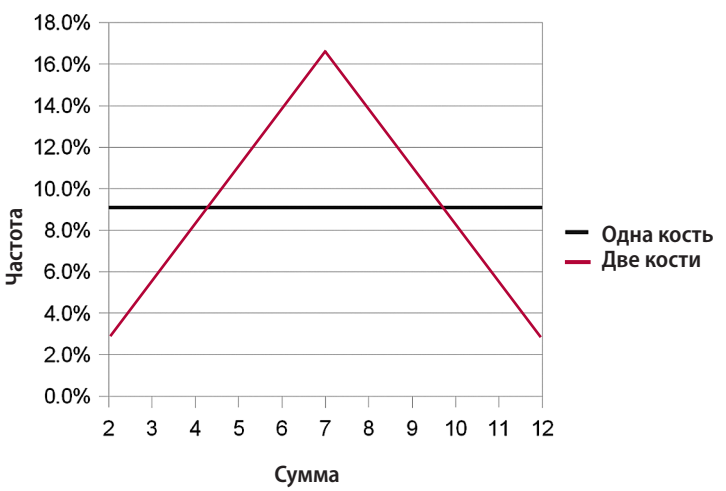
```
сумма 2 выпала 22 раз
сумма 3 выпала 61 раз
сумма 4 выпала 93 раз
сумма 5 выпала 111 раз
сумма 6 выпала 141 раз
сумма 7 выпала 163 раз
сумма 8 выпала 134 раз
сумма 9 выпала 117 раз
сумма 10 выпала 74 раз
сумма 11 выпала 62 раз
сумма 12 выпала 22 раз
```

Как видите, самые большие и самые малые значения выпадают намного реже значений из середины диапазона. Чаще всего выпадают варианты 6 и 7. Это значительно отличается

от результата, полученного для кости с одиннадцатью сторонами. Проделав операцию много раз и рассчитав процент выпадения каждого значения, мы получим таблицу:

Сумма очков	Одна кость с 11-ю гранями, %	Две кости с 6-ю гранями, %
2	9,1	2,8
3	9,1	5,6
4	9,1	8,3
5	9,1	11,1
6	9,1	13,9
7	9,1	16,7
8	9,1	13,9
9	9,1	11,1
10	9,1	8,3
11	9,1	5,6
12	9,1	2,8

На графике эти значения будут выглядеть так:



С чем связана такая разница? Объяснение нам может дать наука, которая называется *теорией вероятности*. По сути, серединные значения появляются чаще, потому что эти суммы при броске двух костей можно получить большим числом способов.

При броске двух костей возникает множество комбинаций. Вот их список с суммами:

```
1 + 1 = 2   1 + 2 = 3   1 + 3 = 4   1 + 4 = 5   1 + 5 = 6   1 + 6 = 7
2 + 1 = 3   2 + 2 = 4   2 + 3 = 5   2 + 4 = 6   2 + 5 = 7   2 + 6 = 8
3 + 1 = 4   3 + 2 = 5   3 + 3 = 6   3 + 4 = 7   3 + 5 = 8   3 + 6 = 9
4 + 1 = 5   4 + 2 = 6   4 + 3 = 7   4 + 4 = 8   4 + 5 = 9   4 + 6 = 10
5 + 1 = 6   5 + 2 = 7   5 + 3 = 8   5 + 4 = 9   5 + 5 = 10  5 + 6 = 11
6 + 1 = 7   6 + 2 = 8   6 + 3 = 9   6 + 4 = 10  6 + 5 = 11  6 + 6 = 12
```

То есть у нас есть 36 возможных комбинаций. А теперь посмотрим, сколько раз в этих комбинациях фигурирует каждая сумма:

- Сумма 2 появляется 1 раз.
- Сумма 3 появляется 2 раза.
- Сумма 4 появляется 3 раза.
- Сумма 5 появляется 4 раза.
- Сумма 6 появляется 5 раз.
- Сумма 7 появляется 6 раз.
- Сумма 8 появляется 5 раз.
- Сумма 9 появляется 4 раза.
- Сумма 10 появляется 3 раза.
- Сумма 11 появляется 2 раза.
- Сумма 12 появляется 1 раз.

Как видите, способов получить в сумме 7 больше, чем способов получить 2. Получить 7 можно, выбросив 1+6, 2+5, 3+4, 4+3, 5+2 или 6+1. А 2 мы получаем только в случае выпадения комбинации 1+1. Значит, при многократном подбрасывании костей сумма 7 должна выпадать чаще, чем сумма 2. Именно такой результат мы получили с помощью программы для двух костей.

Генерирующие случайные события компьютерные программы позволяют экспериментировать с вероятностями и определять, какой результат даст большее количество попыток. В жизни вам потребовалось бы длительное время, чтобы 1000 раз бросить пару костей и записать все результаты. Компьютер же решает эту задачу за долю секунды!

Десять подряд

Перед тем как перейти к следующей теме, еще немного поэкспериментируем с теорией вероятности. В начале главы мы рассматривали подбрасываемую монетку и рассуждали о том, насколько возможно ее приземление одной и той же стороной вверх десять раз подряд. Почему не поставить эксперимент и не посмотреть, насколько часто такое бывает? Разумеется, подобное случается крайне редко, и чтобы увидеть десять орлов подряд, потребуется бросать монетку много раз. Попробуем сделать 1 000 000 бросков! С реальной монеткой это заняло бы у нас… очень долгое время.

Если предположить, что один бросок занимает 5 секунд, значит, в минуту вы делаете 12 бросков, а в час — 720. Бросая монетку по 12 часов в день (в конце концов, вам же нужно есть и спать), вы сможете сделать 8640 попыток. То есть на

миллион бросков вам потребуется 115 дней (около 4 месяцев). Но компьютер выполнит это задание за несколько секунд. (Ну хорошо, нам потребуется несколько минут, так как сначала нужно будет набрать код программы.)

Программа должна не только бросать монетку, но и ожидать момента выпадения 10 орлов подряд. Это можно сделать с помощью переменной, называемой *счетчиком* (counter).

Нам потребуется два счетчика. Один для количества выпавших подряд орлов, назовем его `heads_in_row`; второй для количества выпавших 10 раз подряд орлов, назовем его `ten_heads_in_row`. Вот что будет делать наша программа.

- При выпадении орла счетчик `heads_in_row` будет увеличиваться на 1.
- При выпадении решки счетчик `heads_in_row` будет возвращаться к значению 0.
- Как только счетчик `heads_in_row` достигнет значения 10, увеличим показания счетчика `ten_heads_in_row` на 1, а счетчику `heads_in_row` присвоим значение 0 и начнем процедуру заново.
- В конце программы выведем сообщение с информацией о количестве выпавших 10 раз подряд орлов.

Соответствующий код представлен в листинге 23.3.

Листинг 23.3. Подсчет выпадающих 10 раз подряд орлов

```
from random import *
coin = ["Heads", "Tails"]
heads_in_row = 0
ten_heads_in_row = 0
for i in range (1000000):
    if choice(coin) == "Heads":
        heads_in_row += 1
    else:
        heads_in_row = 0
    if heads_in_row == 10:
        ten_heads_in_row += 1
        heads_in_row = 0

print "Мы получили 10 орлов подряд", ten_heads_in_row, "раз."
```

Бросаем монетку

Получив 10 орлов подряд, увеличиваем счетчик на единицу

После запуска программы я получил такой результат:

```
Мы получили 10 орлов подряд 510 раз.
```

Я запустил программу несколько раз, и каждый раз получал число в районе 500. Это означает, что при миллионе бросков десять орлов подряд выпадают примерно 500 раз, то есть это событие происходит один раз на 2000 бросков (1 000 000 / 500 = 2000).

Колода карт

Другим случайным событием, фигурирующим во множестве программ, является выбор карты. Так как колода тасуется, нельзя заранее сказать, какая карта будет следующей. Тем более что при каждом перемешивании колоды порядок меняется.

Как вы уже знаете, каждый бросок кости или монетки обладает равной вероятностью, потому что монетка (или кость) не обладают последействием. Но в случае с картами это не так. По мере того как вы берете карты, в колоде их остается все меньше (по крайней мере в большинстве игр). Это меняет вероятность вытаскивания каждой из оставшихся карт.

К примеру, шанс вытащить из полной колоды четверку червей составляет 1/52, или около 2 %. Дело в том, что колода состоит из 52 карт, и при этом там только одна четверка червей. Если же вы вытащили уже половину карт из колоды (при этом четверка червей вам еще не попадалась), шанс ее достать составит уже 1/26, или около 4 %. Если к моменту, когда вы доберетесь до последней карты, четверка червей еще останется в колоде, шанс ее вытащить составит 1/1, или 100 %. Вы наверняка вытащите именно эту карту, так как других в колоде просто не осталось.

Я пытаюсь показать, что программируя игры с картами, нужно следить за тем, какие карты извлечены из колоды. Проще всего это реализуется при помощи списка. Можно начать со списка из 52 карт и воспользоваться функцией `random.choice()` для выбора случайной карты. Выбранная карта удаляется из списка функцией `remove()`.

Тасуем колоду

В реальной жизни мы тасуем колоду, обеспечивая случайный порядок следования карт. В результате можно взять верхнюю карту, и она окажется случайной. Функция `random.choice()` выбирает карту из списка случайным образом. Так как нам не нужно всенепременно брать «верхнюю» карту, колоду можно не тасовать. Будем выбирать карту из произвольного места. Это все равно что разложить карты веером и сказать: «Возьми любую карту!» Чтобы проделывать подобное перед каждым ходом реального игрока, вам потребуется некоторое время, но в компьютерной программе все происходит очень быстро.

Объект card

Нужно сделать так, чтобы *список* вел себя как «колода» карт. Но как быть с самими картами? В каком виде мы их будем хранить? Как строки? Как целые числа? Что нам нужно знать о каждой карте?

Для карточных игр нам нужны следующие сведения о картах:

- *масть* (suit) — бубны (diamonds), червы (hearts), трефы (clubs) или пики (spades);
- *достоинство* (rank) — туз (ace), 2, 3, … 10, валет (jack), дама (queen), король (king).
- *значение* (value) — для нумерованных карт (с 2 по 10) обычно совпадает с достоинством, для валета, дамы и короля, как правило, равняется 10, а для туза может равняться 1, 11 или другому значению в зависимости от игры.

Достоинство	Значение	Достоинство	Значение
Туз	1 или 11	8	8
2	2	9	9
3	3	10	10
4	4	Валет	10
5	5	Дама	10
6	6	Король	10
7	7		

Нам нужно следить за этими тремя свойствами в совокупности. Это возможно при помощи списка, но потребуется запоминать, какой из карт соответствует каждый элемент. В качестве альтернативы можно создать объект карты со следующими атрибутами:

```
card.suit
card.rank
card.value
```

К этому мы добавим еще пару атрибутов `suit_id` и `rank_id`:

- `suit_id` — это номер масти от 1 до 4, где 1 — бубны, 2 — червы, 3 — пики, 4 — трефы.
- `rank_id` — это число от 1 до 13, где:
  1 — туз;
  2 — 2;
  3 — 3;
  …
  10 — 10;
  11 — валет;
  12 — дама;
  13 — король.

Добавив эти два атрибута, мы легко смоделируем 52 карты с помощью вложенного цикла `for`. Внутренний цикл определит достоинство карты (от 1 до 13), а внешний — ее масть (от 1 до 4). Метод `__init__()` объекта `card` на основе атрибутов `suit_id` и `rank_id` создаст остальные атрибуты, задающие масть, достоинство и значение. При этом мы легко сможем сравнивать достоинства двух карт.

Для простоты работы с объектом `card` следует добавить еще пару атрибутов. Сведения о карте будут отображаться как «4H» или «4 of Hearts» (4 червей), а для карт с фигурами — как «JD» или «Jack of Diamonds» (бубновый валет). Атрибуты для короткой и длинной версий имени карты `short_name` и `long_name` позволят программе вывести сокращенное и подробное описание любой карты. Создадим класс для объекта (листинг 23.4).

Листинг 23.4. Класс Card

```
class Card:
    def __init__(self, suit_id, rank_id):
        self.rank_id = rank_id
        self.suit_id = suit_id
        if self.rank_id == 1:
            self.rank = "Ace"
            self.value = 1
        elif self.rank_id == 11:
            self.rank = "Jack"
            self.value = 10
        elif self.rank_id == 12:
            self.rank = "Queen"
            self.value = 10
        elif self.rank_id == 13:
            self.rank = "King"
            self.value = 10
        elif 2 <= self.rank_id <= 10:
            self.rank = str(self.rank_id)
            self.value = self.rank_id
        else:
            self.rank = "RankError"
            self.value = -1

        if self.suit_id == 1:
            self.suit = "Diamonds"
        elif self.suit_id == 2:
            self.suit = "Hearts"
        elif self.suit_id == 3:
            self.suit = "Spades"
        elif self.suit_id == 4:
            self.suit = "Clubs"
        else:
            self.suit = "SuitError"
        self.short_name = self.rank[0] + self.suit[0]
        if self.rank == '10':
            self.short_name = self.rank + self.suit[0]
        self.long_name = self.rank + " of " + self.suit
```

Создаем атрибуты rank и value

Создаем атрибут suit

❶ *Выполняем проверку ошибок*

В процессе проверки ошибок ❶ мы смотрим, чтобы атрибуты `rank_id` и `suit_id` попали в корректный диапазон и принадлежали к типу `int`. В случае отрицательного результата проверки при выводе карты мы увидим надпись вроде «7 of SuitError», или «RankError of Clubs».

В строке, задающей параметр `short_name`, связывается число или первая буква достоинства карты (6 или Jack) с первой буквой масти (Diamonds). Для короля червей (King of Hearts) параметр `short_name` будет иметь значение KH. А у шестерки пик (6 of Spades) он получит значение 6S.

Листинг 23.4 — это не программа целиком, а всего лишь определение нашего класса `Card`. Оно пригодится нам и в других программах, поэтому имеет смысл превратить его в *модуль*. Сохраним этот код под именем ***cards.py***.

Теперь нужно получить экземпляры карт — их нам потребуется целая колода! Для тестирования класса `Card` заставим программу создать колоду из 52 карт, выбрать случайным образом 5 карт и показать их атрибуты. Код, реализующий эту задачу, представлен в листинге 23.5.

Листинг 23.5. Создание колоды карт

```
import random
from cards import Card

deck = []

for suit_id in range(1, 5):
    for rank_id in range(1, 14):
        deck.append(Card(suit_id, rank_id))

hand = []
for cards in range(0, 5):
    a = random.choice(deck)
    hand.append(a)
    deck.remove(a)

print
for card in hand:
    print card.short_name, '=' ,card.long_name, " Value:", card.value
```

Импортируем модуль cards

❶ *Создаем колоду при помощи вложенного цикла for*

❷ *Сдаем на руки игроку пять случайным образом выбранных карт*

❶ Внутренний цикл просматривает каждую карту определенной масти, а внешний перебирает масти (13 карт × 4 масти = 52 карты). ❷ После этого код выбирает из колоды пять карт и сдает их игроку. При этом карты из колоды удаляются.

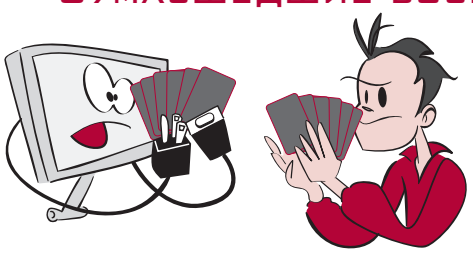
Запустив код из листинга 23.5, вы получите примерно такой результат:

```
7D = 7 of Diamonds Value: 7
9H = 9 of Hearts Value: 9
KH = King of Hearts Value: 10
6S = 6 of Spades Value: 6
KC = King of Clubs Value: 10
```

При следующем запуске вы получите другой набор из пяти карт. И сколько бы вы ни запускали программу, одна карта никогда не попадет в один и тот же набор два раза.

Итак, мы научились моделировать колоду карт и брать из нее пять случайных карт. Теперь ничто не мешает нам написать карточную игру! В следующем разделе мы создадим игру, в которую вы сможете играть со своим компьютером.

Сумасшедшие восьмерки



Возможно, вы уже слышали об игре «Сумасшедшие восьмерки» или даже играли в нее.

Смоделировать карточную игру для нескольких игроков сложно. Ведь в большинстве случаев игроки не должны видеть карт друг друга. А если все смотрят в монитор одного компьютера, это попросту невозможно. Поэтому самым лучшим вариантом тут является игра пользователя против компьютера. В Сумасшедшие восьмерки можно играть и вдвоем, поэтому именно эту игру мы возьмем в качестве основы для нашей программы.

Вот правила игры. Каждый игрок получает по пять карт. Остальные карты сложены в стопку рубашкой вверх, и из этой колоды на стол выкладывается верхняя карта. Вам нужно избавиться от своих карт раньше соперника и до того, как в колоде закончатся карты:

1. Игрок, когда до него доходит очередь, должен:
   - положить карту, масть которой совпадает с мастью выложенной карты;
   - положить карту, достоинство которой совпадает с достоинством выложенной карты;
   - положить восьмерку.
2. Если игрок положил восьмерку, он может «поменять масть», то есть указать обязательную масть для следующего игрока.
3. Если у игрока нет подходящей карты, он берет карту из колоды.
4. Выигрывает тот, кто первым избавится от своих карт. Очки начисляются в зависимости от того, сколько и каких карт осталось на руках у соперника:
   - 50 очков за каждую восьмерку;
   - 10 очков за каждую карту с фигурой;

- номинальное значение остальных карт;
- 1 очко за каждого туза.

5. Игра заканчивается, когда никто не может сделать ход, а в колоде уже не осталось карт. В этом случае обоим игрокам начисляются очки в зависимости от оставшихся у соперника карт.
6. Можно играть до достижения заранее заданной суммы или пока не надоест. Во втором случае выигрывает тот, кто набрал больше очков.

Первым делом нам нужно внести ряд изменений в объекты `card`. Дело в том, что в Сумасшедших восьмерках за карту с достоинством 8 начисляется 50, а не стандартные 8 очков. Разумеется, можно отредактировать метод `__init__` из класса `Card`, но это повлияет на все игры, в которых будет использоваться модуль `cards`. Поэтому лучше внести изменения в основную программу, не трогая определение класса. Это можно сделать следующим способом:

```
deck = []
for suit in range(1, 5):
    for rank in range(1, 14):
        new_card = Card(suit, rank)
        if new_card.rank == 8:
            new_card.value = 50
        deck.append(new_card)
```

В данном фрагменте кода перед добавлением новой карты мы проверяем, не восьмерка ли это. При положительном результате проверки карте присваивается значение 50.

Теперь все готово и пора приступить к коду самой игры. Программа должна уметь:

- следить за выкладываемыми картами;
- получать от игрока сигнал о необходимых действиях (выложить карту или взять карту из колоды);
- если игрок выкладывает карту, убедиться в корректности его выбора:
  - карта должна быть подходящей;
  - карта должна быть на руках у игрока;
  - карта должна соответствовать верхней выложенной карте либо по масти, либо по достоинству, либо это должна быть восьмерка.
- после выкладывания восьмерки спрашивать новую масть (и проверять корректность сделанного выбора);
- играть за компьютер (об этом мы вскоре поговорим подробно);
- определять момент наступления конца игры;
- считать очки.

Сейчас мы по очереди подробно рассмотрим все эти требования. Для некоторых достаточно пары строк кода, а с другими придется повозиться. Во втором случае мы будем писать функции, которые потом вставим в основной цикл.

Основной цикл

Перед тем как перейти к рассмотрению деталей, давайте поговорим о том, как должен выглядеть основной цикл программы. По сути это последовательность ходов игрока и компьютера, продолжающаяся до выигрыша одного из соперников или до блокировки. Это можно реализовать при помощи кода из листинга 23.6.

Листинг 23.6. Основной цикл игры Сумасшедшие восьмерки

```
init_cards()
while not game_done:
    blocked = 0
    player_turn()
    if len(p_hand) == 0:
        game_done = True
        print
        print "Ты выиграл!"
    if not game_done:
        computer_turn()
    if len(c_hand) == 0:
        game_done = True
        print
        print "Компьютер выиграл!"
    if blocked >= 2:
        game_done = True
        print "Ходов больше нет. ИГРА ОКОНЧЕНА."
```

Очередь игрока

У игрока (p_hand) не осталось карт, значит, он выиграл

Очередь компьютера

У компьютера (c_hand) не осталось карт, значит, он выиграл

❶ *Оба игрока заблокированы, игра окончена*

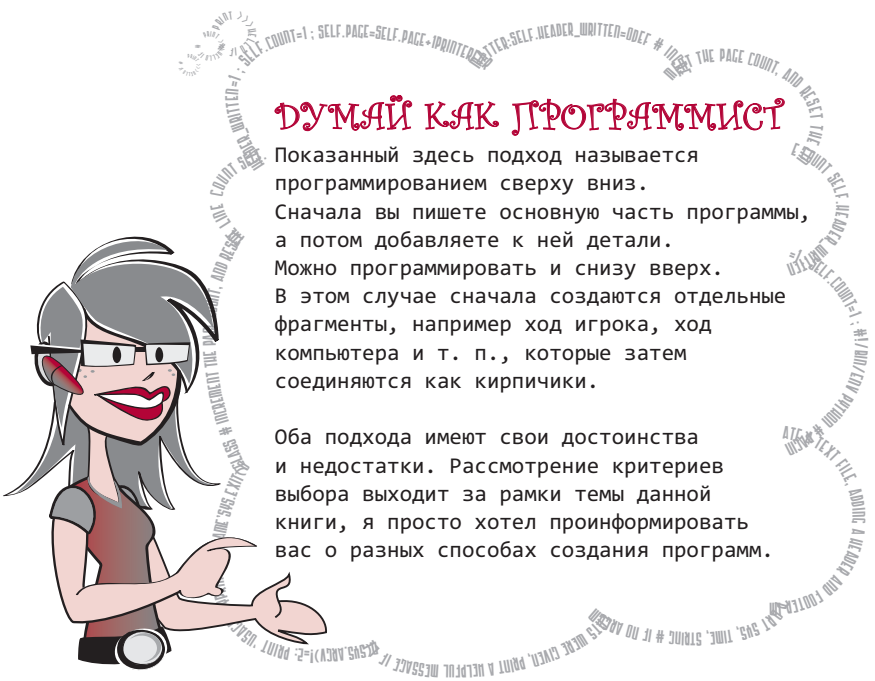
Часть основного цикла определяет момент окончания игры. Он наступает, когда у компьютера или у игрока не остается карт или когда ни одна из сторон не может сделать следующий ход. Переменная `blocked` в коде задается как для игрока, так и для компьютера. ❶ Нам нужно дождаться значения `blocked = 2`, гарантирующего блокировку обоих участников игры.

