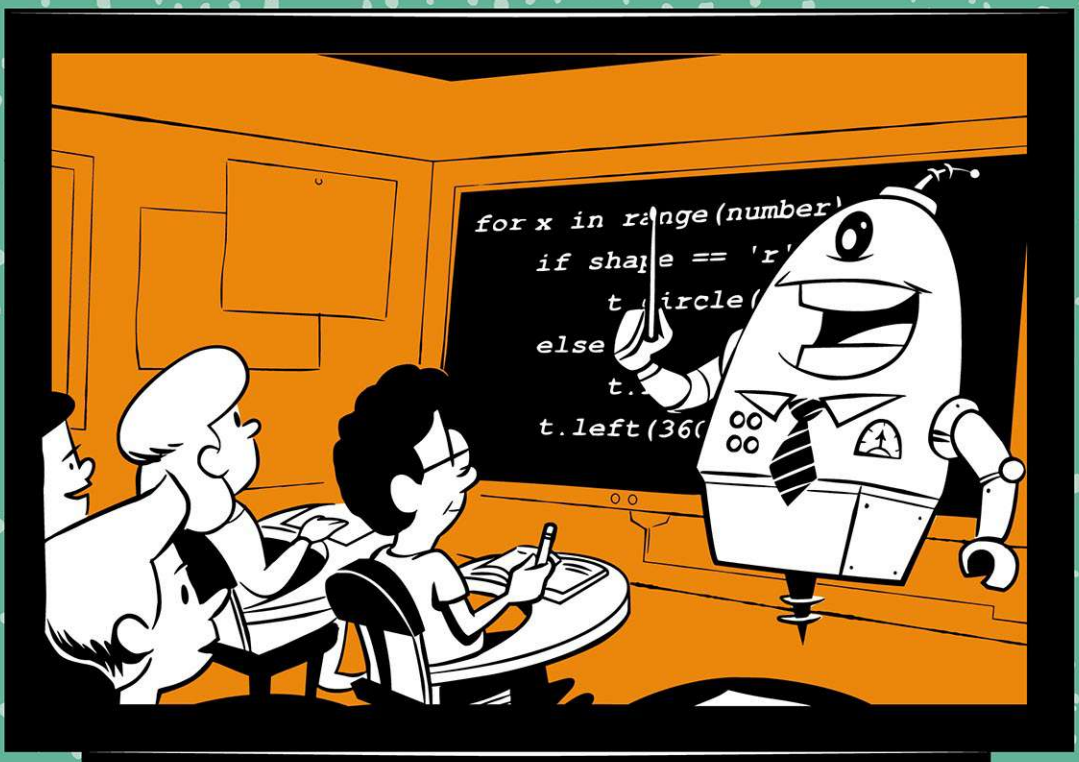
PYTHON ДЛЯ ДЕТЕЙ И РОДИТЕЛЕЙ

ИГРАЙ И ПРОГРАММИРУЙ

БРАЙСОН ПЭЙН

ДЛЯ ДЕТЕЙ СТАРШЕ 9 ЛЕТ И ИХ РОДИТЕЛЕЙ



no starch press

ПРОГРАМИРОВАНИЕ
ДЛЯ ДЕТЕЙ

BRYSON PAYNE

TEACH YOUR KIDS TO CODE

A PARENT-FRIENDLY GUIDE TO PYTHON PROGRAMMING

no starch press

БРАЙСОН ПЭЙН

PYTHON ДЛЯ ДЕТЕЙ И РОДИТЕЛЕЙ

ИГРАЙ И ПРОГРАММИРУЙ

МОСКВА
2017

УДК 004.43-053.2
ББК 32.973.26-018.1
П97

Bryson Payne
TEACH YOUR KIDS TO CODE
Copyright © 2015 by Bryson Payne. Title of English-language original:
Teach Your Kids to Code, ISBN 978-1-59327-614-0, published by No Starch Press.
All rights reserved

Пэйн, Брайсон.

П97 Python для детей и родителей / Брайсон Пэйн ; [пер. с англ. М.А. Райтмана]. — Москва : Издательство «Э», 2017. — 352 с. : ил. — (Программирование для детей).

Программирование — одна из самых востребованных профессий в наше время, и она останется таковой в ближайшем будущем.

Научите своих детей программировать уже сейчас с помощью этой книги! В книге представлен язык Python, один из самых популярных и простых. Вы найдете здесь много упражнений – полезных, интересных и забавных, поэтому ваш ребенок не заскучает. Материал написан доступно и просто, поэтому ему не составит труда освоить азы программирования.

УДК 004.43-053.2
ББК 32.973.26-018.1

Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность.

Пособие для развивающего обучения

ПРОГРАММИРОВАНИЕ ДЛЯ ДЕТЕЙ

Пэйн Брайсон

PYTHON ДЛЯ ДЕТЕЙ И РОДИТЕЛЕЙ

Директор редакции *Е. Капьёв*
Ответственный редактор *Е. Истомина*
Художественный редактор *А. Гусев*

ООО «Издательство «Э»
123308, Москва, ул. Зорге, д. 1. Тел. 8 (495) 411-68-86.

Өндіруші: «Э» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел. 8 (495) 411-68-86.
Тауар белгісі: «Э»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды қабылдаушының
өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-89/90/91/92, факс: 8 (727) 251 58 12 вн. 107.
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта Өндіруші «Э»

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Э»

Өндірген мемлекет: Ресей
Сертификация қарастырылмаған

Подписано в печать 19.12.2016. Формат 70x100 1/16.
Печать офсетная. Усл. печ. л. 28,52.
Тираж экз. Заказ

6+

ISBN 978-5-699-95169-7

9 785699 951697 >

ISBN 978-5-699-95169-7

BOOK24.RU
ИНТЕРНЕТ-МАГАЗИН

В электронном виде книги издательства «Эксмо» вы можете купить на www.litres.ru
ЛитРес:

© Райтман М.А., перевод на русский язык, 2017
© Оформление. ООО «Издательство «Э», 2017

ОТЗЫВЫ О КНИГЕ

«Ясный и четкий текст, привлекательные иллюстрации и потрясающие приложения. Получать удовольствие от этого справочника по программированию смогут и родители, и дети». — Аарон Уолкер, эксперт по безопасности в киберпространстве, NASA.

«Дает строительные блоки для замечательного будущего в быстро меняющемся мире технологий». — Джоан Тейлор, бывший вице-президент Global Telecommunications, IBM.

«У каждого ребенка на планете должна быть эта книга, и у родителя тоже». — Джеймс Е. Дэниел-младший, учредитель App Studios, LLC.

«Жаль, что у меня не было такой книги в детстве». — Скотт Хэнд, разработчик программного обеспечения, CareerBuilder.

ОГЛАВЛЕНИЕ

Отзывы о книге **5**

Введение. Что такое программирование и почему оно полезно для детей? **10**

Почему дети должны изучать программирование? 11
Где дети могут научиться писать код? 12
Как пользоваться этой книгой 13

Глава 1. Основы Python: Знакомство со средой **16**

Начало работы с Python. 18
Написание программ на Python. 20
Запуск программ на Python. 21
Что вы узнали 22
Задачи по программированию 23

Глава 2. «Черепашья» графика: рисование с Python **25**

Наша первая программа `turtle` 25
Черепашка в ударе. 30
Черепашка закругляется 31
Добавим красок 33
Одна переменная, управляющая всей программой. 39
Что вы узнали 42
Задачи по программированию 43

Глава 3. Числа и переменные: Python делает подсчеты **45**

Переменные: место, где мы храним данные. 45
Числа и математика в Python 48
Строки: реальные символы в Python 57
Улучшим нашу спираль с помощью текста 59
Списки: храним все в одном месте 61
Python делает ваше домашнее задание. 64
Что вы узнали 66
Задачи по программированию 68

Глава 4. Циклы — это весело (повторите пару раз) **69**

Создание собственных циклов `for` 71
Улучшение программы с розеткой с помощью пользовательского ввода 75
Игровые циклы и циклы while 78
Семейная спираль. 81
Сведем все вместе: спираль уходит в народ 85
Что вы узнали 91
Задачи по программированию 92

Глава 5. Условия (Что если?) **94**

Выражение `if` 96

Встречаем булевы выражения 98
Выражения `else` 103
Выражения `elif` 110
Сложные условия: `if`, `and`, `or`, `not` 111
Секретные послания 114
Что вы узнали 122
Задачи по программированию 124

Глава 6. Случайное веселье и игры: на удачу! 126

Игра на угадывание 127
Цветные случайные спирали 130
Камень, ножницы, бумага 138
Выберите карту, любую карту 141
Кидаем кубики: игра в кости в стиле яцзы 149
Калейдоскоп 157
Что вы узнали 161
Задачи по программированию 164

Глава 7. Функции: да, у этого есть название 166

Соберем все вместе с функциями 167
Параметры: покормите свою функцию 171
`Return`: важно не то, что ты получаешь, важно то, что ты возвращаешь 179
Прикосновение интерактивности 184
ClickKaleidoscope 195
Что вы узнали 199
Задачи по программированию 201

Глава 8. Таймеры и анимация: как поступил бы Дисней? 202

Использование графического интерфейса Pygame 202
Правильный тайминг: двигайся и прыгай 213
Что вы узнали 231
Задачи по программированию 233

Глава 9. Взаимодействие с пользователем: подключаемся к игре 237

Добавление интерактивности: щелкни и перетащи 238
Улучшенная интерактивность: взрыв из смайликов 245
SmileyPop, версия 1.0 256
Что вы узнали 260
Задачи по программированию 261

Глава 10. Программирование игр: кодинг для развлечения 263

Создание каркаса игры: Smiley Pong, версия 1.0 264
Усложнение и конец игры: Smiley Pong, версия 2.0 278
Добавление новых функций: SmileyPop 2.0 285
Что вы узнали 294
Задачи по программированию 296

Приложение А. Установка Python в среде Windows, macOS и Linux 298

Python для Windows 298

Python для macOS 307
Python для Linux 313

Приложение Б. Установка и настройка Pygame в среде Windows, macOS и Linux 315

Pygame для Windows 315
Pygame для macOS 320
Pygame для Linux 324

Приложение В. Создание ваших собственных модулей 326

Создание модуля `colorspiral` 327
Дополнительные ресурсы 331

Приложение Г. Установка Pygame для Python 3 в среде macOS и Linux 332

Pygame для Python 3.4 в среде macOS 332
Pygame для Python 3 в среде Linux 341

Глоссарий 343

Об авторе 347

Об иллюстраторе 347

Благодарности 348

Предметный указатель 349

Алексу и Маку,

двум моим любимым программистам

Введение

ЧТО ТАКОЕ ПРОГРАММИРОВАНИЕ И ПОЧЕМУ ОНО ПОЛЕЗНО ДЛЯ ДЕТЕЙ?

Компьютерное программирование, или *написание программного кода,* — один из важнейших навыков, которым должен обладать каждый ребенок. Мы используем программы для решения математических задач, игр, они помогают нам повысить эффективность труда, выполнять однообразные задания, хранить и повторно использовать информацию, создавать что-то новое, а также поддерживать связь с друзьями и всем миром. Понимание принципов программирования делает всю эту мощь компьютеров легкодоступной.

Каждый может научиться программировать: это аналогично решению головоломки или загадки. Все, что нужно, — использовать логику, опробовать решение, еще немного поэкспериментировать — и наконец решить задачу. Время научиться программировать настало уже *сейчас*! Мы живем в совершенно особый исторический период: никогда раньше миллиарды людей не могли ежедневно общаться друг с другом, как мы можем общаться сегодня с помощью компьютеров. Мы живем в мире, полном новых возможностей: от электромобилей и роботов-сиделок до радиоуправляемых квадрокоптеров, доставляющих посылки и даже пиццу.

Если ваши дети начнут обучаться программированию уже сегодня, то они смогут оказать помощь в формировании облика этого быстро изменяющегося мира.

Почему дети должны изучать программирование?

Существует множество веских причин изучать компьютерное программирование, но для меня эти две самые важные.

1. Программирование — это весело.
2. Программирование — это ценный профессиональный навык.

Программирование — это весело

Новые технологии становятся частью нашей повседневной жизни. Каждая компания, благотворительная организация и любое дело могут ощутить на себе преимущества новых технологий. На рынке представлены приложения, помогающие пользователям покупать, передавать, вступать, играть, быть волонтером, общаться и делиться — программы помогают во всем, что вы только можете себе представить.

Хотели ли ваши дети создать собственный уровень любимой видеоигры? Программисты делают это! А как насчет создания их собственного приложения на телефон? Они могут осуществить этот замысел, написав программный код такого приложения на компьютере! Любая программа, игра, система или приложение, которые им довелось увидеть, были созданы с помощью блоков программного кода, аналогичных тем, что представлены в этой книге. Когда дети программируют, они принимают активное участие в развитии технологий: они не просто *развлекаются*, они делают технологии *увлекательными*!

Программирование — это ценный профессиональный навык

Программирование — это *важнейший* навык XXI века. Сегодня работа как никогда ранее требует способности решать поставленные задачи, новые технологии становятся неотъемлемой частью строительства карьеры во все большем количестве отраслей.

Бюро трудовой статистики США предсказывает, что в следующие пять лет на рынке будет создано более 8 миллионов рабочих мест, так или иначе связанных с новыми технологиями. Согласно справочнику занятости за 2014–2015 годы, семь из десяти быстро набирающих популярность должностей, для которых не требуется наличие степени магистра или доктора, приходятся на компьютерную науку и информационные технологии (ИТ).

Марк Цукерберг был лишь студентом колледжа, работавшим в своей комнате в общежитии, когда создал первую версию Facebook в 2004 году. Спустя всего 10 лет 1,39 миллиарда людей *в месяц* использовали Facebook (источник: **newsroom.fb.com/company-info/**). Еще никогда в истории идея, продукт или услуга не могли обзавестись аудиторией более миллиарда человек менее чем за 10 лет. Facebook демонстрирует уникальную способность новых технологий достигать большого количества людей так быстро, как никогда ранее.

Где дети могут научиться писать код?

Эта книга — только лишь начало. Сегодня есть небывалое количество мест, где дети могут обучиться программированию. Сайты, такие как Code.org (рис. 1), Codeacademy и бессчетное количество подобных им, обучают основам и более углубленным аспектам написания программного кода на любом языке программирования. Когда вы с детьми завершите изучение этой книги, ваши дети могут записаться на бесплатные курсы с помощью сайтов, таких как EdX, Udacity или Coursera, для более углубленного обучения.

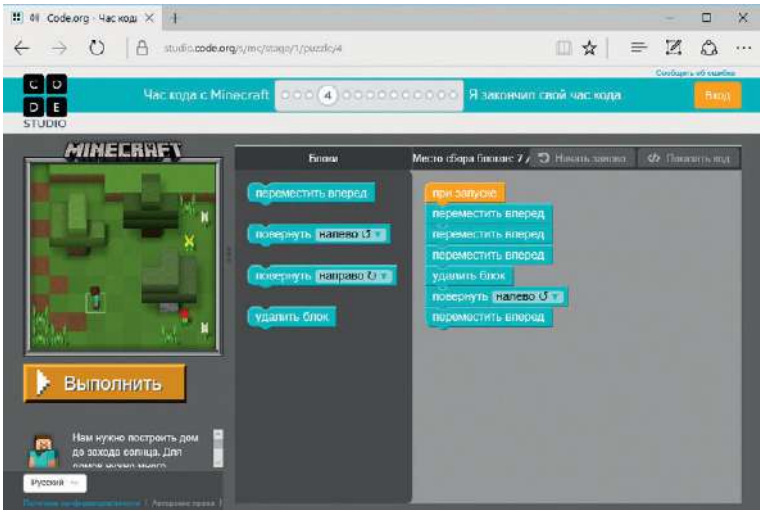


Рис. 1. *Code.org в игровой форме обучит вас программированию на большом количестве языков*

«Клубы программистов» — отличный способ сделать обучение развлекательным вместе с друзьями. Получение степени в колледже по одному из соответствующих профилей все же является одним из лучших способов

подготовки к началу карьеры. Но, если вы сейчас не рассматриваете колледж как один из доступных вариантов, ваши дети могут начать составлять свое портфолио программиста и демонстрировать навыки написания программного кода и решения поставленных задач уже сегодня.

Как пользоваться этой книгой

Эта книга написана не только для детей, но и для родителей, учителей, студентов и взрослых, которые хотели бы изучить основы компьютерного программирования, как для развлечения, так и для получения доступа к новым рабочим местам в высокотехнологической экономике. Не имеет значения, сколько вам лет, вы все можете получить удовольствие, изучая основы программирования. Наилучший подход в данном случае заключается в сочетании работы и экспериментов.

Исследуйте!

Изучение программирования может оказаться увлекательным, если вы желаете пробовать что-то новое. По мере следования программам, излагаемым в данной книге, пробуйте изменять цифры и текст в коде, чтобы увидеть, что произойдет с программой. Даже если вы выведете ее из строя, исправляя программу, вы научитесь чему-то новому. В худшем случае все, что вам понадобится сделать, — это снова ввести пример кода из книги или открыть последнюю рабочую версию программы. Цель изучения кода — попробовать что-то новое, обзавестись новым навыком — и получить возможность решать задачи новым способом. Убедитесь, что ваши дети играют с программой: тестируют код, изменяя что-либо в нем, сохраняя программу, запуская ее, наблюдая за результатами и исправляя ошибки при необходимости.

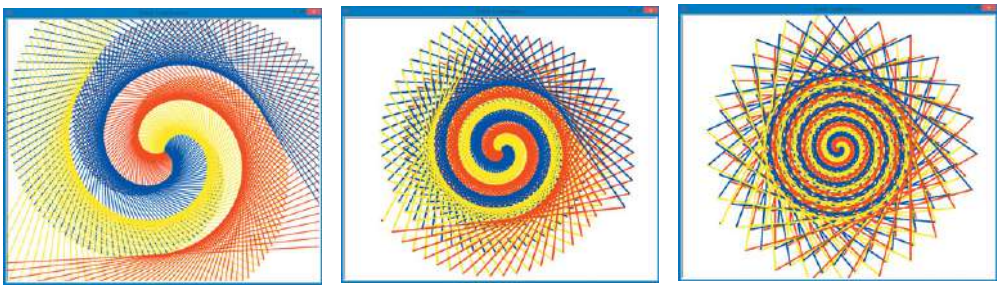


Рис. 2. *Три спиральных узора, которые я создал, просто пробуя ввести разные значения в строке кода одной из программ*

Так, например, я написал небольшой программный код для отображения спирального узора (рис. 2), затем вернулся, изменил некоторые цифры там и сям и попробовал запустить программу снова. Так я получил второй узор, абсолютно не похожий на первый, но от этого не менее потрясающий. Я снова поменял еще несколько цифр кода — и получил взамен еще один красивый уникальный узор. Видите, что можно сделать, просто экспериментируя с кодом программы?

Делайте это вместе!

Эксперименты с кодом — это хороший способ понять, как работают программы, но еще более эффективно работать совместно с кем-то. Учите ли вы ребенка, студента или обучаетесь самостоятельно, всегда гораздо *веселее* экспериментировать вместе, кроме того, это еще и более *эффективно*.

Например, особенность методики преподавания музыки Судзуки в том, что родители посещают занятия вместе с детьми и даже изучают программу немного вперед, чтобы помогать ребенку. Раннее начало обучения также является еще одной визитной карточкой метода Судзуки: дети могут приступить к формальному обучению в возрасте трех или четырех лет.

Я начал знакомить своих сыновей с программированием, когда им было два и четыре года, я поощрял их веселье, разрешал изменять небольшие кусочки каждой программы, например параметры цветов, форм или размеров форм.

Я изучил программирование, когда мне было 13. Я просто набирал на компьютере примеры из книг, затем изменял их, заставляя программы выполнять новые действия. Теперь, преподавая курсы по компьютерной науке, я часто даю студентам программу и поощряю их эксперименты с кодом, чтобы они могли создать нечто новое.

Если вы используете эту книгу для самостоятельного обучения, вы можете работать и вместе с другими людьми, например найдя друга, с которым можно было бы прорабатывать примеры, или открыв клуб по программированию после занятий, или среди знакомых (на сайтах **coderdojo.com** и **codeacademy.com/afterschool/** вы можете получить советы или почерпнуть несколько идей). Программирование в команде — это спорт!

Ресурсы во Всемирной паутине

Файлы с кодом всех программ из этой книги можно скачать по адресу **https://eksmo.ru/files/Python_deti.zip**. Для более углубленного изучения

скачайте эти программы и поэкспериментируйте с ними. Если вы зашли в тупик, воспользуйтесь примером решения. Заходите на сайт!

Программирование = Решение задач

Сколько бы ни было лет вашему ребенку: два, и он только учится считать, или 22, и он ищет новые вызовы, — эта книга и концепции, излагаемые в ней, освещают дорогу к достойному и вдохновляющему времяпрепровождению и лучшим карьерным перспективам. Люди, умеющие программировать и, соответственно, решать задачи быстро и эффективно, очень ценятся в современном мире, поэтому им достается интересная и благодарная работа.

Конечно, не все проблемы современного мира могут быть решены с помощью технологий, однако технология способствует общению, сотрудничеству, повышает информированность, позволяет действовать, причем делает это с неслыханными раньше скоростью и размахом. Если вы умеете писать программный код, вы можете решать задачи. Люди, способные решать поставленные перед ними задачи, обладают возможностью сделать этот мир лучше, поэтому начните программировать уже сегодня.

Глава 1

ОСНОВЫ PYTHON: ЗНАКОМСТВО СО СРЕДОЙ

Практически внутри всего может находиться компьютер: будь то телефон, автомобиль, часы, игровая консоль, тренажер, медицинское устройство, промышленное оборудование, поздравительная открытка или робот. С помощью компьютерного программирования, или *написания программного кода*, мы даем компьютеру инструкцию выполнить ту или иную задачу. Понимание принципов написания кода делает всю мощь компьютера легкодоступной для вас.

Компьютерные программы, также называемые *приложениями*, сообщают компьютеру, какие действия от него требуются. Так, веб-приложение может приказать компьютеру отслеживать вашу любимую музыку, игровое приложение сообщает компьютеру, как отображать эпическое поле боя с реалистичной графикой, простое приложение может дать инструкцию компьютеру, как нарисовать красивую спираль, такую как шестиугольный орнамент на рис. 1.1.

Рис. 1.1. *Цветной спиралевидный орнамент*

Некоторые приложения содержат тысячи строк кода, а другие — всего несколько, как, например, программа *NiceHexSpiral.py*, код которой показан на рис. 1.2.

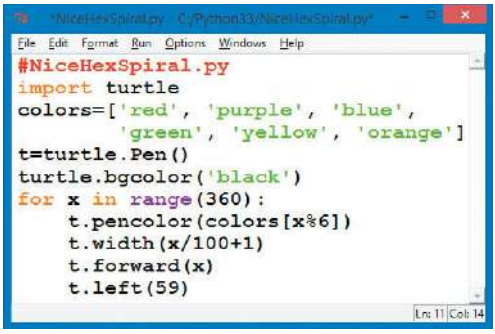


Рис. 1.2. *NiceHexSpiral.py — это короткая программа на языке Python, рисующая спираль, наподобие показанной на рис. 1.1*

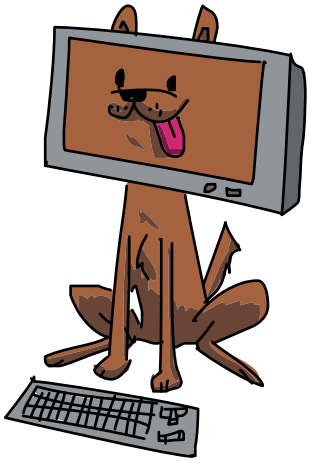
Эта короткая программа рисует цветную спираль, показанную на рис. 1.1. В качестве иллюстрации для этой книги мне хотелось использовать какую-то приятную картинку, поэтому я решил эту задачу с помощью компьютерной программы. Сначала я набросал идею, а затем начал писать программный код.

В этой главе мы скачаем, установим и научимся использовать программы, которые помогут нам писать код и создавать различные приложения, любые, какие вы только можете себе представить.

Начало работы с Python

Чтобы начать писать программный код, вам необходимо начать говорить на компьютерном языке. Компьютерам требуются пошаговые инструкции, и они могут понимать только определенные языки. Точно так, как человек из России может не понимать по-английски, компьютеры понимают только языки, специально созданные для них. Компьютерный, или программный, код пишется на языках программирования, таких как Python, C++, Ruby или JavaScript. Эти языки позволяют нам «разговаривать» с компьютером и сообщать ему команды. Представьте, что вы обучаете свою собаку трюкам: когда вы произносите команду «сидеть», она садится, говорите «голос» — она лает. Собака понимает эти простые команды, однако остальные ваши слова — нет.

Также у компьютеров есть определенные ограничения, однако они готовы выполнить все, что вы попросите на их языке. Язык, которым мы будем пользоваться в этой книге, называется *Python* — простой и мощный язык программирования. Язык Python преподается на вводных курсах по компьютерной науке в старших классах и колледжах, кроме того, он используется для создания одних из самых мощных приложений на свете, таких как Gmail, Google Maps и YouTube.



Чтобы вы могли пользоваться языком Python на своем компьютере, давайте совместно сделаем следующие три шага.

1. Скачаем Python.
2. Установим Python на компьютер.
3. Протестируем Python, написав одну или две простые программы.

1. Скачивание Python

Python — это бесплатная программа, которую легко загрузить на компьютер с официального веб-сайта, изображенного на рис. 1.3.