Листинг 23.6 — это не полный вариант программы, поэтому попытавшись его запустить, вы получите сообщение об ошибке. Это всего лишь основной цикл. Для завершения программы нам нужен ряд дополнительных фрагментов.

Этот код дает возможность сыграть один раз. Возможность начать новую партию даст вставка этого кода в цикл `while`:

```
done = False
p_total = c_total = 0
while not done:
     [play a game... see listing 23.6]
play_again = raw_input("Начать заново (Y/N)? ")
    if play_again.lower().startswith('y'):
        done = False
    else:
        done = True
```

Костяк программы готов. Теперь добавим фрагменты, делающие то, что нам нужно.



ДУМАЙ КАК ПРОГРАММИСТ

Показанный здесь подход называется программированием сверху вниз. Сначала вы пишете основную часть программы, а потом добавляете к ней детали. Можно программировать и снизу вверх. В этом случае сначала создаются отдельные фрагменты, например ход игрока, ход компьютера и т. п., которые затем соединяются как кирпичики.

Оба подхода имеют свои достоинства и недостатки. Рассмотрение критериев выбора выходит за рамки темы данной книги, я просто хотел проинформировать вас о разных способах создания программ.

ПЕРВАЯ КАРТА

Раздав карты на руки игрокам, мы переворачиваем верхнюю карту колоды, чтобы начать игру. Все выкладываемые игроками карты кладутся в эту новую стопку лицевой стороной вверх. Создадим для них такой же список, как для выдаваемых на руки карт (см. листинг 23.5). Но в данном случае нам не нужно следить за всеми попадающими в этот список картами. Важна только последняя выложенная карта, и нам будет достаточно одного экземпляра объекта `Card`. Вот код, описывающий ход игрока или компьютера:

```
hand.remove(chosen_card)
up_card = chosen_card
```

Активная масть

Активная масть в обычном случае совпадает с мастью последней выложенной игроком карты. Но есть и исключение. Выложивший восьмерку игрок может поменять масть на свое усмотрение. Например, если сделав ход восьмеркой червей, игрок выбирает трефы, его соперник может выложить только восьмерку или карту крестовой масти.

То есть нам нужно следить за активной мастью, так как она может отличаться от масти на последней выложенной карте. Для этого мы воспользуемся переменной `active_suit`:

```
active_suit = card.suit
```

Активная масть будет обновляться при каждом ходе, а выложивший восьмерку игрок сможет выбирать значение этой переменной на свое усмотрение.

Ход игрока

На этом этапе нам в первую очередь нужно узнать, что именно хочет сделать игрок. Он может выложить карту из имеющегося у него на руках набора или взять дополнительную карту из колоды. В GUI-версии программы игрок щелчком выбирал бы карту для своего хода или щелкал бы по колоде, чтобы взять дополнительную карту. Но мы пишем текстовую версию, значит, информацию о своих действиях игрок будет вводить с клавиатуры, мы же должны проверить введенные данные на корректность и определить, что именно следует сделать.

Чтобы получить представление о данных, которые может ввести игрок, рассмотрим пример игры. Вводимые данные сделаны в коде более **жирными**:

```
Сумасшедшие восьмерки

У вас есть: 4S, 7D, KC, 10D, QS Открыта карта: 6C
Ваши действия? Введите имя карты или "Draw", чтобы взять еще карту: KC
Вы положили KC (трефовый король)
Компьютер выкладывает 8S (8 пик) и меняет масть на бубны

У вас есть: 4S, 7D, 10D, QS Открыта карта: 8S Масть: Бубны
Ваши действия? Введите имя карты или "Draw", чтобы взять еще карту: 10D
Вы положили 10D (10 бубей)
Компьютер выкладывает QD (бубновую даму)

У вас есть: 4S, 7D QS Открыта карта: QD
Ваши действия? Введите имя карты или "Draw", чтобы взять еще карту: 7D
Вы положили 7D (7 бубей)
Компьютер выкладывает 9D (9 бубей)
```

```
У вас есть: 4S, QS Открыта карта: 9D
Ваши действия? Введите имя карты или "Draw", чтобы взять еще карту: QM
Этот ход недопустим. Повторите попытку: QD
Эта карта у вас отсутствует. Повторите попытку: QS
Некорректный ход. Вы можете положить карту такой же масти, такого же
достоинства, 8 или взять еще карту
Повторите попытку: Draw
Вы взяли 3C
Компьютер берет карту

У вас есть: 4S, QS, 3C Открыта карта: 9D
Ваши действия? Введите имя карты или "Draw", чтобы взять еще карту: Draw
Вы взяли 8C
Компьютер выкладывает 2D

У вас есть: 4S, QS, 3C, 8C Открыта карта: 2D
Ваши действия? Введите имя карты или "Draw", чтобы взять еще карту: 8C
Вы положили 8C (8 треф)

У вас есть: 4S, QS, 3C Выберите масть: S
Вы выбрали пики
Компьютер берет карту

У вас есть: 4S, QS, 3C Открыта карта: 8C Масть: пики
Ваши действия? Введите имя карты или "Draw", чтобы взять еще карту: QS
Вы положили QS (пиковую даму)
.
.
.
```

Это далеко не вся игра, но, думаю, идею вы поняли. Игрок, набирая, например, QS или Draw, дает программе представление о своих действиях. Программа проверяет корректность введенных данных. Для этого мы воспользовались строковыми методами (рассмотренными в главе 21).

Показ карт игрока

Перед тем как спросить о намерениях игрока, нужно показать, какие карты имеются у него на руках и какая карта открыта в данный момент. Вот как это реализуется:

```
print "\nУ вас есть: ",
for card in p_hand:
    print card.short_name,
print " Открыта карта: ", up_card.short_name
```

При выкладывании восьмерки нужно также сообщить игроку активную масть. Поэтому добавим в код еще несколько строк (листинг 23.7).

Листинг 23.7. Показ имеющихся у игрока карт

```
print "\nУ вас есть: ",
for card in p_hand:
    print card.short_name,
print " Открыта карта: ", up_card.short_name
if up_card.rank == '8':
    print" Масть", active_suit
```

Как и листинг 23.6, листинг 23.7 не является полной программой. Мы продолжаем создавать отдельные части. Но после запуска этого кода (в составе программы), мы получим:

```
У вас есть: 4S, QS, 3C Открыта карта: 8C Масть: Spades
```

Если вы предпочитаете использовать длинные имена вместо коротких, результат работы этого фрагмента будет выглядеть так:

```
Your hand: 4 of Spades, Queen of Spades, 3 of Clubs
Открыта карта: 8 of Clubs Масть: Spades
```

Далее мы будем рассматривать примеры с короткими именами.

Выбор игрока

Теперь нужно спросить, что игрок собирается предпринять, и обработать его ответ. Он может выполнить две вещи:

- сделать ход;
- взять дополнительную карту.

В первом случае следует убедиться в корректности его хода. Как уже упоминалось, нужно проверить три фактора.

- Существует ли названная игроком им карта? (Вдруг он пытается положить четверку зефиров?)
- Есть ли у него на руках указанная карта?
- Является ли ход корректным? (Совпадает ли верхняя карта в колоде с положенной игроком картой по масти или по достоинству либо положена восьмерка?)

Но если вдуматься, нам достаточно удостовериться, что выложенная игроком карта находилась среди

НОВЫЕ СЛОВА

Проверить в данном случае означает удостовериться в правильности чего-то, что разрешено или имеет смысл.

имеющихся у него в руках. Ведь игрок выбирает из выводимого на экран списка, в котором в принципе не может появиться четверка зефиров, так как в колоде такая карта попросту отсутствует.

Поэтому рассмотрим код, принимающий и проверяющий выбор игрока (листинг 23.8).

Листинг 23.8. Сделанный игроком ход

```
print "Ваши действия? ",
response = raw_input ("Введите имя карты или 'Draw', чтобы взять карту: " )
valid_play = False
while not valid_play:
    selected_card = None
    while selected_card == None:
        if response.lower() == 'draw':
            valid_play = True
            if len(deck) > 0:
                card = random.choice(deck)
                p_hand.append(card)
                deck.remove(card)
                print "Вы взяли", card.short_name
            else:
                print "В колоде больше нет карт"
                blocked += 1
            return
        else:
            for card in p_hand:
                if response.upper() == card.short_name:
                    selected_card = card
            if selected_card == None:
                response = raw_input("Этой карты у вас нет. Повторите:")

    if selected_card.rank == '8':
        valid_play = True
        is_eight = True
    elif selected_card.suit == active_suit:
        valid_play = True
    elif selected_card.rank == up_card.rank:
        valid_play = True

    if not valid_play:
        response = raw_input("Этот ход недопустим. Повторите попытку: ")
```

Выполняется, пока игрок не укажет корректное значение

Получаем от игрока сведения о карте или выдаем дополнительную карту

❶ *В случае значения draw берем карту из колоды и добавляем ее к картам игрока*

Получено значение draw, поэтому возвращаемся в основной цикл

Проверяем наличие выбранной карты у игрока до тех пор, пока не будет названа одна из имеющихся на руках карт или не взята дополнительная карта

Игра восьмеркой корректна всегда

Проверяем, совпадает ли масть выложенной карты с мастью открытой карты

Проверяем, совпадает ли достоинство выложенной карты с достоинством открытой карты

(Еще раз напоминаю, что это не полная версия программы, запускать ее нельзя.)

❶ В этой точке игрок должен сделать ход или взять дополнительную карту. Во втором случае, если в колоде еще остались карты, следует добавить взятую карту в список имеющихся у игрока карт.

Если же игрок сделал ход, нужно удалить выложенную карту из числа имеющихся у него на руках и превратить ее в открытую карту:

```
p_hand.remove(selected_card)
up_card = selected_card
active_suit = up_card.suit
print "Вы положили", selected_card.short_name
```

В случае хода восьмеркой игрок должен выбрать новую масть. Так как функция `player_turn()` уже имеет достаточную длину, поместим процедуру смены масти в новую функцию `get_new_suit()`. Ее код представлен в листинге 23.9.

Листинг 23.9. Смена масти после хода восьмеркой

```
def get_new_suit():
    global active_suit
    got_suit = False
    while not got_suit:
        suit = raw_input("Выберите масть: ")
        if suit.lower() == 'd':
            active_suit = "Diamonds"
            got_suit = True
        elif suit.lower() == 's':
            active_suit = "Spades"
            got_suit = True
        elif suit.lower() == 'h':
            active_suit = "Hearts"
            got_suit = True
        elif suit.lower() == 'c':
            active_suit = "Clubs"
            got_suit = True
        else:
            print "Некорректная масть. Повторите попытку. ",
    print "Вы выбрали", active_suit
```

Повторяем, пока игрок не укажет корректную масть

Итак, мы полностью запрограммировали действия игрока. Теперь осталось обучить игре в Сумасшедшие восьмерки наш компьютер.

Ход компьютера

После хода игрока приходит очередь компьютера, а значит, нужно объяснить ему правила игры. Компьютер должен придерживаться тех же правил, выбрав при этом карту для своего хода. Для этого он должен знать, когда следует:

- положить восьмерку (и выбрать новую масть);
- положить другую карту;
- взять дополнительную карту.

Для простоты мы заставим компьютер всегда класть восьмерку, если у него есть такая карта. Это не оптимальная, зато самая простая стратегия.

Выложив восьмерку, компьютер обязан выбрать новую масть. Проще всего это сделать, посчитав количество карт каждой масти среди выданных компьютеру «на руки» и выбрав масть, к которой принадлежит большинство карт. Это опять же далеко не оптимальная стратегия, но ее проще всего запрограммировать.

Если среди доступных карт нет ни одной восьмерки, карты просматриваются в цикле и выбираются наиболее подходящие для хода варианты, а из них выбирается карта с максимальным значением.

При отсутствии подходящих вариантов компьютер берет карту. Если оказывается, что в колоде не осталось карт, компьютер блокируется, как и игрок.

Снабженный небольшими пояснениями код, отвечающий за ходы компьютера, показан в листинге 23.10.

Листинг 23.10. Ходы компьютера

```
def computer_turn():
    global c_hand, deck, up_card, active_suit, blocked
    options = []
    for card in c_hand:
        if card.rank == '8':
            c_hand.remove(card)
            up_card = card
            print " Компьютер выкладывает ", card.short_name
            #всего мастей: [diamonds, hearts, spades, clubs]
            suit_totals = [0, 0, 0, 0]
            for suit in range(1, 5):
                for card in c_hand:
                    if card.suit_id == suit:
                        suit_totals[suit-1] += 1
            long_suit = 0
            for i in range (4):
```

Выкладываем восьмерку

Считаем карты каждой масти, при этом масть, к которой принадлежит большинство карт, получает преимущество

```
                if suit_totals[i] > long_suit:
                    long_suit = i
            if long_suit == 0: active_suit = "Diamonds"
            if long_suit == 1: active_suit = "Hearts"
            if long_suit == 2: active_suit = "Spades"
            if long_suit == 3: active_suit = "Clubs"
            print " Компьютер меняет масть на ", active_suit
            return
        else:
            if card.suit == active_suit:
                options.append(card)
            elif card.rank == up_card.rank:
                options.append(card)

if len(options) > 0:
    best_play = options[0]
    for card in options:
        if card.value > best_play.value:
            best_play = card

    c_hand.remove(best_play)
    up_card = best_play
    active_suit = up_card.suit
    print " Компьютер выкладывает ", best_play.short_name

else:
    if len(deck) >0:
        next_card = random.choice(deck)
        c_hand.append(next_card)
        deck.remove(next_card)
        print " Компьютер берет карту"
    else:
        print " У компьютера не осталось ходов"
        blocked += 1
print "Карт у компьютера %i" % (len(c_hand))
```

Делаем масть, получившую преимущество, активной

Завершаем действия компьютера и возвращаемся в основной цикл

Проверяем, какие карты подходят для хода

Проверяем наиболее подходящий вариант (с максимальным значением)

Выкладываем карту

Берем дополнительную карту, так как корректных вариантов хода нет

В колоде не осталось карт, компьютер блокируется

Мы почти закончили, осталось всего несколько деталей. Возможно, вы обратили внимание, что действия компьютера запрограммированы в виде функции, работающей с несколькими глобальными переменными. Эти переменные можно было передавать в функцию, но мы выбрали вариант, больше напоминающий реальный мир. Ведь колода общедоступна, любой может взять из нее карту.

Ходы игрока так же оформлены в виде функции, но мы опустили первую часть ее определения. Вот как оно выглядит:

```
def player_turn():
    global deck, p_hand, blocked, up_card, active_suit
    valid_play = False
    is_eight = False
    print "\nУ вас есть: ",
    for card in p_hand:
        print card.short_name,
    print " Открыта карта: ", up_card.short_name
    if up_card.rank == '8':
        print " Масть", active_suit
    print "Ваши действия? ",
    response = raw_input("Введите карту или 'Draw', чтобы взять карту: ")
```

Теперь осталось проследить за количеством набираемых игроками очков и научиться определять победителя!

НАБРАННЫЕ ОЧКИ

Для завершения игры осталось добавить процедуру подсчета очков. Нужно определить, сколько очков получает игрок, суммировав значения оставшихся на руках у соперника карт. Выводиться должен как счет текущей игры, так и общий счет. После добавления соответствующих процедур главный цикл будет выглядеть так, как показано в листинге 23.11.

Листинг 23.11. Главный цикл с процедурами подсчета очков

```
done = False
p_total = c_total = 0
while not done:
    game_done = False

    blocked = 0
    init_cards()
    while not game_done:
        player_turn()
        if len(p_hand) == 0:
            game_done = True
            print
            print "Ты выиграл!"
            # Здесь отображается счет игры
            p_points = 0
            for card in c_hand:
                p_points += card.value
            p_total += p_points
            print "Вы получили %i очков" % p_points
```

❶ *Раздаем карты обоим игрокам*

Игрок выиграл

Добавляем очки в соответствии с оставшимися у компьютера картами

Добавляем набранные за эту игру очки к общему счету

```
        if not game_done:
            computer_turn()
        if len(c_hand) == 0:
            game_done = True
            print
            print "Компьютер выиграл!"
            # Здесь отображается счет игры
            c_points = 0
            for card in p_hand:
                c_points += card.value
            c_total += c_points
            print "Компьютер получает %i очков" % c_points
        if blocked >= 2:
            game_done = True
            print "Ходов больше нет. ИГРА ОКОНЧЕНА."
            player_points = 0
            for card in c_hand:
                p_points += card.value
            p_total += p_points
            c_points = 0
            for card in p_hand:
                c_points += card.value
            c_total += c_points
            print "Вы получаете %i очков" % p_points
            print "Компьютер получает %i очков" % c_points
    play_again = raw_input("Сыграть еще раз (Y/N)? ")
    if play_again.lower().startswith('y'):
        done = False
        print "\nВсего вами набрано %i очков" % p_total
        print " компьютером набрано %i очков.\n" % c_total
    else:
        done = True
print "\n Окончательный счет:"
print "Вы: %i Компьютер: %i" % (p_total, c_total)
```

Компьютер выиграл

Добавляем набранные за эту игру очки к общему счету

Добавляем очки в соответствии с оставшимися у игрока картами

Оба игрока блокированы, очки считаются у обоих

Показываем счет игры

Показываем общий счет на данный момент

Показываем окончательный общий счет

❶ Функция `init_cards()`, которая здесь не показана, задает состояние колоды, раздает по 5 карт на руки игроку и компьютеру и выкладывает на стол первую карту.

Листинг 23.11 все еще не является полной программой, и, попытавшись его запустить, вы получите сообщение об ошибке. Но если вы выполняли все упражнения, в редакторе будет набрана практически вся программа. Весь листинг слишком велик для публикации на страницах книги (практически 200 строк кода плюс пустые строки плюс комментарии), но его можно найти в папке *\Examples* или загрузить с сайта *www.manning.com/books/hello-world-second-edition*. Для редактирования и запуска программы пользуйтесь IDLE.

Что мы узнали

В этой главе мы узнали:

- что такое случайность (непредсказуемость) и случайное событие;
- что такое теория вероятностей;
- как при помощи модуля `random` генерировать случайные события;
- как моделировать броски монетки и костей;
- как моделировать взятие карты из перетасованной колоды;
- как играть в Сумасшедшие восьмерки (если вы этого еще не знали).

Проверь себя

1. Расскажите, что такое «случайное событие». Приведите два примера.
2. Почему броски кости с одиннадцатью сторонами, на которых выбиты от 2 до 12 точек, дают результат, отличный от бросков двух костей с шестью сторонами, сумма точек которых попадает в интервал от 2 до 12?
3. Какими двумя способами в Python можно смоделировать бросание костей?
4. Переменная какого типа используется для одной карты?
5. Переменная какого типа используется для колоды карт?
6. Какой метод удаляет из колоды взятую игроком карту?

Эксперименты

Повторите эксперимент «Десять подряд» из листинга 23.3, меняя количество попыток. Как часто орел выпадает пять раз подряд? Шесть, семь, восемь и т. д. раз подряд? Видите ли вы какую-то закономерность?

ГЛАВА 24

Компьютерное моделирование

Вы когда-нибудь сталкивались с «виртуальными животными», называемыми тамагочи? Это маленькие игрушки с экраном, чтобы видеть питомца, и набором кнопок, позволяющих накормить его, когда он голоден, уложить спать, если он устал, поиграть с ним, если ему скучно, и т. п. По своим свойствам виртуальный питомец не отличается от реального живого животного. Это пример компьютерного моделирования — тамагочи представляет собой маленький компьютер.

В предыдущей главе вы узнали, что такое случайные события, и научились встраивать их в свои программы. В некотором смысле это тоже моделирование. *Моделированием* (simulation) называется любая компьютерная имитация явлений реального мира. Мы создавали компьютерные модели монеток, костей, колоды карт.

Эта глава продолжает ваше знакомство с программами, имитирующими явления реального мира.

Моделирование реального мира

Компьютерное моделирование получило распространение по многим причинам. Порой реальные эксперименты требуют изрядного времени или поездок на дальние расстояния, к тому же иногда они могут оказаться еще и опасными. Помните, в предыдущей главе мы подкидывали монетку миллион раз? Вряд ли у кого-то хватит времени на подобный эксперимент, в то время как компьютер справился с ним за несколько секунд.

Иногда ученые хотят получить ответ на вопрос: «Что произойдет, если..?» Что произойдет, если в Луну врежется астероид? Заставить настоящий астероид столкнуться с Луной мы не в состоянии, а вот компьютерная модель может дать нам представление о том, что произойдет в этом случае. Улетит ли Луна в космос? Врежется ли она в Землю? Как изменится ее орбита?

Далеко не все этапы обучения пилотов и космонавтов проводятся в реальных условиях. Во-первых, такая учеба стоила бы очень дорого! Во-вторых, вы бы согласились стать пассажиром самолета, которым управляет пилот-ученик? На выручку приходят

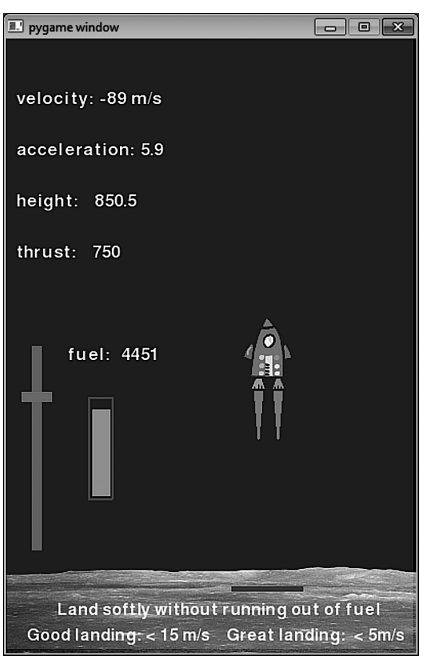
симуляторы, представляющие элементы управления в реальном самолете или космическом корабле.

Моделирование позволяет:

- экспериментировать, не имея специального оборудования (кроме своего компьютера) и не подвергая опасности ничью жизнь;
- ускорять или замедлять время;
- одновременно проводить множество экспериментов;
- пытаться проделывать вещи, которые в реальности стоят денег, опасны или даже просто нереализуемы.

В нашей первой модели будет изучаться сила тяжести. Мы попробуем посадить на Луну космический корабль с ограниченным количеством топлива. То есть двигателем нам придется пользоваться крайне расчетливо. Это упрощенная версия популярной в прошлом аркадной игры Lunar Lander (Лунный посадочный модуль).

Лунный посадочный модуль



Поместим космический корабль на расстоянии от лунной поверхности. Сила тяжести начнет тянуть его вниз, и чтобы замедлить скорость снижения и обеспечить мягкую посадку, нам придется включить двигатели.