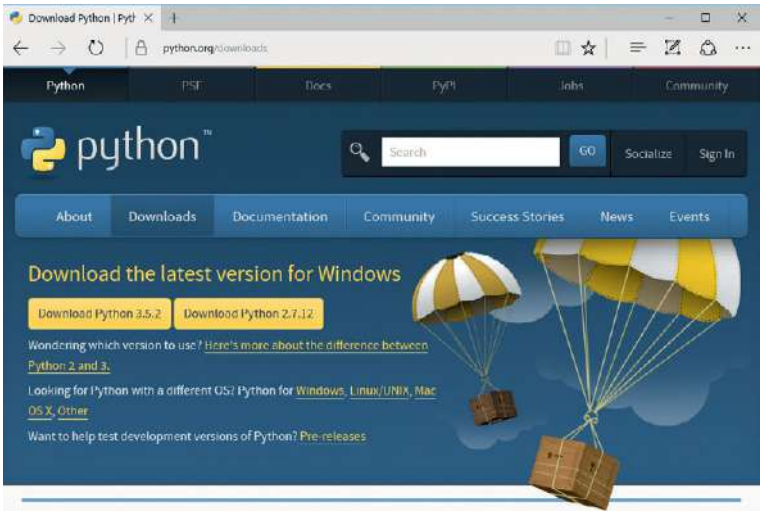


Рис. 1.3. *Сайт Python упрощает загрузку Python на компьютер*

С помощью своего браузера перейдите на сайт **python.org**, наведите указатель мыши на кнопку меню **Downloads** и щелкните по кнопке, текст которой начинается с **Python 3**.

2. Установка Python

Найдите только что загруженный файл (возможно, он расположен в папке *Загрузки* (*Downloads*)), дважды щелкните мышью по этому файлу, чтобы запустить программу и установить Python и редактор IDLE на компьютер. IDLE — это программа, которую мы будем использовать для ввода и запуска наших программ на языке Python. Более подробные инструкции по установке программного обеспечения см. Приложение А.

3. Тестирование Python

В меню **Пуск** (Start) или папке *Приложения* (*Applications*) найдите программу IDLE и запустите ее. Вы увидите текстовое окно для ввода команд, наподобие того, что изображено на рис. 1.4. Эта программа называется оболочкой Python. *Оболочка* — это окно или экран, позволяющие пользователю вводить команды и строки кода.

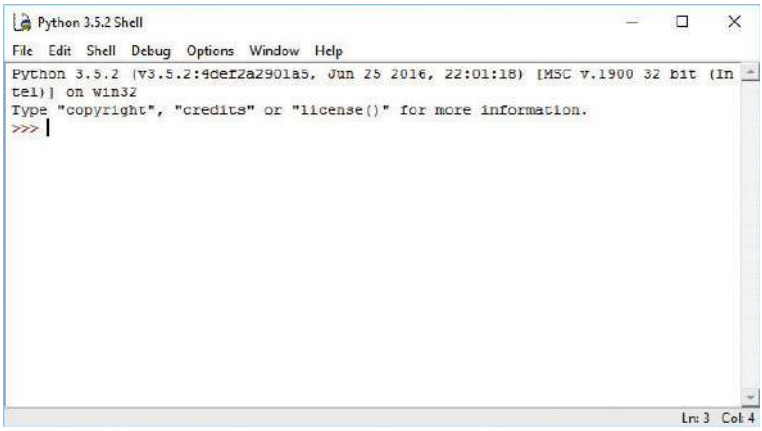


Рис. 1.4. *Оболочка Python IDLE — наш командный пункт для изучения Python*

Строка >>> называется *приглашением*, и ее наличие означает, что компьютер готов принять вашу первую команду. Компьютер просит вас сказать ему, что требуется сделать. Введите:

```
print("Здравствуй, мир!")
```

и нажмите на клавиатуре клавишу **Enter** или **Return**. На экране вы должны увидеть, что Python напечатал текст, который вы указали в скобках: `Здравствуй, мир!` Ну вот: вы написали свою первую программу!

Написание программ на Python

Обычно вам потребуется писать программы длиной более чем одна строка, поэтому Python поставляется вместе с *редактором*, позволяющим писать более длинные программы. В окне IDLE перейдите к меню **File** (Файл) и выберите команду **New Window** (Новое окно) или **New File** (Новый файл). На экране появится пустое окно с заголовком *Untitled*.

Давайте напишем чуть более длинную программу на Python. В новом, пустом, окне введите следующие три строки кода:

```
# YourName.py
name = input("Как тебя зовут?\n")
print("Привет, ", name)
```

Первая строка называется *комментарием*. Комментарии, начинающиеся с символа «решетка» (#), представляют собой заметки программиста или

какие-то напоминания, игнорируемые компьютером. В данном примере комментарий — это лишь заметка, напоминающая нам об имени программы. Вторая строка просит пользователя ввести свое имя и запоминает его как `name`. Третья строка печатает слово `"Привет, "`, после которого следует имя, введенное пользователем. Обратите внимание, что текст `"Привет, "` и `name` разделяет запятая (`,`).

Запуск программ на Python

Перейдите к пункту **Run** (Запуск) строки меню, расположенной над текстом вашей программы, и выберите команду **Run Module** (Запуск модуля). Данное действие *запустит*, или передаст в обработку, инструкции вашей программы. Сначала система попросит вас сохранить программу. Давайте назовем наш файл *YourName.py*. Данное действие сообщит вашему компьютеру, что программу нужно сохранить как файл с именем *YourName.py*, где расширение *.py* обозначает, что перед нами программа на языке Python.

После сохранения и запуска файла вы увидите, что окно оболочки Python запустило программу, выведя на экран вопрос `Как тебя зовут?`. Введите свое имя на следующей строчке и нажмите клавишу **Enter**. Программа напечатает слово `Привет`, после которого будет следовать введенное вами имя. Так как вы не просили программу сделать что-либо еще, то программа будет завершена — и вы опять увидите приглашение >>>, как показано на рис. 1.5.

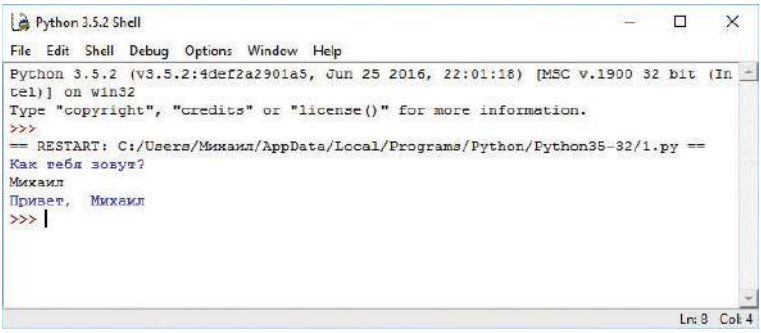


Рис. 1.5. *Компьютер знает мое имя!*

Маленьким ученикам, таким как мой трехлетний сын, очень интересно объяснять, что программа просит их ввести свое имя. Макс уже знает буквы своего имени, поэтому он набирает на клавиатуре *м-а-к-с*, и ему очень нравится, когда я говорю, что программа ответила ему `Привет, макс`. Спросите своего ребенка, не хотел ли бы он, чтобы программа ответила ему что-то другое. Макс

попросил, чтобы программа сказала «Здравствуй», поэтому я изменил программу так, чтобы третья строка кода гласила `Здравствуй,` вместо `Привет,`.

Потом я изменил третью строчку следующим образом:

```
print("Здравствуй, ", name, name, name, name, name)
```

Максу очень понравилось, что программа ответила ему: `Здравствуй, макс макс макс макс макс`. Попробуйте поэкспериментировать со второй и третьей строкой кода с тем, чтобы компьютер задавал разные вопросы и выводил на экран разные ответы.

Что вы узнали

Изучение программирования подобно решению загадок или головоломок. Вы начинаете с постановки проблемы, отвечаете, что знаете, и узнаете нечто новое в процессе. К моменту завершения вы потренировали мозг и ответили на вопрос. Надеюсь, вам было интересно.

В этой главе мы решили нашу первую сложную задачу: мы установили язык программирования Python на наши компьютеры, чтобы приступить к написанию кодов. И это было очень просто, всего лишь загрузить файл, установить и запустить его.

В следующих главах вы научитесь решать задачи с помощью программного кода. Вы начнете с простых визуальных задач, таких как рисование фигур на экране компьютера (планшета или телефона), а потом узнаете, как создавать простые игры, такие как «Угадай число», «Камень, ножницы, бумага» и «Понг».

Используя ту базу, которую заложите, создавая эти первые программы, вы сможете перейти к программированию игр, мобильных приложений, веб-приложений и многого другого.

На данный момент вы должны…

- Иметь полнофункциональную среду Python и текстовый редактор.
- Уметь вводить программные команды напрямую в оболочку Python.
- Уметь писать, сохранять, запускать и редактировать короткие программы в IDLE.
- Быть готовыми к испытанию более сложных, более интересных программ из главы 2.

Задачи по программированию

В конце каждой главы у вас будет возможность попрактиковаться в том, что вы изучили, а также создать более крутые программы, решив пару задач. (Если вы зайдете в тупик, вы можете перейти на сайт **https://eksmo.ru/files/Python_deti.zip** для получения готовых решений.)

#1. ЧЕПУХА

У простого приложения *YourName.py* есть все необходимые компоненты для создания такой гораздо более интересной программы, как старомодная словесная игра «Чепуха» (если вы никогда не пробовали поиграть в такую, перейдите на сайт **madlibs.com**).

Давайте изменим нашу программу *YourName.py* и сохраним ее как *MadLib.py*. Вместо того чтобы спрашивать имя пользователя, мы попросим ввести прилагательное, существительное и глагол прошедшего времени мужского рода единственного числа и сохраним введенные данные в трех разных переменных точно так же, как мы поступали с переменной `name` в исходной программе. Потом мы выведем на печать какое-нибудь предложение, например «Этот *прилагательное существительное глагол* на ленивую рыжую собаку». После внесения изменений программный код должен выглядеть следующим образом:

MadLib.py

```
adjective = input("Введите прилагательное: ")
noun = input("Введите существительное: ")
verb = input("Введите глагол в прошедшем времени: ")
print("Ваша чепуха:")
print("Этот ", adjective, noun, verb, "на ленивую ↵
       рыжую собаку.")
```

Символы стрелки ↵ в конце некоторых строк кода означают перенос строки. Набирать этот символ не нужно!

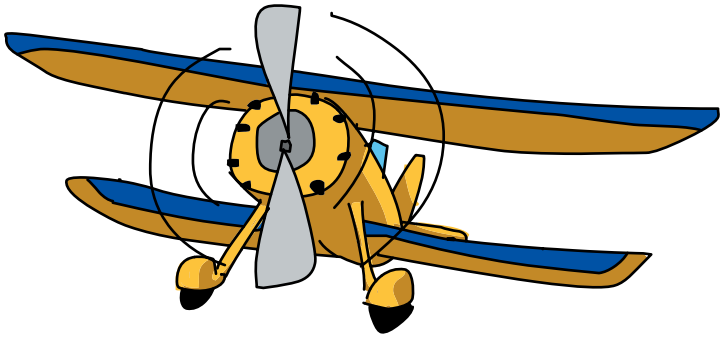
Вы можете ввести любое прилагательное, существительное и глагол по выбору (главное, чтобы они согласовывались в роде и числе).

Далее показано то, что вы должны увидеть на экране после сохранения и запуска программы *MadLib.py* (я ввел `умный`, `учитель` и `чихнул`):

```
>>>
Введите прилагательное: умный
Введите существительное: учитель
Введите глагол прошедшего времени: чихнул
Ваш MadLib:
Этот умный учитель чихнул на ленивую рыжую собаку.
>>>
```

#2. Новая Чепуха!

Давайте сделаем нашу игру MadLib чуть более интересной. Создайте новую версию *MadLib.py*, сохранив ее под именем *MadLib2.py*. Добавьте строку, которая бы запрашивала ввод названия животного с предлогом «на». Затем измените выражение `print`, убрав слово `собаку` и добавив переменную `animal` после предложения, взятого в кавычки (не забудьте в выражении `print` добавить запятую перед новой переменной). При желании вы можете изменить предложение еще сильнее. К примеру, `Эта веселая меловая доска рыгнула на ленивого коричневого геккона`, или что-нибудь еще более забавное!



Глава 2

«ЧЕРЕПАШЬЯ» ГРАФИКА: РИСОВАНИЕ С PYTHON

В этой главе мы напишем короткие и простые программы, с помощью которых создадим красивые и сложные орнаменты. Для этого мы воспользуемся *графикой* `turtle`. При работе с графикой `turtle` вы пишете инструкции для виртуальной, или воображаемой, черепашки*, сообщающие ей о том, как перемещаться по экрану. Черепашка носит с собой ручку, а вы можете проинструктировать черепашку воспользоваться этой ручкой для рисования линий по мере перемещения по экрану. Написав код, инструктирующий черепашку перемещаться крутыми траекториями, вы можете заставить ее нарисовать потрясающие картинки.

Используя графику `turtle`, вы не только можете создавать впечатляющие орнаменты с помощью всего лишь нескольких строк кода, но также можете отследить движение черепашки, дабы понять, каким образом каждая строка кода влияет на траекторию движения черепашки. Это поможет вам понять *логику* программного кода.

Наша первая программа `turtle`

Давайте напишем нашу первую программу с использованием графики `turtle`. Введите следующий код в новое окно программы IDLE и сохраните файл под именем *SquareSpiral1.py*. (Вы также можете скачать эту и другие программы из данной книги по адресу **https://eksmo.ru/files/Python_deti.zip.**)

* *Turtle* — черепаха (*англ.*). *Прим. пер.*

SquareSpiral1.py

```
# SquareSpiral1.py — Рисование квадратной спирали
import turtle
t = turtle.Pen()
for x in range(100):
    t.forward(x)
    t.left(90)
```

После запуска программы мы увидим на экране довольно приятную картинку (рис. 2.1).

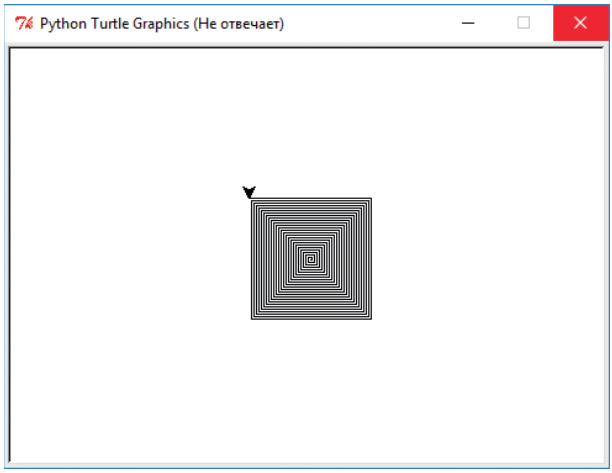
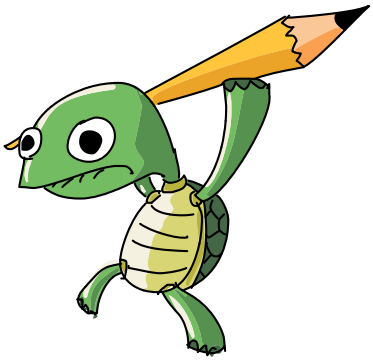


Рис. 2.1. *Гипнотическая квадратная спираль, созданная с помощью короткой программы SquareSpiral1.py*

Как это работает

Чтобы понять, как работает программа, давайте проанализируем ее код строка за строкой.



Первая строка программного кода в файле *SquareSpiral1.py* — это комментарий. Как вы уже узнали в главе 1, комментарий начинается с символа «решетка» (#). Комментарии позволяют нам писать в тексте программы заметки для самих себя или других людей, которые будут читать наш программный код в дальнейшем.

Компьютер не читает и не пытается понять то, что написано после символа «решетки». Комментарий — это заметка для нас, говорящая, какое действие выполняет программа. В данном случае я поместил в комментарий имя программы и краткое описание того, что она делает.

Вторая строка *импортирует* возможность рисовать с использованием графики `turtle`. Импортирование уже написанного кода — одна из самых крутых возможностей программирования. Если вы запрограммировали что-то интересное или полезное, то у вас есть возможность поделиться этим с другими людьми или же повторно использовать самому. Некоторые крутые программисты Python создают *библиотеку* — набор повторно используемого кода, чтобы помочь другим программистам использовать графику `turtle` на языке Python, несмотря на то что впервые графика `turtle` появилась в языке программирования Logo в 1960-х годах*. Когда вы вводите команду `import turtle`, вы сообщаете своей программе о намерении использовать код, написанный этими программистами Python. Маленькая черная стрелка на рис. 2.1 представляет черепашку, рисующую ручкой по мере передвижения по экрану.

Третья строка нашей программы, `t = turtle.Pen()`, сообщает компьютеру, что мы будем использовать букву `t` в качестве обозначения ручки черепашки. Такой подход позволит нам рисовать ручкой черепашки по мере ее перемещения по экрану путем ввода команды `t.forward()` вместо ввода команды полностью: `turtle.Pen().forward()`. Буква `t` будет нашим сокращением для передачи команд черепашке.

Четвертая строка самая сложная. Здесь мы создает *цикл*, который повторяет набор инструкций определенное количество раз (иными словами, программа *зацикливает* выполнение этих строк кода). Этот конкретный цикл устанавливает диапазон, или список из `100` чисел (от `0` до `99`). (Компьютеры почти всегда начинают отсчет с `0`, а не с `1`, как это делаем мы.) Цикл проводит букву `x` через все числа заданного диапазона таким образом, что `x` изначально равна `0`, затем она становится равной `1`, потом `2` и так далее — до тех пор, пока компьютер не досчитает до `99`, выполнив тем самым `100` шагов.

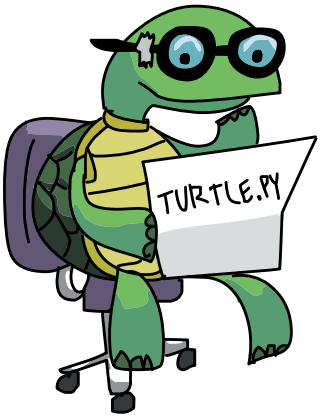
* Язык программирования Logo был создан в 1967 году как обучающий язык программирования, и даже теперь, спустя пять десятилетий, он еще актуален при изучении основ написания программного кода. Круто, да?

Буква x называется *переменной** (в программе *YourName.py* из главы 1 слово name было переменной). Переменная хранит значение, которое может меняться по мере выполнения программы. Мы будем использовать переменные практически во всех создаваемых программах, поэтому лучше познакомиться с ними пораньше.

Следующие две строки табулированы, то есть сдвинуты вправо относительно левой кромки. Это значит, что они находятся *внутри цикла* и относятся к строке над ними. Таким образом, эти строки текста будут повторно выполнены каждый раз, когда переменной x присваивается новое число в диапазоне от 0 до 99, то есть 100 раз.

Что происходит

Давайте посмотрим, что происходит, когда Python впервые читает данный набор инструкций. Команда `t.forward(x)` сообщает ручке черепашки передвинуться на x точек на экране. Так как x равняется 0, ручка не передвинется. Последняя строка, `t.left(90)`, говорит черепашке повернуть налево на 90 градусов.



Из-за наличия цикла for программа продолжает выполнение и возвращается на стартовую позицию цикла. Компьютер прибавляет 1, чтобы передвинуть переменную x на следующую позицию в диапазоне, и, так как 1 все еще находится в диапазоне от 0 до 99, цикл продолжает выполнение. Теперь переменная x равняется 1, поэтому ручка передвигается вперед на 1 точку, а затем вновь поворачивается влево на 90 градусов из-за наличия команды `t.left(90)`. Все это повторяется снова и снова. Цикл повторится в последний раз, когда переменная x будет равняться 99. К этому моменту ручка уже будет рисовать длинные линии по внешнему краю ранее отрисованной квадратной спирали.

Ниже приведена пошаговая визуализация цикла по мере того, как значение переменной x возрастает от 0 до 100:

* Маленькие читатели могут называть x неизвестным, наподобие того, значение которого вычисляется при решении уравнения x + 4 = 6. Более старшие читатели могут узнать x с уроков алгебры или других курсов математики, именно оттуда ранние программисты позаимствовали понятие «переменная». В написании кода много хорошей математики, в дальнейшем мы увидим даже несколько любопытных примеров из геометрии.

```
for x in range(100):
    t.forward(x)
    t.left(90)
```

- *Цикл от 0 до 4: Рисуются первые четыре линии (после x = 4).*
- *Цикл от 5 до 8: Рисуются еще четыре линии, на экране появляется квадрат.*
- *Цикл от 9 до 12: Квадратная спираль увеличивается до 12 линий (три квадрата).*

Точки, или *пиксели*, на экране вашего компьютера, скорее всего, слишком малы — и вы не можете четко видеть их. Но по мере приближения значения переменной `x` к `100` черепашка рисует линии, состоящие из все большего и большего количества пикселей. Иными словами, по мере увеличения значения переменной `x` команда `t.forward(x)` рисует все более длинные линии.

Стрелка черепашки на экране рисует линию, затем поворачивает налево, рисует еще немного, поворачивает налево — и снова рисует, и так повторяется снова и снова, при этом длина линий с каждым разом увеличивается.

К концу выполнения программы у нас на экране гипнотическая квадратная фигура. Четырехкратный поворот налево на 90 градусов позволяет нам сформировать квадрат, аналогично тому, как четыре поворота вокруг здания вернут нас к точке начала движения.

Однако в нашем примере рисуется спираль, потому что при каждом повороте налево мы проходим чуть больше, чем в предыдущий раз. Длина первой нарисованной нами линии составляет всего один шаг (когда `x = 1`), потом два шага (при следующем повторении цикла), потом три, четыре и так далее до 100 шагов, когда длина линии составит 99 пикселей. Опять же, пиксели на вашем экране так малы, что вы не можете различить отдельные точки, но они там есть. Кроме того, вы можете наблюдать, что линии становятся все длиннее при прибавлении пикселей.

Делая все повороты на 90 градусов, мы получаем идеальную квадратную фигуру.

Черепашка в ударе

Давайте посмотрим, что произойдет, если мы изменим одно из чисел в программе. Один из способов узнать о программе что-то новое — это посмотреть, что произойдет, если поменять одну из ее частей. Не всегда результат порадует вас, но вы можете узнать что-то новое, даже если что-то пойдет не так.

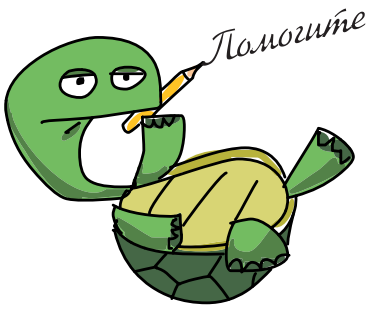
Измените только последнюю строку программы на `t.left(91)` и сохраните изменения в файле *SquareSpiral2.py*.

SquareSpiral2.py

```
import turtle
t = turtle.Pen()
for x in range(100):
    t.forward(x)
    t.left(91)
```

Я уже говорил, что поворот на 90 градусов создает идеальный квадрат. Поворот на угол чуть больше 90 градусов, в данном случае на 91 градус, немного искажает квадрат. И поскольку к следующему повороту квадрат уже немного искажен, новая форма выглядит все менее и менее похожей на квадрат по мере выполнения программы. В результате такое искажение создает приятную спиралевидную фигуру, которая, как вы можете видеть на рис. 2.2, начинает сворачиваться, наподобие винтовой лестницы.

Этот пример также является хорошей наглядной демонстрацией того, как ошибка лишь в одном числе может привести к большим изменениям результата выполнения программы. Один градус кажется незначительным, если только вы не сбились на один градус 100 раз (что в сумме дает 100 градусов) или 1000 градусов или если вы используете программу, чтобы посадить самолет...



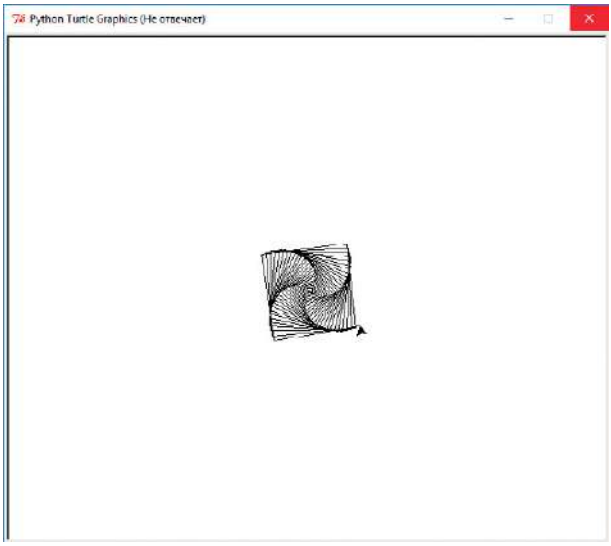


Рис. 2.2. Благодаря одному небольшому изменению программа из квадратной спирали превращается в винтовую лестницу

Если вы еще не знаете, что такое градусы, не переживайте: просто поиграйте с числами и понаблюдайте за изменениями. Увеличьте количество рисуемых программой линий до 200, 500 или 50, меняя значение в скобках после ключевого слова `range`.

Также попробуйте поменять угол в последней строке на 91, 46, 61 или 121 градус и так далее. Помните о необходимости сохранять программу каждый раз. Затем запустите отредактированную программу и посмотрите, как изменился рисуемый ею орнамент. Старшие читатели, знакомые с основами геометрии, увидят знакомые фигуры, зависящие от установленных углов, и, возможно, смогут предсказать фигуру по выбранному углу еще до начала выполнения программы. Младшие читатели могут получить удовольствие от изменения параметров программы, и это упражнение, возможно, всплывет в их памяти когда-нибудь на уроке геометрии.

Черепашка закругляется

Графика `turtle` способна рисовать и более интересные фигуры, чем просто прямые линии. Мы еще вернемся к квадратной фигуре в следующем разделе, а пока свернем немного в сторону и посмотрим еще некоторые возможности библиотеки Turtle языка Python.

Давайте изменим еще одну строку кода: `t.forward(x)`. Ранее мы видели, что эта команда, или *функция*, перемещает ручку черепашки вперед на `x` пикселей и рисует отрезок прямой линии, затем черепашка поворачивает и повторяет это действие снова. А если мы изменим эту строку кода для изображения чего-то более сложного, например окружности?

К счастью для нас, команда для рисования окружности заданного размера, то есть *радиуса*, так же проста, как команда для изображения прямой линии. Замените `t.forward(x)` на `t.circle(x)`, как показано в листинге кода ниже.