Рассмотрим интерфейс программы. Маленькая полоска на шкале слева является регулятором подачи топлива (*fuel*). Двигая ее вверх и вниз, вы управляете мощностью двигателя. Расходомер показывает количество оставшегося топлива, а текст в левом верхнем углу дает сведения о скорости (*velocity*), ускорении (*acceleration*), высоте (*height*) и силе реактивной тяги (*thrust*).

Моделирование посадки

Для имитации посадки космического корабля нужно понимать, каким образом сила тяжести и реактивная тяга двигателей уравновешивают друг друга.

В нашей модели сила тяжести постоянна. На самом деле это не совсем так, но на небольшом расстоянии от лунной поверхности ее можно считать практически постоянной.

Реактивная тяга зависит от количества сжигаемого топлива. Она может как превышать силу тяжести, так и быть меньше нее. При выключенном двигателе реактивная тяга равна нулю, и на корабль действует только сила тяжести.

НОВЫЕ СЛОВА

В английском языке русскому слову «скорость» соответствуют два значения: *velocity* и *speed*. Первое подразумевает направление движения. Так, выражение «пятьдесят миль в час» описывается словом *speed*, в то время как «пятьдесят миль в час на север» — это уже *velocity*. В обычной речи люди часто используют один термин вместо другого, но в нашей программе важно, в какую сторону — вверх или вниз — движется космический корабль, поэтому в окне программы вы видите метку *velocity*.

Ускорением называется темп изменения скорости. Положительное ускорение означает увеличение скорости, а отрицательное — ее уменьшение.

Сила, воздействующая на космический корабль, определяется как сумма этих двух сил. Так как они направлены в разные стороны, одна из них будет положительной, другая отрицательной.

Зная воздействующую на корабль силу, мы можем по формуле определить его скорость и положение в пространстве.

В нашей модели должны учитываться следующие параметры:

- высота космического корабля над лунной поверхностью, его скорость и ускорение;
- масса корабля (которая меняется по мере расхода топлива);
- реактивная тяга, или сила двигателей. Чем сильнее тяга, тем быстрее сгорает топливо;
- количество топлива на борту корабля. По мере сжигания топлива вес корабля уменьшается, но вместе с топливом заканчивается и реактивная тяга;
- действующая на корабль сила тяжести. Этот параметр зависит от размеров Луны, массы корабля и горючего.

И СНОВА PYGAME

Для этой модели мы снова воспользуемся модулем Pygame. За единицу времени возьмем отсчеты Pygame-часов. После каждого такта мы будем проверять действующую на корабль силу, обновлять данные о его высоте, скорости, ускорении и массе оставшегося топлива и на основе этой информации обновлять графику и текст.

В данном случае мы имеем дело с очень простой анимацией, поэтому использовать спрайт в качестве изображения космического корабля смысла нет. Но он пригодится нам при рисовании ручки регулятора топлива (серого прямоугольника), которую игрок будет двигать мышью. Указателем уровня топлива послужит нарисованная методом `draw`.

`rect()` пара прямоугольников. В качестве текста, как и в игре PyPong, мы используем объекты `pygame.font`.

Код имеет смысл разбить на следующие фрагменты:

- инициализация игры — настройка Pygame-окна, загрузка изображений и присвоение переменным начальных значений;
- определение спрайтового класса для ручки регулятора топлива;
- вычисление высоты, скорости, ускорения и потребляемого горючего;
- вывод информации;
- обновление уровня топлива;
- вывод языков пламени от реактивного двигателя (размер которых зависит от реактивной тяги);
- блитирование всех элементов на экран, проверка событий мыши, обновление положения ручки регулятора топлива, проверка, произошло ли приземление — все это пойдет в основной цикл Pygame-событий;
- вывод надписи «Игра окончена» и итоговой статистики.

Код игры Lunar Lander показан в листинге 24.1. Вы можете найти его в файле *Listing_24-1.py* в папке *\Examples\LunarLander* или на сайте *www.manning.com/books/hello-world-second-edition*. Там же находятся графические фрагменты (космический корабль и лунная поверхность). Познакомьтесь с кодом и примечаниями к нему и убедитесь, что вы понимаете принцип его работы. Не волнуйтесь по поводу формул расчета высоты, скорости и ускорения — с ними вы познакомитесь в школе на уроках физики, сдадите экзамен и благополучно все забудете (если, конечно, не собираетесь выбрать себе профессию, связанную с космосом). Хотя, возможно, эта программа позволит вам их запомнить!

Листинг 24.1. Lunar Lander

```
import pygame, sys

pygame.init()
screen = pygame.display.set_mode([400,600])
screen.fill([0, 0, 0])
ship = pygame.image.load('lunarlander.png')
moon = pygame.image.load('moonsurface.png')
ground = 540
start = 90
clock = pygame.time.Clock()
ship_mass = 5000.0
fuel = 5000.0
velocity = -100.0
gravity = 10
```

Инициализируем программу

Посадочная площадка y = 540

```
height = 2000
thrust = 0
delta_v = 0
y_pos = 90
held_down = False
```

Инициализируем программу

```
class ThrottleClass(pygame.sprite.Sprite):
    def __init__(self, location = [0,0]):
        pygame.sprite.Sprite.__init__(self)
        image_surface = pygame.surface.Surface([30, 10])
        image_surface.fill([128,128,128])
        self.image = image_surface.convert()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.centery = location
```

Класс Sprite для ручки регулятора топлива

```
    def calculate_velocity():
        global thrust, fuel, velocity, delta_v, height, y_pos
        delta_t = 1/fps
        thrust = (500 - myThrottle.rect.centery) * 5.0
        fuel -= thrust /(10 * fps)
        if fuel < 0: fuel = 0.0
        if fuel < 0.1: thrust = 0.0
        delta_v = delta_t * (-gravity + 200 * thrust / (ship_mass + fuel))
        velocity = velocity + delta_v
        delta_h = velocity * delta_t
        height = height + delta_h
        y_pos = ground - (height * (ground - start) / 2000) - 90
```

Отсчитываем один виток Рудате-цикла

Физическая формула

Преобразуем позицию y ручки регулятора топлива в реактивную тягу

Вычитаем горючее в зависимости от тяги

Вычисляем высоту, скорость, ускорение и горючее

Преобразуем высоту в Рудате-координату y

```
    def display_stats():
        v_str = "velocity: %i m/s" % velocity
        h_str = "height: %.1f" % height
        t_str = "thrust: %i" % thrust
        a_str = "acceleration: %.1f" % (delta_v * fps)
        f_str = "fuel: %i" % fuel
        v_font = pygame.font.Font(None, 26)
        v_surf = v_font.render(v_str, 1, (255, 255, 255))
        screen.blit(v_surf, [10, 50])
        a_font = pygame.font.Font(None, 26)
        a_surf = a_font.render(a_str, 1, (255, 255, 255))
        screen.blit(a_surf, [10, 100])
        h_font = pygame.font.Font(None, 26)
        h_surf = h_font.render(h_str, 1, (255, 255, 255))
        screen.blit(h_surf, [10, 150])
        t_font = pygame.font.Font(None, 26)
        t_surf = t_font.render(t_str, 1, (255, 255, 255))
        screen.blit(t_surf, [10, 200])
```

Показываем статистику при помощи объектов шрифта

```
        f_font = pygame.font.Font(None, 26)
        f_surf = f_font.render(f_str, 1, (255, 255, 255))
        screen.blit(f_surf, [60, 300])

    def display_flames():
        flame_size = thrust / 15
        for i in range (2):
            startx = 252 - 10 + i * 19
            starty = y_pos + 83
            pygame.draw.polygon(screen, [255, 109, 14], [(startx, starty),
                                (startx + 4, starty + flame_size),
                                 (startx + 8, starty)], 0)

    def display_final():
        final1 = "Игра окончена"
        final2 = "Скорость приземления %.1f м/с" % velocity
        if velocity > -5:
            final3 = "Прекрасное приземление!"
            final4 = "Я слышал, что NASA ищет сотрудников!"
        elif velocity > -15:
            final3 = "Ох! Грубовато, но вы выжили."
            final4 = "В следующий раз получится лучше."
        else:
            final3 = "Черт! Вы разбили корабль за 30 миллионов долларов."
            final4 = "Как вы попадете домой?"
        pygame.draw.rect(screen, [0, 0, 0], [5, 5, 350, 280],0)
        f1_font = pygame.font.Font(None, 70)
        f1_surf = f1_font.render(final1, 1, (255, 255, 255))
        screen.blit(f1_surf, [20, 50])
        f2_font = pygame.font.Font(None, 40)
        f2_surf = f2_font.render(final2, 1, (255, 255, 255))
        screen.blit(f2_surf, [20, 110])
        f3_font = pygame.font.Font(None, 26)
        f3_surf = f3_font.render(final3, 1, (255, 255, 255))
        screen.blit(f3_surf, [20, 150])
        f4_font = pygame.font.Font(None, 26)
        f4_surf = f4_font.render(final4, 1, (255, 255, 255))
        screen.blit(f4_surf, [20, 180])
        pygame.display.flip()

myThrottle = ThrottleClass([15, 500])
running = True
while running:
    clock.tick(30)
    fps = clock.get_fps()
    if fps < 1: fps = 30
```

Показываем статистику при помощи объектов шрифта

Рисуем треугольники пламени

Показываем пламя в реактивном двигателе при помощи двух треугольников

Показываем итоговый счет после окончания игры

Создаем объект регулятора

Запускаем основной цикл Pygame-событий

```
    if height > 0.01:
        calculate_velocity()
        screen.fill([0, 0, 0])
        display_stats()
        pygame.draw.rect(screen, [0, 0, 255], [80, 350, 24, 100], 2)
        fuelbar = 96 * fuel / 5000
        pygame.draw.rect(screen, [0,255,0],
                 [84,448-fuelbar,18, fuelbar], 0)
        pygame.draw.rect(screen, [255, 0, 0],
                 [25, 300, 10, 200],0)
        screen.blit(moon, [0, 500, 400, 100])
        pygame.draw.rect(screen, [60, 60, 60],
                 [220, 535, 70, 5],0)
        screen.blit(myThrottle.image, myThrottle.rect)
        display_flames()
        screen.blit(ship, [230, y_pos, 50, 90])
        instruct1 = "Приземлитесь аккуратно, не израсходовав все топливо"
        instruct2 = "Хорошая посадка: < 15 м/с Великолепная посадка: < 5 м/с"
        inst1_font = pygame.font.Font(None, 24)
        inst1_surf = inst1_font.render(instruct1, 1, (255, 255, 255))
        screen.blit(inst1_surf, [50, 550])
        inst2_font = pygame.font.Font(None, 24)
        inst2_surf = inst1_font.render(instruct2, 1, (255, 255, 255))
        screen.blit(inst2_surf, [20, 575])
        pygame.display.flip()

    else:
        display_final()
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            held_down = True
        elif event.type == pygame.MOUSEBUTTONUP:
            held_down = False
        elif event.type == pygame.MOUSEMOTION:
            if held_down:
                myThrottle.rect.centery = event.pos[1]
                if myThrottle.rect.centery < 300:
                    myThrottle.rect.centery = 300
                if myThrottle.rect.centery > 500:
                    myThrottle.rect.centery = 500
pygame.quit()
```

Контур указателя уровня топлива

Количество топлива

Ползунок регулятора горючего

Луна

Посадочная площадка

Рычаг управления двигателем

Корабль

Рисуем все

Игра окончена. Показываем итоговый счет

Проверяем сдвигаемый мышью регулятор

Обновляем положение регулятора

Попробуйте запустить программу. Возможно, вы окажетесь хорошим пилотом! Если игра покажется вам слишком легкой, отредактируйте код, увеличив силу тяжести и вес

корабля, уменьшив количество топлива или задав другие начальную высоту и скорость. Программист тут вы, и только вы решаете, как должна функционировать игра.

Симулятор посадочного модуля в основном связан с силой тяжести. Остальные разделы этой главы мы посвятим другому важному аспекту моделирования — времени. И создадим модель, в которой нам потребуется следить за временем.

Слежение за временем

Время является важным фактором для множества компьютерных моделей. Иногда возникает необходимость его ускорить, заставив процесс протекать быстрее, чем в реальном мире, чтобы не приходилось сидеть и долго ждать его завершения. Иногда его требуется замедлить, чтобы появилась возможность как следует рассмотреть процессы, протекающие слишком быстро. А иногда нужно, чтобы программа работала в *реальном времени*, полностью имитируя процесс реального мира. Но во всех случаях вашим программам не обойтись без часов, которые будут отвечать за измерение времени.

В любом компьютере есть встроенные часы, которые вы можете использовать. Мы уже рассматривали несколько примеров работы со временем:

- в главе 8 функция `time.sleep()` помогла нам создать таймер обратного отсчета;
- в Pygame-программах мы использовали функции `time.delay` и `clock.tick` для управления скоростью анимации или частотой кадров. Еще мы прибегали к функции `get_fps()` для проверки скорости воспроизведения анимации, что тоже представляет собой средство измерения времени (среднего времени для каждого кадра).

Итак, мы уже следили за временем в исполняющихся программах, но иногда это требуется и для еще не запущенных программ. Например, модель виртуального домашнего любимца никто не будет оставлять включенной круглосуточно. Поиграв с такой программой, пользователь ее выключает на некоторое время. И было бы логично, если бы при этом питомец мог устать, проголодаться или пойти спать. То есть программа должна знать, сколько времени прошло с момента ее последнего запуска.

Если сохранить файл перед выключением программы, добавив небольшой фрагмент информации — текущее время, тогда при следующем старте эти сведения будут прочитаны. Программа сравнит записанное и текущее время и путем несложных расчетов определит, сколько прошло с момента ее последнего запуска.

В Python для работы со временем и датами существует специальный объект. С ним мы сейчас познакомимся.

Новые слова

Время, сохраненное в файле для последующего считывания, называют *временной меткой* (timestamp).

Объекты времени

Классы таких Python-объектов, как объекты даты и времени, определяются в модуле `datetime`. Этот модуль содержит классы, позволяющие работать с датами, временем и разницей, или *дельтой* (delta), между двумя датами или между двумя отсчетами времени.

НОВЫЕ СЛОВА

Словом *дельта* называется буква греческого алфавита.

Буквы греческого алфавита часто применяются в физике и математике для обозначения определенных величин. Дельта обозначает разницу между двумя величинами.

Сначала мы воспользуемся объектом `datetime`. (Да, имя класса в данном случае совпадает с именем модуля.) В объект `datetime` входят год, месяц, день, час, минута и секунда. Он создается следующим образом (запустите этот код в интерактивном режиме):

```
>>> import datetime
>>> when = datetime.datetime(2012, 10, 24, 10, 45, 56)
```

Имя модуля *Имя класса*

Посмотрим, что у нас получилось:

```
>>> print when
2012-10-24 10:45:56
```

Мы создали объект `datetime` с именем `when`, содержащий сведения о дате и времени.

При создании объекта `datetime` имеет место следующий порядок следования параметров (чисел в скобках): год, месяц, день, час, минута, секунда. Но если вы не сможете этого запомнить, используйте любой удобный вам порядок. Главное — сообщить его интерпретатору Python:

```
>>> when = datetime.datetime(hour=10, year=2012, minute=45, month=10,
second=56, day=24)
```

С объектами `datetime` возможны и другие операции. Можно формировать отдельные фрагменты, содержащие сведения (о годе, дне или минуте). Можно получить дату и время в виде отформатированной строки. Запустите этот код в интерактивном режиме:

```
>>> print when.year
2012
>>> print when.day
24
>>> print when.ctime()
Wed Oct 24 10:45:56 2012
```

Вывод даты и времени в виде строки

Получение отдельных фрагментов объекта datetime

Объект `datetime` содержит одновременно дату и время. Если вас интересует только дата, воспользуйтесь классом `date`, в который входит только информация о годе, месяце и дне. А в ситуации, когда требуется только время, имеет смысл прибегнуть к классу `time`, включающему в себя часы, минуты и секунды. Вот как выглядят эти классы:

```
>>> today = datetime.date(2012, 10, 24)
>>> some_time = datetime.time(10, 45, 56)
>>> print today
2012-10-24
>>> print some_time
10:45:56
```

Как и в случае с объектом `datetime`, вы можете менять порядок следования параметров, просто его обязательно нужно обозначить:

```
>>> today = datetime.date(month=10, day=24, year=2012)
>>> some_time = datetime.time(second=56, hour=10, minute=45)
```

Существует способ разбить объект `datetime` на объекты `date` и `time`:

```
>>> today = when.date()
>>> some_time = when.time()
```

А за объединение объектов `date` и `time` с целью формирования объекта `datetime` отвечает метод `combine()` класса `datetime` из модуля `datetime`:

```
>>> when = datetime.datetime.combine(today, some_time)
```

Имя модуля *Имя класса* *Метод*

Теперь, когда вы познакомились с объектом `datetime` и некоторыми его свойствами, давайте рассмотрим способы сравнения двух моментов времени и определения, сколько времени прошло от одного момента до другого.

Вычитание двух дат

В компьютерных моделях часто возникает необходимость узнать, сколько времени прошло с определенного момента. Например, программа, имитирующая домашнего любимца, должна знать, как давно его кормили в последний раз. Именно эта информация позволит ей определить, насколько голоден питомец.

Решать подобные задачи помогает класс `timedelta` из модуля `datetime`. Надеюсь, вы помните, что слово *дельта* означает разницу. Поэтому название `timedelta` указывает на разницу между двумя временны́ми метками.

Так выглядит процесс создания класса `timedelta` и последующего вычитания двух дат:

```
>>> import datetime
>>> yesterday = datetime.datetime(2012, 10, 23)
>>> tomorrow = datetime.datetime(2012, 10, 25)
>>> difference = tomorrow - yesterday
>>> print difference
2 days, 0:00:00
>>> print type(difference)
<type 'datetime.timedelta'>
```

Получаем разницу между двумя датами

Между завтра и послезавтра умещается 2 дня

Разница является объектом timedelta

Обратите внимание, что результатом вычитания двух объектов `datetime` является не еще один объект `datetime`, а объект `timedelta`. Это преобразование интерпретатор Python выполняет автоматически.

МАЛЕНЬКИЕ ФРАГМЕНТЫ ВРЕМЕНИ

До сих пор мы рассматривали время, измеряемое в секундах. Но объекты времени (`date`, `time`, `datetime` и `timedelta`) поддерживают куда большую точность. С их помощью можно измерять даже микросекунды, что составляет одну миллионную долю секунды.

В качестве примера рассмотрим метод `now()`, сообщающий текущие показания встроенных в компьютер часов:

```
>>> print datetime.datetime.now()
2012-10-24 21:25:44.343000
```

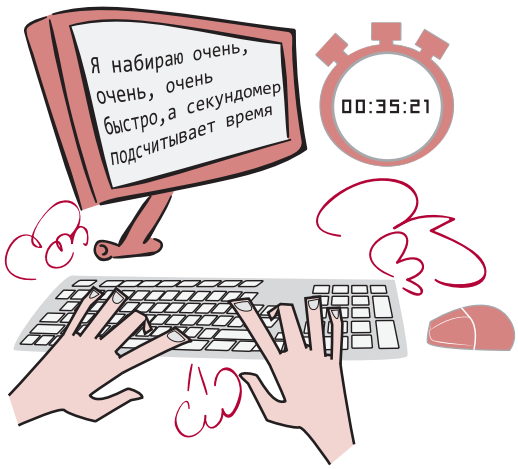
Обратите внимание, что результат работы этого метода содержит не секунды, а доли секунд:

```
44.343000
```

На моем компьютере последние три цифры всегда равны нулю, так как часы в моей операционной системе отсчитывают только миллисекунды (тысячные секунд). С моей точки зрения, это более чем достаточная точность!

Важно знать, что хотя эти данные имеют вид числа с плавающей точкой, хранятся они как два целых числа, отдельно указывающих количество секунд и количество микросекунд: 44 секунды и 343 000 микросекунды. Преобразовать эти данные в число с плавающей точкой можно по специальной формуле. Предположим, у нас есть объект `time` с именем `some_time` и вы хотите получить количество секунд в форме вещественного числа. Вот как это делается:

```
seconds_float = some_time.second + some_time.microsecond / float(1000000)
```



Функция `float()` гарантирует, что вам будет возвращено вещественное число, а не результат деления нацело.

При помощи метода `now()` и объекта `timedelta` можно протестировать вашу скорость набора текста. Программа из листинга 24.2 выводит на экран случайную фразу, которую должен набрать пользователь. Затем подсчитывается, сколько времени занял этот процесс, и вычисляется скорость набора. Попробуйте сами.

Листинг 24.2. Измерение временны́х интервалов — тест на скорость набора

```
import time, datetime, random

messages = [
    "Из всех деревьев мы врезались в то, которое смогло дать нам сдачи.",
    "Если не прекратить его попытки спасти вам жизнь, он вас убьет.",
    "Нашу сущность намного лучше демонстрируют действия, а не возможности.",
    "Я маг, а не размахивающий палкой бабуин.",
    "Величие порождает зависть, зависть — злобу, злоба — ложь.",
    "В мечтах мы попадаем в наш и только наш мир.",
    "Я убежден, что истина, как правило, предпочтительнее лжи.",
    "Казалось, что рассвет следует за полночью с неприличной поспешностью."
    ]

print "Проверка скорости набора. Введите следующую фразу. Я засеку время."
time.sleep(2)
print "\nПриготовиться..."
time.sleep(1)
print "\nСосредоточиться..."
time.sleep(1)
print "\nНачали:"
message = random.choice(messages)
print "\n " + message
start_time = datetime.datetime.now()
typing = raw_input('>')
end_time = datetime.datetime.now()
diff = end_time - start_time
typing_time = diff.seconds + diff.microseconds / float(1000000)
cps = len(message) / typing_time
wpm = cps * 60 / 5.0
```

Используем модуль time для функции sleep()

Используем модуль time для функции sleep()

Выводим инструкции

Выбираем фразу из списка

Запускаем часы

Останавливаем часы

Рассчитываем затраченное время

Для расчета скорости набора предполагаем, что 1 слово = 5 символов

```
print "\nВы ввели %i символов за %.1f секунд." % (len(message),
                      typing_time)

print "Это %.2f символов в секунду, или %.1f слов в минуту" %(cps, wpm)
if typing == message:
    print "Вы не сделали ни одной ошибки."
else:
    print "Но вы сделали по крайней мере одну ошибку."
```

Выводим результат в отформатированном виде

Существует еще одна особенность объектов `timedelta`, о которой нужно помнить. В отличие от объектов `datetime`, содержащих данные о годах, месяцах, днях, часах, минутах, секундах (и микросекундах), объект `timedelta` содержит только данные о днях, секундах и микросекундах. Информацию о месяцах или годах следует рассчитывать самостоятельно, исходя из количества дней. А разницу в минутах или часах нужно вычислять, исходя из количества секунд.