CircleSpiral1.py

```
import turtle
t = turtle.Pen()
for x in range(100):
    t.circle(x)
    t.left(91)
```

Ого! Изменение всего одной команды с `t.forward(x)` на `t.circle(x)` дало нам гораздо более сложную фигуру (как можно видеть на рис. 2.3). Функция `t.circle(x)` дает команду программе нарисовать окружность с радиусом `x` в текущей позиции. Обратите внимание, что получившийся рисунок имеет нечто общее с квадратной спиралью: на рисунке видны четыре набора круглых спиралей, что совпадает с количеством сторон квадратной спирали в прошлом примере. Это происходит потому, что мы по-прежнему поворачиваем влево на угол чуть больше 90 градусов каждый раз при выполнении команды `t.left(91)`. Если вы изучали геометрию, то знаете, что вокруг каждой точки можно выполнить поворот на 360 градусов, что также соответствует четырем 90-градусным углам квадрата ($4 \times 90 = 360$). Черепашка отрисовывает эту спиралевидную фигуру, поворачивая каждый раз влево на угол чуть больше 90 градусов.

Вы можете заметить одно отличие: круглая спираль больше квадратной чуть ли не в два раза. Это произошло потому, что команда `t.circle(x)` использует значение переменной `x` в качестве *радиуса* окружности, являющегося расстоянием от центра окружности до ее края, или половиной ширины окружности. Радиус, заданный переменной `x`, означает, что *диаметр* окружности, или ее ширина, будет в два раза больше переменной `x`. То есть команда `t.circle(x)` отрисовывает окружность шириной 2 пикселя,

когда x равняется 1, 4 пикселя, когда x равняется 2, и так до достижения ширины окружности 198 пикселей при x = 99. А это почти 200 пикселей в ширину, что почти в два раза больше длины стороны самого большого квадрата. Таким образом, круглая спираль почти в два раза больше квадратной спирали и, может быть, в два раза круче!

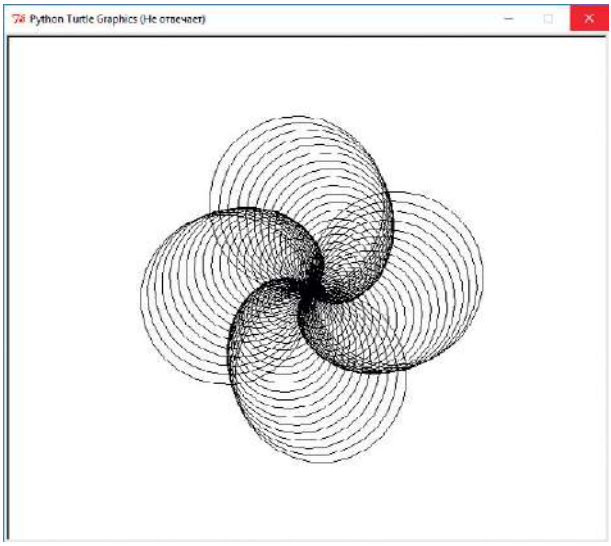


Рис. 2.3. *Еще одно небольшое изменение дает нам красивый набор из четырех спиралевидных окружностей*

Добавим красок

Спирали, конечно, приятные фигуры, но было бы лучше сделать их чуть более красочными? Давайте вернемся к коду квадратной спирали и добавим в программу еще одну строку, сразу после строки `t = turtle.Pen()`, строку, устанавливающую красный цвет ручки:

SquareSpiral3.py

```
import turtle
t = turtle.Pen()
t.pencolor("red")
for x in range(100):
    t.forward(x)
    t.left(91)
```

Запустите программу — и вы увидите более яркую версию квадратной спирали (рис. 2.4).

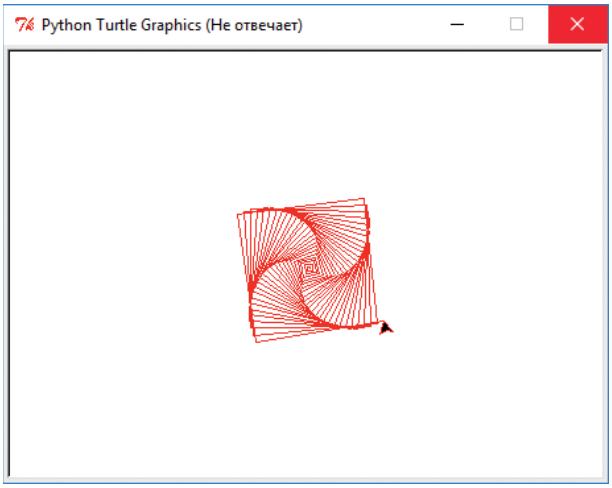


Рис. 2.4. *Квадратная спираль становится более цветной*

Попробуйте заменить параметр `"red"` (красный) другим простым цветом, например синим (`"blue"`) или зеленым (`"green"`), — и вновь запустите программу. С библиотекой Turtle вы можете использовать большое количество разнообразных цветов, даже такие странные, как «лососевый» (`"salmon"`) или «лимонный шифон» (`"lemon chiffon"`). Посетите сайт **www.tcl.tk/man/tcl8.4/TkCmd/colors.htm**, чтобы просмотреть полный список цветов. Перекрашивание всей спирали — неплохой шаг, но если мы захотим перекрасить каждую *сторону* спирали в отдельный цвет? Для этого нам нужно внести в программу еще несколько изменений.

Четырехцветная спираль

Давайте обдумаем *алгоритм* (то есть набор шагов), который бы превратил нашу одноцветную спираль в четырехцветную. Большинство шагов совпадают с шагами рассмотренных ранее программ, но в этом случае нам нужно добавить несколько поворотов.

1. Импортировать модуль `turtle` и установить черепашку.
2. Сообщить компьютеру, какие цвета мы хотели бы использовать.
3. Настроить цикл на отрисовку 100 линий спирали.

4. Выбрать отдельный цвет для каждой из сторон спирали.
5. Передвигать черепашку вперед для отрисовки каждой стороны.
6. Повернуть черепашку влево и приготовить ее для начала рисования следующей стороны.

В первую очередь нам потребуется использовать *список* цветов, а не один, поэтому мы создадим переменную-список с именем `colors` и поместим в этот список четыре цвета:

```
colors = ["red", "yellow", "blue", "green"]
```

Этот список из четырех цветов будет давать отдельный цвет для каждой из сторон квадрата. Обратите внимание, что мы поместили список цветов в квадратные скобки `[` и `]`. Убедитесь, что название каждого цвета заключено в кавычки, как слова, которые мы выводили на печать в главе 1, потому что эти названия цветов являются *строками*, или текстовыми значениями, которые мы в скором времени передадим функции `pencolor`. Как говорилось ранее, мы используем переменную `colors` для хранения списка из четырех цветов, поэтому, когда бы нам ни потребовалось воспользоваться одним из цветов из списка, мы используем для обозначения цвета ручки переменную `colors`. Запомните, что переменные хранят значения, которые могут изменяться. Эта особенность отражена прямо в названии: «переменная» — от слова «перемена».

Следующее, что нам нужно сделать, — изменять цвет ручки *каждый раз*, когда программа проходит через цикл отрисовки. Чтобы сделать это, нам необходимо перенести функцию `t.pencolor()` в группу инструкций, расположенную под циклом `for`. Нам также необходимо сообщить функции `pencolor()`, какой именно цвет из представленных в списке мы хотим использовать.

Введите следующий код и запустите его.

ColorSquareSpiral.py

```
import turtle
t = turtle.Pen()
colors = ["red", "yellow", "blue", "green"]
for x in range(100):
    t.pencolor(colors[x%4])
    t.forward(x)
    t.left(91)
```

Использование списка из четырех цветов логично, поэтому мы применим его в запускаемом примере (рис. 2.5). Чем дальше, тем лучше.

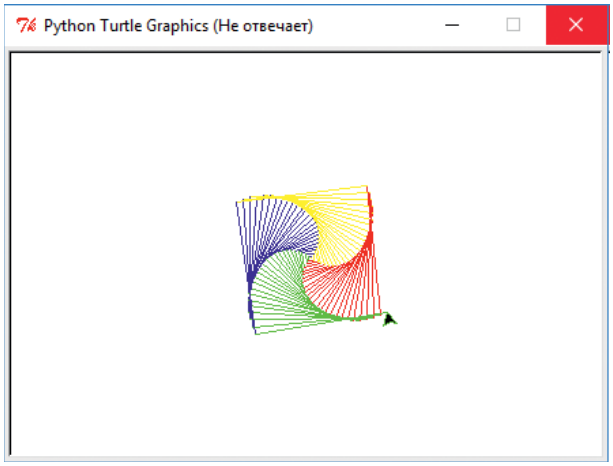


Рис. 2.5. *Гораздо более полноцветная версия программы квадратной спирали*

Единственное нововведение в программе — параметр `(colors[x%4])` функции `pencolor`. `x` внутри выражения — это та же самая переменная, которую мы используем в других программах, поэтому значение `x` будет увеличиваться от `0` до `99`, как и раньше. Имя переменной `colors` в скобках сообщает Python о необходимости использования списка цветов с именем `colors`, который мы добавили в программу.

Выражение `[x%4]` сообщает Python о том, что мы будем использовать первые четыре цвета списка `colors`, пронумерованные от 0 до 3, и выбирать один из этих цветов каждый раз при изменении значения переменной `x`. Сейчас наш список состоит только из четырех цветов, поэтому мы будем проходить по нему снова и снова:

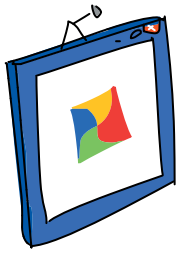
```
colors = ["red", "yellow", "blue", "green"]
           0       1         2        3
```

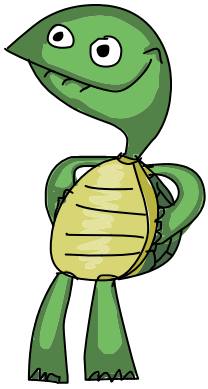
Символ % в выражении `[x%4]` представляет операцию *взятие по модулю*, результатом выполнения которой будет *остаток* от деления (5 ÷ 4 равняется 1 с остатком 1, так как 4 можно нацело разделить на 5 только с остатком 1, остаток от деления 6 ÷ 4 равняется 2, и так далее). Оператор взятия по модулю может быть полезен при необходимости циклично пройти по нескольким элементам списка, аналогично тому, как мы поступаем со списком из четырех цветов.

За 100 шагов выражение `colors[x%4]` пройдет по всем четырем цветам (0, 1, 2 и 3 для красного, желтого, синего и зеленого) 25 раз. Если у вас есть время (и увеличительное стекло), то вы можете насчитать 25 красных, 25 желтых, 25 синих и 25 зеленых сегментов на рис. 2.5. При первом проходе по циклу рисования Python использует первый цвет в списке, красный. При втором проходе используется желтый, и так далее. Затем, при пятом проходе, Python вновь вернется к красному цвету, потом к желтому, и так далее. После каждого четвертого прохода цикл будет вновь возвращаться к красному цвету.

Изменение цветов фона

Давайте опять что-нибудь изменим и посмотрим, сможем ли мы создать что-то еще более красивое, чем орнамент на рис. 2.5. Например, как заметил мой пятилетний сын Алекс, желтые стороны очень плохо видны. Как желтый карандаш на белой бумаге, желтые пиксели на экране практически не видны из-за белого цвета фона. Давайте исправим это, заменив цвет фона на черный. Введите следующую строку в любом участке программы после команды `import`:





```
turtle.bgcolor("black")
```

Благодаря всего одной строчке кода у нас на экране появилась более

приятная картинка: теперь все цвета очень отчетливо видны на черном фоне. Обратите внимание, что мы не изменили ни одного параметра ручки черепашки (представленной в программе переменной `t`). Вместо этого мы изменили параметры экрана черепашки, а именно цвет фона. Команда `turtle.bgcolor()` позволяет изменять цвет всего экрана для рисования на любой поименованный цвет Python. В строке `turtle.bgcolor("black")` в качестве цвета фона мы выбрали черный цвет, поэтому красный, желтый, синий и зеленый цвета теперь выглядят очень красиво.

Пока мы не закрыли эту программу, мы можем изменить диапазон range() нашего цикла до 200 или еще большего значения, чтобы спираль отрисовывалась с большими сторонами. На рис. 2.6 изображена новая версия картинки с 200 линиями и черным фоном.

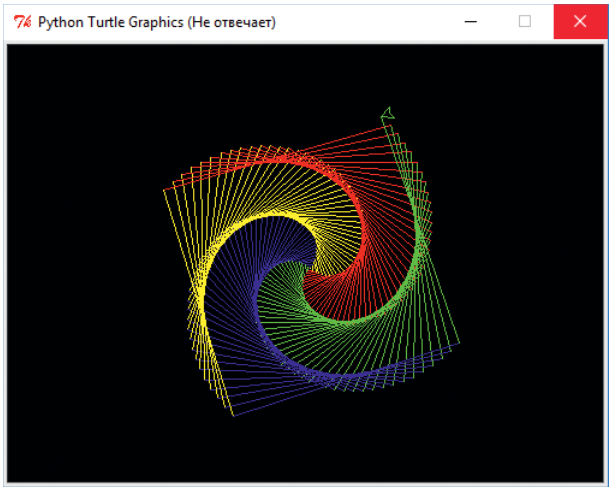


Рис. 2.6. *Наша квадратная спираль прошла долгий путь из простого начала*

Всегда желающий усовершенствовать мои программы Алекс попросил меня внести еще одно изменение: что если мы теперь заменим отрезки прямых линий на окружности? Не будет ли это самой крутой картинкой из всех? Должен согласиться: такая картинка еще круче. Вот код этой программы:

ColorCircleSpiral.py

```
import turtle
t = turtle.Pen()
turtle.bgcolor("black")
```

```
colors = ["red", "yellow", "blue", "green"]
for x in range(100):
    t.pencolor(colors[x%4])
    t.circle(x)
    t.left(91)
```

Результат выполнения этой программы показан на рис. 2.7.

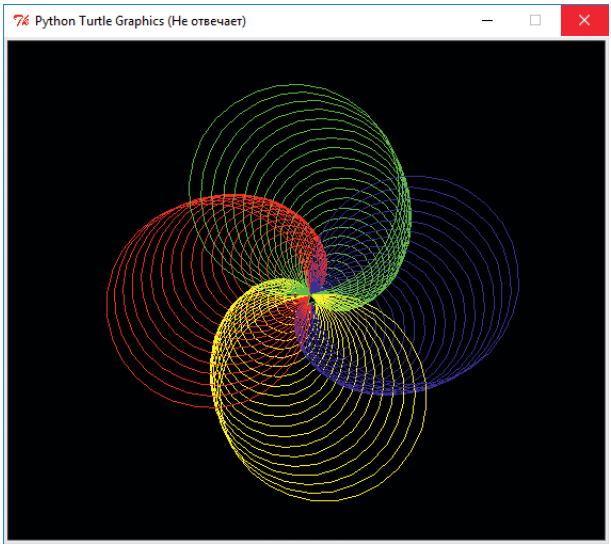


Рис. 2.7. *Восемь простых и элегантных строк кода от Алекса, рисующие потрясающую цветную круглую спираль*

Одна переменная, управляющая всей программой

Мы уже использовали переменные для изменения цвета, размера и угла поворота спиралевидных фигур. Давайте добавим еще одну переменную, `sides`, которая будет устанавливать количество сторон фигуры. Как эта новая переменная изменит нашу спираль? Чтобы узнать, запустите эту программу, *ColorSpiral.py*.

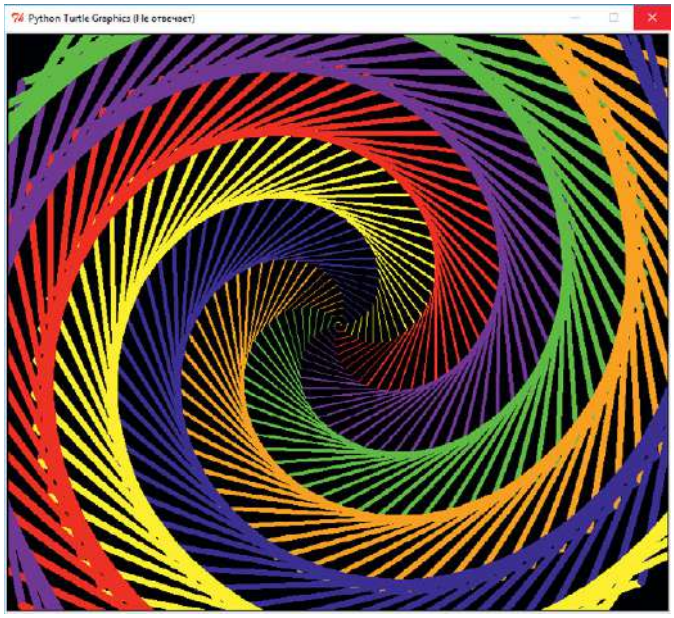
ColorSpiral.py

```
import turtle
t = turtle.Pen()
turtle.bgcolor("black")
# Вы можете задать от 2 до 6 граней — и получить крутые
# фигуры!
```

```
sides = 6
colors = ["red", "yellow", "blue", "orange", "green", ↵
          "purple"]
for x in range(360):
    t.pencolor(colors[x%sides])
    t.forward(x * 3/sides + x)
    t.left(360/sides + 1)
    t.width(x*sides/200)
```

Вы можете уменьшить значение переменной `sides` с `6` до `2` (рисовать одну сторону не очень интересно, а большие числа использовать вы не сможете, если не добавите в шестую строку кода другие цвета). Изменяйте, сохраняйте и запускайте программу сколько угодно. На рис. 2.8 показаны картинки, созданные программой при `sides = 6`, `sides = 5` и так далее вплоть до `sides = 2`, странной плоской спирали, показанной на рис. 2.8д. Вы также можете изменить порядок цветов, увеличить или уменьшить числа в любой функции цикла рисования. Если вы сломаете программу, просто вернитесь к исходному файлу *ColorSpiral.py* и поиграйте еще немного.

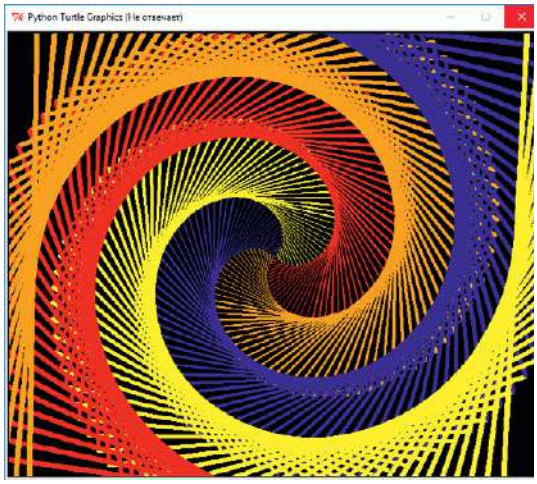
В программе *ColorSpiral.py* мы использовали одну новую команду: `t.width()`. Эта команда изменяет толщину ручки черепашки. В нашей программе ручка становится все шире по мере увеличения длины рисуемых ею фигур. Мы еще вернемся к этой и подобным программам в главах 3 и 4, когда вы приобретете навыки создания таких программ с нуля.



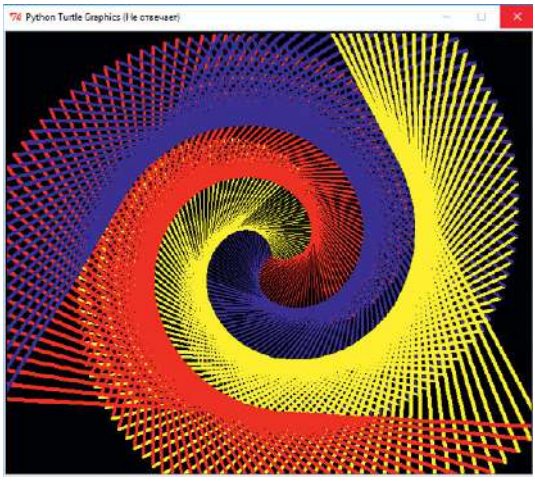
а

Python Turtle Graphics (Не отвечает)

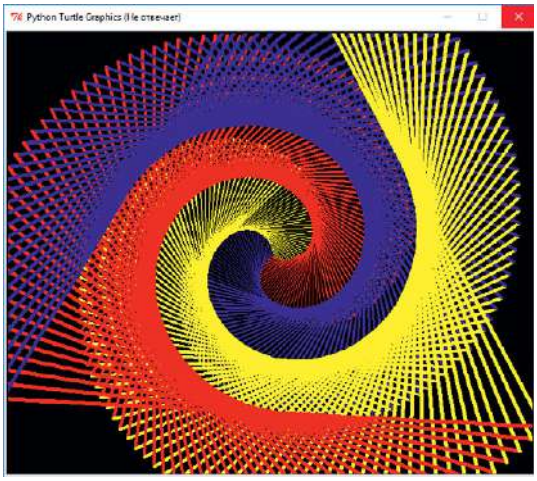
б


Python Turtle Graphics (Не отвечает)

в


Python Turtle Graphics (Не отвечает)

г



д

Рис. 2.8. *Пять цветных фигур, созданных путем значения переменной* `sides` *от 6 (а) до 2 (д)*

Что вы узнали

В этой главе мы нарисовали несколько впечатляющих цветных фигур на языке `Python` с помощью библиотеки инструментов Turtle. Мы подключили эту библиотеку к нашей программе с помощью команды `import`, вы также узнали, что повторное использование кода таким образом — одна из мощнейших возможностей программирования. Написав что-то полезное или позаимствовав код, которым щедро поделился другой человек, мы не только экономим время, но и можем создавать новые элегантные решения, импортировав этот код.

Вас также познакомили с переменными, такими как `x` и `sides`, используемыми в наших программах. Эти переменные сохраняют, или запоминают, число либо значение, которое мы можем многократно использовать и даже изменять при выполнении программы. В следующей главе вы узнаете больше о возможностях переменных, а также о том, как `Python` может помочь вам с выполнением домашнего задания по математике!

На данный момент вы должны уметь делать следующее.

- Рисовать простую графику с помощью библиотеки Turtle.
- Использовать переменные для сохранения числовых значений и строк.
- Изменять, сохранять и запускать программы в IDLE.

Задачи по программированию

Попробуйте решить эти задачи, чтобы попрактиковаться и закрепить знания, полученные в этой главе. (Если вы зайдете в тупик, вы можете перейти на сайт **https://eksmo.ru/files/Python_deti.zip** для получения готовых решений.)

#1. Изменение количества сторон

На с. 39 в программе *ColorSpiral.py* мы использовали переменную `sides`, а также несколько раз изменяли значение этой переменной, редактируя текст программного кода, сохраняя его и снова запуская программу. Попробуйте изменить значение переменной `sides`, например сделав его равным `5`. Сохраните программу и запустите ее вновь, чтобы увидеть, как это отразилось на рисунке. Теперь попробуйте значения `4`, `3`, `2` и даже `1`! Теперь к списку цветов на шестой строке программного кода добавьте два или больше цветов, в кавычках, разделенные запятыми. Увеличьте количество сторон, чтобы использовать порядковые номера новых цветов, — попробуйте задать 8, 10 или больше граней.

#2. Сколько сторон?

А если вы хотите, чтобы пользователь мог самостоятельно задать количество сторон при запуске программы? Используя знания, полученные в главе 1, вы можете попросить пользователя ввести количество сторон и сохранить введенные данные в переменной `sides`. Нашим следующим дополнительным шагом будет *оценка* числа, введенного пользователем. Функция `eval()` позволит нам узнать, какое число ввел пользователь:

```
sides = eval(input("Введите количество сторон от 2 ↵
до 6: "))
```

В программе *ColorSpiral.py* замените строку `sides = 6` на вышеприведенную. Ваша новая программа спросит, сколько сторон пользователь хотел бы отобразить, а затем программа нарисует запрашиваемую пользователем фигуру. Опробуйте!

#3. Мячик из цветных резинок

Попытайтесь поменять программу *ColorSpiral.py* для отрисовки более сложной и абстрактной фигуры, добавив еще один поворот внутри цикла рисования. Добавьте такую строку, как `t.left(90)`, в конец цикла `for`, чтобы сделать углы фигуры более острыми (не забудьте сделать отступ, чтобы выражение оставалось в цикле). Результат, показанный на рис. 2.9, похож на некую геометрическую игрушку или, возможно, мячик, сделанный из цветных канцелярских резинок для денег.

Рис. 2.9. *Добавление дополнительных 90-градусных поворотов превращает программу ColorSpiral.py в RubberBandBall.py*

Сохраните программу в файл с именем *RubberBandBall.py*. Также на сайте **https://eksmo.ru/files/Python_deti.zip** вы можете найти исходный код этой программы.

Глава 3

ЧИСЛА И ПЕРЕМЕННЫЕ: PYTHON ДЕЛАЕТ ПОДСЧЕТЫ

Мы использовали язык Python для создания интересных программ, рисующих цветные картинки за счет всего нескольких строк кода, но наши программы были ограничены в возможностях. Мы просто запускали их и наблюдали, как они сами создают картинки. А что если бы мы захотели *взаимодействовать* с нашими программами на Python? В этой главе мы узнаем, как заставить Python запросить имя пользователя и даже предложить ему сделать домашнее задание по математике!

Переменные: место, где мы храним данные

В главах 1 и 2 мы использовали *переменные* (возможно, вы помните переменную `name` из нашей первой программы в главе 1 или переменные `x` и `sides` из главы 2). Теперь давайте посмотрим, что такое в действительности переменные и как они работают.

Переменная — это нечто такое, что нужно запомнить вашему компьютеру на время работы программы. Когда Python «запоминает» что-либо, он сохраняет эту информацию в памяти компьютера. Python может запоминать *значения* нескольких типов, например числовые значения (такие как `7`, `42` или даже `98.6`) и строки (буквы, символы, слова, предложения — все, что вы можете набрать на клавиатуре). В Python, как

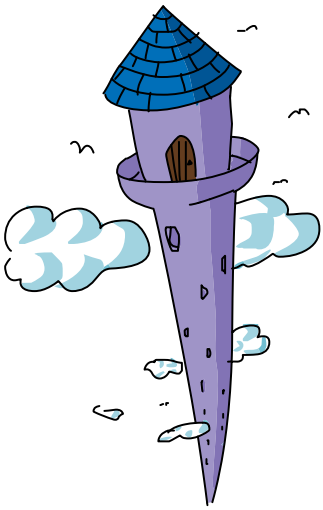
и в большинстве современных языков программирования, мы *присваиваем* значение переменной с помощью знака равенства (=). Выражение присваивания, такое как `x = 7`, говорит компьютеру запомнить число 7 и вывести его при каждом вызове переменной `x`. Мы также используем знак равенства для присвоения переменной строки клавиатурных символов, главное — не забыть заключить присваиваемую строку в кавычки (") как показано ниже:

```
my_name = "Брайсон"
```

Выше мы присвоили значение `"Брайсон"` переменной `my_name`. Кавычки до и после слова `"Брайсон"` говорят о том, что перед нами строка.

Каждый раз при присвоении переменной какого-либо значения сначала (слева от знака равенства) необходимо указать имя этой переменной, а затем (справа от знака равенства) — присваиваемое значение. Как правило, переменные называются простыми именами, описывающими содержащееся в переменной значение (как в случае с переменной `my_name`, хранящей мое имя). Такой подход упрощает запоминание имен переменных и их последующее использование. Однако следует запомнить несколько правил, которым должны соответствовать имена создаваемых переменных.

Во-первых, имя переменной должно начинаться с буквы*.



Во-вторых, остальные символы в имени переменной должны быть буквами, цифрами или нижним подчеркиванием (_). Внутри имени переменной пробелы не допускаются (например, имя `my name` приведет к появлению сообщения о синтаксической ошибке, так как Python подумает, что вы перечислили две переменные и разделили их пробелом).

В-третьих, имена переменных в Python *чувствительны к регистру*. Это значит, что если в имени переменной мы использовали только буквы *нижнего регистра* (например, `abc`),

* Здесь и далее имеются в виду буквы латинского алфавита, т.к. использование букв русского алфавита в именах переменных является недопустимым. *Прим. пер.*

то воспользоваться значением, сохраненном в переменной, мы можем, только указав имя переменной в точно таком же виде, в том же регистре. Например, чтобы воспользоваться значением переменной `abc`, нам необходимо ввести с клавиатуры `abc`: использовать соответствующие буквы *верхнего регистра* (`ABC`) нельзя. Таким образом, переменная `My_Name` отличается от `my_name`, а `MY_NAME` отличается от них обеих. В этой книге для имен переменных мы будем использовать только буквы нижнего регистра и разделять слова символом `_`.

Давайте попробуем создать программу, использующую несколько переменных. Введите следующий программный код в новое окно IDLE и сохраните его с именем *ThankYou.py*.

ThankYou.py.

```
my_name = "Брайсон"
my_age = 43
your_name = input("Как тебя зовут? ")
your_age = input("Сколько тебе лет? ")
print("Меня зовут", my_name, ", и мне", my_age, "лет.")
print("Тебя зовут", your_name, ", и тебе", your_age, ".")
print("Спасибо за покупку моей книги,", your_name, "!")
```

При запуске программы мы говорим компьютеру запомнить имя `"Брайсон"` в переменной `my_name` и возраст `43` года в `my_age`. Затем мы просим пользователя (человека, запустившего данную программу) ввести его имя и возраст, а затем говорим компьютеру запомнить введенные данные в переменных `your_name` и `your_age`. Мы используем функцию языка Python `input()` для сообщения языку Python, что мы хотим, чтобы пользователь *ввел* данные с клавиатуры. *Вводом* называется информация, переданная в программу во время выполнения, — в данном случае имя пользователя и возраст. Предложение в скобках, заключенное в кавычки (`"Как тебя зовут? "`), называется *приглашением*, так как оно приглашает пользователя, или, иными словами, задает ему вопрос, ответ на который требуется ввести с клавиатуры.

В последних трех строках программы мы сообщим компьютеру вывести на печать значение, сохраненное в переменной `my_name` и трех других переменных. Переменной `your_name` мы даже воспользуемся дважды, а компьютер все правильно запомнит, в том числе и то, что пользователь ввел с клавиатуры.

Эта программа запоминает мое имя и возраст, запрашивает имя и возраст пользователя, а затем выводит на печать приятное сообщение, адресованное пользователям, как показано на рис. 3.1.

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
Python 3.2.5 (default, May 15 2013, 23:06:03) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Как тебя зовут? Алекс
Сколько тебе лет? 5
Меня зовут Брайсон , и мне 43 лет.
Тебя зовут Алекс , и тебе 5 .
Спасибо за покупку моей книги, Алекс !
>>> |
                                                                Ln: 10 Col: 4
```

Рис. 3.1. *Программа с четырьмя переменными и создаваемый ею вывод*

Числа и математика в Python

У компьютера есть потрясающий талант запоминать значения. Мы можем использовать одну и ту же переменную в одной и той же программе тысячи раз — и компьютер будет всякий раз выдавать нам правильное значение, при условии, что мы правильно его запрограммировали. Компьютеры также очень хороши в математических операциях (сложение, вычитание и так далее). Ваш компьютер способен совершать более *одного миллиарда* (1 000 000 000, или тысячи миллионов) математических операций в *секунду*!

Это гораздо быстрее, чем наша возможность вычислять числа в уме. Хотя со многими задачами мы все еще справляемся лучше, скоростной счет — это соревнование, в котором компьютер всегда одержит победу. Язык Python дает нам доступ к этому мощному ресурсу математических вычислений, предоставляя для этого числа двух основных видов. Кроме того, язык позволяет вам использовать весь набор символов для выполнения математических действий над этими числами, начиная от + и − и до более сложных.

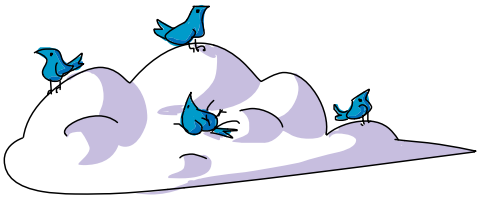
Числа Python

Два основных вида чисел, использующихся в языке Python, называются *целыми числами* (целые числа, включая отрицательные числа, такие как `7`, `−9` или `0`) и *числами с плавающей точкой* (десятичные дроби, например `1.0`, `2.5`, `0.999` или `3.14159265`). Существуют также два дополнительных

вида чисел, которые мы практически не используем в данной книге. Первым видом являются *Булевы*, которые хранят значения ИСТИНА или ЛОЖЬ (наподобие ответов на задания «Истина или Ложь» в школьных тестах), а вторым видом — *комплексные числа*, которые могут содержать даже воображаемые числовые значения (это может восхитить вас, если вы несколько знакомы с высшей математикой, однако в этой книге мы стараемся не отрываться от реальности).

Целые числа полезны при счете (переменная `x` из главы 2 отсчитывала количество линий рисуемой спирали), а также для базовых математических подсчетов (`2 + 2 = 4`). Как правило, мы указываем свой возраст с помощью целых чисел, то есть, когда вы говорите, что вам 5, 16 или 42, вы используете целые числа. Когда вы считаете до 10, вы также используете целые числа.

Числа с плавающей точкой, или десятичные дроби, идеально подходят, когда мы хотим указать только часть чего-либо, например 3.5 мили, 1.25 пиццы или $25.97. Конечно, на языке Python мы не будем указывать единицы (мили, пиццы, доллары), только число с дробной частью. Таким образом, если мы хотим сохранить в переменную стоимость нашей пиццы (`cost_of_pizza`), то мы можем присвоить соответствующее значение следующим образом: `cost_of_pizza = 25.97`. Нам лишь нужно запомнить, что используемые единицы — это доллары, евро или любая другая валюта.



Операторы Python

Математические символы, такие как + (плюс) и − (минус), называются *операторами*, так как они оперируют (или выполняют вычисления) числами в уравнении. Когда мы говорим вслух «4 + 2», чтобы ввести это на калькуляторе, мы хотим сложить числа 4 и 2, чтобы получить их сумму, 6.

В языке Python используется большинство операторов, которыми вы пользуетесь на уроках математики, в том числе +, − и скобки, `()`, как показано

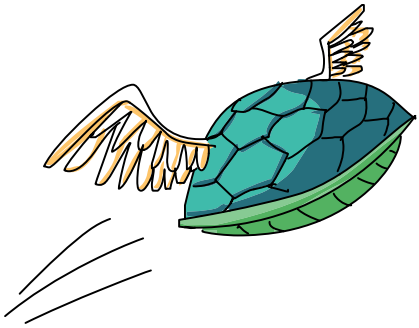
в табл. 3.1. Однако, некоторые операторы отличаются от используемых в школе: так, оператор умножения представлен звездочкой (*) вместо ×, а оператор деления — косой чертой, /, вместо ÷). В этом разделе мы более подробно познакомимся с перечисленными операторами.

Таблица 3.1. Основные математические операторы в языке Python

Математический символ	Оператор Python	Операция	Пример	Результат
+	+	Сложение	`4 + 2`	6
−	-	Вычитание	`4 - 2`	2
×	*	Умножение	`4 * 2`	8
÷	/	Деление	`4 / 2`	2.0
4^2	**	Экспонента или степень	`4 ** 2`	16
()	()	Скобки (группировка)	`(4 + 2) * 3`	18

Математические вычисления в оболочке Python

Сейчас самое время испытать математику Python, в этот раз давайте воспользуемся оболочкой Python. Как вы, возможно, помните из главы 1, оболочка Python дает вам прямой доступ к возможностям Python без необходимости написания целой программы. Иногда оболочку называют *командной строкой*, так как вы можете вводить команды строка за строкой и сразу видеть результат их выполнения. Вы можете ввести математическую задачу (называемую в программировании *выражением*), например 4 + 2, непосредственно в строку с приглашением (символ >>> с мигающим курсором) в оболочке Python. Затем, по нажатию клавиши **Enter**, вы увидите *результат* данного выражения, или ответ на задачу.



Попробуйте ввести несколько примеров, перечисленных в табл. 3.1, и посмотрите, какой ответ даст Python. На рис. 3.2 показан образец ответов. Не стесняйтесь также пробовать вводить и собственные уравнения.

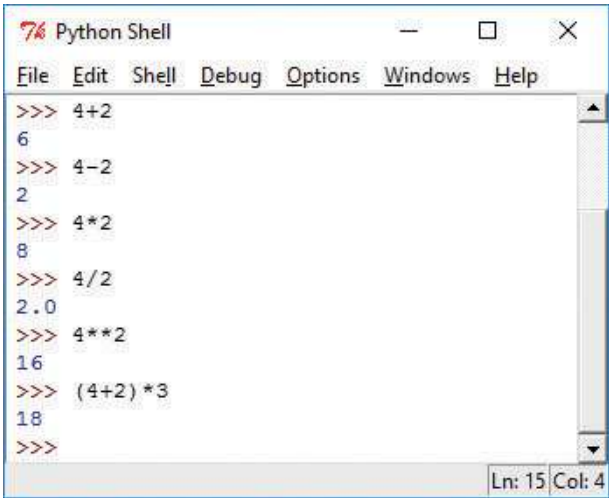


Рис. 3.2. *Введите примеры математических задач (выражений) из табл. 3.1, а Python даст на них ответы!*

Синтаксические ошибки: что вы сказали?

Вводя команды в оболочке Python, вы можете узнать, что такое *синтаксические ошибки*. Если Python или любой другой язык программирования не может понять введенную вами команду, то он может вывести в ответ сообщение об ошибке с текстом, например, `"Syntax Error"`. Это означает, что возникла проблема с тем, *каким образом* вы попросили компьютер выполнить какое-либо задание, то есть с синтаксисом.

Синтаксис — это набор правил, которым необходимо следовать при создании предложений или *выражений* на языке. В случае с компьютерным программированием мы называем ошибку в выражении синтаксической ошибкой, если мы делаем ошибку в предложении на английском языке, мы можем назвать ее грамматической ошибкой. Разница в том, что, в отличие от людей, говорящих на английском языке, компьютеры не воспринимают плохую грамматику *в принципе.* Python, как и большинство языков программирования, очень хорош в вычислениях, но только при условии, что мы следуем правилам синтаксиса, ведь компьютер не распознает ничего из сказанного нами с ошибками в синтаксисе. Взгляните на рис. 3.3, здесь

даны примеры синтаксических ошибок, после которых указаны выражения в формате, воспринимаемом Python.

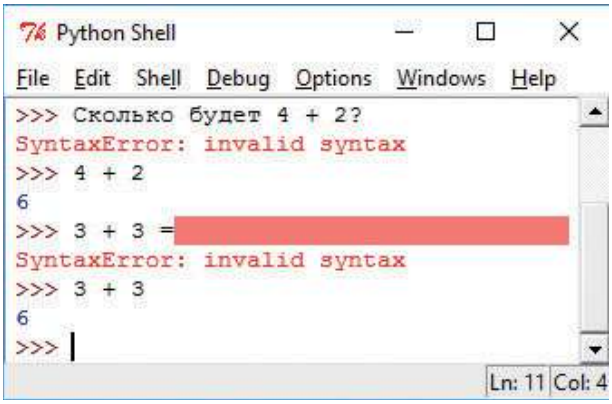


Рис. 3.3. *Обучаемся говорить на языке Python*

Мы можем спросить Python по-русски «Сколько будет 4 + 2?», на что Python ответит: `"SyntaxError: invalid syntax"`, дабы показать нам, что он не понял, что мы попросили его сделать. Если мы даем Python корректное выражение, `4 + 2`, то Python каждый будет давать верный ответ: `6`. Даже один лишний символ, как знак равно в конце выражения `3 + 3 =`, сбивает Python с толку, так как Python видит знак равенства как оператор присваивания, присваивающий значение переменной. Если мы введем `3 + 3` и нажмем клавишу **Enter**, Python поймет нашу команду и всегда даст правильный ответ: `6`.

Мы можем положиться на компьютер в том, что он ответит правильно и быстро каждый раз, когда мы представляем ему правильный ввод. Этот факт является одной из важнейших особенностей программирования. Мы можем переложить на компьютер задачи по быстрым и точным подсчетам — при условии, что мы правильно запрограммировали эти компьютеры на языке, который они понимают. Именно этому вы обучаетесь, изучая программирование на языке Python.

Переменные в оболочке Python

Как мы уже обсуждали, оболочка Python дает нам прямой доступ к возможностям программирования на языке Python без необходимости писать полноценные самостоятельные программы. При вводе в оболочку Python мы также можем использовать переменные, такие как `x` и `my_age`, все, что нам нужно, — присвоить им определенные значения, чему вы научились в первом примере этой главы.

Если вы введете x = 5 в строке приглашения (>>>), Python сохранит значение 5 в памяти компьютера как переменную x и будет помнить это значение до тех пор, пока вы не скажете Python поменять его (например, ввод x = 9 присвоит переменной x новое значение 9). На рис. 3.4 показаны примеры работы оболочки Python.

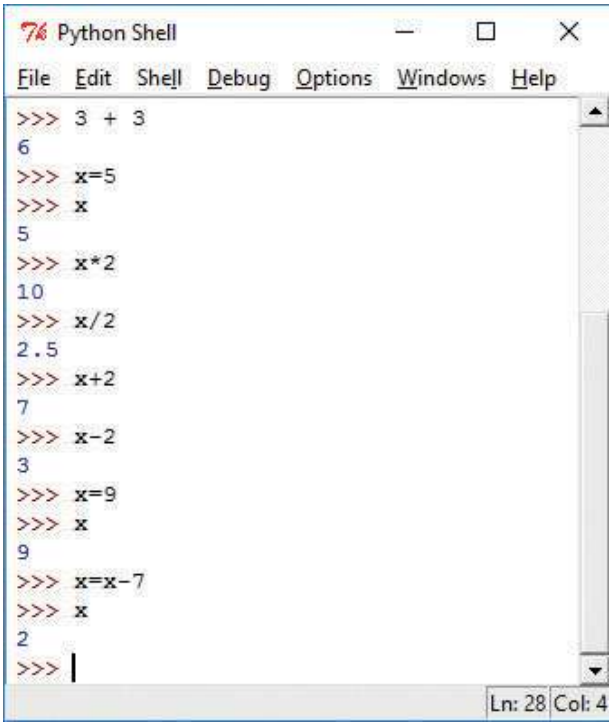


Рис. 3.4. *Python будет помнить наши переменные так долго, как нам это нужно*

Обратите внимание, что в последнем выражении присваивания мы использовали переменную x *по обеим сторонам* знака равенства: x = x – 7. На уроке алгебры это было бы неверным выражением, так как x не может равняться x – 7. Но в программе компьютер *сначала* выполняет правую сторону уравнения, вычисляя значение выражения x – 7 *до* присвоения этого значения переменной x в левой части. Переменные справа от знака равенства заменяются их значениями. В данном случае значение переменной x равняется 9, поэтому компьютер подставляет 9 в выражение x – 7 и получает выражение 9 – 7, которое равняется 2. Наконец, переменной слева от знака равенства, x, присваивается результат вычисления выражения в правой части. Значение переменной x меняется только по окончании процесса присвоения.

Прежде чем переходить к демонстрационной программе, давайте обсудим дополнительную функцию математики в Python. В табл. 3.1 и рис. 3.2 и 3.4 мы использовали оператор деления — косую черту (/), и Python выдал в ответ десятичную дробь. Для выражения `4 / 2` Python выдал значение `2.0`, а не `2`, которое мы вполне могли ожидать. Это произошло потому, что в языке Python используется так называемое *истинное деление*, которое считается более легким для понимания и вероятность возникновения ошибок в котором гораздо ниже.

Мы увидели положительный эффект от истинного деления на языке Python на рис. 3.4, когда попросили вычислить значение выражения `x / 2` при `x` равном `5`. Python говорит нам, что `5`, поделенное на `2`, равняется `2.5`, что *является* ожидаемым результатом. Это деление подобно делению поровну пяти пицц между двумя командами: каждой команде достается по `2.5` пиццы (результат деления `5 / 2`). В некоторых языках программирования оператор деления возвращает только целое число (в нашем случае это было бы число `2`). Просто запомните, что Python «делит пиццу».

Программирование с операторами: калькулятор пиццы

Говоря о пицце, давайте представим, что мы владельцы нашей собственной пиццерии.

Давайте напишем небольшую программу, которая подсчитает общую стоимость простого заказа пиццы с учетом налога с продаж. Предположим, что мы заказываем одну или несколько пицц одинаковой стоимости и что мы делаем наш заказ в городе Атланта. По законам штата налог с продаж не включается в меню, но прибавляется при завершении покупки. Ставка этого налога 8 процентов, то есть за каждый доллар, уплаченный за пиццу, мы должны заплатить восемь центов налога с продаж. На словах мы можем смоделировать данную программу следующим образом.

1. Спросить человека, сколько пицц он желает заказать.
2. Запросить стоимость каждой пиццы, указанную в меню.
3. Подсчитать стоимость каждой пиццы как подытог.
4. Подсчитать начисляемый налог с продаж по ставке 8 процентов от подытога.
5. Добавить сумму налога с продаж к подытогу и получить итоговую стоимость.

6. Показать пользователю общую сумму к оплате, в том числе налог.

Мы уже знаем, как попросить пользователя ввести данные. Чтобы выполнить вычисления с введенными числами, нам потребуется еще одна функция: `eval()`. Функция `eval()` *оценивает*, или выявляет, значение числа, введенного с клавиатуры. Клавиатурный ввод в языке Python всегда воспринимается как строка текстовых символов, поэтому мы используем функцию `eval()`, чтобы преобразовать текстовый ввод в число. Таким образом, если мы введем в программу `"20"`, функция `eval(20)` выдаст нам числовое значение `20`, которое мы сможем впоследствии использовать в математических формулах для вывода новых чисел, например стоимость 20 пицц. Когда речь заходит о работе с числами на языке Python, функция `eval()` является достаточно мощным решением.

Теперь, когда мы знаем, как преобразовать пользовательский ввод в числа, с которыми можно производить вычисления, мы переведем пронумерованные шаги плана программы в настоящий код.

Для каждого примера программы вы можете попробовать написать собственную программу, прежде чем смотреть на код, приведенный в книге. Начните с написания комментариев (#), обозначающих шаги, которые нужно сделать для решения проблемы. После этого заполните программные шаги под каждым комментарием, сверяясь с кодом в книге, когда вам нужна подсказка.

Введите нижеприведенный код в новое окно и сохраните его в файл с именем *AtlantaPizza.py*.

AtlantaPizza.py

```
#AtlantaPizza.py — простой калькулятор стоимости пиццы

#Спросить человека, сколько пицц он хочет,
#получить число с помощью функции eval()
number_of_pizzas = eval(input("Сколько пицц вы хотите? "))
#Запросить стоимость пиццы по меню
cost_per_pizza = eval(input("Сколько стоит пицца? "))

#Подсчитать общую стоимость пиццы как подытог
subtotal = number_of_pizzas * cost_per_pizza
```

```
#Подсчитать сумму налога с продаж по ставке 8% от подытога
tax_rate = 0.08 #Сохранить 8% как дробное значение 0.08
sales_tax = subtotal * tax_rate

#Приплюсовать налог с продаж к подытогу для подсчета итога
total = subtotal + sales_tax

#Показать пользователю общую сумму к оплате,
#в том числе налог
print("Полная стоимость $", total)
print("В том числе $", subtotal, " за пиццу и")
print("$", sales_tax, "налог с продаж")
```

Это программа сводит в одно мощное решение все, что вы узнали о переменных и операторах. Прочтите код программы и убедитесь, что понимаете, как работает каждая из ее частей. Как бы вы изменили программу, чтобы она работала с другой ставкой налога с продаж?

Обратите внимание, что мы включили шаги программы в виде комментариев с символом «решетка» (#). Учтите, что комментарии предназначены для чтения человеком. Редактор IDLE выделяет комментарии красным цветом, напоминая нам о том, что Python проигнорирует эти участки кода. Это наш алгоритм, набор шагов, которые должна сделать наша программа. Алгоритмы аналогичны рецептам: если мы выполним все шаги в нужном порядке, программа получится замечательной!

Когда мы записываем наш алгоритм словами (в виде комментариев с #) *и* кодом (в виде программных выражений), то достигаем двух целей. Во-первых, мы уменьшаем количество ошибок в программе за счет уверенности, что не пропустили ни один шаг. Во-вторых, мы упрощаем себе и другим чтение и понимание нашей программы. Вы должны завести себе привычку писать комментарии с самого начала, и мы будем это делать по всей книге. Если вы не хотите печатать все комментарии, программа все равно будет запускаться. Комментарии

нужны для того, чтобы помочь вам разобраться, какое действие выполняет программа.

После того как вы написали свою программу, можете ее запустить и вступить с ней во взаимодействие, выполнив команду **Run ⇒ Run Module** (Запуск ⇒ Запуск модуля). На рис. 3.5 показан образец вывода программы.

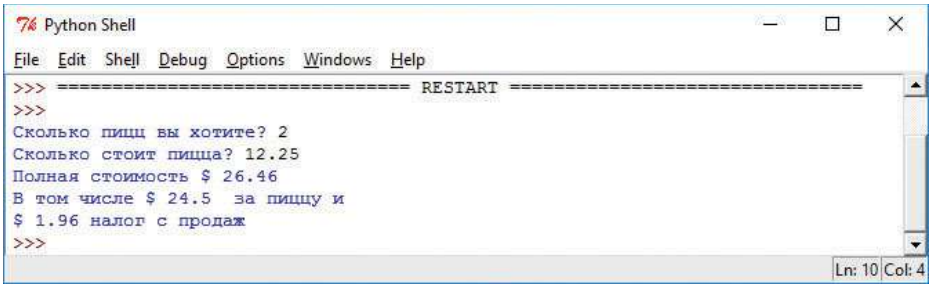


Рис. 3.5: *Пробный запуск программы калькулятора пиццы AtlantaPizza.py*

Строки: реальные символы в Python

Мы увидели, что Python отлично умеет работать с числами, но что, если мы хотим общаться с людьми? Люди лучше понимает слова и предложения, а не просто цифры. Чтобы написать программы, которыми смогут пользоваться люди, нам потребуется другой тип переменных, известный под названием *строки*. Строки — это то, как мы называем *текст* или клавиатурные символы на языке программирования, иными словами, это группы букв, цифр и символов. Ваше имя — это строка, как и название вашего любимого цвета, даже этот абзац (или даже вся книга) — это длинная строка букв, пробелов, цифр, символов, перемешанных друг с другом.

Единственная разница между строками и числами заключатся в том, что с помощью строк мы не можем производить вычисления. Строки, как правило, представляют собой имена, слова или иную информацию, которую мы не можем ввести в калькулятор. Обычно строки используются для печати. Например, в программе в начале главы мы попросили пользователя ввести свое имя, чтобы потом вывести его на печать.

Давайте проделаем это снова в новой программе. Мы запросим имя пользователя, сохраним его в переменной `name`, а затем выведем его на экран 100 раз. Как и в крутых примерах с рисованием спиралей из глав 1 и 2, мы воспользуемся *циклом* для стократного повторения печати имени

пользователя. Введите следующий код в новое окно IDLE и сохраните программу в файл с именем *SayMyName.py*.

SayMyName.py

```
# SayMyName.py — заполняет экран именем пользователя
# Запросить имя пользователя
name = input("Как тебя зовут? ")
# Распечатать имя 100 раз
for x in range(100):
    # После имени вставить пробел, а не перенос строки
    print(name, end = " ")
```

В выражении `print()` в последней строке программы появилось кое-что новое: теперь эта функция содержит *ключевое слово-аргумент*. В данном случае *ключевым словом* является слово `end`, с помощью которого мы говорим программе *заканчивать* каждое выражение `print()` пробелом (между кавычками находится пробел: `" "`) вместо обычного символа переноса строки. Как правило, на языке Python выражения `print()` заканчиваются символом переноса строки, это аналогично нажатию клавиши **Enter** на клавиатуре. Но с помощью данного ключевого слова-аргумента мы сообщаем Python о том, что не хотим распечатывать текст каждый раз на новой строке.

Чтобы более четко увидеть это изменение, модифицируйте последнюю строку программного кода указанным ниже образом — и запустите программу:

```
print(name, end = " молодец! ")
```

Если вы запустите измененную программу, то увидите, что компьютер распечатает фразу "*Ваше имя* молодец!" 100 раз! Ключевое слово-аргумент `end = "молодец! "` позволяет нам изменить работу выражения `print()`. Теперь каждая строка заканчивается словом `" молодец! "`, а не символом новой строки **Return** или **Enter**.

В языках программирования *аргумент* не является чем-то плохим и связанным со спором, это просто способ передачи функции, такой как `print()`, определенной команды за счет включения дополнительных значений внутри скобок функции. Эти значения внутри скобок функции `print()` являются аргументами, а специальное ключевое слово-аргумент означает, что мы используем ключевое слово `end`, чтобы изменить

то, каким образом функция `print()` заканчивает каждую из распечатываемых строк. Когда мы изменяем конец каждой строки с символа новой строки на символ простого пробела, слова добавляются в конец текущей строки без *возврата каретки*, то есть без начала новой строки до тех пор, пока в текущей строке не останется свободного места для отображения текста, который перенесется на следующую строку. Взгляните на результат на рис. 3.6.

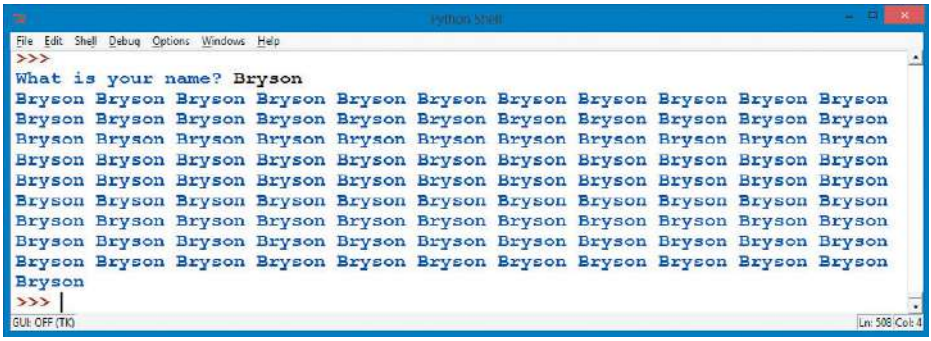


Рис. 3.6. *Python заполняет экран моим именем, когда я запускаю программу SayMyName.py*

Улучшим нашу спираль с помощью текста

Строки настолько популярны, что даже для графической библиотеки Turtle языка Python реализованы функции, принимающие в качестве ввода строки и выводящие их на экран. Функция библиотеки Turtle, просящая пользователя ввести строку, или текст, — `turtle.textinput()`. Данная функция открывает всплывающее окно, которое запрашивает текстовый ввод и позволяет нам сохранить введенное строковое значение. На рис. 3.7 показано небольшое графическое окно, открываемое библиотекой Turtle при вызове функции `turtle.textinput("Введите свое имя", "Как тебя зовут?")`. В вызове функции `turtle.textinput()` библиотеки Turtle два аргумента. Первый аргумент — `"Введите свое имя"` — заголовок всплывающего окна. Второй аргумент — `"Как тебя зовут?"` — текст приглашения, запрашивающего у пользователя нужную нам информацию.

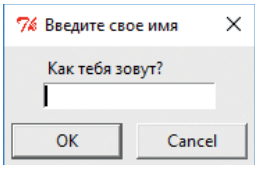


Рис. 3.7. *Окно ввода текста графической библиотеки Turtle*

Функция для написания текста на экране Turtle — `write()`. Данная функция рисует текст цветом ручки черепашки и в ее текущем местоположении на экране. Мы можем использовать функции `write()` и `turtle.textinput()` для объединения возможностей строк и цветной графики Turtle. Давайте попробуем! В следующей программе мы настроим графику Turtle по аналогии с предыдущими программами, но вместо рисования линий и окружностей мы запросим имя пользователя, а затем нарисуем его на экране в виде цветной спирали. Введите следующий текст в новое окно и сохраните его в файл с именем *SpiralMyName.py*.

SpiralMyName.py

```
  # SpiralMyName.py — печатает цветную спираль из имени
  # пользователя

  import turtle # Установка графики Turtle
  t = turtle.Pen()
  turtle.bgcolor("black")
  colors = ["red", "yellow", "blue", "green"]

  # Запрос имени пользователя с помощью всплывающего окна
  # textinput
❶ your_name = turtle.textinput("Введите свое имя", ↵
     "Как тебя зовут?")

  # Нарисовать на экране спираль из имени, повторенного 100 раз
  for x in range(100):
      t.pencolor(colors[x%4]) # По очереди выбрать все 4 цвета
❷     t.penup() # Не рисовать обычные линии спирали
❸     t.forward(x*4) # Просто переместить черепашку по экрану
❹     t.pendown() # Написать имя пользователя, увеличивая
                  # каждый раз шрифт
❺     t.write(your_name, font = ("Arial", int((x + 4) / 4), ↵
      "bold"))
      t.left(92) # Повернуть налево, как в других спиралях
```

Бóльшая часть кода программы *SpiralMyName.py* не отличается от рассмотренных ранее программ, рисующих спирали. Однако в этом случае мы просим пользователя ввести его имя во всплывающем окне `turtle.textinput` в участке кода ❶ и сохраняем введенное имя в переменной `your_name`. Мы также изменили цикл рисования, добавив команду поднятия ручки черепашки от экрана в участке ❷, таким образом, когда мы перемещаем черепашку вперед в участке ❸, она не оставляет след,

то есть не рисует обычную линию спирали. Мы хотим, чтобы спираль была представлена именем пользователя, поэтому после перемещения черепашки в участке ❸ мы с помощью команды `t.pendown()` в участке ❹ говорим программе снова начать рисовать. Затем с помощью команды write в участке ❺ мы говорим черепашке написать имя пользователя `your_name` на экране при каждом выполнении цикла. Окончательный результат — красивая спираль, составленная из многократно повторенного имени моего сына Макса, — показан на рис. 3.8.

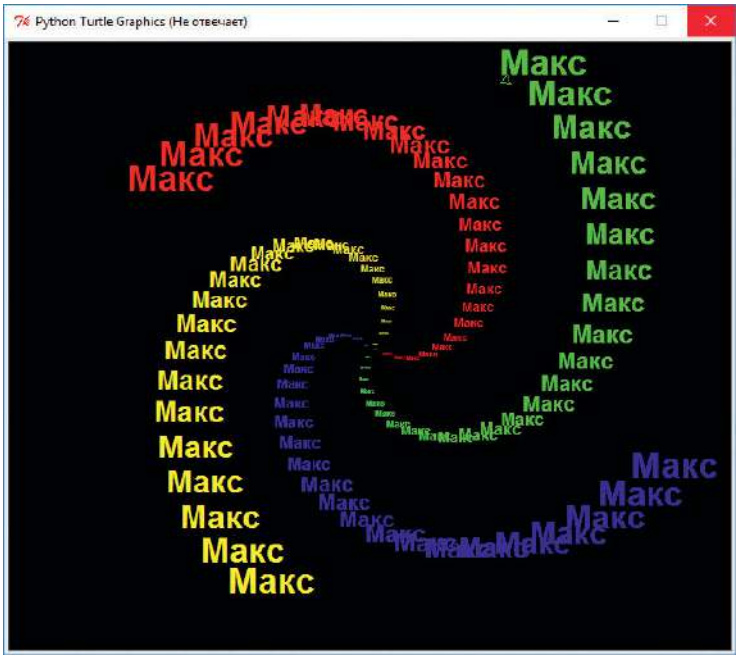


Рис. 3.8. *Цветная текстовая спираль*

Списки: храним все в одном месте

Вдобавок к строкам или числовым значениям переменная также может содержать списки. *Список* — это набор значений, разделенных запятыми, перечисляемых в квадратных скобках `[]`. В списках можно сохранять значения любого типа, будь то числа или строки, а также мы можем создавать списки списков.

В наших программах, рисующих спирали, мы сохраняли список строк — `["red", "yellow", "blue", "green"]` — в переменной `colors`. Затем, когда нашей программе требовалось воспользоваться

каким-то цветом, мы просто вызывали функцию `t.pencolor()` и говорили ей воспользоваться списком цветов `colors`, в котором функция могла бы найти названия следующего используемого цвета. Давайте добавим в наш список еще несколько названий цветов и познакомимся с еще одной функцией ввода пакета Turtle `numinput()`.

К красному, желтому, голубому и зеленому давайте добавим еще четыре поименованных цвета: оранжевый (`"orange"`), сиреневый (`"purple"`), белый (`"white"`) и серый (`"gray"`). Далее, мы хотим спросить у пользователя, сколько сторон должно быть у рисуемой фигуры. По аналогии с функцией `turtle.textinput()`, запрашивающей у пользователя строку текста, функция `turtle.numinput()` позволяет пользователю ввести число.

Мы воспользуемся функцией `numinput()`, чтобы запросить у пользователя количество сторон (от 1 до 8), а также предоставим пользователю опцию *по умолчанию* — 4, означающую, что если пользователь не введет число, то программа автоматически воспользуется параметром 4 для установки количества сторон. Введите следующий код в новое окно и сохраните его в файл с именем *ColorSpitalInput.py*.

ColorSpiralInput.py

```
import turtle # Настройка графики turtle

t = turtle.Pen()
turtle.bgcolor("black")
# Настройка списка из любых 8 действительных имен цветов
# Python
colors = ["red", "yellow", "blue", "green", "orange", ↵
          "purple", "white", "gray"]
# Запросить у пользователя количество сторон, от 1 до 8,
# количество по умолчанию — 4
sides = int(turtle.numinput("Сколько сторон", ↵
                "Сколько сторон вы хотите (1-8)?", 4, 1, 8))
# Нарисовать красочную спираль с количеством сторон
# по выбору пользователя
for x in range(360):
❶    t.pencolor(colors[x % sides]) # Использовать правильное
     # количество цветов
❷    t.forward(x * 3 / sides + x) # Изменить размер
     # в соответствии с количеством сторон
❸    t.left(360 / sides + 1) # Поворот на 360 градусов
     # количество сторон, плюс 1
❹    t.width(x * sides / 200) # Увеличить размер ручки
     # по мере передвижения во внешнюю сторону
```

Данная программа использует количество сторон, введенное пользователем, для выполнения определенных вычислений каждый раз при отрисовке новой стороны. Давайте посмотрим на четыре пронумерованные строки кода внутри цикла.

В строке ❶ программа изменяет цвет ручки черепашки, при этом приводя в соответствие количество цветов с количеством сторон (в треугольниках используется три цвета для каждой из сторон, в квадратах четыре цвета и так далее). В строке ❷ мы изменяем длины каждой линии в зависимости от количества сторон (таким образом, треугольники не выглядят на экране значительно меньшими, чем восьмиугольники).

В строке ❸ мы поворачиваем черепашку на правильный угол. Чтобы получить нужный угол поворота, мы делим 360 на количество сторон, что даст нам *внешний угол*, или угол, на который мы должны повернуть, чтобы отрисовать правильную фигуру с заданным количеством сторон. Например, окружность — это 360 градусов с одной «стороной», квадрат составляется из четырех углов по 90 градусов каждый (в сумме 360 градусов), чтобы обойти шестиугольник, вам потребуется сделать шесть поворотов на 60 градусов (что также даст в сумме 360 градусов), и так далее.

Наконец, в строке ❹ мы увеличиваем ширину, или толщину, ручки по мере того, как отходим все дальше от центра экрана. На рис. 3.9 показаны орнаменты — результат задания восьми и трех сторон.

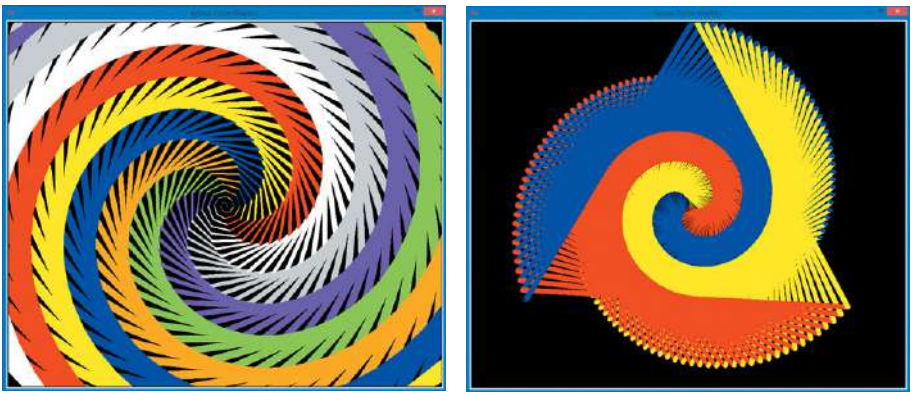


Рис. 3.9. *Картины, нарисованные программой ColorSpiral.py с восемью (слева) и тремя (справа) сторонами*

Python делает ваше домашнее задание

Мы увидели, что Python — это мощный и интересный язык программирования, который может обрабатывать данные любого типа: числа, строки, списки и даже сложные математические выражения. Сейчас же вы воспользуетесь возможностями языка Python для создания чего-то очень практически полезного: для выполнения вашего домашнего задания!

Мы напишем короткую программу, в которой используются строки и числа, воспользуемся функцией `eval()`, чтобы превратить математические задачи в ответы. Ранее в этой главе я уже говорил, что функция `eval()` может преобразовать строку `"20"` в число 20. Функция `eval()` может делать не только это: она может превратить `"2 * 10"` в число 20. Когда функция `eval()` обрабатывает строку клавиатурных символов, она делает это точно так же, как и оболочка Python. Таким образом, когда мы вводим математическую задачу в качестве входных данных, запуск `eval()` для обработки ввода может дать нам ответ на задачу.

Напечатав исходную задачу, введенную пользователем, а затем выведя `eval(`*задача*`)`, мы можем показать исходную задачу и ответ на нее в одной строке. Вспомните операторы из табл. 3.1: если вам было необходимо решить задачу 5 ÷ 2, вы бы ввели `5 / 2`, а для задачи 4^2 вы ввели бы `4 ** 2`. Ниже приведена наша программа, *MathHomework.py*, как она выглядит, если совместить все вышеописанное.

MathHomework.py

```
print("MathHomework.py")
# Попросить пользователя ввести математическую задачу
problem = input("Введите задачу или 'q', чтобы выйти: ")
# Выполнять программу, пока пользователь не введет 'q'
while (problem!= "q"):
    # Показать задачу и ответ с помощью eval()
    print("Ответ на ", problem, "- это", eval(problem))
    # Запросить еще одну задачу
    problem = input("Введите еще одну задачу или 'q', ↵
                     чтобы выйти: ")
    # Этот цикл while будет выполняться, пока вы не введете
    # 'q' для выхода
```

Выражение `while` будет продолжать просить пользователя ввести задачу и выводить ответ до тех пор, пока пользователь не нажмет на клавиатуре клавишу **Q** и не выйдет из программы.

Несмотря на то что эта короткая программа пока не может помочь нам с алгеброй, она все-таки способна на нечто большее, чем простейшая арифметика. Помните наше обсуждение истинного деления, использующегося в языке Python? Мы назвали это деление «делением пиццы», так как оно позволяло нам разделить пиццы поровну между любым количеством человек. Python, конечно, способен выполнять и целочисленное деление, нам нужно лишь познакомиться с двумя новыми операторами.

Для чего вам может понадобиться целочисленное деление? Предположим, учитель дал вам и трем вашим друзьям 10 упаковок шоколадного молока, и вы хотите разделить их поровну так, чтобы каждый из вас получил по одинаковому числу упаковок. Вас четверо (вы плюс три ваших друга), таким образом, 10 ÷ 4 равняется 2.5. К сожалению, вы не можете просто взять и разрезать упаковку молока пополам. Если бы у вас были стаканы, вы могли бы разделить молоко между двумя друзьями, но давайте представим, что поблизости кружек нет. Для честности вам можно было бы каждому взять по две упаковки и отдать учителю оставшиеся две. Это очень похоже на деление в столбик: две оставшиеся коробки, которые вы возвращаете учителю, — это *остаток* от деления 10 на 4. В математике мы иногда отмечаем остаток от деления в столбик следующим образом: 10 ÷ 4 = 2 (ост. 2). Иными словами, 10, разделенное на 4, равняется *фактором* числа 2 с остатком 2. Это значит, что 4 помещается поровну в 10 два раза с остатком 2.

На языке Python целочисленное деление осуществляется с помощью двух операторов `косая черта`, `//`. Так, `10 // 4` равняется `2`, а `7 // 4` равняется `1` (так как 4 помещается в 7 только 1 раз, с остатком 3). Оператор `//` возвращает нам частное от деления, но что же с остатком? Чтобы получить остаток от деления, мы используем оператор взятия по модулю, который в языке Python представлен символом %. Не путайте % с процентом — на языке Python проценты записываются в виде десятичных дробей (таким образом, 5% становится 0.05), а оператор % — это *всегда взятие по модулю*, или остаток от целочисленного деления. Чтобы получить остаток от деления на Python, введите `10 % 4` (возвращает остаток `2`) или `7 % 4` (остаток равняется `3`). На рис. 3.10 приведены результаты нескольких математических операций, в том числе целочисленного деления, и остатки, получаемые с помощью операторов `//` и %.

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
>>> ================================ RESTART ================================
>>>
MathHomework.py
Введите задачу или 'q' чтобы выйти: 5/2
Ответ на  5/2 - это 2.5
Введите еще одну задачу или 'q' чтобы выйти: 5 // 2
Ответ на  5 // 2 - это 2
Введите еще одну задачу или 'q' чтобы выйти: 5 % 2
Ответ на  5 % 2 - это 1
Введите еще одну задачу или 'q' чтобы выйти: (2 + 4) * (2 + 2)
Ответ на  (2 + 4) * (2 + 2) - это 24
Введите еще одну задачу или 'q' чтобы выйти: 4 ** 2
Ответ на  4 ** 2 - это 16
Введите еще одну задачу или 'q' чтобы выйти: q
>>>
                                                                Ln: 17 Col: 4
```

Рис. 3.10. *Python решает ваше домашнее задание*

В тексте книги мы будем продолжать использовать оператор % в программах, наподобие созданных для рисования спиралей, чтобы удержать числа в фиксированном диапазоне.

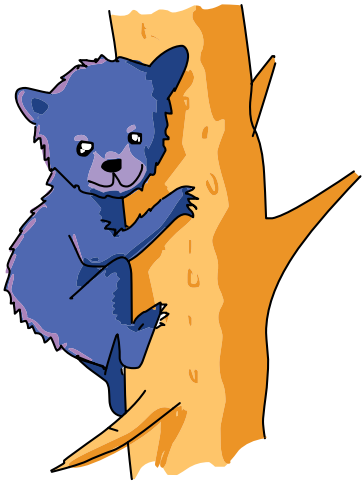
Что вы узнали

В этой главе вы увидели, каким образом можно сохранять информацию разного вида, в том числе списки, числа и строки, в переменных. Вы также изучили правила именования переменных в языке Python (латинские буквы, подчеркивания, числа; чувствительность к регистру, никаких пробелов), а также как присвоить переменным значения с помощью оператора — знака равенства (`my_name = "Алекс"` или `my_age = 5`).

Кроме того, вы узнали о существовании целых чисел и чисел с плавающей точкой (десятичные дроби). Вы изучили различные математические операторы языка Python, а также их отличия от символов, которые вы можете встретить в учебнике по математике. Мы продемонстрировали вам, как пользоваться строками из слов, букв и символов, а также — как дать Python понять и оценить определенные строки, аналогично примеру, когда мы использовали число, введенное пользователем, для выполнения вычислений.

Вы увидели несколько примеров синтаксических ошибок, а также узнали, как можно избежать некоторых из них при написании программ. Вы узнали о существовании переменных типа список, которые вы можете

использовать для сохранения списков значений любых видов, например `colors = ["red", "yellow", "blue", "green"]`. Вы даже узнали, каким образом Python может помочь вам в выполнении простых вычислений, в том числе в делении столбиком.



Узнав в главе 4, как использовать переменные для создания собственных циклов, как использовать компьютер для принятия решений (глава 5), а также как запрограммировать компьютер для игр (глава 6), и изучив следующие главы, вы получите комплексное понимание того, что есть переменная и что такое типы данных. Переменные — это первоочередной и важнейший инструмент программирования, помогающий нам решить даже самые сложные задачи, начиная от видеоигр и заканчивая программным обеспечением для спутников и медицинского оборудования. Разделив серьезные задачи на небольшие части, мы с легкостью можем решить их. Поработайте с примерами из этой главы, создавайте собственные примеры до тех пор, пока не поймете принципы работы переменных в достаточной для перехода к следующей главе степени.

На данный момент вы должны уметь делать следующее.

- Создавать собственные переменные для сохранения чисел, строк и списков.
- Обсуждать разницу между типами чисел в Python.
- Использовать основные математические операторы Python для выполнения вычислений.
- Объяснить разницу между строками, числами и списками.
- Расписать короткие программы в виде шагов на русском языке, а затем записать эти шаги в качестве комментариев, которые могут помочь вам создать программный код.
- Запросить пользовательский ввод в различных ситуациях, а затем воспользоваться введенными данными в ваших программах.

Задачи по программированию

Чтобы попрактиковаться в том, что вы изучили в этой главе, попробуйте решить эти задачи. (Если вы зашли в тупик, перейдите на сайт **https://eksmo.ru/files/Python_deti.zip** для загрузки готовых решений.)

#1. Круглые спирали

Вернитесь к программе *ColorCircleSpiral.py* из главы 2 (с. 38), которая рисовала окружности вместо прямых линий на каждой стороне спирали. Повторно запустите эту программу, чтобы определить, какие строки кода необходимо добавить в программу *ColorSpiralInput.py* или удалить из нее, чтобы иметь возможность рисовать круглые спирали с любым количеством сторон в диапазоне от одной до восьми. Когда вы решите эту задачу, сохраните программу в файл с именем *ColorSpiralInput.py*.

#2. Настраиваемые спирали с именами

Было бы здорово иметь возможность спросить пользователя, сколько сторон должно быть у спирали, спросить имя пользователя, а затем нарисовать спираль из введенного имени с правильным количеством сторон и цветов, не так ли? Давайте посмотрим, сможете ли вы определить, какие участки программы *SpiralMyName.py* (с. 60) необходимо включить в программу *ColorSpiralInput.py* (с. 62), чтобы разработать эту новую, впечатляющую программу. Когда у вас получится решить эту задачу (или если вы создадите что-то еще более интересное), сохраните новую программу в файл с именем *ColorMeSpiralled.py*.

Глава 4

ЦИКЛЫ — ЭТО ВЕСЕЛО (ПОВТОРИТЕ ПАРУ РАЗ)

Мы используем циклы, начиная с нашей первой программы, рисующей повторяющиеся фигуры. Теперь настало время научиться создавать с нуля собственные циклы. Когда программе требуется вновь и вновь повторить какое-либо действие, циклы позволяют нам повторять нужные шаги без необходимости вводить каждый из них по отдельности. На рис. 4.1 показан визуальный пример: розетка, созданная четырьмя циклами.

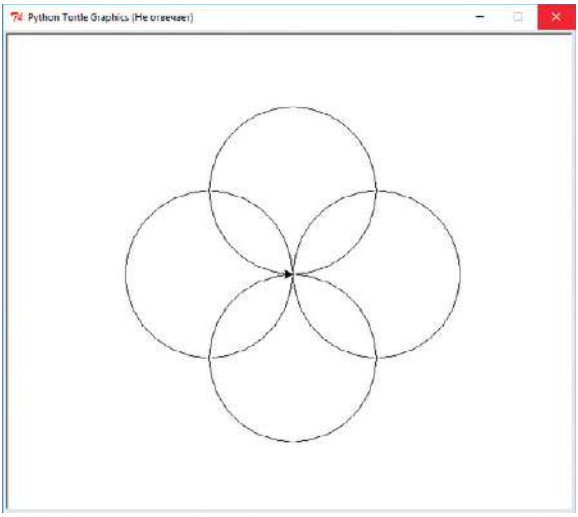


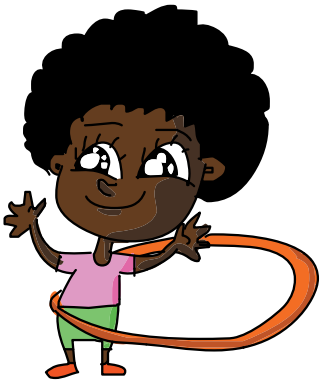
Рис. 4.1. *Орнамент розетка с четырьмя окружностями*

Давайте поразмыслим, как мы можем написать программу, которая рисовала бы четыре накладывающиеся друг на друга окружности, как показано на рисунке. Как вы уже видели в главе 2, команда `circle()` библиотеки Turtle рисует окружность указываемого в скобках радиуса. Окружности на рисунке расположены так, будто бы они ориентированы на экране по сторонам света: север, юг, восток и запад, удалены друг от друга на 90 градусов, а мы, кстати, знаем, как повернуть влево или вправо на 90 градусов. Поэтому мы могли бы написать четыре пары выражений для отрисовки окружности, затем повернуть на 90 градусов — и нарисовать еще одну окружность, как и сделано в следующей программе. Введите этот код в новое окно и сохраните в файл с именем *Rosette.py*.