Сохранение времени в файле

В начале главы мы говорили о том, что периодически возникает необходимость сохранить информацию о времени в файле (на жестком диске), чтобы она не пропала после завершения работы программы. Так, воспользовавшись функцией `now()` в момент завершения работы программы и сохранив полученный результат, вы сможете при следующем запуске программы проверить текущее время и вывести на экран такую информацию:

```
В последний раз вы запускали эту программу 2 дня, 7 часов, 23 минуты назад.
```

Разумеется, большинству программ это просто не нужно, но бывают случаи, когда сведения о времени простоя являются ключевыми. Примером такой программы может служить Виртуальный питомец. Как и в популярных некоторое время назад тамагочи, в нашей программе нужно следить за тем, сколько времени прошло с момента последнего обращения. Если вы не играли, скажем, два дня, ваш питомец явно очень голоден! Но запрограммировать подобное поведение можно только при условии, что вы знаете, сколько времени прошло с момента последнего кормления. При этом учитывается и то время, когда программа не работала.

Сохранить время в файле можно двумя способами. Можно записать в файл строку:

```
timeFile.write ("2012-10-24 14:23:37")
```

В этом случае для чтения временно́й метки применяется строковый метод `split()`, разбивающий строку на отдельные части с такой информацией, как день, месяц, год, час, минута, секунда.

Еще можно воспользоваться модулем `pickle`, с которым вы познакомились в главе 22. Он позволяет сохранять в файле переменные любого типа, в том числе объекты. Так как слежение за временем осуществляется с помощью объектов `datetime`, модуль `pickle` легко сохранит их в файле и выполнит чтение из файла.

Рассмотрим простой пример, в котором на экран выводится информация о последнем времени запуска программы. Нам нужно:

- найти сериализованный файл и открыть его. Модуль, позволяющий определить, существует ли указанный файл, называется `os` (сокращенное от «операционная система»). Мы воспользуемся встроенным в этот модуль методом `isfile()`;
- обнаружение файла указывает на то, что программа ранее запускалась, и мы нашли сведения о времени этого запуска (указанное в сериализованном файле время);
- далее следует записать в файл текущее время;
- если программа запускается в первый раз, файл с сериализованными данными пока отсутствует, и мы покажем сообщение, информирующее о создании такого файла.

Код этой программы вы найдете в листинге 24.3. Запустите его и посмотрите, как он работает.

Листинг 24.3. Сохранение времени в файле при помощи модуля pickle

```
import datetime, pickle
import os

first_time = True
if os.path.isfile("last_run.pkl"):
    pickle_file = open("last_run.pkl", 'r')
    last_time = pickle.load(pickle_file)
    pickle_file.close()
    print "В последний раз эта программа запускалась ", last_time
    first_time = False

pickle_file = open("last_run.pkl", 'w')
pickle.dump(datetime.datetime.now(), pickle_file)
pickle_file.close()
if first_time:
    print "Создан новый файл с временем запуска."
```

Импортируем модули datetime, pickle и os

Проверяем наличие сериализованного файла

Открываем сериализованный файл для чтения (если он существует)

Десериализуем объект datetime

Открываем (или создаем) файл для записи текущего времени

Сериализуем объект datetime, содержащий текущее время

Теперь у нас есть все фрагменты для создания простой модели виртуального питомца. Этим мы и займемся в следующем разделе.

Виртуальный питомец

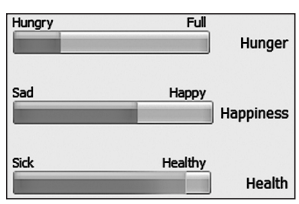
Мы напишем упрощенную программу Virtual Pet (Виртуальный питомец), которая представляет собой компьютерную модель. Игрушки с виртуальными питомцами продаются в магазинах (они напоминают брелок с маленьким экраном). Существуют сайты Neopets и Webkinz, на которых можно завести себе виртуального домашнего любимца. И все это тоже компьютерные модели. Они имитируют поведение живых существ, которые могут испытывать голод, одиночество, усталость и т. п. Для поддержки нормальной жизнедеятельности этих животных нужно кормить, играть с ними или даже показывать доктору.

Наш виртуальный питомец будет далеко не таким сложным и реалистичным, как продающиеся в магазинах или доступные в Интернете. Я хочу, чтобы вы получили основное представление о создании подобных программ, а чрезмерно сложный код может затруднить восприятие. Впоследствии ничто не мешает вам доработать и улучшить простую версию программы в соответствии с собственными предпочтениями.

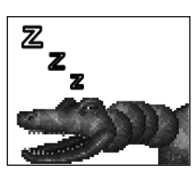
Вот основные функциональные возможности нашей программы.

- Вы можете проделать четыре основных действия: покормить питомца (*feed*), погулять с ним (*walk*), поиграть с ним (*play*) или показать врачу (*doctor*).

- Вы можете следить за тремя аспектами состояния питомца: голод (*hunger*), счастье (*happiness*) и здоровье (*health*).



- Питомец может бодрствовать или спать.



- Со временем питомец начинает испытывать голод, который постепенно усиливается. Он ослабевает при кормлении.
- Чувство голода возникает медленнее, если питомец спит.
- Если приступить к выполнению какого-либо действия, когда питомец спит, он проснется.
- Если питомец слишком голоден, параметр *happiness* начинает уменьшаться.
- Если питомец голоден слишком сильно, начинает уменьшаться параметр *health*.
- Прогулка с питомцем увеличивает как параметр *happiness*, так и параметр *health*.

- Игра с питомцем увеличивает параметр *happiness*.
- Параметр *health* также можно увеличить, показав питомца доктору.
- Мы будем использовать шесть вариантов изображения:
  - спящий питомец;
  - бодрствующий, но ничем не занятый питомец;
  - гуляющий питомец;
  - играющий питомец;
  - питающийся питомец;
  - питомец у доктора.

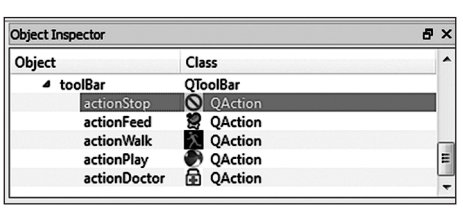
Изображения будут анимированными. Посмотрим, как все это оформить в программе.

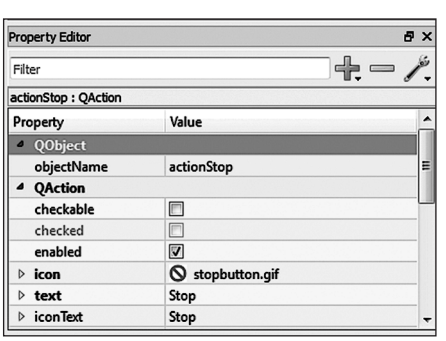
GUI

Мы с Картером при помощи модуля PyQt создали для нашей программы GUI-интерфейс. В нем есть кнопки (то есть значки на панели инструментов), управляющие различными действиями, индикаторы разных жизненных параметров, а также место для картинки, демонстрирующей состояние питомца.

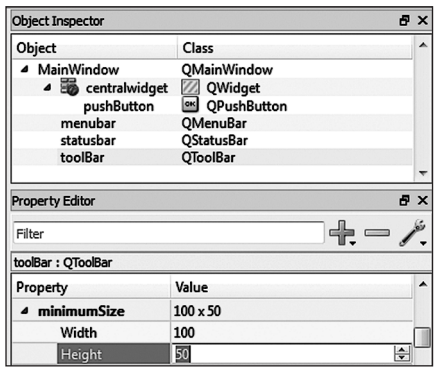
Обратите внимание на заголовок *Virtual Pet* (Виртуальный питомец). Как отредактировать заголовок окна? В окне Qt-дизайнера откройте новую форму и щелчком выделите объект *MainWindow* в инспекторе объектов. Затем в редакторе свойств найдите свойство *windowTitle* и присвойте ему значение *Virtual Pet* (или выберите заголовок по вкусу).

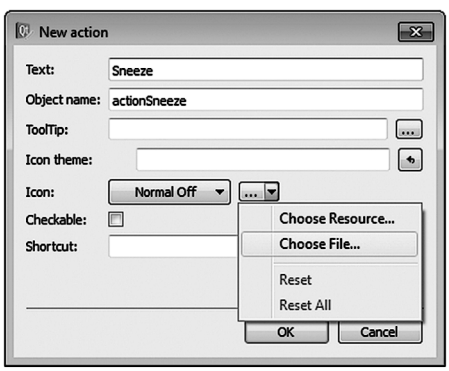
Группа отвечающих за различные действия кнопок создана при помощи PyQt-виджета, который называется *Toolbar* (Панель инструментов). С панелью инструментов, как и с меню, связан набор *действий*, просто в случае с панелью каждому действию соответствует свой *значок*.





Чтобы добавить панель инструментов, щелкните правой кнопкой мыши в основном окне и выберите в меню команду *Add Toolbar* (Добавить панель инструментов). В верхней части формы появится панель инструментов очень маленького размера. Выделите ее в инспекторе объектов и в редакторе свойств найдите свойство *minimumSize*. Присвойте минимальной ширине значение 100, а минимальной высоте — 50.





Для добавления на панель инструментов действий (значков) в окне Qt-дизайнера перейдите на вкладку *Action Editor* (Редактор действий) в правом нижнем углу. Щелкните правой кнопкой мыши в произвольном месте открывшегося окна и выберите команду *New* (Создать). Откроется диалоговое окно для добавления нового действия. Вам нужно ввести имя действия в поле *Text* (Текст), и Qt-дизайнер автоматически создаст имя объекта. Затем найдите в центре раскрывающийся список, обозначенный тремя точками (…), и щелкните на расположенной справа от него кнопке со стрелкой. Выберите в открывшемся меню команду *Choose File* (Выбрать файл) и укажите файл с изображением, которое вы планируете использовать для данной кнопки.

Щелкните по кнопке *OK*, и новое действие будет создано. Оно сразу же появится в списке редактора действий. Останется перетащить его на панель инструментов. После этого выбранный вами графический фрагмент превратится в кнопку. Qt-дизайнер автоматически подгонит его размер под размер панели инструментов.

Индикатором здоровья служит виджет *Progress Bar* (Индикатор процесса). Для вывода графики мы воспользовались виджетом *Push Button* (Обычная кнопка), с которым вы уже знакомы, отредактировав его свойства таким образом, чтобы он перестал выглядеть как кнопка, а стал похож на картинку.

Весь дополнительный текст был реализован при помощи виджетов *Label* (Метка).

Вы можете самостоятельно создать подобный GUI-интерфейс в Qt-дизайнере или загрузить нашу версию (из папки *examples*) и исследовать виджеты и их свойства.

Алгоритм

Для написания кода нам нужно конкретизировать поведение нашего питомца. Вот каким алгоритмом мы воспользуемся:

- поделим «день» питомца на 60 частей, которые назовем *циклами*. Каждый цикл занимает 5 секунд реального времени, соответственно, виртуальный день нашего питомца составит 5 минут;
- бодрствование будет занимать 48 циклов, после чего последуют 12 циклов сна. Питомца можно разбудить, но есть вероятность, что он рассердится!
- голод, счастье и здоровье будут оцениваться по шкале от 0 до 8;
- в бодрствующем состоянии голод увеличивается на единицу за каждый цикл, а счастье на единицу уменьшается за каждые 2 цикла (за исключением ситуации, когда вы выгуливаете питомца или играете с ним);
- во время сна голод увеличивается на единицу каждые 3 цикла;
- во время еды голод уменьшается на 2 единицы за каждый цикл;
- когда вы играете с питомцем, счастье увеличивается на 1 за каждый цикл;
- во время прогулки счастье и здоровье увеличиваются на 1 за каждые 2 цикла;
- во время посещения доктора здоровье увеличивается на 1 за каждый цикл;
- если голод достигает 7, здоровье уменьшается на 1 за каждые 2 цикла;
- если голод достигает 8, здоровье уменьшается на 1 за каждый цикл;
- у разбуженного питомца счастье уменьшается на 4 единицы;
- если программа не работает, питомец либо бодрствует (ничего не делая), либо спит;
- при повторном запуске вычисляется количество прошедших циклов, и все показатели обновляются.

Кажется, что правил слишком много, но запрограммировать подобное поведение не сложно. Более того, вы, скорее всего, захотите добавить собственные варианты поведения, чтобы сделать игру более интересной. Код с пояснениями вы найдете далее.

Простая анимация

Создать анимацию можно и без Pygame-модуля. В PyQt это делается при помощи *таймера*. Таймер — это элемент, генерирующий *события* с определенной частотой. После чего вам остается написать *обработчик события* срабатывания таймера. Аналогичным образом мы писали обработчики пользовательских событий, таких как щелчок по кнопке, просто в этом случае событие создается не пользователем, а программой. Сработавший таймер генерирует так называемые *события тайм-аута*.

В нашем GUI-интерфейсе для виртуального питомца есть два таймера: один отвечает за анимацию, а второй считает циклы. Анимация должна обновляться каждые полсекунды (0.5 секунды), в то время как один цикл занимает 5 секунд.

При срабатывании отвечающего за анимацию таймера мы меняем картинку с питомцем. Каждому состоянию (еда, игра и т. п.) соответствует собственный набор анимированных изображений. При этом все наборы хранятся в виде списков. Анимация возникает за

счет циклического вывода элементов списка на экран. Список программа выбирает в зависимости от текущей активности питомца.

ПОПРОБУЙ ЭТО СНОВА

В нашей программе мы воспользуемся новой конструкцией. Это блок `try-except`. Если программа может сделать нечто, приводящее к ошибке, имеет смысл добавить в нее код обработки этой ошибки. В противном случае ее работа просто прекратится. Блок `try-except` отвечает именно за обработку ошибок.

Так, при попытке открыть несуществующий файл вы получаете сообщение об ошибке. Если эту ошибку не обработать, произойдет остановка программы. А возможно, вы предпочли бы попросить пользователя заново ввести имя файла. Вдруг в первый раз он просто опечатался. Блок `try-except` позволит вам обойти ошибку и продолжить работу.

Вот как выглядит пример с открытием файла:

```
try:
    file = open("somefile.txt", "r")
except:
    print "Невозможно открыть файл. Хотите повторно ввести имя файла?"
```

Действие, которое вы хотите попытаться сделать (потенциально приводящее к ошибке), помещается в блок `try`. В данном случае мы пытаемся открыть файл. В случае успеха операции часть `except` пропускается. Если же код из блока `try` приводит к ошибке, запускается код из блока `except`, сообщающий программе, что ей делать в случае ошибки. Это можно представить следующим образом:

```
try:
    сделать это (больше ничего не делать...)
except:
    в случае ошибки сделать это
```

С помощью инструкций блока `try-except` интерпретатор Python выполняет так называемую *обработку ошибок* (error handling). Именно обработка ошибок позволяет писать код, в котором что-то может пойти не так, гарантируя при этом работоспособность программы. Подробно мы эту тему рассматривать не будем, но я хотел дать вам основные представления об обработке ошибок, так как с этой процедурой вы встретитесь в коде программы Virtual Pet.

Давайте посмотрим на сам код, представленный в листинге 24.4. Он снабжен подробными комментариями. Если вы не хотите набирать его самостоятельно (он достаточно длинный), можете воспользоваться файлом из папки *\Examples\VirtualPet*. Еще его можно

загрузить с сайта книги *www.manning.com/books/hello-world-second-edition*. Там же находится созданный модулем PyQt файл с пользовательским интерфейсом и вся прилагающаяся графика. Запустите этот код и убедитесь, что понимаете принцип его работы.

Листинг 24.4. VirtualPet.py

```
import sys, pickle,datetime
from PyQt4 import QtCore, QtGui, uic

formclass, baseclass = uic.loadUiType("mainwindow.ui")

class MyForm(baseclass, formclass):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.doctor = False
        self.walking = False
        self.sleeping = False
        self.playing = False
        self.eating = False
        self.time_cycle = 0
        self.hunger = 0
        self.happiness = 8
        self.health = 8
        self.forceAwake = False
        self.sleepImages = ["sleep1.gif","sleep2.gif","sleep3.gif",
                            "sleep4.gif"]
        self.eatImages = ["eat1.gif", "eat2.gif"]
        self.walkImages = ["walk1.gif", "walk2.gif", "walk3.gif",
                           "walk4.gif"]
        self.playImages = ["play1.gif", "play2.gif"]
        self.doctorImages = ["doc1.gif", "doc2.gif"]
        self.nothingImages = ["pet1.gif", "pet2.gif", "pet3.gif"]

        self.imageList = self.nothingImages
        self.imageIndex = 0

        self.actionStop.triggered.connect(self.stop_Click)
        self.actionFeed.triggered.connect(self.feed_Click)
        self.actionWalk.triggered.connect(self.walk_Click)
        self.actionPlay.triggered.connect(self.play_Click)
        self.actionDoctor.triggered.connect(self.doctor_Click)

        self.myTimer1 = QtCore.QTimer(self)
        self.myTimer1.start(500)
        self.myTimer1.timeout.connect(self.animation_timer)
        self.myTimer2 = QtCore.QTimer(self)
```

Инициализируем значения

Списки изображений для анимации

Связываем обработчики событий с кнопками панели инструментов

Настраиваем таймеры

```
        self.myTimer2.start(5000)
        self.myTimer2.timeout.connect(self.tick_timer)

        filehandle = True
        try:
            file = open("savedata_vp.pkl", "r")
        except:
            filehandle = False
        if filehandle:
            save_list = pickle.load(file)
            file.close()
        else:
            save_list = [8, 8, 0, datetime.datetime.now(), 0]
        self.happiness = save_list[0]
        self.health = save_list[1]
        self.hunger = save_list[2]
        timestamp_then = save_list[3]
        self.time_cycle = save_list[4]

        difference = datetime.datetime.now() - timestamp_then
        ticks = difference.seconds / 50
        for i in range(0, ticks):
            self.time_cycle += 1
            if self.time_cycle == 60:
                self.time_cycle = 0
            if self.time_cycle <= 48:
                self.sleeping = False
                if self.hunger < 8:
                    self.hunger += 1
            else:
                self.sleeping = True
                if self.hunger < 8 and self.time_cycle % 3 == 0:
                    self.hunger += 1
            if self.hunger == 7 and (self.time_cycle % 2 ==0) \
                and self.health > 0:
                self.health -= 1
            if self.hunger == 8 and self.health > 0:
                self.health -=1
        if self.sleeping:
            self.imageList = self.sleepImages
        else:
            self.imageList = self.nothingImages

    def sleep_test(self):
        if self.sleeping:
            result = (QtGui.QMessageBox.warning(self, 'WARNING',
"Are you sure you want to wake your pet up? He'll be unhappy about it",
```

Настраиваем таймеры

Пытаемся открыть сериализованный файл

Если файл открывается, читаем оттуда

Если сериализованный файл не открывается, используем значения, предлагаемые по умолчанию

Извлекаем из списка отдельные значения

Проверяем, сколько времени прошло с момента последнего запуска

Моделируем все циклы, прошедшие за время простоя

Бодрствует

Спит

Выбираем корректную анимацию — бодрствование или сон

Тип диалогового окна

```
                    QtGui.QMessageBox.Yes | QtGui.QMessageBox.No,
                    QtGui.QMessageBox.No))
            if result == QtGui.QMessageBox.Yes:
                self.sleeping = False
                self.happiness -= 4
                self.forceAwake = True
                return True
            else:
                return False
        else:
            return True

    def doctor_Click(self):
        if self.sleep_test():
            self.imageList = self.doctorImages
            self.doctor = True
            self.walking = False
            self.eating = False
            self.playing = False

    def feed_Click(self):
        if self.sleep_test():
            self.imageList = self.eatImages
            self.eating = True
            self.walking = False
            self.playing = False
            self.doctor = False

    def play_Click(self):
        if self.sleep_test():
            self.imageList = self.playImages
            self.playing = True
            self.walking = False
            self.eating = False
            self.doctor = False

    def walk_Click(self):
        if self.sleep_test():
            self.imageList = self.walkImages
            self.walking = True
            self.eating = False
            self.playing = False
            self.doctor = False
```

Выводимые на экран кнопки

Кнопка, предлагаемая по умолчанию

Перед выполнением действия проверяем, не спит ли питомец

Обработчик события кнопки doctor

Обработчик события кнопки feed

Обработчик события кнопки play

Обработчик события кнопки walk

```
    def stop_Click(self):
        if not self.sleeping:
            self.imageList = self.nothingImages
            self.walking = False
            self.eating = False
            self.playing = False
            self.doctor = False

    def animation_timer(self):
        if self.sleeping and not self.forceAwake:
            self.imageList = self.sleepImages
        self.imageIndex += 1
        if self.imageIndex >= len(self.imageList):
            self.imageIndex = 0
        icon = QtGui.QIcon()
        current_image = self.imageList[self.imageIndex]
        icon.addPixmap(QtGui.QPixmap(current_image),
                       QtGui.QIcon.Disabled, QtGui.QIcon.Off)
        self.petPic.setIcon(icon)
        self.progressBar_1.setProperty("value", (8-self.hunger)*(100/8.0))
        self.progressBar_2.setProperty("value", self.happiness*(100/8.0))
        self.progressBar_3.setProperty("value", self.health*(100/8.0))

    def tick_timer(self):
        self.time_cycle += 1
        if self.time_cycle == 60:
            self.time_cycle = 0
        if self.time_cycle <= 48 or self.forceAwake:
            self.sleeping = False
        else:
            self.sleeping = True
        if self.time_cycle == 0:
            self.forceAwake = False
        if self.doctor:
            self.health += 1
            self.hunger += 1
        elif self.walking and (self.time_cycle % 2 == 0):
            self.happiness += 1
            self.health += 1
            self.hunger += 1
        elif self.playing:
            self.happiness += 1
            self.hunger += 1
        elif self.eating:
            self.hunger -= 2
        elif self.sleeping:
```

Обработчик события кнопки stop

Обработчик события анимационного таймера (каждые 0.5 секунды)

Обновляем изображение питомца (анимация)

Запускаем обработчик события основного 5-секундного таймера

Проверяем, спит или бодрствует питомец

Добавляем или удаляем элементы в зависимости от типа активности

```
            if self.time_cycle % 3 == 0:
                self.hunger += 1
        else:
            self.hunger += 1
            if self.time_cycle % 2 == 0:
                self.happiness -= 1
        if self.hunger > 8: self.hunger = 8
        if self.hunger < 0: self.hunger = 0
        if self.hunger == 7 and (self.time_cycle % 2 ==0) :
            self.health -= 1
        if self.hunger == 8:
            self.health -=1
        if self.health > 8: self.health = 8
        if self.health < 0: self.health = 0
        if self.happiness > 8: self.happiness = 8
        if self.happiness < 0: self.happiness = 0
        self.progressBar_1.setProperty("value", (8-self.hunger)*(100/8.0))
        self.progressBar_2.setProperty("value", self.happiness*(100/8.0))
        self.progressBar_3.setProperty("value", self.health*(100/8.0))

    def closeEvent(self, event):
        file = open("savedata_vp.pkl", "w")
        save_list = [self.happiness, self.health, self.hunger, \
                     datetime.datetime.now(), self.time_cycle]
        pickle.dump(save_list, file)
        event.accept()

    def menuExit_selected(self):
        self.close()

app = QtGui.QApplication(sys.argv)
myapp = MyForm()
myapp.show()
app.exec_()
```

Добавляем или удаляем элементы в зависимости от типа активности

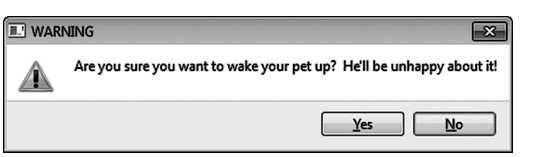
Проверяем, чтобы значения не выходили за границы диапазона

Обновляем индикаторы процесса

Символ продолжения строки

Сохраняем в сериализованном файле состояние и временну́ю метку

Функция `sleep_test()` пользуется диалоговым окном с «предупреждающим сообщением» модуля PyQt. Пара параметров сообщает ей, какие кнопки следует отобразить и какая из них является кнопкой, предлагаемой по умолчанию. Этот момент объяснен в примечаниях к листингу 24.4. Открывшееся диалоговое окно (при попытке разбудить питомца) выглядит так:



Не волнуйтесь, если вы не до конца поняли, как работает код. При желании можете самостоятельно прочитать о библиотеке PyQt.