Rosette.py

```
import turtle
t = turtle.Pen()
t.circle(100)   #Отрисовка первой окружности (показывает
                #на север)
t.left(90)      #Затем черепашка поворачивает налево
                #на 90 градусов
t.circle(100)   #Отрисовка второй окружности (показывает
                #на запад)
t.left(90)      #Затем черепашка поворачивает налево
                #на 90 градусов
t.circle(100)   #Отрисовка третьей окружности (показывает
                #на юг)
t.left(90)      #Затем черепашка поворачивает налево
                #на 90 градусов
t.circle(100)   #Отрисовка четвертой окружности (показывает
                #на восток)
t.left(90)      #Затем черепашка поворачивает налево
                #на 90 градусов
```

Этот код, конечно, работает, но не кажется ли вам, что он несколько однообразный? Мы четырежды написали код для отрисовки окружности и трижды — для поворота влево. Из примеров со спиралями мы знаем, что можно написать блок кода единожды и повторно использовать его в цикле `for`. В этой главе мы узнаем, как писать такие циклы самостоятельно. Давайте попробуем прямо сейчас!



Создание собственных циклов `for`

Чтобы создать собственный цикл, первое, что нужно сделать, — определить повторяющиеся шаги. Инструкции, повторяемые в предыдущей программе, — это `t.circle(100)` для отрисовки окружности с радиусом 100, а также `t.left(90)` для выполнения поворота налево перед отрисовкой следующей окружности. Во-вторых, необходимо определить, сколько раз требуется повторять эти шаги. Мы хотим нарисовать четыре окружности, поэтому давайте начнем с четырех повторений.

Теперь, определив повторяющиеся инструкции и количество операций отрисовки окружностей, создадим наш первый цикл `for`. Цикл `for` *выполняет итерации* по перечню элементов или выполняет единичное повторение каждого из элементов в перечне, например числа от 1 до 100 или от 0 до 9. Нам нужно, чтобы цикл повторился четырежды — по одному разу для каждой окружности, следовательно, необходимо настроить список из четырех чисел.

Встроенная функция `range()` позволяет с легкостью создавать списки чисел. Простейшая команда для создания списка (диапазона) из n чисел выглядит как `range(n)`. Она позволит создать список из n-ого количества чисел от 0 до $n - 1$ (от 0 до числа, на один меньшего, чем n).

Например, команда `range(10)` позволяет создавать список из 10 чисел от 0 до 9. Давайте введем несколько тестовых команд `range()` в окно командной строки IDLE и посмотрим, как это работает. Чтобы распечатать список, воспользуемся также функцией `list()`, в которую мы передадим наш список. Введите следующий код после приглашения >>>:

```
>>> list(range(10))
```

IDLE выведет на экран `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`: список из 10 чисел, начиная с 0. Для получения более коротких или длинных списков вам необходимо указывать другие числа в скобках функции `range()`:

```
>>> list(range(3))
[0, 1, 2]
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Как видите, ввод команды `list(range(3))` создаст список из трех чисел, начиная с 0, а ввод команды `list(range(5))` создаст список из пяти чисел, начиная с 0.

Использование цикла `for` для создания розетки из четырех окружностей

Для создания розетки нам потребуется повторить выполнение команды отрисовки окружности четыре раза, и поможет нам в этом функция `range(4)`. Синтаксис, или порядок слов нашего цикла `for`, выглядит следующим образом:

```
for x in range(4):
```

В первую очередь мы указываем ключевое слово `for`, после чего передаем переменную `x`, которая будет счетчиком, или *итератором*. Ключевое слово `in` говорит Python поочередно провести переменную `x` через все значения в заданном с помощью списка диапазоне, в свою очередь, `range(4)` задает список значений от 0 до 3, `[0, 1, 2, 3]`. Не забывайте, что компьютер обычно начинает считать с 0, а не с 1, как люди.

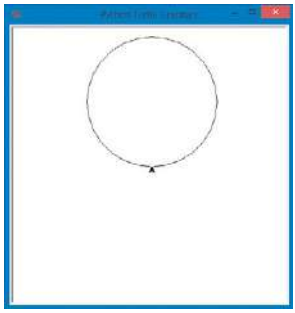
Чтобы дать понять компьютеру, какие инструкции следует повторить, мы используем *отступы*. Мы делаем отступ для каждой повторяемой в цикле команды, нажимая для этого клавишу **Tab** в окне нового файла. Наберите новую версию нашей программы и сохраните ее в файл с именем *Rosette4.py*.

Rosette4.py

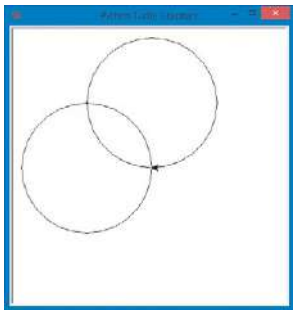
```
import turtle
t = turtle.Pen()
for x in range(4):
    t.circle(100)
    t.left(90)
```

Это гораздо более короткая версия программы *Rosette.py*, которую удалось сократить благодаря использованию цикла `for`, при этом новая программа создает те же самые четыре окружности, что и версия без цикла. Новая программа четырежды выполняет строки 3, 4 и 5, генерируя розетку из четырех окружностей: по одной в верхней, в левой, в нижней и в правой области окна. Давайте пошагово проанализируем цикл по мере отрисовки окружностей, составляющих розетку.

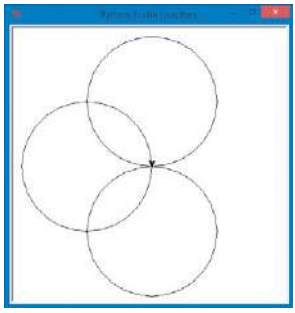
1. Во время первого прохода по циклу переменная-счетчик `x` имеет начальное значение `0`, то есть первое значение в списке, задающем диапазон `[0, 1, 2, 3]`. Мы рисуем нашу первую окружность в верхней части экрана с помощью функции `t.circle(100)`, а затем поворачиваем черепашку влево на 90 градусов с помощью команды `t.left(90)`.



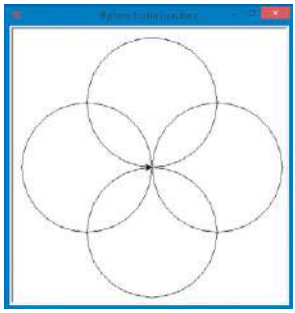
2. Python возвращается к началу цикла и устанавливает значение переменной `x` равным `1`, второму значению в списке `[0, 1, 2, 3]`. После этого компьютер отрисовывает вторую окружность в левой части экрана и поворачивает черепашку влево на 90 градусов.



3. Python вновь возвращается к началу, увеличивая значение переменной `x` до `2`, после чего компьютер отрисовывает третью окружность в нижней части окна — и поворачивает черепашку влево.



4. При четвертом и последнем проходе по циклу Python увеличивает значение переменной `x` до `3`, а затем выполняет команды `t.circle(100)` и `t.left(90)`, чтобы нарисовать четвертую окружность в правой части окна и повернуть черепашку. Теперь розетка завершена.



Изменение цикла, чтобы сделать розетку из шести окружностей

Теперь, когда мы создали с нуля собственный цикл `for`, не могли бы вы изменить программу самостоятельно так, чтобы она рисовала что-нибудь новое? Что, если мы захотим нарисовать розетку с шестью окружностями, а не с четырьмя? Что нам может понадобиться изменить в нашей программе? Сделайте небольшую паузу, чтобы поразмыслить, как можно решить эту проблему.

* * *

Ну что, есть идеи? Давайте пройдемся по задаче вместе. Во-первых, мы знаем, что нам в этот раз нужно шесть окружностей, а не четыре, поэтому в цикле `for` будет необходимо изменить диапазон: `range(6)`. Но если мы изменим только это, то не увидим никаких отличий в рисунке, потому что мы продолжим рисовать поверх уже нарисованных четырех окружностей, отстоящих друг от друга на 90 градусов. Если нам нужно нарисовать розетку из шести окружностей, то мы должны будем разделить ее на 6 поворотов влево, а не на 4. Вокруг центра рисунка можно описать 360 градусов: четыре поворота по 90 градусов провели нас на 4 × 90 = 360 градусов вокруг центра. Если же мы разделим 360 на 6, а не на 4, то получим по 360 ÷ 60 = 60 градусов для каждого поворота. Таким образом, при выполнении команды `t.left()` нам нужно поворачивать влево на 60 градусов при каждом прохождении цикла, то есть `t.left(60)`.

Измените свою программу с розеткой и сохраните ее в файл с именем *Rosette6.py*.

Rosette6.py

```
  import turtle
  t = turtle.Pen()
❶ for x in range(6):
❷     t.circle(100)
❸     t.left(60)
```

В этот раз выражение `for` в позиции ❶ проведет переменную `x` по шести значениям: от `0` до `5`, таким образом, программа повторит шаги с отступами ❷ и ❸ шесть раз. В позиции ❷ мы все еще рисуем окружность с радиусом `100`. Однако в позиции ❸ мы поворачиваем каждый раз только на 60 градусов, или на одну шестую от 360 градусов, таким образом, в этот раз вокруг центра экрана будет нарисовано шесть окружностей, как показано на рис. 4.2.

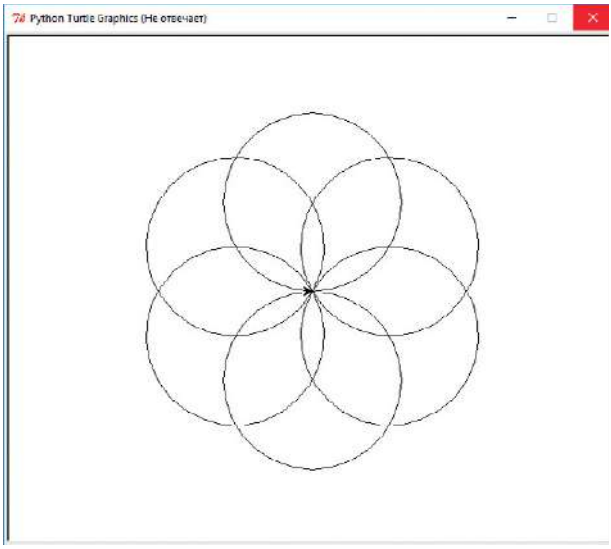


Рис. 4.2. *Розетка из шести окружностей*

Розетка из шести окружностей выглядит даже лучше, чем розетка из четырех, и благодаря циклу `for` код для отрисовки шести окружностей не длиннее ни на строку, чем код для отрисовки четырех: нам всего лишь нужно было изменить два числа! Так как мы меняли эти два числа, возможно, вас уже подмывает заменить их на переменную. Давайте подчинимся соблазну. Давайте дадим пользователям возможность нарисовать розетку с *любым* количеством окружностей.

Улучшение программы с розеткой с помощью пользовательского ввода

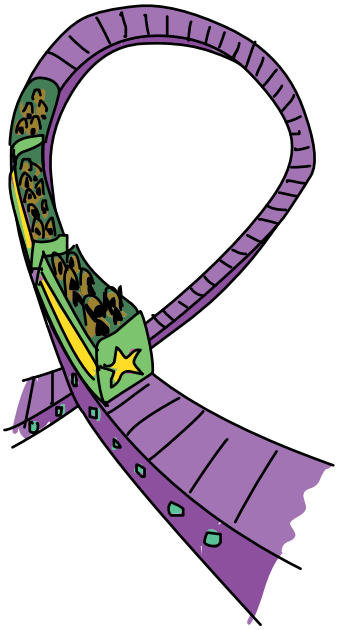
В этом разделе мы воспользуемся функцией `turtle.numinput()`, с которой мы познакомились в главе 3 (см. программу *ColorSpiralInput.py*), чтобы написать программу, которая будет просить пользователя ввести число, а затем будет рисовать розетку с этим числом окружностей. Мы установим

введенное пользователем число в качестве размера конструктора `range()`. Затем все, что нам останется сделать, — это разделить 360 градусов на это число, и мы найдем количество градусов, на которые следует поворачивать влево при каждом прохождении по циклу. Введите и запустите следующий код в файл с именем *RosetteGoneWild.py*.

RosetteGoneWild.py

```
  import turtle
  t = turtle.Pen()
  #Попросить пользователя задать кол-во кругов в розетке,
  #по умолчанию 6
❶ number_of_circles = int(turtle.numinput("Количество ↵
  окружностей", "Сколько окружностей в вашей розетке?", 6))
❷ for x in range(number_of_circles):
❸     t.circle(100)
❹     t.left(360/number_of_circles)
```

В позиции ❶ мы присваиваем переменной `number_of_circles` сразу две функции. Мы используем функцию `numinput` библиотеки Turtle, чтобы спросить у пользователя, сколько окружностей требуется нарисовать. Первое значение — `Количество окружностей` — это заголовок всплывающего окна, второе — `Сколько окружностей в вашей розетке?` — текст, который появится в окне, тогда как последнее значение, `6` — это значение по умолчанию, необходимое на случай, если пользователь не введет ничего. Функция `int()`, внутри которой расположена функция `numinput()`, преобразует введенное пользователем число в целочисленное значение, которое мы можем использовать в вызове функции `range()`. Мы сохраняем введенное пользователем число в переменной `number_of_circles` для дальнейшего использования в качестве размера диапазона `range()` цикла отрисовки розетки.



Цикл представлен выражением `for` в позиции ❷. В цикле используется переменная `number_of_circles` для поочередного присвоения переменной `x` этого количества значений. Команда начала отрисовки окружностей диаметром 100 пикселей осталась неизменной и находится в позиции ❸. В позиции ❹ мы делим полный оборот в 360 градусов на количество окружностей для того, чтобы отрисовать окружности на равном расстоянии друг от друга и от центра экрана. Например, если пользователь введет в качестве количества окружностей число `30`, то 360 ÷ 30 даст нам поворот на 12 градусов между каждой из 30 окружностей вокруг центральной точки, как показано на рис. 4.3. Запустите программу и попробуйте ввести собственные числа. Вы можете даже отрисовать розетку из 30 или 200 окружностей (однако вам придется подождать, пока Python будет рисовать такое большое количество окружностей!). Измените программу так, чтобы сделать ее вашей собственной: поменяйте цвет фона или цвет розетки, увеличьте или уменьшите диаметр окружностей или сделайте окружности больше *и* меньше! Играйте с программами по мере их создания и представляйте те интересные возможности, которыми эти программы могут обладать! На рис. 4.4 показано, что придумал мой пятилетний сын Алекс, добавив в программу *RosetteGoneWild.py* всего три дополнительные строки кода. Перейдите на сайт **https://eksmo.ru/files/Python_deti.zip** для загрузки исходного кода данной программы.

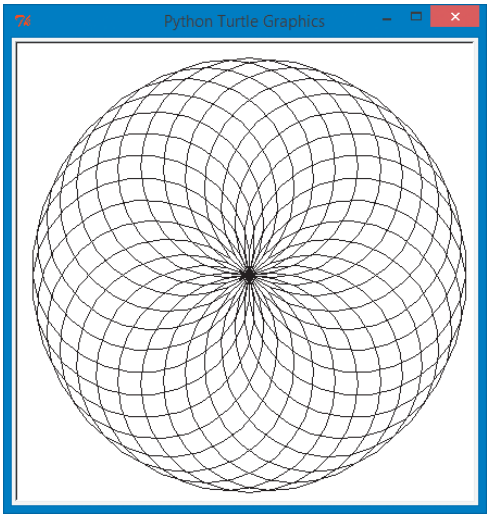


Рис. 4.3. *Пользовательская розетка с 30 окружностями*

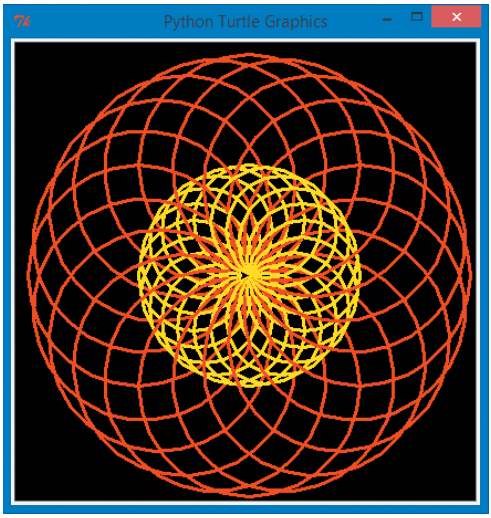


Рис. 4.4. *Немного воображения и чуть-чуть кода могут превратить нашу программу с розеткой в разноцветное веселье!*

Игровые циклы и циклы while

Цикл `for` — это мощное средство, однако мощь его также не беспредельна. В частности, что если мы захотим, чтобы выполнение цикла прекратилось, если произойдет какое-либо событие, вместо прохождения по всему длинному списку чисел? Или что если мы не уверены в том, сколько раз следует запускать цикл?

Например, представьте себе *игровой цикл*, когда нам необходимо написать программу, в частности игру, где пользователь сам выбирает, нужно ли продолжать играть или же выйти из программы. Мы, как программисты, не можем заранее знать, сколько раз пользователь захочет сыграть в нашу игру или запускать нашу программу. При этом мы должны предоставлять пользователям возможность сыграть снова без необходимости каждый раз перезапускать программу. Представьте, что необходимо перезагружать Xbox или PlayStation каждый раз, когда вы захотите повторно сыграть в ту или иную игру, или что приходится играть в одну и ту же игру десять раз, прежде чем перейти к следующей? Радости от такой игры было бы значительно меньше.

Одним из способов решения проблемы игрового цикла является использование циклов другого типа, а именно цикла `while`. В отличие от цикла `for` (Для), цикл `while` (Пока) не выполняет итерации по заранее установленному списку значений. Цикл `while` может проверить *условие*, или ситуацию, и решить, выполнить ли еще один проход либо прекратить выполнение. Синтаксис выражения `while` выглядит следующим образом:

```
while условие:
    табулированные выражения
```

Условие как правило — это некое логическое выражение, иными словами, тест Истина/Ложь. Примером из обыденной жизни может служить прием пищи и питье. Пока (`while`) вы голодны, вы едите. Когда «да» перестает быть ответом на вопрос «Я голоден?», вы прекращаете есть. Пока (`while`) вы хотите пить, вы выпиваете еще один стакан воды. Когда вы перестаете чувствовать жажду, вы прекращаете пить. Голод и жажда — это условия, и, когда эти условия становятся ложными (`false`), вы выходите из «циклов» приема пищи и питья. Цикл `while` продолжает повторение выражений в цикле (табулированные выражения) до тех пор, пока условие остается истинным.

Условия Истина/Ложь в циклах `while` зачастую подразумевают сравнение значений. Мы можем сказать: «Значение переменной `x` больше 10? До тех пор пока больше, выполнять код. Когда `x` уже не больше 10, прекратить выполнение кода». Иными словами, мы выполняем код, *пока* условие `x > 10` возвращает значение `True` (Истина). Знак больше (>) — это *оператор сравнения*, еще одна разновидность операторов, отличающаяся от арифметических операторов, например + (плюс) или – (минус). Операторы сравнения, такие как > (больше), < (меньше), == (равняется) или != (не равняется), позволяют вам сравнить два значения и увидеть, больше или меньше одно из этих значений по сравнению с другим, или же они равны или неравны. `x` меньше 7? Да или нет? `True` или `False`? В зависимости от результата, `True` или `False`, вы можете дать инструкцию программе выполнить один из двух блоков кода.

У циклов `for` и `while` есть несколько схожих черт. Во-первых, как и цикл `for`, цикл `while` при необходимости вновь и вновь повторяет выполнение определенного набора выражений. Во-вторых, при использовании обоих циклов, `while` и `for`, мы сообщаем Python о том, какие выражения необходимо повторять, путем вставки отступа перед этими выражениями с помощью клавиши **Tab**.

Давайте испробуем программу с циклом `while`, чтобы увидеть его в действии. Введите следующий код (или загрузите его по адресу **https://eksmo.ru/files/Python_deti.zip**) и запустите программу.

SayOutNames.py

```
  # Спросить у пользователя его имя
❶ name = input("Как тебя зовут? ")
  # Печатать имена, пока мы не захотим выйти
❷ while name!= "":
  # Напечатать имя 100 раз
❸     for x in range(100):
  # Печатать имена через пробел, а не с новой строки
❹         print(name, end = " ")
❺     print() # После цикла for пропустить строку
      # Запросить еще одно имя или выйти
❻     name = input("Введите еще имя или нажмите [Enter], ↵
                   чтобы выйти: ")
❼ print("Спасибо за игру!")
```

Мы начинаем нашу программу с того, что спрашиваем имя пользователя в позиции ❶ и сохраняем ответ в переменной `name`. Имя нам необходимо как условие для проверки в цикле `while`, поэтому нам нужно запросить имя *перед* началом выполнения цикла. Затем в позиции ❷ мы начинаем выполнение цикла `while`, который будет выполняться до тех пор, пока имя, вводимое пользователем, не является пустой строкой (представленной двумя кавычками, между которыми ничего нет: `""`). Пустая строка — это именно то, каким образом Python воспринимает ввод, когда пользователь просто нажимает клавишу **Enter** для выхода из программы.

В позиции ❸ начинается выполнение цикла `for`, который выведет на печать имя 100 раз, а в позиции ❹ мы говорим выражению `print()` напечатать пробел после каждого имени. Мы будем возвращаться к позиции ❸ и проверять, не достигло ли значение переменной `x` числа `100`, затем печатая имя в позиции ❹ до тех пор, пока введенное имя не заполнит несколько строк на экране. Когда наш цикл `for` завершит стократный вывод имени, мы распечатаем пустую строку без пробела ❺, переведя печать на следующую чистую строку. Теперь наступит время запросить новое имя ❻.

Так как ❻ — это последняя табулированная строка кода под циклом `while` ❷, новое имя, вводимое пользователем, передается обратно в позицию ❷, таким образом, цикл `while` может проверить, является ли оно пустой строкой. Если имя не пустое, то программа может начать выполнение цикла `for` для вывода на печать нового имени 100 раз. Если имя является пустой строкой, это означает, что пользователь нажал клавишу **Enter** для завершения программы. В этом случае цикл `while` в позиции ❷ пропускает весь код до позиции ❼, где мы благодарим пользователя за игру. На рис. 4.5 показан вывод программы, запущенной моими сыновьями.

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
>>>
Как тебя зовут? макс
макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс
макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс
макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс
макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс
макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс
макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс макс
макс макс макс макс
Введите еще имя или нажмите [Enter] чтобы выйти: алекс
алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс ал
екс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алек
с алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс
алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс ал
екс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алек
с алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс
алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс алекс ал
екс алекс алекс алекс алекс алекс алекс
Введите еще имя или нажмите [Enter] чтобы выйти: брайсон
брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон
брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон
брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон
брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон
брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон
брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон
брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон
брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон
брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон брайсон
Введите еще имя или нажмите [Enter] чтобы выйти: бев
бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев
бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев
бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев
бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев
бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев бев
Введите еще имя или нажмите [Enter] чтобы выйти:
Спасибо за игру!
>>>
Ln: 15 Col: 4
```

Рис. 4.5. *Мои сыновья запустили программу SayOurNames.py и ввели имена всех членов нашей семьи!*

Семейная спираль

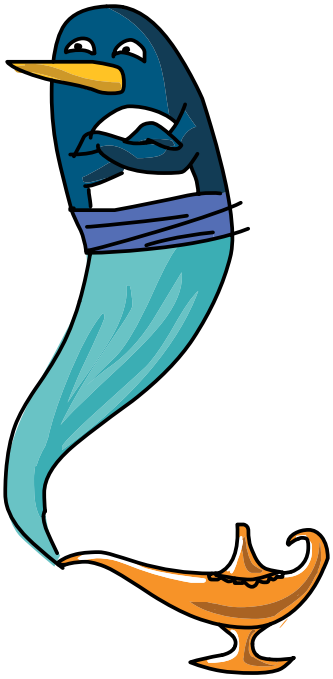
Теперь, когда мы знаем, как запросить список имен и распечатать их на экране, давайте объединим цикл печати с одной из наших программ из главы 3, *SpiralMyName.py*, на с. 60, чтобы создать разноцветную спираль из имен наших родственников и друзей.

Наша новая объединенная программа будет отличаться от программы, повторяющей имена, *SayOurNames.py*, сразу по нескольким аспектам. Однако самое важное отличие в том, что для отрисовки спирали мы не можем просто вводить каждое имя по отдельности — для этого нам потребуется

иметь все имена сразу, чтобы мы могли распечатывать их в определенной последовательности по мере передвижения по спирали.

В программе *SayOurNames.py* мы могли запросить только одно имя пользователя, но для новой графической программы со спиралью нам потребуется хранить все имена в списке, как мы поступали ранее с цветами. Затем, по мере продвижения по циклу, мы сможем одновременно выбирать новое имя и цвет на каждом углу спирали. Чтобы этого достичь, нам потребуется создать пустой список:

```
family = [] # Создание пустого списка для имен
            # родственников
```



Когда бы мы ни создавали списки цветов в наших программах, мы уже знали названия тех цветов, которые мы хотели использовать, например красный, желтый, голубой и так далее. В случае со списком родственников нам придется дождаться, пока пользователь не введет необходимые имена. Мы используем пустой список — пару квадратных скобок, `[]`, чтобы показать Python наше намерение использовать список с именем `family`, а также то, что мы не знаем, что именно будет храниться в списке, до момента запуска программы.

После создания пустого списка мы можем запросить имена в цикле `while`, как мы делали в программе *SayOurNames.py*, после чего включить введенные имена в список. *Включить* означает добавить имена в конец списка. В данной программе первое введенное пользователем имя будет добавлено к пустому списку, второе имя будет добавлено после первого, и так далее. По окончании ввода всех имен пользователь сможет нажать клавишу **Enter**, чтобы сообщить программе, что ввод имен закончен. Затем мы воспользуемся циклом `for`, чтобы нарисовать на экране цветную спираль из введенных имен.

Введите и запустите следующий программный код, чтобы увидеть, как циклы `while` и `for` совместно выписывают красивую фигуру.

SpiralFamily.py

```
   import turtle # Настройка графики Turtle
   t = turtle.Pen()
   turtle.bgcolor("black")
   colors = ["red", "yellow", "blue", "green", "orange", ↵
             "purple", "white", "brown", "gray", "pink"]
❶  family = [] # Создать пустой список для имен родственников
   # Запросить первое имя
❷  name = turtle.textinput("Моя семья", "Введите имя или ↵
                           нажмите [ENTER], чтобы выйти:")
   # Продолжить запрашивать имена
❸  while name!= "":
       # Добавить имя к пустому списку
❹      family.append(name)
       # Запросить еще одно имя или выйти
       name = turtle.textinput("Моя семья", " Введите имя или ↵
                               нажмите [ENTER], чтобы выйти:")
   # Нарисовать на экране спираль из имен
   for x in range(100):
❺      t.pencolor(colors[x%len(family)]) # Переключение цветов
❻      t.penup() # Не рисовать обычные прямые линии
❼      t.forward(x*4) # Просто передвинуть черепашку по экрану
❽      t.pendown() # Нарисовать следующее имя родственника
❾      t.write(family[x%len(family)], font = ("Arial", ↵
         int((x+4)/4), "bold"))
❿  t.left(360/len(family) + 2) # Повернуть влево
```

В позиции ❶ мы создаем пустой список `[]` с именем `family`, который будет хранить вводимые пользователем имена. В позиции ❷ мы просим ввести первое имя в окно `turtle.textinput` и начинаем выполнять цикл `while`, чтобы собрать имена всех членов семьи в позиции ❸. Команда, добавляющая значение в конец списка, — `append()`, показана в позиции ❹. Данная функция принимает введенное пользователем имя `name` и прибавляет его к списку `family`. Затем мы просим ввести следующее имя и продолжаем выполнять цикл `while` ❸ до тех пор, пока пользователь не нажмет клавишу **Enter**, сказав нам тем самым, что ввод окончен.

Цикл `for` начинается так же, как и в предыдущих примерах со спиралями, однако в позиции ❺ мы используем новую команду для установки

цвета ручки. Команда `len()` — это сокращение от английского слова *length* — длина. Данная функция сообщает нам длину списка имен, сохраненного в переменной `family`. Например, если вы ввели четыре имени ваших родственников, то функция `len(family)` вернет значение `4`. Для переключения между цветами и выбора своего цвета для каждого имени из списка `family` мы выполняем с возвращаемым значением операцию взятия по модулю, %. Для более крупных семей будет осуществляться переключение между бóльшим количеством цветов (до 10 цветов в списке), тогда как меньшим семьям понадобится меньшее количество цветов.

В позиции ❻ мы используем функцию `penup()`, «поднимая» с экрана ручку черепашки, чтобы при перемещении вперед в позиции ❼ черепашка ничего не нарисовала, поскольку мы будем рисовать имена по углам спирали, при этом между именами линий не будет. В позиции ❽ мы вновь опускаем ручку черепашки, чтобы программа смогла отрисовать следующее имя.

В позиции ❾ мы выполняем много операций. В первую очередь мы сообщаем черепашке имя, которое нужно нарисовать. Обратите внимание, что для переключения между введенными в список `family` именами в выражении`(family[x%len(family)]` используется операция взятия по модулю, %. Программа начнет с первого введенного имени `family[0]`, затем продолжит уже с именами `family[1]` и `family[2]` до тех пор, пока не будет достигнуто последнее имя в списке. Команда `font =`, также входящая в это выражение, сообщает компьютеру, что для печати имен мы хотим использовать шрифт Arial, жирный. Кроме того, эта команда устанавливает увеличение размера шрифта по мере увеличения значения переменной `x`. Выражение `(x + 4) / 4` означает, что, когда программа завершит выполнение цикла при `x = 100`, размер шрифта достигнет приятной величины (100 + 4) / 4 = 26 точек. Изменяя данное уравнение, вы можете увеличивать или уменьшать размеры шрифтов.

Наконец, в позиции ❿ мы поворачиваем черепашку влево `360/len(family)` на плюс 2 градуса. Так, для семьи из четырех человек мы повернем на 90 плюс 2 градуса для получения красивой квадратной спирали, для семьи из шести человек мы совершим повороты на 60 плюс 2 градуса, получая шестиугольную спираль, и так далее. Дополнительные 2 градуса позволят несколько закрутить спираль влево для создания визуального эффекта воронки. На рис. 4.6 мы запустили эту программу, ввели имена членов нашей семьи, а также двух наших котов, Лео и Рокки, чтобы получить чудесный спиралевидный семейный портрет.

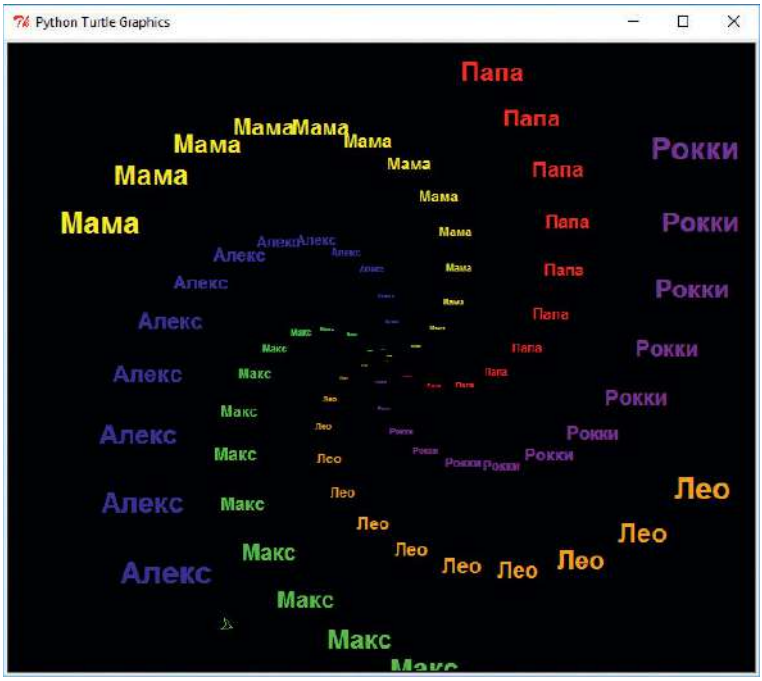


Рис. 4.6. *Семейная спираль семьи Пэйнов, включая двух котов, Лео и Рокки*

Сведем все вместе: спираль уходит в народ

Мы уже видели мощь циклов: они берут участки кода и повторяют их снова и снова, делая рутинную и монотонную работу, которую мы не хотим делать вручную, например печатают имена 100 раз. Давайте несколько усложним циклы и создадим наш собственный *вложенный цикл*, что есть цикл внутри другого цикла (как русские матрешки: загляните внутрь куклы — и увидите там еще одну).

Для того чтобы исследовать возможности вложенных циклов, давайте нарисуем спираль, но не из имен, а из *спиралей*! Вместо того чтобы рисовать в каждом углу спирали имя, как мы делали на рис. 4.6, мы можем нарисовать меньшую спираль. Чтобы добиться этого, нам понадобится большой цикл для рисования спирали на экране, а также маленький цикл внутри него, чтобы нарисовать маленькие спирали вокруг большой.

Прежде чем мы напишем такую программу, давайте изучим, каким образом один цикл вкладывается в другой. Для начала давайте создадим цикл, как обычно, затем, находясь внутри этого цикла, нажмем клавишу **Tab** для создания второго цикла:

```
# Это первый цикл, называемый внешним циклом
for x in range(10):
    # Команды с одним отступом будут выполнены 10 раз
    # Далее — внутренний, или вложенный, цикл
    for y in range(10):
          #Команды с двумя отступами будут выполнены
          #100 (10*10) раз!
```

Первый цикл называется *внешним*, так как он окружает вложенный цикл. Вложенный цикл, в свою очередь, называется *внутренним*, так как находится внутри внешнего. Обратите внимание, что внутри вложенного цикла все строки с двумя отступами (что показывает, что эти строки находятся внутри второго цикла) будут выполнены 10 раз для переменной `y` и 10 раз для переменной `x`, то есть всего 100 раз.

Давайте приступим к написанию программы *ViralSpiral.py*. Мы будем писать ее поэтапно, код законченной программы приводится на с. 90.

```
  import turtle
  t = turtle.Pen()
❶ t.penup()
  turtle.bgcolor("black")
```

Первые несколько строчек кода похожи на строки кода других созданных нами программ, рисующих спирали, за исключением того, что для большой спирали мы не будем отрисовывать линии. Мы планируем заменить линии большой спирали меньшими спиралями, поэтому в позиции ❶ мы используем функцию `t.penup()`, чтобы отнять ручку черепашки от экрана. После этого мы изменяем цвет экрана на черный.

Продолжаем печатать: это еще не все! Следующим шагом с помощью функции `turtle.numinput()` мы спросим у пользователей, сколько сторон спирали они хотели бы видеть на экране. На случай, если пользователь не укажет иное, мы установим значением по умолчанию 4, а также ограничим диапазон возможного количества сторон значениями от 2 до 6.

```
sides = int(turtle.input("Количество сторон", "Сколько ↵
            будет сторон у вашей спирали (2-6)?", 4, 2, 6))
colors = ["red", "yellow", "blue", "green", "purple", ↵
          "orange"]
```

Функция `turtle.numinput()` позволяет указать заголовок диалогового окна для ввода данных, вопрос-приглашение, значение по умолчанию, а также минимальное и максимальное значения в следующем порядке: `turtle.numinput(заголовок, приглашение, по_умолчанию, минимум, максимум)`. В данном примере мы укажем значение по умолчанию `4`, минимальное `2` и максимальное `6`. (Если пользователь попытается ввести `1` или `7`, программа выведет на экран предупреждение, что минимальное допустимое значение `2`, а максимальное — `6`.) Мы также настраиваем список `colors`, содержащий шесть цветов.

Далее мы создадим наш собственный цикл для отрисовки спирали. Внешний цикл поместит черепашку на каждом углу большой спирали.

```
❷ for m in range(100):
      t.forward(m*4)
❸     position = t.position()# Запомнить этот угол спирали
❹     heading = t.heading()# Запомнить направление следования
```

Внешний цикл изменяет значения переменной `m` от `0` до `99`, обеспечивая тем самым 100 повторений цикла ❷. Во внешнем цикле мы перемещаемся вперед, как и в других программах, рисующих спирали, но по достижении угла большой спирали мы приостанавливаемся, чтобы запомнить текущее положение `position` ❸ и направление `heading` ❹. *Положение* — это экранные координаты *(x, y)* черепашки, а *направление* — это курс перемещения черепашки.

Черепашка отклоняется от курса в каждой точке большой спирали с тем, чтобы отрисовать маленькие спирали, следовательно, черепашка должна возвращаться в положение и на курс по завершении рисования каждой маленькой спирали с тем, чтобы сохранить форму большой спирали. Если бы

мы не запоминали положение и направление движения черепашки перед началом каждой маленькой спирали, черепашка бродила бы по всему экрану, начиная рисовать новую маленькую спираль там, где была закончена отрисовка предыдущей маленькой спирали.

Две команды, которые сообщат нам положение и направление движения черепашки, — это `t.position()` и `t.heading()` соответственно. Функция `t.position()` предоставляет нам доступ к местоположению черепашки, содержащему обе координаты черепашки на экране — x (по горизонтали) и y (по вертикали), по аналогии с координатной сеткой. Направление движения черепашки доступно через функцию `t.heading()` и измеряется в диапазоне от 0,0 до 360,0 градуса, где 0,0 градуса указывают в направлении верха экрана. Мы будем сохранять эту информацию в переменных `position` и `heading` перед началом отрисовки каждой маленькой спирали, чтобы у нас была возможность вернуться туда, где мы остановились в отрисовке большой спирали в прошлый раз.

Пришло время внутреннего цикла. В этот раз команды отодвинуты еще дальше вправо. Внутренний цикл нарисует маленькую спираль в каждом углу большой спирали.

```
❺   for n in range(int(m/2)):
        t.pendown()
        t.pencolor(colors[n%sides])
        t.forward(2*n)
        t.right(360/sides - 2)
        t.penup()
❻   t.setx(position[0]) # Вернуться в положение x спирали
❼   t.sety(position[1]) # Вернуться в положение y спирали
❽   t.setheading(heading) # Указать на направление большой
                            спирали
❾   t.left(360/sides + 2) # Нацелиться на следующую точку
                            большой спирали
```

Внутренний цикл ❺ начинается со значения переменной `n = 0` и прекращает выполнение, когда `n = m/2`, или половине `m`. Это сделано, чтобы внутренние спирали были меньше, чем внешняя спираль. Внутренние спирали похожи на наши предыдущие спирали, отличие заключается в том, что мы опускаем ручку перед отрисовкой каждой линии, а затем поднимаем ее после отрисовки каждой линии для того, чтобы большая спираль оставалась чистой.

После отрисовки спирали в позиции ❺ мы возвращаемся к большой спирали в позиции ❻, устанавливая горизонтальное положение черепашки равным значению, сохраненному в позиции ❸. Горизонтальную ось чаще называют *осью икс*, поэтому при установке горизонтального положения мы используем функцию `t.setx()`, иными словами, мы устанавливаем положение черепашки на экране по оси икс. В позиции ❼ `t.sety()` мы устанавливаем положение по оси игрек, то есть положение по вертикали, которое мы также сохранили в позиции ❸. В позиции ❽ `t.setheading()` мы поворачиваем черепашку в направлении, которое мы сохранили в позиции ❹, прежде чем мы перейдем к следующей части большой спирали в позиции ❾.

По окончании выполнения большого цикла, когда значение переменной `m` увеличилось с `0` до `99`, на экране уже будет нарисовано 100 маленьких спиралей, уложенных в форме большой спирали, что создает приятный эффект калейдоскопа, как показано на рис. 4.7.

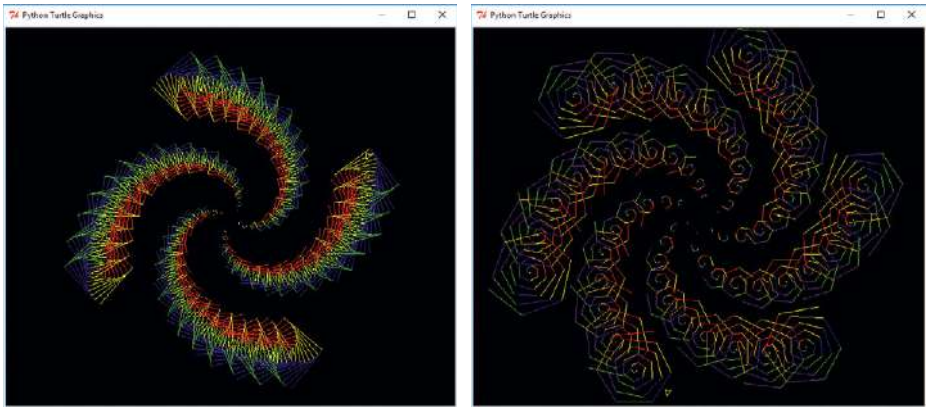


Рис. 4.7. *Квадратная спираль с квадратными спиралями в каждом углу (слева), а также пятигранная (пентагонная) спираль спиралей (справа) из программы* ***ViralSpiral.py***

После запуска программы, во время ожидания завершения отрисовки фигур, вы заметите один недостаток вложенных циклов: для рисования фигур, показанных на рис. 4.7, программе требуется больше времени по сравнению с отрисовкой простых спиралей. Это происходит потому, что наша программа выполняет гораздо больше шагов по сравнению с программами, рисующими простые спирали. На самом деле, если мы задаем программе *ViralSpiral.py* отрисовать шестигранную версию спирали,

то законченный рисунок будет состоять из 2352 отдельных линий! Все эти команды рисования, плюс повороты и установка цвета ручки добавляют приличное количество работы даже для очень быстрого компьютера. Вложенные циклы полезны, однако помните, что дополнительные шаги могут замедлить работу программы. Поэтому мы используем вложенные циклы только в том случае, если эффект от их использования перекрывает временные затраты на выполнение команд этих циклов.

Ниже приведен законченный код программы *ViralSpiral.py*.

ViralSpiral.py

```
import turtle
t = turtle.Pen()
t.penup()
turtle.bgcolor("black")
# Запросить кол-во сторон, по умолчанию 4, мин 2, макс 6
sides = int(turtle.numinput("Количество сторон", "Сколько ↵
          сторон у вашей спирали спиралей? (2-6) ", 4,2,6))
colors = ["red", "yellow", "blue", "green", "purple", "orange"]
# "Внешний" цикл спирали
for m in range(100):
    t.forward(m*4)
    position = t.position() # Запомнить этот угол спирали
    heading = t.heading() # Запомнить текущее направление
    print(position, heading)
    # "Внутренний" цикл спирали
    # Отрисовывает маленькую спираль в каждом углу большой
    for n in range(int(m/2)):
        t.pendown()
        t.pencolor(colors[n%sides])
        t.forward(2*n)
        t.right(360/sides - 2)
        t.penup()
    t.setx(position[0]) # Вернуться в позицию x большой спирали
    t.sety(position[1]) # Вернуться в позицию y большой спирали
    t.setheading(heading) # Указать направление большой спирали
    t.left(360/sides + 2) # Нацелиться на следующую точку
                          # большой спирали
```

Что вы узнали

В этой главе вы узнали о том, как создавать собственные циклы, сначала определяя повторяющиеся шаги программы, а затем перенося их в цикл нужного типа. С помощью цикла `for` вы можете повторять выполнение участка кода заданное количество раз, например 10 повторений с помощью выражения `for x in range (10)`. С помощью цикла `while` вы можете повторять выполнение кода до того, как какое-либо условие станет истинным, или до наступления определенного события, например до того, как пользователь не введет пустую строку (выражение `while name!=""`).

Вы узнали, что ход выполнения программы может быть изменен с помощью создаваемых циклов. Мы использовали функцию `range()` для генерации списков значений, позволявших нам контролировать количество повторений цикла `for`, мы также использовали оператор взятия по модулю, %, для последовательного прохода по значениям в списке и для переключения цветов в списке, выбора имен цветов из списка и пр.

Мы использовали пустой список, `[]`, и функцию `append()` для добавления информации, вводимой пользователем для последующего использования в программе. Вы узнали, что функция `len()` может сообщить вам длину списка, а точнее, количество элементов, которые этот список содержат.

Вы узнали, каким образом можно запоминать текущее положение черепашки и направление следования с помощью функций `t.position()` и `t.heading()`, а также научились возвращать черепашку на исходную позицию и задавать сохраненное направление с помощью команд `t.setx()`, `t.sety()` и `t.setheading()`.

Наконец, вы увидели, как использовать вложенные циклы для повторения одного набора инструкций внутри другого набора, сначала распечатав на экране список имен, а затем создав узор из спиралей с эффектом калейдоскопа. Кроме того, мы рисовали на экране линии, окружности и строки из слов или имен.

На данный момент вы должны уметь делать следующее.

- Создавать собственные циклы `for` для повторения набора инструкций заданное количество раз.
- Использовать функцию `range()` для генерации списков значений, контролирующих выполнение цикла `for`.

- Создавать пустые списки и добавлять в них значения с помощью функции `append()`.
- Создавать собственные циклы `while` для повторения блока кода, пока условие остается истинным (`True`), или до тех пор, пока условие не станет ложным (`False`).
- Объяснить принцип работы каждого типа цикла, их выражение в программном коде Python.
- Дать примеры ситуаций, в которых вы использовали бы циклы каждого из типов.
- Создавать и редактировать программы, в которых используются вложенные циклы.

Задачи по программированию

Чтобы попрактиковаться в том, что вы изучили в этой главе, попробуйте решить эти задачи. (Если вы зашли в тупик, перейдите на сайт **https://eksmo.ru/files/Python_deti.zip** для загрузки готовых решений.)

#1. Спиралевидные розетки

Подумайте над тем, как изменить программу *ViralSpiral.py*, чтобы маленькие спирали сменить на розетки, как в программе *Rosette6.py* (с. 74) и *RosetteGoneWild.py* (с. 76). Подсказка: сначала замените внутренний цикл на внутренний цикл, рисующий розетки, затем добавьте код для изменения цветов и размеров каждой розетки. В качестве дополнительной опции незначительно изменяйте толщину ручки по мере увеличения радиусов кругов. Завершив, сохраните новую программу в файл с именем *SpiralRosettes.py*. На рис. 4.8 показано изображение, созданное одним из вариантов решения этой задачи.

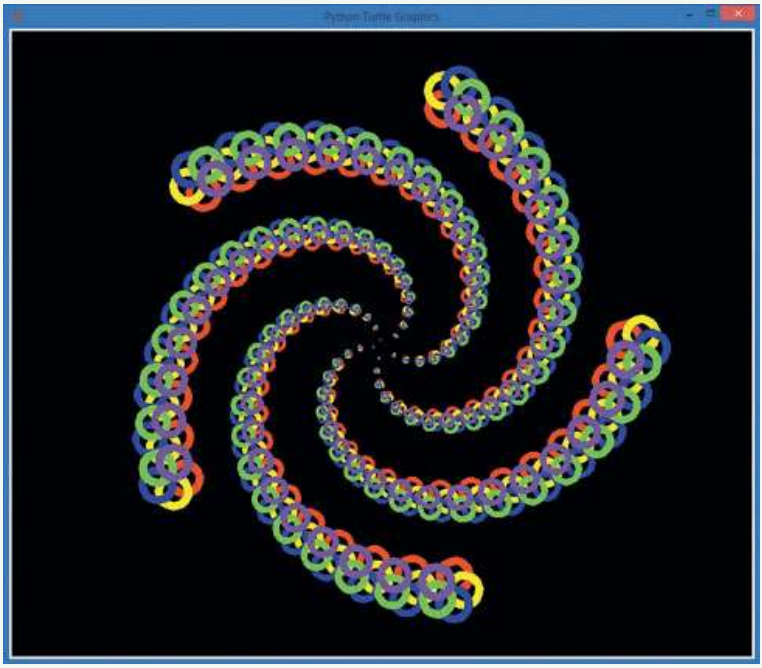


Рис. 4.8. *Спираль из розеток, нарисованная одним из решений Задачи по программированию № 1*

#2. Спираль семейных спиралей

Было бы круто нарисовать спираль спиралей из имен членов вашей семьи? Взгляните на программу *SpiralFamily.py* (с. 83), а затем еще раз просмотрите код программы *ViralSpiral.py*. Создайте вложенный цикл внутри цикла `for` программы *SpiralFamily.py*, который отрисовывал бы маленькие спирали. Затем измените внешний цикл так, чтобы он запоминал положение и направление черепашки прежде, чем переходить к отрисовке каждой маленькой спирали, и устанавливал эти значения вновь перед переходом к следующей точке большой спирали. Когда у вас все получится, сохраните программу в файл с именем *ViralFamilySpiral.py*.

Глава 5

УСЛОВИЯ (ЧТО ЕСЛИ?)

В дополнение к точности еще одно качество делает компьютеры столь мощными — это возможность оценивать информацию и быстро принимать небольшие решения. Так, например, термостат постоянно проверяет температуру и включает отопление или охлаждение, как только температура становится ниже или выше определенного числа. Сенсоры современных автомобилей реагируют на резко остановившийся впереди идущий автомобиль и нажимают на тормоза гораздо быстрее, чем это смогли бы сделать мы. Фильтры нежелательной почты отворачивают от нас десятки электронных писем, чтобы содержать папку входящих писем в чистоте.

В каждом из этих случаев компьютер проверяет набор условий: не слишком ли низкая температура? Есть ли что-нибудь на пути автомобиля? Похоже ли это электронное письмо на спам?

В главе 4 мы уже видели выражение, использующее условие для принятия решения: выражение `while`. В примерах с этим выражением условие говорило циклу `while`, сколько раз повторить набор команд. А если бы нам потребовалось принять решение, *стоит ли* выполнять набор инструкций в принципе? Представьте, если бы мы могли написать программу, в которой позволили бы пользователю решить, хочет ли он отобразить на экране спираль круглой, квадратной или иной формы? А что если бы мы хотели, чтобы окружности *сочетались* с другими фигурами, как показано на рис. 5.1?

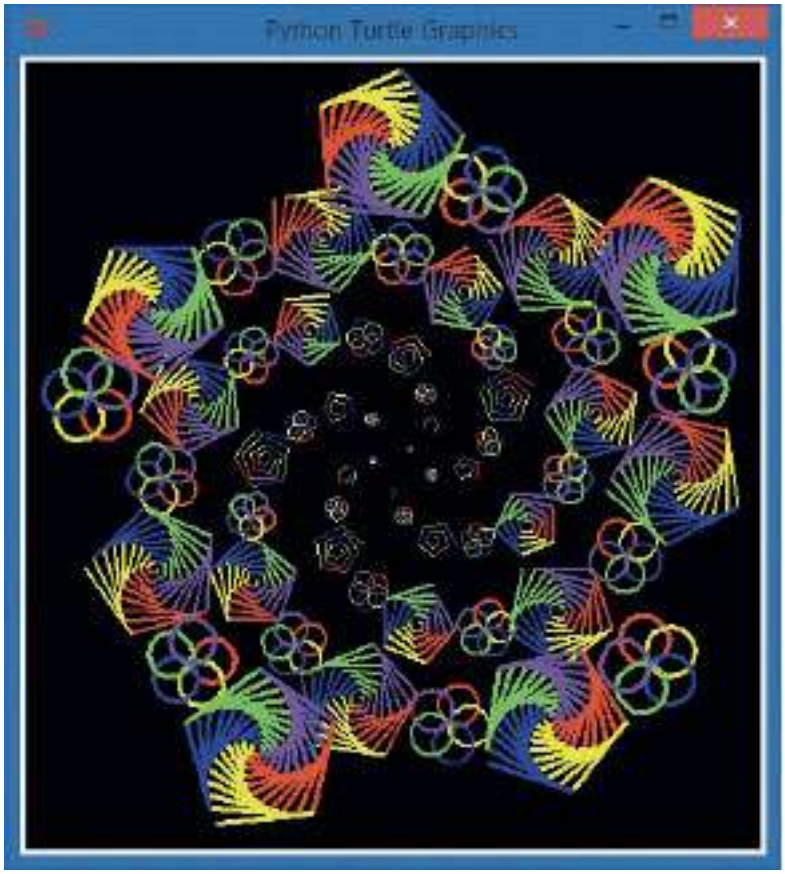


Рис. 5.1. *Спираль розеток и спираль спиралей: очень любезно со стороны выражения* `if`

Выражение, делающее все это возможным, — это выражение `if` (Если). Данное выражение спрашивает, является ли некое условие истинным, и в зависимости от ответа на этот вопрос выражение принимает решение, выполнить ли набор инструкций или же пропустить их. Если температура воздуха в здании нормальная, то системы отопления и охлаждения не запускаются, но, если в здании слишком жарко или слишком холодно, система будет включена. Если на улице идет дождь, вы возьмете с собой зонт, в противном случае — оставите его дома. В этой главе вы узнаете, как запрограммировать компьютер на принятие решений в зависимости от того, является ли определенное условие истинным или ложным.