В этой главе мы всего лишь слегка затронули возможности компьютерного моделирования. Вы получили базовые представления об имитации условий реального мира, например силы тяжести и времени, но компьютерное моделирование широко применяется в физике, инженерном деле, медицине, финансовой сфере и многих других областях. Многие модели крайне сложны, и их реализация даже на самых быстрых компьютерах занимает дни или даже недели. Но даже маленький виртуальный питомец на брелоке представляет собой компьютерную модель, а порой самые простые модели являются самыми интересными.

Что мы узнали

В этой главе мы узнали:

- что такое компьютерные модели и зачем они нужны;
- как моделировать силу тяжести, ускорение и реактивную тягу;
- как следить за временем и моделировать его течение;
- как с помощью сериализации сохранять в файле временны́е метки;
- как обработать ошибку (`try-except`);
- как генерировать периодические события при помощи таймера.

Проверь себя

1. Назовите три причины применения компьютерных моделей.
2. Перечислите три вида компьютерных моделей, которые вы видели или о которых слышали.
3. Объект какого типа применяется для хранения разницы между двумя датами или двумя моментами времени?

Эксперименты

1. Добавьте к программе Lunar Lander проверку на уход с орбиты. Если корабль уходит за пределы окна, а его скорость превышает +100 м/с, программа должна прекратить свою работу и вывести сообщение: «Лунная гравитация на вас больше не действует. Сегодня никаких прилунений!»
2. Добавьте к игре Lunar Lander возможность повторения попыток приземления без необходимости заново загружать программу.
3. Добавьте к GUI-интерфейсу программы Virtual Pet кнопку ***Pause*** (Пауза), которая будет останавливать, или «замораживать», время для питомца. В зафиксированном состоянии он должен оставаться даже после прекращения работы программы. (Подсказка: скорее всего, вам нужно сохранить «остановленное» состояние в файле с помощью сериализации.)

Лыжник. Объяснение

Помните игру Skier (Лыжник)? В главе 10 вы набирали ее код (по крайней мере я надеюсь на это!), а затем запустили. Разумеется, код был снабжен комментариями, но дополнительных разъяснений я не предоставил. Набор и запуск кода, даже если вы не до конца понимаете принцип его работы, является замечательным способом изучения как программирования в целом, так и конкретного языка.

Теперь вы намного лучше знакомы с языком Python, и, скорее всего, вам будет любопытно понять, как функционирует программа Skier. Именно этому посвящена данная глава.

Лыжник

Сперва мы программируем лыжника. В процессе игры вы, вероятно, обратили внимание, что лыжник двигается по экрану только вперед и назад. Смещаться вверх и вниз он не умеет. Иллюзия спуска «вниз» возникает за счет прокрутки фона (деревьев и флагов).

У нас пять изображений спускающегося по холму лыжника: одно для спуска вниз, два для поворота направо (на небольшой угол и резкий поворот) и два для поворота налево (на небольшой угол и резкий поворот). В начале программы из этих изображений был создан список, при этом в список изображения были помещены в определенном порядке:

```
skier_images = ["skier_down.png",
                "skier_right1.png", "skier_right2.png",
                "skier_left2.png", "skier_left1.png"]
```

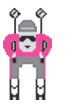
Вскоре вы поймете, почему в данном случае порядок имеет значение. Для слежения за ориентацией лыжника используется переменная `angle`. Она может принимать значения от –2 до +2:

–2 — резкий поворот налево;
–1 — плавный поворот налево;
0 — спуск вниз;
+1 — плавный поворот направо;
+2 — резкий поворот направо.

(Лево и право рассматриваются с нашей точки зрения, а не с точки зрения лыжника.)

Значение переменной `angle` позволяет выбрать корректное изображение. Более того, можно напрямую воспользоваться значением этой переменной в качестве индекса списка изображений.

- `skier_images[0]` — спуск вниз:



- `skier_images[1]` — плавный поворот направо:

- `skier_images[2]` — резкий поворот направо:

А теперь небольшая хитрость. Помните, в главе 12, где мы рассматривали списки, говорилось, что отрицательный индекс заставляет начать обратный отсчет с конца списка? Поэтому в нашем случае мы можем написать:

- `skier_images[-1]` — плавный поворот налево (в обычном случае это был бы элемент `skier_images[4]`):

- `skier_images[-2]` — резкий поворот налево (в обычном случае это был бы элемент `skier_images[3]`):

Теперь вы поймете, зачем был нужен строгий порядок следования элементов списка:

- `angle` = +2 (резкий поворот направо) = `skier_images[2]`;
- `angle` = +1 (плавный поворот направо) = `skier_images[1]`;
- `angle` = 0 (спуск вниз) = `skier_images[0]`;
- `angle` = –1 (плавный поворот налево) = `skier_images[-1]` (также известен как `skier_images[4]`);
- `angle` = –2 (резкий поворот налево) = `skier_images[-2]` (также известен как `skier_images[3]`).

Мы создали для лыжника класс, который входит в Pygame-класс `Sprite`. Лыжник зафиксирован на расстоянии 100 пикселов от верхней границы окна, а в первый момент он будет располагаться точно по центру, то есть его координата *x* = 320, так как ширина окна составляет 640 пикселов. Значит, начальные координаты лыжника составляют [320, 100]. Вот первая часть определения его класса:

```
class SkierClass(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
```

```
        self.image = pygame.image.load("skier_down.png")
        self.rect = self.image.get_rect()
        self.rect.center = [320, 100]
        self.angle = 0
```

Метод класса будет отвечать за ориентацию лыжника, то есть менять переменную `angle`, загружать корректное изображение и определять скорость спуска. Скорость обладает компонентами *x* и *y*. Мы будем двигать спрайт с лыжником только влево и вправо (меняя его *x*-скорость). При этом от *y*-скорости зависит быстрота прокрутки фона (насколько быстро лыжник «спускается» с холма). При спуске вниз эта скорость увеличивается, а в моменты поворотов уменьшается. Вот формула расчета скорости:

```
speed = [self.angle, 6 - abs(self.angle) * 2]
```

Функция `abs` в этой строчке кода даст нам абсолютное значение переменной `angle`. То есть мы проигнорируем знак (+ или –). Для скорости спуска вниз не имеет значения, в какую сторону поворачивает лыжник. Нам хватит информации о крутизне этого поворота.

Вот метод, управляющий поворотами:

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

Также нам требуется метод смещения лыжника вправо и влево. Он должен гарантировать, что лыжник не выйдет за пределы окна:

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

Для направления лыжника вправо и влево мы будем пользоваться кнопками со стрелками. Добавив сюда код инициализации и цикла событий, мы получим работающую программу, содержащую только лыжника. Она представлена в листинге 25.1.

Листинг 25.1. Фрагмент игры Лыжник

```
import pygame, sys, random

skier_images = ["skier_down.png",
                "skier_right1.png", "skier_right2.png",
                "skier_left2.png", "skier_left1.png"]

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

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

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

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

pygame.init()
screen = pygame.display.set_mode([640,640])
clock = pygame.time.Clock()
skier = SkierClass()
speed = [0, 6]

running = True
while running:
    clock.tick(30)
    for event in pygame.event.get():
        if event.type == pygame.QUIT: running = False
        if event.type == pygame.KEYDOWN:
```

Различные изображения лыжника в зависимости от направления его движения

Не разрешаем лыжнику поворачиваться больше чем на +/–2

Двигаем лыжника влево и вправо

Перерисовываем экран

Проверяем нажатие клавиш

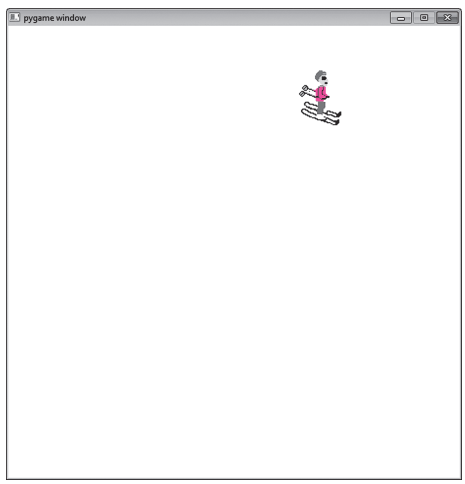
Главный цикл Pygame-событий

```
            if event.key == pygame.K_LEFT:
                speed = skier.turn(-1)
            elif event.key == pygame.K_RIGHT:
                speed = skier.turn(1)
        skier.move(speed)
        animate()

pygame.quit()
```

Правая стрелка поворачивает вправо

Левая стрелка поворачивает влево

Главный цикл Pygame-событий



Запустив программу из листинга 25.1, вы увидите одинокого лыжника (без очков и препятствий), которого можно будет поворачивать влево и вправо.

Препятствия

Теперь нам нужно решить, как создать препятствия — деревья и флаги. Для простоты эту часть мы начнем с чистого листа — лыжника в нашей программе не будет. Код лыжника и код препятствий мы совместим только в самом конце.

Размер игрового окна составляет 640 × 640 пикселов. Чтобы избежать слишком близкого расположения препятствий, поделим окно сеткой 10 × 10. В результате получим 100 квадратиков, каждый из которых будет иметь площадь 64 × 64 пиксела. Спрайты препятствий имеют куда меньший размер, поэтому между препятствиями, оказавшимися в соседних ячейках, обязательно будет пустое пространство.

создание отдельных препятствий

Первым делом нужно сгенерировать отдельные препятствия. Для этого создадим класс `ObstacleClass`. Как и в случае с лыжником, мы воспользуемся для этого классом `Sprite` из библиотеки Pygame:

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

создание карты препятствий

Теперь распределим препятствия по экрану. Их количество должно быть достаточным для заполнения сетки размером 10 × 10 ячеек, что соответствует экрану размером 640 × 640 пикселов. Случайным образом расставим 10 препятствий (флагов и деревьев) в 100 квадратах сетки. Каждое препятствие может быть флагом или деревом. То есть на экране может оказаться 2 флага и 8 деревьев, 7 флагов и 3 дерева или любая другая комбинация чисел, дающих в сумме 10. Числа в данном случае выбираются случайным образом. Случайным является и местоположение препятствия внутри сетки.

Но мы должны следить за тем, чтобы два препятствия не оказались в одной точке пространства. Поэтому будем фиксировать данные о выбранных местах. Воспользуемся переменной `locations`, которая представляет собой список уже занятых мест. При размещении нового препятствия мы сразу проверим, не занято ли выбранное нами место:

```
def create_map():
    global obstacles
    locations = []
    for i in range(10):
        row = random.randint(0, 9)
        col = random.randint(0, 9)
        location = [col * 64 + 32, row * 64 + 32 + 640]
        if not (location in locations):
            locations.append(location)
            type = random.choice(["tree", "flag"])
            if type == "tree": img = "skier_tree.png"
            elif type == "flag": img = "skier_flag.png"
            obstacle = ObstacleClass(img, location, type)
            obstacles.add(obstacle)
```

10 препятствий на один экран

(x, y) местоположение препятствия

Гарантируем, что два препятствия не окажутся в одной точке

Картер, ты очень внимателен! Дело в том, что в момент запуска игры мы не хотим видеть заполненный препятствиями экран. Лыжник начинает спускаться с чистого холма, а препятствия должны появиться ниже. То есть вид холма с препятствиями изначально скрыт под нижней границей окна. Это достигается добавлением значения 640 (высота нашего окна) к координате *y* каждого препятствия.

После запуска игры препятствия по нашему замыслу должны начать перемещаться вверх. Такая прокрутка реализуется изменением координаты *y* каждого из них. Скорость этого изменения зависит от того, насколько быстро лыжник спускается вниз. Мы поместили эти операции в метод `update()`, входящий в класс `ObstacleClass`:

```
def update(self):
    global speed
    self.rect.centery -= speed[1]
```

Глобальная переменная `speed` — это скорость нашего лыжника. Она состоит из двух значений, представляющих *x*- и *y*-скорости, и для выделения второго компонента мы используем индекс [1]. По мере прокрутки первой сцены с препятствиями под нижним краем экрана должна генерироваться следующая сцена. Как определить подходящий для этого момент? При помощи переменной `map_position`, следящей за величиной прокрутки. Этот код вставляется в основной цикл:

```
running = True
    while running:
        clock.tick(30)
        for event in pygame.event.get():
            if event.type == pygame.QUIT: running = False

        map_position += speed[1]

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

Следим за величиной прокрутки экрана

Если прокрутка сцены завершена, создаем новую версию экрана с препятствиями

Для перерисовки всех деталей сцены, как и в коде с одним лыжником, используется функция `animate()`. После сбора компонентов в одно целое код заполненного препятствиями экрана будет выглядеть так, как показано в листинге 25.2.

Листинг 25.2. Игра Лыжник — только препятствия

```
import pygame, sys, random

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

    def update(self):
        global speed
        self.rect.centery -= speed[1]
```

Класс для спрайтов, представляющих препятствия (деревья и флаги)

```
def create_map():
    global obstacles
    locations = []
    for i in range(10):
        row = random.randint(0, 9)
        col = random.randint(0, 9)
        location = [col * 64 + 32, row * 64 + 32 + 640]
        if not (location in locations):
            locations.append(location)
            type = random.choice(["tree", "flag"])
            if type == "tree": img = "skier_tree.png"
            elif type == "flag": img = "skier_flag.png"
            obstacle = ObstacleClass(img, location, type)
            obstacles.add(obstacle)

def animate():
    screen.fill([255, 255, 255])
    obstacles.draw(screen)
    pygame.display.flip()

pygame.init()
screen = pygame.display.set_mode([640,640])
clock = pygame.time.Clock()
speed = [0, 6]
obstacles = pygame.sprite.Group()
map_position = 0
create_map()

running = True
while running:
    clock.tick(30)
    for event in pygame.event.get():
        if event.type == pygame.QUIT: running = False
    map_position += speed[1]

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

    obstacles.update()
    animate()

pygame.quit()
```

10 препятствий на экран

Создаем один «экран» с препятствиями размером 640 x 640

Предотвращаем попадание двух препятствий в одну точку

Перерисовываем все

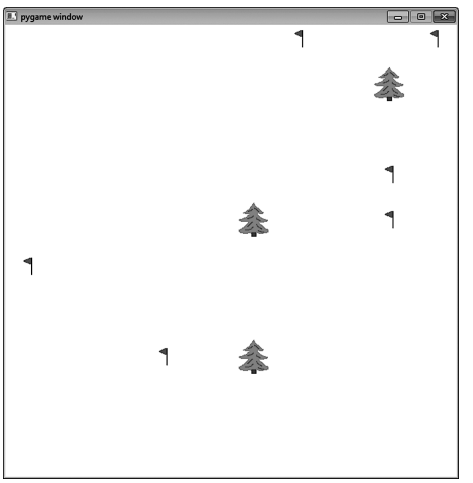
Инициализируем все

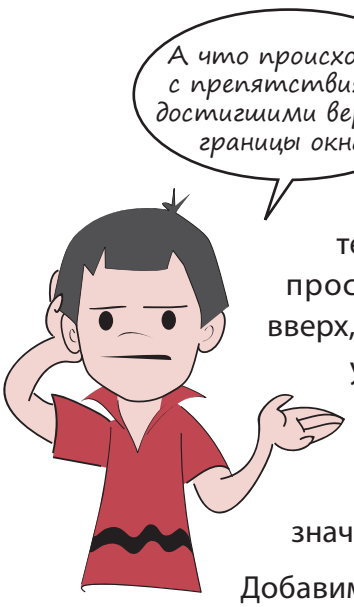
Основной цикл

Создаем за нижним краем окна новый блок препятствий

Следим за степенью прокрутки препятствий

Запустив код из листинга 25.2, вы увидите перемещающиеся вверх по экрану деревья и флаги:





Это хороший вопрос, Картер. В текущей версии кода они просто продолжают подниматься вверх, а их координаты *y* все больше уходят в отрицательные значения. Чем дольше длится игра, тем больше спрайтов с препятствиями мы создаем. В какой-то момент это может стать причиной замедления работы программы по причине нехватки памяти. А это значит, что нужно сделать небольшую уборку.

Добавим в метод `update()` проверку на выход препятствия за границу экрана. В случае положительного результата этой проверки препятствие будет удаляться. В Pygame для этой цели используется метод `kill()`. Вот измененный код метода `update()`:

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

Проверяем, не вышел ли спрайт за границу экрана

Избавляемся от спрайта

Теперь все готово для объединения карты препятствий и лыжника.

- Нам потребуются классы `SkierClass` и `ObstacleClass`.
- Функция `animate()` должна будет рисовать как лыжника, так и препятствия.
- Код инициализации обязан создавать лыжника и начальную карту препятствий.
- В основной цикл войдут как обработка ключевых событий лыжника, так и создание новых блоков препятствий.

По сути, листинг 25.3 — объединение листингов 25.1 и 25.2.

Листинг 25.3. Лыжник и препятствия

```
import pygame, sys, random

skier_images = ["skier_down.png", "skier_right1.png", "skier_right2.png",
                "skier_left2.png", "skier_left1.png"]

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

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

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

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

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

Код лыжника

Код препятствия

```
def create_map():
    global obstacles
    locations = []
    for i in range(10):
        row = random.randint(0, 9)
        col = random.randint(0, 9)
        location = [col * 64 + 32, row * 64 + 32 + 640]
        if not (location in locations):
            locations.append(location)
            type = random.choice(["tree", "flag"])
            if type == "tree": img = "skier_tree.png"
            elif type == "flag": img = "skier_flag.png"
            obstacle = ObstacleClass(img, location, type)
            obstacles.add(obstacle)
```

Код препятствия

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

Перерисовка лыжника и препятствий

```
pygame.init()
screen = pygame.display.set_mode([640,640])
clock = pygame.time.Clock()
points = 0
speed = [0, 6]
skier = SkierClass()
obstacles = pygame.sprite.Group()
create_map()
map_position = 0
```

Создаем лыжника

Создаем препятствие

Инициализируем все

```
running = True
while running:
    clock.tick(30)
    for event in pygame.event.get():
        if event.type == pygame.QUIT: running = False

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                speed = skier.turn(-1)
            elif event.key == pygame.K_RIGHT:
                speed = skier.turn(1)
    skier.move(speed)

    map_position += speed[1]
```

Основной цикл

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

    obstacles.update()
    animate()

pygame.quit()
```

Основной цикл

Запустив код из листинга 25.3, вы сможете управлять спуском лыжника с холма на фоне перемещающихся вверх препятствий. При этом скорость лыжника как при движении вправо-влево, так и при спуске строго вниз зависит от того, как он повернут. Результат практически достигнут.

Осталось добавить еще пару фрагментов:

- распознавание типа препятствия, с которым произошло столкновение;
- подсчет и вывод очков.

Распознавание столкновений мы рассматривали в главе 16. В коде из листинга 25.3 спрайты препятствий уже собраны в группу, поэтому можно воспользоваться функцией `spritecollide()` для определения момента столкновения. После этого останется определить тип участвующего в нем спрайта и выбрать вариант действий:

- при столкновении с деревом картинка меняется на изображение упавшего лыжника, а из общего счета вычитается 100:

- при столкновении с флажком к счету добавляется 10 очков, а флаг исчезает.

Реализующий это код следует поместить в основной цикл:

```
hit = pygame.sprite.spritecollide(skier, obstacles, False)
    if hit:
        if hit[0].type == "tree" and not hit[0].passed:
            points = points - 100

            skier.image = pygame.image.load("skier_crash.png")
            animate()
            pygame.time.delay(1000)
            skier.image = pygame.image.load("skier_down.png")
            skier.angle = 0
            speed = [0, 6]
            hit[0].passed = True

        elif hit[0].type == "flag" and not hit[0].passed:
            points += 10
            hit[0].kill()
```

Распознаем столкновения

Удар о дерево

В течение секунды демонстрируем «упавшего» лыжника

Продолжаем спуск

Запоминаем, что с этим деревом уже было столкновение

Столкновение с флагом

Удаляем флаг

Переменная `hit` указывает, с каким именно спрайтом столкнулся спрайт лыжника. Она представляет собой список, содержащий всего один элемент, так как в каждый момент времени лыжник может натолкнуться только на одно препятствие. Фактически, лыжник сталкивается с препятствием `hit[0]`.

Переменная `passed` указывает, что произошел наезд на дерево. Она гарантирует, что после того как спуск продолжится, лыжник не столкнется снова с тем же самым деревом. Теперь нужно показать набранные очки. Нам хватит трех строк кода. В разделе инициализации создадим объект `font`, представляющий собой экземпляр Pygame-класса `Font`:

```
font = pygame.font.Font(None, 50)
```

В основном цикле визуализируем объект `font` с текстом, задающим новое количество очков:

```
score_text = font.render("Score: " +str(points), 1, (0, 0, 0))
```

А в функцию `animate()` вставим вывод очков в верхнем левом углу окна:

```
screen.blit(score_text, [10, 10])
```

На этом все. Объединив все части, вы получите код из представленного в главе 10 листинга 10.1. Просто теперь вы лучше его понимаете. Это поможет вам при разработке и проектировании собственных игр.

Что мы узнали

В этой главе мы узнали:

- как функционируют все части программы Skier;
- как создать прокручивающийся фон.

Эксперименты

1. Отредактируйте игру Skier, сделав ее более сложной. Для этого можно:
   - с течением времени увеличивать скорость движения лыжника;
   - по мере спуска увеличивать количество деревьев в сцене;
   - добавить «лед», усложняющий поворот.
2. В программе SkiFree, которая послужила прототипом нашей игры Skier, действовал случайным образом появляющийся и преследующий лыжника снежный человек. Если вы хотите по-настоящему испытать свои силы, добавьте аналогичный элемент в нашу игру. Вам нужно будет найти или самостоятельно создать подходящий спрайт и отредактировать код, добавив туда нужное поведение.

ГЛАВА 26

Искусственный интеллект

В этой книге мы рассматривали процесс создания игр. Но есть одна вещь, которой мы пока не касались — игровой искусственный интеллект (Artificial Intelligence, AI). Практически все игры, начиная с вышедшей еще в 1980 году игры Pac-Man, обладают своего рода разумом, позволяющим компьютеру играть против человека, и в этой главе мы поговорим о том, как он создается.

Описание игры Python Battle

Искусственный интеллект, который мы будем создавать, является частью игры Python Battle (Битва питонов). Это простая игра с простыми правилами. На каждом шаге можно продвинуться вперед, развернуться или атаковать. Атакуемый робот теряет одно очко «здоровья». Робот, потерявший все очки, проигрывает. Атаковать можно только клетку, расположенную в непосредственной близости.

Но наша игра будет иметь одну особенность: в нее играют роботы. Вы должны написать стратегию, или искусственный интеллект, их перемещений и запустить игру, чтобы посмотреть, как один робот борется с другим. Разумеется, как код искусственного интеллекта, так и сама игра Python Battle будут написаны на языке Python.

Как обычно, если вы пользовались прилагаемой к книге программой установки, игра Python Battle уже имеется на жестком диске вашего компьютера. Я добавил туда три AI-сценария: сценарий CircleAI рисует окружности вокруг игрового поля, сценарий RandomAI перемещает и поворачивает робота случайным образом, сценарий NullAI ничего не делает. Заставьте их бороться друг с другом и посмотрите, кто выиграет.

Запуск игры Python Battle

Выполните следующие действия.

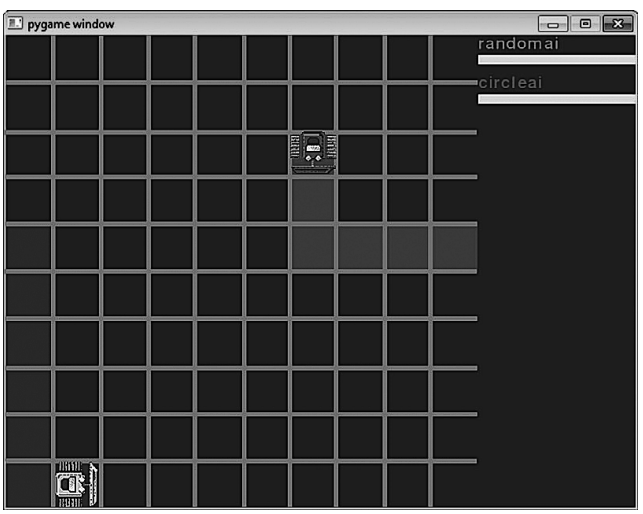
1. Убедитесь, что выбранные вами AI-сценарии находятся в той же папке, что и файл *PythonBattle.py*.
2. Запустите файл *PythonBattle.py*.

3. Вы увидите приглашение:

```
Enter red AI:
```

   Введите имя выбранного вами AI-сценария. Расширение *.py* опустите. Например, если вы хотите протестировать сценарий *CircleAI.py*, введите `circleai`.

4. Проделайте ту же операцию для второго AI-сценария.
5. Наблюдайте за боем и смотрите, кто выиграет!
6. В конце закройте Pygame-окно.



Правила игры

Итак, вы пронаблюдали несколько битв. Теперь имеет смысл разобраться, как все это работает. На каждом шаге робот может выполнить одно из шести действий:

- сместиться на одну клетку вперед;
- сместиться на одну клетку назад;
- повернуться налево;
- повернуться направо;
- атаковать расположенную перед ним клетку;
- остаться на месте.

Кроме того, робот может в любой момент рассмотреть игровое поле. Цель игры — 10 раз атаковать врага.

Надеюсь, наблюдая за битвой роботов CircleAI и RandomAI, вы обратили внимание на красные и синие квадраты? Дело в том, что поле, на которое шагнул робот, приобретает его цвет (красный или синий). Если после 1000 шагов ни один из роботов не атаковал противника или оба робота имеют одинаковые показатели здоровья, выигрыш определяется по максимальному количеству окрашенных клеток.

Создание робота

Запрограммируем двух роботов в соответствии со сценариями CircleAI и RandomAI. Первым делом, следует разработать стратегию. Начнем с простейшего варианта.

1. Если враг рядом, его нужно атаковать.
2. Если в направлении перемещения находится стена, следует повернуть в сторону.
3. В противном случае робот движется вперед.

Это не самая лучшая стратегия, но для победы над роботом CircleAI ее должно хватить. Если этого не произойдет, никто не мешает нам внести коррективы.

Приступим к коду. Создайте новый файл (я назвал его *better_than_circleai.py*) и введите туда следующие строки:

```
class AI:
    def __init__(self):
        pass
    def turn(self):
        pass
```

Это базовый код, с которого будут начинать все роботы. Функция `__init__()` вызывается в начале игры в момент создания AI. Функция `turn()` вызывается на каждом шаге и определяет действия робота. Классу следует присвоить имя `AI`, в противном случае игра Python Battle не сможет найти код, отвечающий за искусственный интеллект.

Реализуем задуманное. За перемещения робота будут отвечать следующие функции:

- `self.robot.lookInFront()`;
- `self.robot.turnRight()`;
- `self.robot.turnLeft()`;
- `self.robot.goForth()`;
- `self.robot.attack()`.

Названия этих методов строятся по схеме `self.robot.xx()`, так как часть `self` относится к искусственному интеллекту, а не к роботу, которым он управляет. При этом объект `AI` никуда не двигается и никого не атакует. Он отвечает за передачу роботу команд, а уже тот выполняет перемещения и атаки.

Приступим к методу `turn()`. Удалите часть `pass` и добавьте к своему коду такую строку:

```
self.robot.goForth()
```

Запустив работа, вы увидите, что пока он умеет только двигаться вперед. В результате он быстро упирается в край игрового поля. Наткнувшись на противника, робот останавливается. Ситуацию можно исправить при помощи функции `self.robot.lookInFront()`. Если перед роботом появляется противник, которого можно атаковать, он возвращает значение `"bot"`. Отредактируйте функцию `turn()` следующим образом:

```
if self.robot.lookInFront() == "bot":
    self.robot.attack()
else:
    self.robot.goForth()
```

Как только на пути робота появится препятствие, он пойдет в атаку. Но после столкновения со стеной робот все равно остановится. В этой ситуации функция `self.robot.lookInFront()` возвращает значение `"wall"`. Добавьте в функцию `turn()` между элементами `self.robot.attack()` и `else` следующие строки:

```
elif self.robot.lookInFront() == "wall":
    self.robot.turnRight()
```

Теперь после запуска кода робот будет перемещаться кругами! Наткнувшись на стену, он поворачивает направо, дойдя до следующей стены, снова поворачивает направо и т. д. Для полного разворота после столкновения со стеной робот должен два раза повернуть направо. Это делается за два хода. И нужно, чтобы после первого хода робот «помнил», что он находится в процессе разворота на 180°. Для этого в класс `AI` можно добавить новое свойство (переменную) для слежения за выполняемыми действиями. Вставьте в функцию `__init__()` следующую строку:

```
self.currentlyDoing = "forward"
```

Это заставит робота начать с движения вперед. А в процессе разворота этой переменной присваивается значение `"turnRight"`, «напоминающее» роботу, что ему нужно еще раз повернуть направо. Окончательный код после редактирования функции `turn()` представлен в листинге 26.1.

Листинг 26.1. Искусственный интеллект робота

```
class AI:
    def __init__(self):
        self.currentlyDoing = "forward"
    def turn(self):
        if self.robot.lookInFront() == "bot":
            self.robot.attack()
        elif self.robot.lookInFront() == "wall":
            self.robot.turnRight()
            self.currentlyDoing = "turnRight"
        elif self.currentlyDoing == "turnRight":
            self.robot.turnRight()
            self.currentlyDoing = "forward"
        else:
            self.robot.goForth()
```

Запустив этот код, вы обнаружите, что искусственный интеллект обладает рядом недостатков. Внесем несколько изменений, которые позволят нашему роботу победить своего соперника CircleAI.

Более сложный робот

Наш первый робот был очень простым и не имел шансов в бою с CircleAI. Для победы нам требуется по-настоящему выигрышная стратегия. До этого момента наш робот ходил кругами, пока перед ним не оказывался враг. А нам нужно задействовать все команды. Кроме того, выигрышная стратегия требует более детальной проработки.

Вот ряд методов, которые мы опустили в предыдущем разделе. Они могут помочь нам в разработке выигрышной стратегии:

- `self.robot.goBack()` — позволяет роботу двигаться назад;
- `self.robot.checkSpace(space)` — позволяет проверять любой фрагмент игрового поля;
- `self.robot.checkSpace((3,3))` — сообщает нам, что находится в клетке (3,3). Если там пусто, возвращает значение `"blank"`. В противном случае возвращает значение `"bot"` (если в этой клетке находится враг), `"me"` (если там располагался сам робот) или `"wall"` (если указанная клетка выходит за границу игрового поля);
- `self.robot.locateEnemy()` — дает информацию о положении и направлении движения врага;
- `self.robot.position` — дает информацию о собственном положении;
- `self.robot.rotation` — дает информацию о своем направлении движения;
- `self.robot.calculateCoordinates(direction, distance, position)` — назначение этих функций мы рассмотрим позже. Сначала следует познакомиться с системой координат в игре Python Battle.

Система координат

Система координат в игре Python Battle использует диапазон от (1,1) до (10,10). Она начинается в левом верхнем углу, как это принято в Pygame. Во всех направлениях игровое поле окружают стены. Данные о позиции робота в этой системе координат дает функция `self.robot.position`.

направления

Направления хранятся в виде чисел от 0 до 3, где 0 соответствует верху (север), 1 указывает вправо (восток), 2 — вниз (юг), а 3 — влево (запад). При повороте робота вправо к значению направления прибавляется единица. При повороте влево единица, соответственно, отнимается. Все очень просто. Определить ориентацию робота можно при помощи функции `self.robot.rotation`.

функция calculateCoordinates()

Функция `calculateCoordinates()` имеет три параметра: `distance` (расстояние), `direction` (направление) и `position` (текущее положение). Она определяет координату

в пространстве как `distance` квадратов от точки `position` в направлении `direction`. К примеру, функция `calculateCoordinates(2,3,(5,5))` определяет координату, которая находится двумя квадратами левее (3 соответствует направлению «влево») квадрата с координатами (5, 5). Теперь запишем стратегию. Она будет прямолинейной:

1) двигаемся в направлении врага;
2) по возможности атакуем.

Начнем с базового кода предыдущей версии робота:

```
class AI:
    def __init__(self):
        pass
    def turn(self):
        if self.robot.lookInFront() == "bot":
            self.robot.attack()
```

Этот код реализует вторую часть нашей стратегии: «по возможности атакуем». Осталось написать код первой части. Добавьте к функции `turn()` такие строки:

```
else:
    self.goTowards(self.robot.locateEnemy()[0])
```

Мы вызываем метод `self.goTowards()` класса `AI` с положением врага в качестве аргумента. Метод `self.robot.locateEnemy()` возвращает список с положением и ориентацией врага. Но этот код не работает, пока мы не определим метод `self.goTowards()`.

```
def goTowards(self,enemyLocation):
    myLocation = self.robot.position
    delta = (enemyLocation[0]-myLocation[0],
             enemyLocation[1]-myLocation[1])
```

Определяем дельту или разницу между положением вашего робота и положением цели. Смотрим, в каком направлении следует повернуться, чтобы оказаться к врагу лицом:

```
if abs(delta[0]) > abs(delta[1]):
    if delta[0] < 0:
        targetOrientation = 3
    else:
        targetOrientation = 1
else:
    if delta[1] < 0:
        targetOrientation = 0
    else:
        targetOrientation = 2
```

Лицом влево

Лицом вправо

Лицом вверх

Лицом вниз

Теперь нужно двинуться в этом направлении. Если вы уже развернулись в нужную сторону, это реализуется просто:

```
if self.robot.rotation == targetOrientation:
    self.robot.goForth()
```

Иначе нужно определить, в какую сторону вам следует повернуться. Первым делом считаем, сколько поворотов налево требуется для ориентации в корректном направлении:

```
else:
    leftTurnsNeeded = (self.robot.rotation - targetOrientation) % 4
```

Затем приходит время собственно поворота. Если для корректной ориентации нам требуется более двух поворотов влево, имеет смысл один раз повернуться вправо:

```
if leftTurnsNeeded <= 2:
    self.robot.turnLeft()
else:
    self.robot.turnRight()
```

Полностью код нашего робота представлен в листинге 26.2.

Листинг 26.2. Более сложный робот

```
class AI:
    def __init__(self):
        pass
    def turn(self):
        if self.robot.lookInFront() == "bot":
            self.robot.attack()
        else:
            self.goTowards(self.robot.locateEnemy()[0])
    def goTowards(self,enemyLocation):
        myLocation = self.robot.position
        delta = (enemyLocation[0]-myLocation[0],
                 enemyLocation[1]-myLocation[1])
        if abs(delta[0]) > abs(delta[1]):
            if delta[0] < 0:
                targetOrientation = 3
            else:
                targetOrientation = 1
        else:
            if delta[1] < 0:
                targetOrientation = 0
            else:
                targetOrientation = 2
```

Лицом влево

Лицом вправо

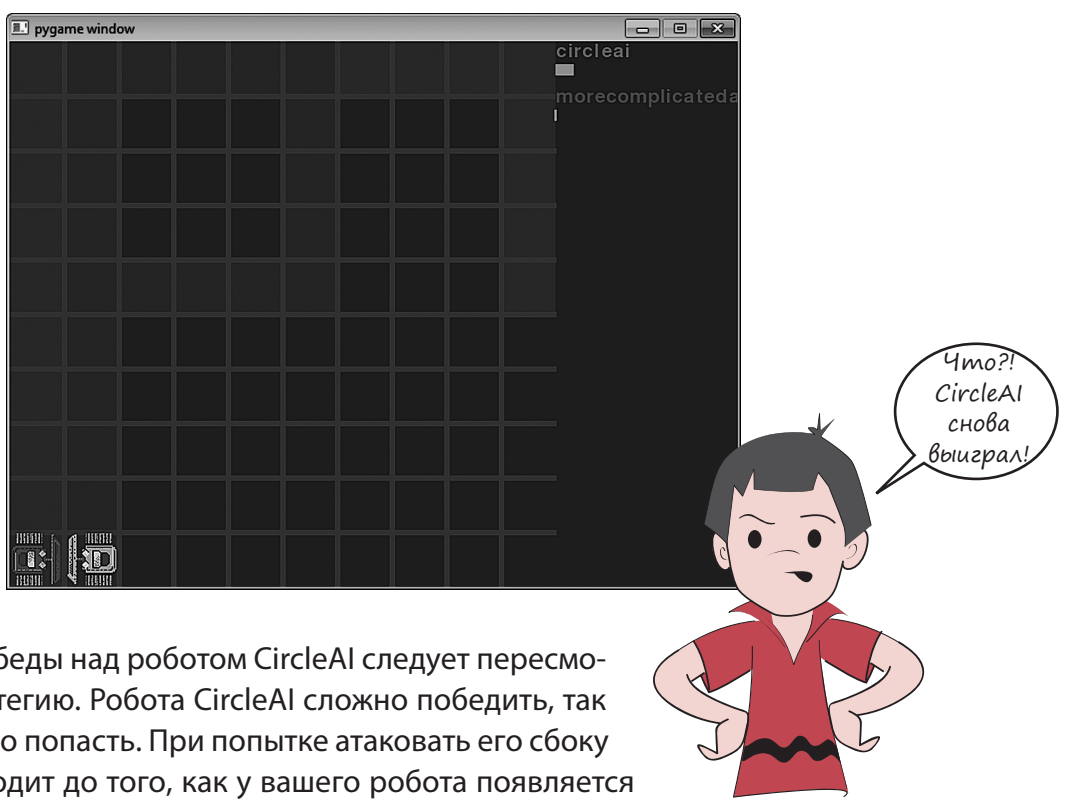
Лицом вверх

Лицом вниз

```
        if self.robot.rotation == targetOrientation:
            self.robot.goForth()
        else:
            leftTurnsNeeded = (self.robot.rotation - targetOrientation) % 4
            if leftTurnsNeeded <= 2:
                self.robot.turnLeft()
            else:
                self.robot.turnRight()
```

Попробуем выступить против робота CircleAI. Я уверен, что все наши усилия будут вознаграждены, и победа останется за нами! Сохраните этот искусственный интеллект под именем *morecomplicatedai.py* и снова запустите игру Python Battle:

```
>>>
Введите красный AI: circleai
Введите синий AI: morecomplicatedai
.
.
.
Красный выигрывает с показателем здоровья 10!
```



Полагаю, для победы над роботом CircleAI следует пересмотреть нашу стратегию. Робота CircleAI сложно победить, так как в него сложно попасть. При попытке атаковать его сбоку или сзади он уходит до того, как у вашего робота появляется

возможность второй атаки. А так как он двигается вдоль стены, его можно атаковать только с одной стороны. Попытка атаковать его спереди с большой вероятностью приведет к тому, что он нанесет удар первым и в результате выиграет. Несмотря на далекую от совершенства стратегию, победить робота CircleAI крайне сложно.

Вероятно, существует возможность разработать искусственный интеллект, способный постоянно выигрывать у робота CircleAI. Но это в случае, когда вы заранее знаете, что бороться предстоит с CircleAI. Если же вы не в курсе используемой соперником стратегии, задача усложняется еще больше!

Что мы узнали

В этой главе мы узнали:

- как в играх используют искусственный интеллект для создания противников, обладающих интеллектом;
- как написать собственный искусственный интеллект в качестве части игры Python Battle.

Эксперименты

Пересмотрите мою стратегию и попытайтесь создать робота, способного победить CircleAI.

Ответы на задания

Здесь вы найдете ответы на вопросы, которые задавались в разделах «Проверь себя» и «Эксперименты» в конце каждой главы. Разумеется, в некоторых случаях может быть несколько корректных решений, особенно при создании кода, но вы можете использовать наши ответы для проверки правильности своих рассуждений.

ПРОВЕРЬ СЕБЯ

1. В операционной системе Windows среда IDLE запускается через стартовое меню, в котором нужно найти пункт *Python 2.7*, а затем — *IDLE* (*Python GUI*). В операционной системе Mac OS X нужно щелкнуть по значку *IDLE* на панели *Dock*, если приложение было добавлено на эту панель. Альтернативным способом запуска является двойной щелчок по строке *IDLE.app* в папке *Python 2.7*, вложенной в папку *Applications*. В операционной системе Linux все зависит от установленного у вас диспетчера окон, но в общем случае вы можете воспользоваться меню *Applications* или *Programs*. Следует заметить, что многие пользователи Linux не работают с IDLE, а запускают интерпретатор Python с терминала или открывают для работы с кодом редактор *vi* или *emacs*.
2. Команда `print` отображает в окне вывода (в нашем первом примере это было IDLE-окно) указанный пользователем текст.
3. В Python символом умножения является * (звездочка).
4. При запуске программы среда IDLE выводит на экран такую строку:

```
>>> ====================== RESTART ======================
```

5. Альтернативой словосочетанию «запустить программу» является «выполнить программу».

ЭКСПЕРИМЕНТЫ

1. Количество минут в неделе можно вычислить так:

```
>>> print 7 * 24 * 60
```

(7 дней в неделю, 24 часа в день, 60 минут в час). Вы должны получить ответ 10 080.

2. Программа для вывода на экран трех строк (имени, даты рождения и любимого цвета) выглядит примерно так:

```
print "Меня зовут Уоррен Сэнд."
print "Я родился 1 января 1970."
print "Мой любимый цвет голубой."
```

ПРОВЕРЬ СЕБЯ

1. Интерпретатор Python воспринимает переменную как строку, если заключить ее в кавычки.
2. Ответ на вопрос «Можно ли изменить значение, присвоенное созданной вами переменной?» зависит от того, что вы понимаете под словом «изменение». Например:

```
myAge = 10
```

Если сначала вы выполните эту операцию, ничто не мешает вам затем сделать так:

```
myAge = 11
```

То есть вы изменили значение, назначенное переменной myAge, перенеся ярлык myAge на другой элемент: со значения 10 на значение 11. Но вы не превратили 10 в 11, и вместо: «Вы изменили значение переменной» корректнее будет сказать: «Вы присвоили переменной другое значение».

3. Имена TEACHER и TEACHEr относятся к двум разным переменным, потому что имена переменных чувствительны к регистру букв.
4. Для интерпретатора Python имена 'Blah' и "Blah" одинаковы. Оба элемента являются строками, а интерпретатор Python разрешает использовать кавычки любого типа. Главное — чтобы открывающая кавычка принадлежала к тому же типу, что и закрывающая.
5. Для интерпретатора Python '4' и 4 — это не одно и то же. Первый элемент является строкой (хотя и состоящей всего из одного символа), так как помещен в кавычки. Второй же элемент представляет собой число.
6. Ответ: 2Teacher. Имена переменных в Python не могут начинаться с числа.
7. "10" — это строка, так как символы заключены в кавычки.

ЭКСПЕРИМЕНТЫ

1. В интерактивном режиме код будет выглядеть примерно так:

```
>>> temperature = 25
>>> print temperature
25
```

2. Можно написать такой код:

```
>>> temperature = 40
>>> print temperature
40
```

Или такой:

```
>>> temperature = temperature + 15
>>> print temperature
40
```

3. Можно написать такой код:

```
>>> firstName = "Fred"
>>> print firstName
Fred
```

4. С переменными программа, вычисляющая количество минут в дне, приобретет такой вид:

```
>>> DaysPerWeek = 7
>>> HoursPerDay = 24
>>> MinutesPerHour = 60
>>> print DaysPerWeek * HoursPerDay * MinutesPerHour
10080
```

5. Узнать, сколько минут было бы в неделе с 26-часовым днем, позволяет следующий код:

```
>>> HoursPerDay = 26
>>> print DaysPerWeek * HoursPerDay * MinutesPerHour
10920
```

Проверь себя

1. Для умножения в Python используется символ * (звездочка).
2. Интерпретатор Python ответит так: 8 / 3 = 2. Поскольку числа 8 и 3 являются целыми, Python 2 округляет в меньшую сторону до ближайшего целого. (Учтите, что в интерпретаторе Python 3 вы получите 2.66666666667, так как в отличие от Python 2 он по умолчанию не выполняет деление нацело.)
3. Для получения остатка пользуйтесь оператором %. То есть 8 % 3.
4. Чтобы получить в результате операции 8 / 3 десятичное число, преобразуйте в десятичное число делимое или делитель: 8.0 / 3 или 8 / 3.0. (Напоминаю, что Python 3 выполняет эту операцию автоматически.)
5. В Python запись 6 * 6 * 6 * 6 можно переписать так: 6 ** 4.
6. Число 17 000 000 в экспоненциальном представлении записывается так: 1.7e7.
7. Экспоненциальное представление 4.56e-5 соответствует значению 0.0000456 в обычном представлении.

Эксперименты

1. Эти варианты решения не единственные, вы можете найти свое собственное решение.
   1) Вычисление суммы, которую каждый должен оплатить в ресторане:

```
>>> print 35.27 * 1.15 / 3
>>> 13.5201666667
```

   После округления получим, что каждый должен заплатить $13.52.

   2) Вычисление площади и периметра прямоугольника:

```
length = 16.7
width = 12.5
Perimeter = 2 * length + 2 * width
Area = length * width
print 'Длина = ', length, ' Ширина = ', width
print "Площадь = ", Area
print "Периметр = ", Perimeter
```

   Результат работы программы:

```
Длина = 16.7 Ширина = 12.5
Площадь = 208.75
Периметр = 58.4
```

2. Программа преобразования температуры из градусов Фаренгейта в градусы Цельсия:

```
fahrenheit = 75
celsius = 5.0/9 * (fahrenheit - 32)
print "Fahrenheit = ", fahrenheit, "Celsius =", celsius
```

3. Вычисление времени проезда определенного расстояния с заданной скоростью:

```
distance = 200
speed = 80.0
time = distance / speed
print "время =", time
```

   (Напоминаю, что одно из чисел следует сделать вещественным, иначе вы получите результат, округленный в меньшую сторону до целого.)

4

Проверь себя

1. Функция `int()` всегда округляет результат в меньшую сторону (до следующего целого числа в сторону нуля по числовой оси).
2. Будет ли работать в программе преобразования температуры следующий код?

```
cel = float(5 / 9 * (fahr - 32))
cel = 5 / 9 * float(fahr - 32)
```

Попробуйте его запустить и посмотрите, что получится:

```
>>> fahr = 75
>>> cel = float(5 / 9 * (fahr - 32))
>>> print cel
0.0
```

Почему код не сработал? Помните, что действия в скобках всегда выполняются в первую очередь. То есть происходит следующее:

```
75 - 32 = 43
```

Затем:

```
5 / 9 = 0
```

Так как операции выполняются слева направо, сначала вычисляется `5 / 9`. Но числа 5 и 9 являются целыми, поэтому Python выполняет деление нацело и округляет результат в меньшую сторону. В данном случае ответ меньше 1, и после округления мы получим 0. Соответственно:

```
0 * 43 = 0
```

Затем:

```
float(0) = 0.0
```

К моменту передачи значения функции `float()` уже слишком поздно — ответ уже равен 0! То же самое происходит со вторым уравнением.

3. «Обмануть» функцию `int()`, заставив ее выполнить округление не в меньшую, а в большую сторону, можно, прибавив к передаваемому ей числу значение 0.5. Вот пример (в интерактивном режиме):

```
>>> a = 13.2
>>> roundoff = int(a + 0.5)
>>> roundoff
13
>>> b = 13.7
>>> roundoff = int(b + 0.5)
>>> b
14
```

Если исходное число меньше 13.5, функция `int()` получит число меньше 14 и округлит его до 13. Если же исходное число равно 13.5 или больше этого значения, функция `int()` получит число, равное или превышающее 14, которое округлит в меньшую сторону до 14.

Эксперименты

1. Преобразовать строку в десятичное число позволяет функция `float()`:

```
>>> a = float('12.34')
>>> print a
12.34
```

Но как узнать, что результат является числом, а не строкой? Проверим его тип:

```
>>> type(a)
<type 'float'>
```

2. Функция `int()` позволяет преобразовать десятичное число в целое:

```
>>> print int(56.78)
56
```

Ответ был округлен в меньшую сторону.

3. Функция `int()` позволяет преобразовать строку в целое число:

```
>>> a = int('75')
>>> print a
75
>>> type(a)
<type 'int'>
```

Проверь себя

1. Имеется фрагмент кода:

```
answer = raw_input()
```

Ввод в этом фрагменте кода числа 12 даст строковую переменную `answer`, потому что функция `raw_input()` всегда возвращает строку.

Запустите эту короткую программу и убедитесь сами:

```
print "введите число: ",
answer = raw_input()
print type(answer)
>>> ============== RESTART ==============
>>>
введите число: 12
<type 'str'>
>>>
```

Итак, функция `raw_input()` возвращает строку.

Напоминаю, что в интерпретаторе Python 3 функция `raw_input()` называется просто `input()`.

2. Чтобы заставить функцию `raw_input()` вывести на экран приглашение для ввода данных, поместите в скобки текст в кавычках, например:

```
answer = raw_input("Введите число: ")
```

3. Если вам нужно ввести при помощи функции `raw_input()` целое число, воспользуйтесь функцией `int()` для преобразования возвращаемой функцией `raw_input()` строки в целое. Это можно сделать за два шага:

```
something = raw_input()
answer = int(нечто)
```

Или за один:

```
answer = int(raw_input())
```

4. Ответ на этот вопрос аналогичен предыдущему, просто в данном случае вместо функции `int()` нужно взять функцию `float()`.

ЭКСПЕРИМЕНТЫ

1. В интерактивном режиме ваши инструкции должны выглядеть примерно так:

```
>>> first = 'Уоррен'
>>> last = 'Сэнд'
>>> print first + last
УорренСэнд
```

Ой! Пробел отсутствует. Можно добавить пробел после имени:

```
>>> first = 'Уоррен '
```

Можно также задействовать такой код:

```
>>> print first + ' ' + last
Уоррен Сэнд
```

Еще можно воспользоваться запятой:

```
>>> first = 'Уоррен'
>>> last = 'Сэнд'
>>> print first, last
Уоррен Сэнд
```

2. Пример программы:

```
first = raw_input('введите ваше имя: ')
last = raw_input('введите вашу фамилию: ')
print 'Привет,', first, last, 'как твои дела?'
```

3. Пример программы:

```
length = float(raw_input ('длина комнаты в сантиметрах: '))
width = float(raw_input ('ширина комнаты в сантиметрах: '))
area = length * width
print 'Площадь', area, 'квадратных сантиметров.'
```

4. В программу можно добавить несколько строк из #С:

```
length = float(raw_input ('длина комнаты в сантиметрах: '))
width = float(raw_input ('ширина комнаты в сантиметрах: '))
cost_per_metre = float(raw_input ('стоимость за квадратный метр: '))
area_cm = length * width
area_metre = area_cm / 10000.0
total_cost = area_metre * cost_per_metre
print 'Площадь равна', area_cm, 'квадратных сантиметров.'
print 'Это', area_metre, 'квадратных метров.'
print 'Что будет стоить', total_cost
```

5. Пример программы:

```
kop50 = int(raw_input("Сколько монет по 50 копеек? "))
kop10 = int(raw_input("Сколько монет по 10 копеек? "))
kop5 = int(raw_input("Сколько монет по 5 копеек? "))
kop1 = int(raw_input("Сколько монет по 1 копейке? "))
total = 0.50 * kop50 + 0.10 * kop10 + 0.05 * kop5 + 0.01 * kop1
print "Всего у вас: ", total
```

6

ПРОВЕРЬ СЕБЯ

1. Для вызова в модуле EasyGui информационного окна пользуйтесь функцией `msgbox()`, например:

```
easygui.msgbox("Это ответ!")
```

2. Для ввода строки при работе с модулем EasyGui используйте элемент `enterbox`.
3. Для ввода целого числа используйте элемент `enterbox` (он возвращает строку) и преобразуйте результат в целое. Также можно задействовать элемент `integerbox`.
4. Для ввода числа с плавающей точкой можно воспользоваться элементом `enterbox`, а затем функцией `float()`.
5. Значение, предлагаемое по умолчанию, — это своего рода «автоматический ответ». Его можно использовать, например, в следующей ситуации. В программе, в поля которой все ученики вашего класса вводят свои имена и адреса, в качестве значения, предлагаемого по умолчанию, можно указать город, где вы живете. В результате вводить название города придется только иногородним ученикам.

ЭКСПЕРИМЕНТЫ

1. Вот пример программы преобразования температуры, написанной при помощи модуля EasyGui:

```
# tempgui1.py
# EasyGui-версия программы преобразования температуры
# преобразует градусы Фаренгейта в градусы Цельсия
import easygui

easygui.msgbox('Эта программа преобразует градусы Фаренгейта в Цельсия')
temperature = easygui.enterbox('Введите градусы Фаренгейта:')
Fahr = float(temperature)
Cel = (Fahr - 32) * 5.0 / 9
easygui.msgbox('Это ' + str(Cel) + ' градусов Цельсия.')
```

2. Вот программа, спрашивающая ваше имя и части вашего адреса, а затем показывающая адрес целиком. В данном случае вам потребуется принудительный перенос строк. Для этого используется символ `\n`. Подробно он будет рассматриваться в главе 21, а пока небольшая демонстрация:

```
# address.py
# Вводятся части адреса и отображается адрес целиком
import easygui

name = easygui.enterbox("Как тебя зовут?")
addr = easygui.enterbox("На какой улице ты живешь?")
house = easygui.enterbox("В каком доме ты живешь?")
flat = easygui.enterbox("В какой квартире ты живешь?")
city = easygui.enterbox("В каком городе ты живешь?")
country = easygui.enterbox("В какой стране ты живешь?")
code = easygui.enterbox("Укажи свой почтовый индекс?")

whole_addr = code + ", " + country + ", " + city + "\n"
    + addr + ", " + house + ", " + flat + "\n" + name

easygui.msgbox(whole_addr, "Вот твой адрес:")
```

ПРОВЕРЬ СЕБЯ

1. Запустив программу, мы получим:

```
Меньше 20
```

Так как переменная `my_number` меньше 20, инструкция `if` вернет значение `true`, поэтому будет выполнен следующий за ней блок (в данном случае одна строка).

2. Запустив программу, мы получим

```
20 или больше
```

Так как переменная `my_number` больше 20, инструкция `if` вернет значение `false`, а код в блоке инструкции `if` выполняться не будет. Вместо него будет выполнен код блока `else`.

3. Чтобы проверить, что число больше 30, но меньше или равно 40, можно написать такой код:

```
if number > 30 and number <= 40:
    print 'Число лежит в диапазоне от 30 до 40'
```

Можно сделать и так:

```
if 30 < number <= 40:
    print "Число лежит в диапазоне от 30 до 40"
```

4. Проверка ввода буквы Q в верхнем или нижнем регистре может выглядеть так:

```
if answer == 'Q' or answer == 'q':
    print "вы ввели букву 'Q' "
```

Обратите внимание, что выводимая строка заключена в двойные кавычки, при этом внутри нее фигурируют одинарные кавычки, в которые заключена буква Q. Именно так выводится знак кавычек: берутся еще одни кавычки, охватывающие строку целиком.

Эксперименты

1. Вот вариант программы:

```
# программа вычисления скидки
# 10% при цене меньше или равной $10, 20% при цене больше $10
item_price = float(raw_input ('укажите цену товара: '))
if item_price <= 10.0:
    discount = item_price * 0.10
else:
    discount = item_price * 0.20
final_price = item_price - discount
print 'Вы получили скидку ', discount, 'итоговая цена', final_price
```

В данной программе ответ не округляется до двух знаков после десятичной точки (до центов) и не отображается знак доллара.

2. Вот вариант программы:

```
# программа проверки возраста и пола футбольных игроков
# принимаются девочки от 10 до 12 лет
gender = raw_input("Вы мальчик или девочка? ('m' or 'f') ")
if gender == 'f':
    age = int(raw_input('Сколько вам лет? '))
    if age >= 10 and age <= 12:
        print 'Вы можете играть в этой команде'
    else:
        print 'Вы не прошли по возрасту.'
else:
    print 'В команду принимаются только девочки.'
```

3. Вот вариант программы:

```
# программа проверяет, нужен ли вам бензин.
# Следующая заправка через 200 км
tank_size = int(raw_input('Размер бака (в литрах)? '))
full = int(raw_input ('Заполненность бака в процентах?'))
mileage = int(raw_input ('Расход топлива (км на литр)? '))
range = tank_size * (full / 100.0) * mileage
print 'Вы можете проехать еще', range, 'км.'
print 'Следующая заправка через 200 км.'
```

```
if range <= 200:
    print 'ЗАПРАВЬТЕСЬ СЕЙЧАС!'
else:
    print 'Можно подождать следующей заправки.'
```

Чтобы добавить погрешность в 5 литров, возьмите строку:

```
range = tank_size * (full / 100.0) * mileage
```

Измените ее следующим образом:

```
range = (tank_size - 5) * (full / 100.0) * mileage
```

4. Вот простая программа проверки пароля

```
password = "bigsecret"
guess = raw_input("Введите пароль: ")
if guess == password:
    print "Пароль принят. Добро пожаловать!"
    # здесь должен быть остальной код вашей программы
else:
    print "Пароль неверный. До свидания!"
```

ПРОВЕРЬ СЕБЯ

1. Цикл запустится пять раз.
2. Цикл запустится три раза со следующими значениями: `i = 1`, `i = 3`, `i = 5`.
3. Функция `range(1, 8)` даст вам `[1, 2, 3, 4, 5, 6, 7]`.
4. Функция `range(8)` даст вам `[0, 1, 2, 3, 4, 5, 6, 7]`.
5. Функция `range(2, 9, 2)` даст вам `[2, 4, 6, 8]`.
6. Функция `range (10, 0, -2)` даст вам `[10, 8, 6, 4, 2]`.
7. Для остановки текущей итерации цикла и перехода к следующей используется ключевое слово `continue`.
8. Цикл `while` завершается, если при проверке условия продолжения возвращается значение `false`.

ЭКСПЕРИМЕНТЫ

1. Вот программа, с помощью цикла `for` выводящая на экран таблицу умножения для выбранного пользователем числа:

```
# программа вывода таблицы умножения до 10
number = int(raw_input('Для какого числа нужна таблица умножения? '))
print 'Вот ваша таблица:'
for i in range(1, 11):
    print number, 'x', i, '=', number * i
```

2. Та же самая таблица умножения на основе цикла `while`:

```
# вывод таблицы умножения (с циклом while)
number = int(raw_input(' Для какого числа нужна таблица умножения? '))
print 'Вот ваша таблица:'
i = 1
while i <= 10:
    print number, 'умножить', i, '=', number * i
    i = i + 1
```

3. Вот таблица умножения в указанном пользователем диапазоне:

```
# программа вывода таблицы умножения
# ее размер задает пользователь
```

```
number = int(raw_input('Для какого числа нужна таблица умножения? '))
limit = int(raw_input('До какого множителя хотите дойти? '))
print 'Вот ваша таблица:'
for i in range(1, limit + 1):
    print number, 'умножить', i, '=', number * i
```

Обратите внимание, что в цикле `for` второй параметр функции `range()` является не числом, а переменной. Более подробно мы говорили об этом в главе 11.

9

ЭКСПЕРИМЕНТЫ

Вот пример комментария, который я добавил к программе преобразования температуры:

```
# tempconv1.py
# Программа, преобразующая градусы Фаренгейта в градусы Цельсия
Fahr = 75
Cel = (Fahr - 32) * 5.0 / 9 #вещественное деление, не целое
print "Fahrenheit = ", Fahr, "Celsius = ", Cel
```

10

ЭКСПЕРИМЕНТЫ

Набрав и запустив программу, не забудьте скопировать в папку с программой все файлы с графикой.

11

ПРОВЕРЬ СЕБЯ

1. Для создания переменной цикла в Python достаточно вставить переменную в функцию `range()`:

```
for i in range(numberOfLoops)
```

Или так:

```
for i in range(1, someNumber)
```

2. Для создания вложенного цикла цикл следует поместить в тело другого цикла, например так:

```
for i in range(5):
    for j in range(8):
        print "Привет",
    print
```

Этот код выводит слово «Привет» в строке восемь раз (внутренний цикл), а затем создает пять таких строк (внешний цикл).

3. Будут выведены 15 звездочек.
4. Вот результат работы этого кода:

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

5. Для четырехуровневого дерева решений существует `2**4`, или `2*2*2*2`, возможных вариантов. Это 16 вариантов спуска вниз по дереву.

ЭКСПЕРИМЕНТЫ

1. Вот программа с таймером обратного отсчета, спрашивающая у пользователя, откуда следует начать отсчет:

```
# Таймер обратного отсчета спрашивает, откуда начать
import time
```

```
start = int(raw_input("Таймер обратного отсчета: Сколько секунд? ", ))
for i in range (start, 0, -1):
    print i
    time.sleep(1)
print "ПУСК!"
```

2. Предыдущая программа после добавления к выводимым числам звездочек:

```
# Таймер обратного отсчета спрашивает, откуда начать
# и выводит рядом с каждым числом звездочки
import time
start = int(raw_input("Таймер обратного отсчета: Сколько секунд? ", ))
for i in range (start, 0, -1):
    print i,
    for star in range(i):
        print '*',
    print
    time.sleep(1)
print "ПУСК!"
```

ПРОВЕРЬ СЕБЯ

1. За добавление элемента в список отвечает функция `append()`, `insert()` или `extend()`.
2. За удаление элемента из списка отвечают функции `remove()` и `pop()` или команда `del`.
3. Для получения отсортированной копии списка можно:
   - создать копию списка при помощи среза `new_list = my_list[:]` и выполнить сортировку нового списка: `new_list.sort()`;
   - воспользоваться функцией `sorted()`: `new_list = sorted(my_list)`.
4. Проверку на наличие в списке указанного значения выполняет ключевое слово `in`.
5. Поиск положения элемента в списке выполняет метод `index()`.
6. Кортеж представляет собой недоступный для редактирования список.
7. Список списков можно получить несколькими способами:
   - при помощи вложенных квадратных скобок:

```
>>> my_list = [[1, 2, 3], ['a', 'b', 'c'], ['red', 'green', 'blue']]
```

   - добавив список при помощи функции `append()`:

```
>>> my_list = []
>>> my_list.append([1, 2, 3])
>>> my_list.append(['a', 'b', 'c'])
>>> my_list.append(['red', 'green', 'blue'])
>>> print my_list
 [[1, 2, 3], ['a', 'b', 'c'], ['red', 'green', 'blue']]
```

   - создав и затем объединив отдельные списки:

```
>>> list1 = [1, 2, 3]
>>> list2 = ['a', 'b', 'c']
>>> list3 = ['red', 'green', 'blue']
>>> my_list = [list1, list2, list3]
>>> print my_list
 [[1, 2, 3], ['a', 'b', 'c'], ['red', 'green', 'blue']]
```

8. Для извлечения из списка списков отдельного элемента следует указать два его индекса:

```
my_list = [[1, 2, 3], ['a', 'b', 'c'], ['red', 'green', 'blue']]
my_color = my_list[2][1]
```

Эта переменная имеет значение `'green'`.

9. Словарь представляет собой коллекцию пар ключ–значение.
10. Для добавления в словарь элемента следует указать его ключ и значение:

```
phone_numbers['John'] = '555-1234'
```

11. Для поиска в словаре элемента по ключу применяется индекс:

```
print phone_numbers['John']
```

Эксперименты

1. Программа, требующая ввода пяти имен, помещающая их в список и выводящая результат на экран:

```
nameList = []
print "Введите 5 имен (после каждого нажимайте клавишу Enter):"
for i in range(5):
    name = raw_input()
    nameList.append(name)

print "Это имена:", nameList
```

2. Программа, выводящая исходный список и его отсортированную версию:

```
nameList = []
print "Введите 5 имен (после каждого нажимайте клавишу Enter):"
for i in range(5):
    name = raw_input()
    nameList.append(name)

print "Это имена:", nameList
print "Список после сортировки:", sorted(nameList)
```

3. Программа, выводящая на экран третье имя из списка:

```
nameList = []
print "Введите 5 имен (после каждого нажимайте клавишу Enter):"
for i in range(5):
    name = raw_input()
    nameList.append(name)

print "Третье введенное вами имя:", nameList[2]
```

4. Программа, позволяющая пользователю заменить в списке любое имя:

```
nameList = []
print "Введите 5 имен (после каждого нажимайте клавишу Enter):"
for i in range(5):
    name = raw_input()
    nameList.append(name)

print "Это имена:", nameList
print "Какое имя нужно заменить? (1-5):",
replace = int(raw_input())
new = raw_input("Новое имя: ")
nameList[replace - 1] = new
print "Это имена:", nameList
```

5. Программа, позволяющая пользователю создать словарь со словами и их определениями:

```
user_dictionary = {}
while 1:
    command = raw_input("'a' добавление слова,
                        'l' поиск слова,
                        'q' выход ")

    if command == "a":
        word = raw_input("Введите слово: ")
        definition = raw_input("Введите определение: ")
        user_dictionary[word] = definition
        print "Слово добавлено!"

    elif command == "l":
        word = raw_input("Введите слово: ")
        if word in user_dictionary.keys():
            print user_dictionary[word]
        else:
            print "Это слово пока отсутствует в словаре."

    elif command == 'q':
        break
```


ПРОВЕРЬ СЕБЯ

1. Для создания функции применяется ключевое слово `def`.
2. Для вызова функции достаточно указать ее имя, добавив следом скобки.
3. Передаваемый в функцию аргумент указывается в скобках в момент вызова функции.
4. Функция может иметь неограниченное количество аргументов.
5. Функция возвращает информацию в вызывающий код при помощи ключевого слова `return`.
6. После завершения работы функции локальная переменная уничтожается.

ЭКСПЕРИМЕНТЫ

1. Эта функция представляет собой набор инструкций `print`:

```
def printMyNameBig():
    print "  CCCC       A      RRRRR  TTTTTTT EEEEEE RRRRR "
    print " C    C     A A     R    R    T    E      R    R "
    print "C          A   A    R    R    T    EEEE   R    R "
    print "C         AAAAAAA   RRRRR     T    E      RRRRR "
    print " C    C  A       A  R    R    T    E      R    R "
    print "  CCCC  A         A R     R   T    EEEEEE R     R"
```

   Вызывающая ее программа может выглядеть вот так:

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

2. Вот мой вариант программы, выводящей адрес с семью аргументами:

```
# определим функцию с семью аргументами
def printAddr(name, num, street, city, prov, pcode, country):
    print name
    print num,
    print street
    print city,
    if prov !="":
        print ", "+prov
```

```
        else:
            print ""
        print pcode
        print country
        print

    # вызов функции и передача в нее семи аргументов
    printAddr("Sam", "45", "Main St.", "Ottawa", "ON", "K2M 2E9", "Canada")
    printAddr("Jian", "64", "2nd Ave.", "Hong Kong", "", "235643", "China")
```

3. Ответа тут не будет, просто попробуйте проделать указанную операцию.
4. Функция сложения мелочи:

```
def addUpChange(kop50, kop10, kop5, kop1):
    total = 0.50 * kop50 + 0.10 * kop10 + 0.05 * kop5 + 0.01 * kop1
    return total
```

Вот пример вызывающей такую функцию программы:

```
kop50 = int(raw_input("монет по 50 копеек: "))
kop10 = int(raw_input("монет по 10 копеек: "))
kop5 = int(raw_input("монет по 5 копеек: "))
kop1 = int(raw_input("монет по 1 копейке: "))

total = addUpChange(kop50, kop10, kop5, kop1)

print "Всего у нас: ", total
```

14 ПРОВЕРЬ СЕБЯ

1. Новый тип объекта определяет ключевое слово `class`.
2. Атрибуты представляют собой известные вам элементы объекта. Это содержащиеся в объекте переменные.
3. Методами называются действия, которые можно проделывать с объектом. Это содержащиеся в объекте функции.
4. Класс представляет собой определение, или план, объекта. Превращая план в объект, вы получаете экземпляр.
5. Имя `self` обычно используется для ссылки на экземпляр в методе объекта.
6. Полиморфизм означает вероятность наличия двух и более методов с одинаковыми именами, связанных с разными объектами. Поведение методов в этом случае зависит от того, какому из объектов он принадлежит.
7. Наследованием называется способность объектов получать атрибуты и методы от своих «родителей». «Дочерний» класс (который также называют подклассом, или производным классом) получает от родителя все атрибуты и методы и при этом может иметь собственные атрибуты и методы, отсутствующие у родителя.

ЭКСПЕРИМЕНТЫ

1. Вот пример класса банковского счета:

```
class BankAccount:
    def __init__(self, acct_number, acct_name):
        self.acct_number = acct_number
        self.acct_name = acct_name
        self.balance = 0.0

    def displayBalance(self):
        print "На вашем счету:", self.balance
```

```
    def deposit(self, amount):
        self.balance = self.balance + amount
        print "Вы положили", amount
        print "Теперь на вашем счету:", self.balance

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance = self.balance - amount
            print "Вы сняли", amount
            print "Теперь на вашем счету:", self.balance
        else:
            print "Вы попытались снять", amount
            print "На вашем счету:", self.balance
            print "В операции отказано. Недостаточно средств."
```

А это код, позволяющий протестировать класс и убедиться, что все работает:

```
myAccount = BankAccount(234567, "Warren Sande")
print "Название счета:", myAccount.acct_name
print "Номер счета:", myAccount.acct_number
myAccount.displayBalance()

myAccount.deposit(34.52)
myAccount.withdraw(12.25)
myAccount.withdraw(30.18)
```

2. Для получения процентов от вклада нужно создать подкласс `BankAccount` и метод добавления процентов:

```
add interest:
class InterestAccount(BankAccount):
    def __init__(self, acct_number, acct_name, rate):
        BankAccount.__init__(self, acct_number, acct_name)
        self.rate = rate

    def addInterest (self):
        interest = self.balance * self.rate
        print "начисления по вкладу,", self.rate * 100, "процентов"
        self.deposit (interest)
```

Вот код для тестирования этого класса:

```
myAccount = InterestAccount(234567, "Warren Sande", 0.11)
print "Название счета:", myAccount.acct_name
print "Номер счета:", myAccount.acct_number
myAccount.displayBalance()
myAccount.deposit(34.52)
myAccount.addInterest()
```

Проверь себя

1. Преимущества, которые дает модуль:
   - можно написать код один раз и использовать его в разных программах;
   - можно пользоваться модулями, написанными другими;
   - уменьшается размер файлов с кодом, а значит, облегчается поиск отдельных фрагментов кода;
   - можно пользоваться только частями (модулями), которые нужны для решения конкретной задачи.
2. Модуль создается на языке Python и сохраняется в файле.
3. Модули вставляются в программу при помощи ключевого слова `import`.
4. Импорт модуля аналогичен импорту пространства имен.

5. Импортировать модуль `time` для получения доступа ко всем его именам можно так:

```
import time
```

Или так:

```
from time import *
```

ЭКСПЕРИМЕНТЫ

1. Чтобы создать модуль, поместите код функции вывода вашего имени большими буквами в файл, назвав его, к примеру, *bigname.py*. Импортировать этот модуль и вызвать функцию можно так:

```
import bigname
bigname.printMyNameBig()
```

Или так:

```
from bigname import *
printMyNameBig()
```

2. Для вставки функции `c_to_f()` в пространство имен основной программы можно поступить так:

```
from my_module import c_to_f
```

Или так:

```
from my_module import *
```

3. Короткая программа вывода пяти случайных целых чисел в диапазоне от 1 до 20:

```
import random
for i in range(5):
    print random.randint(1, 20)
```

4. Короткая программа, в течение 30 секунд каждые 3 секунды выводящая случайное десятичное число:

```
import random, time
for i in range(10):
    print random.random()
    time.sleep(3)
```

16

ПРОВЕРЬ СЕБЯ

1. RGB-значение [255, 255, 255] соответствует белому цвету.
2. RGB-значение [0, 255, 0] соответствует зеленому цвету.
3. Нарисовать прямоугольник позволяет Pygame-метод `pygame.draw.rect()`.
4. Для рисования линий, соединяющих множество точек друг с другом, применяется метод `pygame.draw.lines()`.
5. Термин *пиксел* (pixel) является сокращением от английских слов «picture element» и означает точку на экране (или на бумаге).
6. В Pygame-окне координата [0, 0] находится в верхнем левом углу.
7. На диаграмме координату [50, 200] имеет буква B.
8. На диаграмме координату [300, 50] имеет буква D.
9. Метод `blit()` в Pygame-модуле отвечает за копирование изображений.
10. Перемещение или анимирование изображения выполняется в два этапа:
    - изображение стирается со старого места;
    - изображение рисуется на новом месте.

Эксперименты

1. Программа, рисующая на экране несколько разных фигур, находится в файле *TIO_CH16_1.py* в папке *\answers* и на нашем сайте:

```
import pygame, sys
pygame.init()
screen=pygame.display.set_mode((640, 480))
screen.fill((250, 120, 0))
pygame.draw.arc(screen, (255, 255, 0), pygame.rect.Rect(43, 368, 277,
    235), -6.25, 0, 15)
pygame.draw.rect(screen, (255, 0, 0), pygame.rect.Rect(334, 191, 190,
    290))
pygame.draw.rect(screen, (128, 64, 0), pygame.rect.Rect(391, 349, 76,
    132))
pygame.draw.line(screen, (0, 255, 0), (268, 259), (438, 84), 25)
pygame.draw.line(screen, (0, 255, 0), (578, 259), (438, 84), 25)
pygame.draw.circle(screen, (0, 0, 0), (452, 409), 11, 2)
pygame.draw.polygon(screen, (0, 0, 255), [(39, 39), (44, 136), (59,
    136), (60, 102), (92, 102), (94, 131), (107, 141), (111, 50),
    (97, 50), (93, 86), (60, 82), (58, 38)], 5)
pygame.draw.rect(screen, (0, 0, 255), pygame.rect.Rect(143, 90, 23, 63),
    5)
pygame.draw.circle(screen, (0, 0, 255), (153, 60), 15, 5)
clock = pygame.time.Clock()
pygame.display.flip()

running = True
while running:
    clock.tick(60)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
            running = False
pygame.quit()
```

2. Для замещения мяча другим рисунком достаточно поменять имя файла в строке, вставив туда имя другого графического файла:

```
my_ball = pygame.image.load('beach_ball.png')
```

3. Найдите в листинге 16.16 эти две строки:

```
x_speed = 10
y_speed = 10
```

Измените их следующим образом:

```
x_speed = 20
y_speed = 8
```

4. Чтобы заставить мяч отскочить от «невидимой» стены, найдите в листинге 16.16 такую строку:

```
if x > screen.get_width() - 90 or x < 0:
```

Измените ее следующим образом:

```
if x > screen.get_width() - 250 or x < 0:
```

В результате мяч поменяет направление своего движения раньше, чем достигнет края окна. Аналогичным образом можно заставить его отскочить от пола, отредактировав координату *y*.

5. Вот как будет выглядеть листинг 16.6 после вставки метода `display.flip` в цикл `while` и добавления задержки:

```
import pygame, sys, random
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
for i in range (100):
    width = random.randint(0, 250)
    height = random.randint(0, 100)
    top = random.randint(0, 400)
    left = random.randint(0, 500)
    pygame.draw.rect(screen, [0,0,0], [left, top, width, height], 1)
    pygame.display.flip()
    pygame.time.delay(30)
```

Все прямоугольники должны появляться по отдельности, так как мы замедлили выполнение программы и вставили код очистки экрана после вывода каждой фигуры. Написав по аналогичному принципу программу рисования синусоиды, вы сможете понаблюдать за рисованием каждой точки.

17 Проверь себя

1. Обнаружение столкновений по ограничивающим прямоугольникам означает регистрацию момента соприкосновения или перекрывания прямоугольников, в которые вписаны два объекта.
2. В процессе обнаружения столкновений с использованием данных изображений участвуют контуры графического объекта. В этом случае ограничивающие объекты прямоугольники не рассматриваются. Это более точный и реалистичный вариант обнаружения столкновений, но он требует большего количества кода и слегка замедляет работу программы.
3. Следить за группой спрайтовых объектов можно как при помощи обычного Python-списка, так и при помощи группы Pygame-спрайтов.
4. Скорость анимации (частота кадров) в коде контролируется либо добавлением задержки между кадрами, либо методом `pygame.time.Clock`, позволяющим получить определенную частоту. Еще можно указывать, на какое расстояние (на какое количество пикселов) объект смещается в кадре.
5. Метод задержки является менее точным, так как не учитывает время выполнения самого кода в каждом кадре. В итоге невозможно точно предсказать итоговую частоту кадров.
6. Частоту воспроизведения анимации позволяет определить функция `pygame.time.Clock.get_fps()`.

18 Проверь себя

1. Программа реагирует на события клавиатуры и мыши.
2. Фрагмент кода, запускаемый после возникновения события, называется обработчиком события.
3. Для распознавания нажатых клавиш модуль Pygame пользуется событием `KEYDOWN`.
4. Атрибут `pos` сообщает местоположение мыши в момент возникновения события.
5. Определить следующий доступный номер для пользовательского события позволяет метод `pygame.USEREVENT`.
6. Для создания таймера пользуйтесь функцией `pygame.time.set_timer()`.
7. Для вывода текста в Pygame-окне пользуйтесь объектом `font`.
8. Три этапа вывода текста в Pygame-окне:
   - создание объекта `font`;
   - визуализация текста, создание поверхности;
   - блитирование (побитное копирование) этой поверхности на поверхность отображения.

Эксперименты

1. Почему так странно ведет себя ударяющийся о боковую поверхность ракетки шарик? Так как в данном случае речь идет о столкновении, код пытается поменять *y*-направление движения на противоположное (заставив двигаться вверх, а не вниз). Но прилетевший сбоку шарик даже после этой операции продолжит сталкиваться с ракеткой. На следующем шаге цикла (кадром позднее) он снова поменяет направление движения, начав двигаться вниз, и т. д. Проще всего это исправить, заставив шарик после столкновения с ракеткой всегда двигаться «вверх» (отрицательная *y*-скорость). Это не самый лучший способ, так как шарик будет смещаться вверх даже после столкновения с боковой поверхностью ракетки, что в реальности невозможно! Но проблема шарика, скачущего рядом с ракеткой, будет решена. Более реалистичное решение требует больше кода. Скорее всего, нужно будет проверять, с какой стороной ракетки столкнулся шарик, и только после этого программировать его «отскок».
2. На сайте в файле *TIO_CH18_2.py* вы найдете пример кода с добавленным в программу элементом случайности.

19

Проверь себя

1. Существуют звуковые файлы следующих типов: Wave (*.wav*), MP3 (*.mp3*), Ogg Vorbis (*.ogg*) и Windows Media Audio (*.wma*).
2. Для проигрывания музыки применяется модуль `pygame.mixer`.
3. Громкость воспроизведения звукового Pygame-объекта отдельно для каждого из объектов задает метод `set_volume()`.
4. Для задания громкости фоновой музыки пользуйтесь функцией `pygame.mixer.music.set_volume()`.
5. Создать затихающую музыку позволяет метод `pygame.mixer.music.fadeout()`. В качестве аргумента используется число миллисекунд (тысячных секунды), за которое музыка затихает. К примеру, метод `pygame.mixer.music.fadeout(2000)` уменьшит громкость до нуля за 2 секунды.

Эксперименты

Код программы для угадывания чисел после добавления звука — в файле *TIO_CH19_1.py* на нашем сайте.

20

Проверь себя

1. Графические GUI-элементы называют элементами управления, виджетами и компонентами.
2. Клавиша буквы, нажатие которой (одновременно с клавишей *Alt*) позволяет попасть в меню, называется горячей клавишей.
3. Файлы Qt-дизайнера должны иметь расширение *.ui*.
4. Типы компонентов, добавляемых к GUI при помощи модуля PyQt, включают в себя кнопку, флажок, индикатор процесса, раскрывающийся список, переключатель, счетчик, ползунок, текстовое поле, изображение, метку и ряд других. Увидеть полный перечень можно в окне *Widget Box* Qt-дизайнера.
5. Чтобы виджет начал выполнять некие действия, к нему следует добавить обработчик событий.
6. Символ & (амперсанд) в Qt-дизайнере задает горячую клавишу.
7. Содержимое счетчиков в Qt-дизайнере всегда целое.

Эксперименты

1. Версия программы для угадывания чисел, написанная с использованием Qt-дизайнера, доступна на сайте в виде файлов *TIO_CH20_1.py* и *TIO_CH20_1.ui*.
2. Для решения проблемы со счетчиком выделите его в Qt-дизайнере и в редакторе свойств измените свойства `minimum` и `maximum`. Свойству `minimum` имеет смысл присвоить значение в районе –1000, а свойству `maximum` — какое-нибудь очень большое значение, например 1 000 000.

21 Проверь себя

1. Чтобы вывести две инструкции `print` в одну строку, поставьте после первой запятую:

```
print "И как же",
print "тебя зовут?"
```

2. Чтобы добавить при выводе информации пустую строку, вставьте еще инструкцию `print`:

```
print "Hello"
print
print
print
print "World"
```

Также можно воспользоваться символом переноса `\n`:

```
print "Hello\n\n\nWorld"
```

3. Для горизонтального выравнивания пользуйтесь символом табуляции `\t`.
4. Для вывода числа в экспоненциальном представлении пользуйтесь форматирующей строкой `%e` или `%E`:

```
>>> number = 12.3456
>>> print '%e' % number
1.234560e+001
```

Эксперименты

1. Вот пример такой программы:

```
name = raw_input("Как тебя зовут? ")
age = int(raw_input("Сколько тебе лет? "))
color = raw_input("Какой твой любимый цвет? ")

print "Тебя зовут", name,
print "тебе", age, "лет,",
print "твой любимый цвет", color
```

2. Вот код для выравнивания таблицы умножения с помощью символа табуляции:

```
for looper in range(1, 11):
    print looper, "\ttimes 8 =\t", looper * 8
```

Обратите внимание на символ `\t` перед словом `times` и после знака `=`.

3. Программа, вычисляющая все дроби с дробной частью 8:

```
for i in range(1, 9):
    fraction = i / 8.0
    print str(i) + '/8 = %.3f' % fraction
```

Первая часть, `print str(i) + '/8 =`, выводит на экран дробь. Вторая часть, `%.3f' %fraction`, выводит на экран вещественное число с тремя знаками после десятичной точки.

22 Проверь себя

1. В Python объекты, используемые для работы с файлами, называют файловыми.
2. Файловый объект создается при помощи встроенной в Python функции `open()`.
3. Имя файла служит для хранения файла на диске (или любом другом накопителе). Файловый объект используется для работы с файлами в Python. Имя файлового объекта может не совпадать с именем файла на диске.

4. После завершения чтения файла или записи в файл программа должна его закрыть.
5. Данные, записанные в файл, открытый в режиме добавления, вставляются в конец этого файла.
6. Вся информация в файле, открытом для записи, теряется, ее заменяют новые данные.
7. Чтобы начать чтение файла сначала, когда часть его уже прочитана, воспользуйтесь методом `seek()` с аргументом 0:

```
myFile.seek(0)
```

8. Для сохранения Python-объекта в файле средствами модуля `pickle` используйте метод `pickle.dump()`, указывая в качестве аргументов предназначенный для сохранения объект и имя файла:

```
pickle.dump(myObject, "my_pickle_file.pkl")
```

9. Для десериализации сохраненного в файле объекта пользуйтесь методом `pickle.load()`, указывая в качестве аргумента имя файла:

```
myObject = pickle.load("my_pickle_file.pkl")
```

Эксперименты

1. Пример программы, генерирующей нелепые фразы:

```
import random

noun_file = open("nouns.txt", 'r')
nouns = noun_file.readline()
noun_list = nouns.split(',')
noun_file.close()

adj_file = open("adjectives.txt", 'r')
adjectives = adj_file.readline()
adj_list = adjectives.split(',')
adj_file.close()

verb_file = open("verbs.txt", 'r')
verbs = verb_file.readline()
verb_list = verbs.split(',')
verb_file.close()

adverb_file = open("adverbs.txt", 'r')
adverbs = adverb_file.readline()
adverb_list = adverbs.split(',')
adverb_file.close()

noun = random.choice(noun_list)
adj = random.choice(adj_list)
verb = random.choice(verb_list)
adverb = random.choice(adverb_list)

print "The", adj, noun, verb, adverb + '.'
```

   Файлы со словами представляют собой списки слов, отделенных друг от друга запятыми.

2. Программа, сохраняющая введенные данные в текстовом файле:

```
name = raw_input("Введите ваше имя: ")
age = raw_input("Введите ваш возраст: ")
color = raw_input("Введите любимый цвет: ")
food = raw_input("Введите любимое блюдо: ")
```

```
my_data = open("my_data_file.txt", 'w')
my_data.write(name + "\n")
my_data.write(age + "\n")
my_data.write(color + "\n")
my_data.write(food)

my_data.close()
```

3. Программа, сохраняющая данные средствами модуля `pickle`:

```
import pickle
name = raw_input("Введите ваше имя: ")
age = raw_input("Введите ваш возраст: ")
color = raw_input("Введите ваш любимый цвет: ")
food = raw_input("Введите вашу любимую еду: ")

my_list = [name, age, color, food]

pickle_file = open("my_pickle_file.pkl", 'w')
pickle.dump(my_list, pickle_file)

pickle_file.close()
```

23 Проверь себя

1. *Случайным* называется событие, результат возникновения которого заранее неизвестен. Примерами таких событий являются бросок монетки (она может упасть как орлом, так и решкой) и бросок костей (выпадающие на верхней грани числа заранее неизвестны).
2. Бросок кости с одиннадцатью сторонами отличается от броска двух костей с шестью сторонами, так как в первом случае вероятность выпадения всех чисел от 2 до 12 одинакова. В случае двух костей с шестью сторонами некоторые числа (сумма точек на обеих костях) будут появляться реже остальных.
3. Первый способ:

```
import random
sides = [1, 2, 3, 4, 5, 6]
die_1 = random.choice(sides)
```

Второй способ:

```
import random
die_1 = random.randint(1, 6)
```

4. Для представления одной карты мы пользовались объектом.
5. Для представления колоды карт мы пользовались списком. Каждый элемент списка отражал одну карту (один объект).
6. Убрать карту из колоды для списков позволяет метод `remove()`, например: `deck.remove()` или `hand.remove()`.

Эксперименты

Просто проделайте этот эксперимент и посмотрите, что получится.

24 Проверь себя

1. Компьютерное моделирование применяется по следующим причинам:
   - для экономии денег (имитация дорогостоящих экспериментов);
   - для защиты людей и оборудования (имитация опасных экспериментов);

- для выполнения неосуществимых в реальности вещей (мы не сможем сделать так, чтобы большой астероид врезался в Луну);
- для ускорения времени (эксперимент можно провести быстрее, чем это происходит на самом деле), что помогает при изучении медленных процессов, таких как таяние ледников;
- для замедления времени (эксперимент проходит медленнее, чем в реальности), что может помочь при изучении быстрых процессов, таких как передвижение электронов по проводу.

2. Вы можете назвать любые компьютерные модели, которые придут вам в голову. Игры, математические или физические модели, даже прогноз погоды — компьютерное моделирование применяется везде.
3. Для хранения разницы между двумя датами или моментами времени применяется объект `timedelta`.

ЭКСПЕРИМЕНТЫ

Эти программы вы найдете на сайте *www.manning.com/books/hello-world-second-edition*:

1. *TIO_CH24_1.py* — Lunar Lander с проверкой ухода с орбиты.
2. *TIO_CH24_2.py* — Lunar Lander с повторением попыток приземления.
3. *TIO_CH24_3.py* — GUI-интерфейс игры Virtual Pet с кнопкой *Pause*.

ЭКСПЕРИМЕНТЫ

Вот базовый код робота, способного победить соперника CircleAI:

```
class AI:
    def __init__(self):
        self.isFirstTurn = True
    def turn(self):
        if self.isFirstTurn:
            self.robot.turnLeft()
            self.isFirstTurn = False
        elif self.robot.lookInFront() == "bot":
            self.robot.attack()
        else:
            self.robot.doNothing()
```

Он ждет, пока питон CircleAI обойдет вокруг доски, после чего атакует, как только тот приблизится. В данном случае я учитывал принцип работы робота CircleAI, поэтому против других роботов эта стратегия работать не будет. Как я уже упоминал, написать программу для робота, который сможет все время выигрывать, какой бы робот ни выступал в качестве противника, очень сложно.