Выражение if

Выражение `if` — это важный инструмент программирования. Данный инструмент позволяет нам сообщать компьютеру, следует ли выполнять определенную группу инструкций, в зависимости от одного или сразу нескольких условий. Выражением `if` мы можем велеть компьютеру принять решение.

Синтаксис выражения `if`, иными словами, то, как мы кодируем выражение `if`, чтобы компьютер понял его, выглядит следующим образом:

```
if условие:
    табулированные выражения
```

Условие, проверяемое выражением `if`, как правило, является булевым выражением, или тестом Истина/Ложь. Булево выражение может быть либо истинным (`True`), либо ложным (`False`). При использовании булевого выражения с `if` вам необходимо указать действие или набор действий, выполняемых, если выражение истинно. Если условие истинно, программа выполнит табулированные выражения, однако если условие ложно, то программа просто пропустит их и продолжит выполнение оставшегося не табулированного кода.

Программа *IfSpiral.py* иллюстрирует применение выражения `if` в коде.

IfSpiral.py

```
❶ answer = input("Хотите увидеть спираль? д/н:")
❷ if answer == 'д':
❸ print("Работаем…")
  import turtle
  t = turtle.Pen()
  t.width(2)
❹ for x in range(100):
❺     t.forward(x*2)
❻     t.left(89)
❼     print("Ну вот, готово!")
```

Первая строка программы *IfSpiral.py* ❶ просит пользователя ввести `д` или `н` в зависимости от того, хочет ли пользователь отобразить на экране спираль, и сохраняет ответ пользователя в переменной `answer`. В позиции ❷ выражение `if` проверяет, равняется ли введенный ответ `'д'`. Обратите

внимание, что в операторе, проверяющем «равняется ли», используется два знака равенства, ==, что позволяет отличить такой оператор от оператора присваивания, представленного одним знаком равенства, как показано в строке ❶. Оператор == проверяет равенство значения переменной `answer` и литеры `'д'`. Если равенство установлено, то условие в выражении `if` считается истинным. При проверке переменной на предмет содержания единственного символа, введенного пользователем, мы заключаем проверяемую букву или любой другой символ в одинарные кавычки (`'`).

Если условие в строке ❷ истинно, то программа распечатывает на экране сообщение `Работаем...` в строке ❸, а затем рисует на экране спираль. Обратите внимание, что выражение `print` в строке ❸, а также все выражения, рисующие спираль, вплоть до строке ❻ включительно, табулированы. Эти табулированные выражения будут выполнены только в том случае, если условие в строке ❷ истинно. В противном случае программа пропустит весь код до строки ❼, в которой распечатает сообщение `Ну вот, готово!`.

Выражения после цикла `for` в строке ❹ табулированы еще дальше (❺ и ❻), так как они относятся к выражению `for`. Аналогично тому, как мы помещали один цикл внутри другого цикла в главе 4, табулируя вложенный цикл, мы можем поместить цикл внутри выражения `if`, табулируя весь цикл.

Когда спираль закончена, наша программа возвращается к строке ❼ и сообщает пользователю об окончании работы. Именно к этой строке переходит наша программа и в том случае, если пользователь ввел в строке ❶ ответ `н` или любой другой символ, не соответствующий символу `д`. Помните, что весь блок кода от строки ❸ до ❻ будет пропущен, если условие в строке ❷ ложно (`False`).

Введите в новом окне IDLE `IfSpiral.py` или скачайте файл по адресу **https://eksmo.ru/files/Python_deti.zip** и запустите программу несколько раз, чтобы протестировать разные ответы. Если в строке с приглашением вы введете букву д, то увидите спираль, как показано на рис. 5.2.

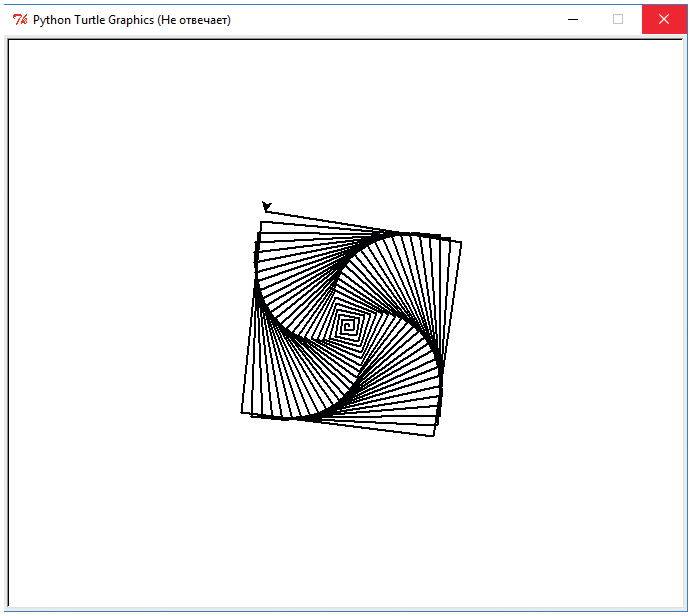


Рис. 5.2. *Если вы ответите* `д` *на вопрос в программе IfSpiral.py, то сможете увидеть вот такую спираль*

Если вы введете символ, отличающийся от буквы д нижнего регистра, или если будет введено несколько символов, то программа выведет текст `Ну вот, готово!` и завершится.

Встречаем булевы выражения

Булевы выражения, или *выражения ветвления*, — это важные инструменты программирования: способность компьютера принимать решения зависит от его способности оценивать булевы выражения как Истинные (`True`) или Ложные (`False`).

Для того чтобы сообщить компьютеру, какое условие необходимо проверить, мы должны использовать компьютерный язык. Синтаксис выражения ветвления на языке Python выглядит следующим образом:

```
выражение1 оператор_ветвления выражение2
```

Каждое выражение может быть переменной, значением или другим выражением. В программе *IfSpiral.py* `if answer == 'д'` было выражением ветвления, при этом переменная `answer` — это первое выражение, а `'д'` — второе. В языке Python кроме оператора == реализованы также другие операторы ветвления. Давайте познакомимся с некоторыми из них.

Операторы сравнения

Наиболее часто встречающиеся операторы ветвления — это *операторы сравнения*, которые позволяют вам протестировать и сравнить друг с другом два значения. Больше или меньше одно значение другого? Они равны? Каждое сравнение, осуществляемое с помощью операторов сравнения, — это условие, которое будет расценено как Истинное (`True`) или Ложное (`False`). Один из повседневных примеров сравнения — это ввод кода доступа от подъездного замка. Булево выражение принимает введенный вами код доступа и сравнивает его с правильным кодом, если ввод совпадает с правильным кодом (равняется), выражение расценивается как Истинное (`True`), и дверь открывается.

Операторы сравнения приведены в табл. 5.1.

Таблица 5.1. Операторы сравнения в языке Python

Математический символ	Оператор Python	Значение	Пример	Результат
<	<	Меньше, чем	1 < 2	True
>	>	Больше, чем	1 > 2	False
≤	<=	Меньше или равно	1 <= 2	True
≥	>=	Больше или равно	1 >= 2	False
=	==	Равно	1 == 2	False
≠	!=	Не равно	1!= 2	True

В главе 3 мы познакомились с математическими операторами. Некоторые операторы языка Python отличаются от соответствующих математических символов. Это сделано, чтобы такие операторы было проще набирать со стандартной клавиатуры. Для операторов *меньше, чем* и *больше, чем* используются привычные всем символы < и > соответственно.

Для оператора *меньше или равно* в языке Python совместно используются знаки меньше и равно, <=, между знаками пробел не ставится. Это же верно и для оператора *больше или равно*, >=. Помните, что между этими двумя знаками ставить пробел *не нужно*, так как это вызовет программную ошибку.

Оператор, позволяющий увидеть, равны ли два значения, представлен двойным знаком равенства, ==, так как одинарный знак равенства уже используется в качестве оператора присваивания. Выражение `x = 5` присваивает значение `5` переменной `x`, тогда как выражение `x == 5` проверяет, *равно* ли значение переменной `x` значению `5`. Полезно читать двойной знак равенства вслух как «равняется». Это поможет вам избежать часто встречающейся ошибки написания неправильного выражения `if x = 5` вместо правильного `if x == 5` («если `x` *равняется* пяти»).

Оператор, проверяющий *неравенство* двух значений, — !=, восклицательный знак и знак равенства. Эту комбинацию может быть легче запомнить, если вы будете проговаривать вслух фразу «не равняется» каждый раз, когда встречаетесь с оператором !=. Так, выражение `if x != 5` может быть прочтено вслух как «если `x` не *равняется* пяти».

Результат теста с использованием оператора ветвления — всегда одно из двух булевых значений, Истина (`True`) или Ложь (`False`). Перейдите в оболочку Python и попробуйте ввести некоторые из выражений, показанных на рис. 5.3. Python ответит вам `True` или `False`.

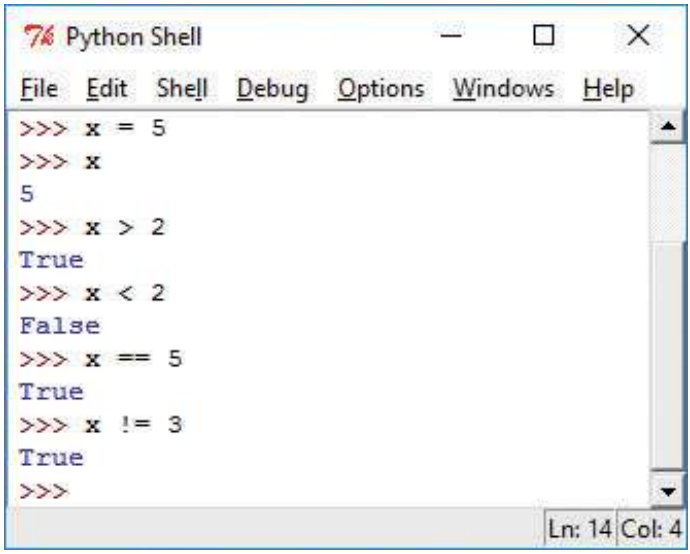


Рис. 5.3. *Тестирование выражений ветвления в оболочке Python*

Начнем с того, что откроем оболочку и введем выражение `x = 5`, чтобы создать переменную `x`, где будет храниться значение `5`. На второй строке мы проверяем значение переменной, вводя просто само имя переменной `x`, на что оболочка отвечает выводом значения этой переменной, `5`. Наше первое выражение ветвления выглядит как `x > 2`, или «икс больше двух». Python ответит нам `True`, так как 5 больше 2. Наше следующее выражение `x < 2` («x меньше 2») ложно, если `x` равняется `5`, соответственно, Python ответит `False`. В оставшихся выражениях ветвления используются операторы `<=` (меньше или равно), `>=` (больше или равно), `==` (равно) и `!=` (не равно).

В языке Python любое выражение ветвления расценивается либо как Истина (`True`), либо как Ложь (`False`). Это единственные булевы значения, использование заглавных букв *T* и *F* при записи значений `True` и `False` обязательно. `True` и `False` — это встроенные константные значения языка Python. Язык Python не поймет вас, если вы введете значение `True` как `true`, без заглавной буквы *T*, то же самое верно и для значения `False`.

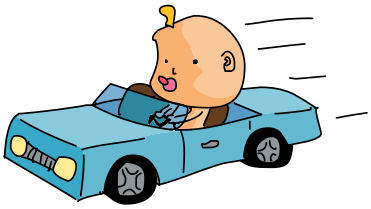
Вы — недостаточно взрослый!

Давайте напишем программу, которая использовала бы булевы выражения ветвления, чтобы понять, достаточно ли вы взрослый, чтобы водить автомобиль. Введите следующий код в новое окно и сохраните его в файл с именем *OldEnough.py*.

OldEnough.py

```
❶ driving_age = eval(input("Каков минимальный возраст для ↵
    получения прав в вашей стране? "))
❷ your_age = eval(input("Сколько вам лет? "))
❸ if your_age >= driving_age:
❹     print("Вы достаточно взрослый для получения прав!")
❺ if your_age < driving_age:
❻     print("Извините, вы не сможете водить еще", ↵
          driving_age - your_age, "лет.")
```

В строке ❶ мы просим пользователя ввести возраст, с которого разрешено управление автомобилем в стране проживания, оцениваем введенное число и сохраняем его в переменной `driving_age`. В строке ❷ мы просим ввести возраст пользователя в настоящий момент и сохраняем его в переменной `your_age`.



Если выражение в строке ❸ расценивается как истинное (`True`), то программа выполняет код строки ❹ и выводит на экран сообщение `Вы достаточно взрослый для получения прав!`. Если условие в строке ❸ расценивается как Ложь (`False`), программа пропускает строку ❹ и переходит к строке ❺. В строке ❺ мы проверяем, не является ли возраст пользователя *меньше, чем* возраст, с которого разрешено управление автомобилем. Если это так, программа выполняет инструкцию в строке ❻ и сообщает пользователю, через сколько лет он сможет получить права, путем вычитания `your_age` из `driving_age` и вывода результата на экран. На рис. 5.4 показаны результаты запуска программы для моего сына и меня.

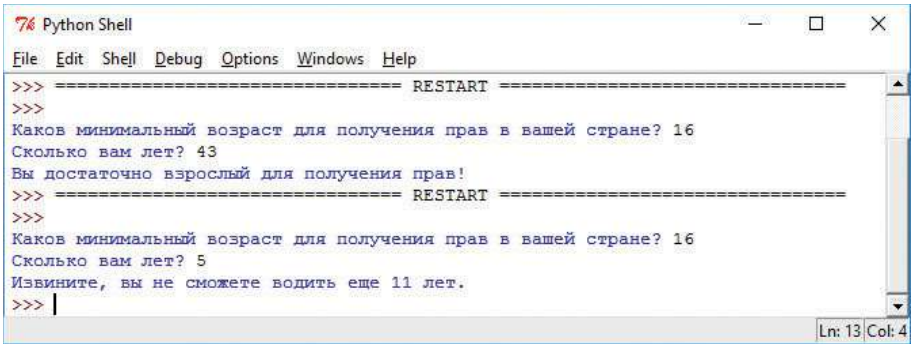


Рис. 5.4. *Я уже достаточно взрослый, чтобы управлять автомобилем в Соединенных Штатах, но мой пятилетний сын — нет*

Единственная проблема этой программы заключается в кажущейся избыточности выражения `if` в строке ❺. Если пользователь достаточно взрослый в строке ❸, то нам нет необходимости проверять, не слишком ли он молод, в строке ❺, потому что нам уже известно обратное. Если же пользователь *недостаточно* взрослый в строке ❸, то нам нет необходимости проверять, не слишком ли он молод, в строке ❺, потому что нам это и так уже известно. Ах, если бы в языке Python была возможность избавиться от ненужного кода… Так случилось, что в языке Python *действительно* предусмотрен более короткий способ обработки таких ситуаций, как эта.

Для более корректного отображения ответа на русском языке в выражение ветвления можно вложить еще три выражения `if`, первое из которых будет проверять, не является ли разница между возрастом пользователя и минимальным возрастом больше 4. Если это так, то программа выведет на экран сообщение `Извините, вы не сможете водить еще 5 лет`.

Второе выражение `if` проверит, не равна ли разница в возрасте 1, — и если равенство установлено, то выведет на экран сообщение `Извините, вы не сможете водить еще 1 год`.

Третье же выражение `if` проверит, не попадает ли разница в возрасте в диапазон значений от 2 до 4 включительно. В случае истинности этого условия на экран будет выведено соответствующее сообщение.

Код расширенной версии программы *OldEnough.py* будет выглядеть следующим образом.

```
driving_age = eval(input("Каков минимальный возраст для ↵
  получения прав в вашей стране? "))
your_age = eval(input("Сколько вам лет? "))
if your_age >= driving_age:
    print("Вы достаточно взрослый для получения прав!")
if your_age < driving_age:
if driving_age - your_age > 4:
    print("Извините, вы не сможете водить еще", ↵
      driving_age - your_age, "лет.")
if driving_age - your_age == 1:
    print("Извините, вы не сможете водить еще", ↵
      driving_age - your_age, "год.")
if driving_age - your_age!= 1 and driving_age - your_age <= 4:
    print("Извините, вы не сможете водить еще", ↵
      driving_age - your_age, "года.")
```

Выражения `else`

Зачастую нам нужно, чтобы наша программа выполняла определенное действие, если условие расценивается как истинное (`True`), и какое-то другое действие, если условие является ложным (`False`). Эта ситуация встречается очень часто, и потому в нашем распоряжении есть сокращение: выражение `else`, позволяющее нам проверить истинность условия без необходимости проверять его ложность. Выражение `else` может быть использовано только после выражения `if`, самостоятельное использование выражения `else` является недопустимым. Синтаксис выглядит следующим образом:

```
if условие:
    табулированные выражения
else:
    другие табулированные выражения
```

Если условие в выражении `if` истинно, то программа выполняет табулированные выражения под выражением `if`, а выражение `else` и соответствующие табулированные выражения пропускаются. Если же условие в выражении `if` ложно, то программа сразу же переходит к выполнению табулированных выражений блока `else`.

Мы можем переписать программу *OldEnough.py*, включив в нее выражение `else`, что поможет нам избавиться от лишней проверки условия (`your_age < driving_age`). Это не только сделает код короче и более удобочитаемым, но также поможет предотвратить возникновение ошибок в коде двух выражений ветвления. Например, при проверке условия `your_age > driving_age` в первом выражении `if` мы могли случайно упустить тот случай, когда `your_age == driving_age`. Используя пару `if` — `else`, мы можем просто проверить условие `if your_age >= driving_age`, чтобы проверить, достаточно ли пользователь взрослый, и проинформировать его об этом, либо просто перейти к выражению `else` и напечатать, сколько лет осталось ждать пользователю.

Ниже приведен код программы *OldEnoughOrElse.py*, улучшенной версии программы *OldEnough.py*, с использованием пары `if` — `else` вместо двух выражений `if`.

OldEnoughOrElse.py

```
driving_age = eval(input("Какой минимальный возраст для ↵
   получения прав в вашей стране? "))
your_age = eval(input("Сколько вам лет? "))
if your_age >= driving_age:
    print("Вы достаточно взрослый для получения прав!")
else:
    print("Извините, вы не сможете водить еще", ↵
      driving_age - your_age, "лет.")
```

Единственное отличие между программами заключается в том, что мы заменили второе выражение `if` и условие более коротким и простым выражением `else`.

По аналогии с предыдущей программой эта программа также может быть расширена для проверки дополнительных условий и вывода на экран корректных ответов на русском языке. Так, благодаря расширению блока `else` программа сможет вывести сообщение «`Извините, вы не сможете водить еще 1 год`», тогда как исходная версия программы выведет

стандартное сообщение «Извините, вы не сможете водить еще 1 лет». Ниже приведен код расширенной версии программы *OldEnoughOrElse.py*, адаптированной нами для русского языка.

```
driving_age = eval(input("Какой минимальный возраст для ↵
   получения прав в вашей стране? "))
your_age = eval(input("Сколько вам лет? "))
if your_age >= driving_age:
    print("Вы достаточно взрослый для получения прав!")
else:
if driving_age-your_age > 4:
    print("Извините, вы не сможете водить еще", ↵
      driving_age - your_age, "лет.")
if driving_age - your_age == 1:
    print("Извините, вы не сможете водить еще", ↵
      driving_age - your_age, "год.")
if driving_age-your_age <=4 and driving_age-your_age!= 1:
   print("Извините, вы не сможете водить еще", ↵
     driving_age - your_age, "года.")
```

Многоугольники или розетки

В качестве наглядного примера мы можем попросить пользователя ввести, хотели бы они нарисовать многоугольник (треугольник, квадрат, пятиугольник и так далее) или же розетку с определенным количеством сторон или окружностей. В зависимости от выбора, сделанного пользователем (`m` для многоугольника и `p` для розетки), мы сможем нарисовать правильную фигуру.

Давайте введем и запустим этот пример, *PolygonOrRosette.py*, в котором используется пара `if` — `else`.

PolygonOrRosette.py

```
  import turtle
  t = turtle.Pen()
  # Запросить у пользователя количество сторон или
  # окружностей, по умолчанию 6
❶ number = int(turtle.numinput("Количество сторон или ↵
    окружностей", "Сколько сторон или окружностей будет ↵
    у фигуры?", 6))
  # Спросить у пользователя, хочет ли он отобразить
  # многоугольник или розетку
❷ shape = turtle.textinput("Какую фигуру вы хотите?", ↵
    "Введите 'м' для многоугольника или 'р' для розетки:")
```

```
❸ for x in range(number):
❹     if shape == 'p':          # Пользователь выбрал розетку
❺         t.circle(100)
❻     else:                     # Многоугольник по умолчанию
❼         t.forward (150)
❽     t.left(360/number)
```

В строке ❶ мы просим пользователя ввести необходимое количество сторон (для многоугольника) или окружностей (для розетки). В строке ❷ мы даем пользователю возможность выбрать между `м` многоугольником и `p` розеткой. Запустите программу несколько раз, попробуйте обе опции с разным количеством сторон или окружностей, чтобы увидеть работу цикла `for` ❸.

Обратите внимание, что строки ❹–❺ табулированы, так как они являются частью цикла `for` в строке ❸. Выражение `if` в строке ❹ проверяет, ввел ли пользователь команду `p` для отрисовки розетки; если это так, то программа выполняет строку ❺, рисуя тем самым окружность — часть розетки. Если же пользователь ввел команду `м` или что-то еще, но не символ `p`, программа переходит по умолчанию к выражению `else` в строке ❻ и рисует прямую линию-грань многоугольника в строке ❼. Наконец, в строке ❽ мы поворачиваем влево на правильное количество градусов (360 градусов делится на количество сторон или лепестков розетки) и продолжаем циклически выполнять строки с ❸ по ❽ до тех пор, пока не завершим отрисовку заданной фигуры. См. пример на рис. 5.5.

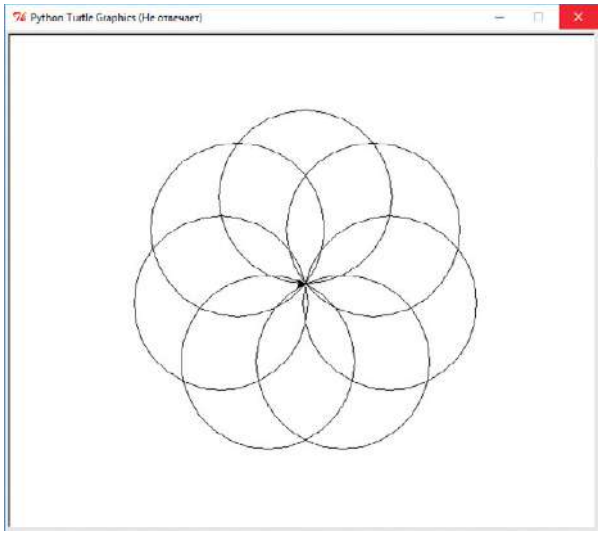


Рис. 5.5. *Рисунок, генерируемый программой PolygonOrRosette.py, если пользователь ввел количество сторон* `7` *и* `p` *для выбора отрисовки розетки*

Чет или нечет?

Выражение `if-else` может тестировать не только пользовательский ввод: мы можем использовать его для изменения фигур, как на рис. 5.1, путем проверки с помощью выражения `if` четности значения переменной — счетчика цикла при каждом изменении ее значения. При каждом прохождении цикла, когда переменная равняется `0`, `2`, `4` и так далее, мы рисуем розетку, однако при каждом нечетном проходе по циклу мы можем нарисовать многоугольник.

Чтобы выполнить это, нам необходимо знать, как проверить, является ли число четным или нечетным. Давайте подумаем, как мы решаем, является ли число четным, то есть делящимся на два. Существует ли какой-то способ узнать, делится ли число на два поровну? «Делится на два поровну» означает, что от деления не остается остатка. Например, число четыре *четное*, или делящееся на два поровну, так как 4 ÷ 2 = 2 без остатка. Число пять *нечетное*, так как 5 ÷ 2 = 2 с остатком 1. Таким образом, при делении четного числа на два остаток равен нулю, а при делении нечетного числа остаток равен единице. Помните оператор остатка? Все верно: это наш друг — оператор взятия по модулю, %.

В коде на языке Python мы можем настроить переменную-счетчик `m` и проверить, является ли значение `m` четным, с помощью выражения `m % 2 == 0`. Иными словами, все, что нам нужно сделать, — проверить, равняется ли остаток от деления `m` на 2 нулю:

```
for m in range(number):
    if (m % 2 == 0):  # Проверка m на четность
          # Выполнить четные действия
    else: # Иначе, m нечетная
          # Выполнить нечетные действия
```

Давайте изменим программу со спиралью так, чтобы на четных углах большой спирали компьютер рисовал лепестки розетки, а на нечетных — многоугольники. Мы будем использовать большой цикл `for` для отрисовки большой спирали, выражение `if-else` для проверки того, нужно ли нарисовать розетку или многоугольник, а также два маленьких внутренних цикла для непосредственной отрисовки розетки или многоугольника. Эта программа будет самая длинная из тех, что мы уже писали, но комментарии помогут разобраться с действиями, производимыми программой. Введите и запустите следующую программу, *RosettesAndPolygons.py*, обратите внимание на проверку правильности табуляции циклов и выражений `if`.

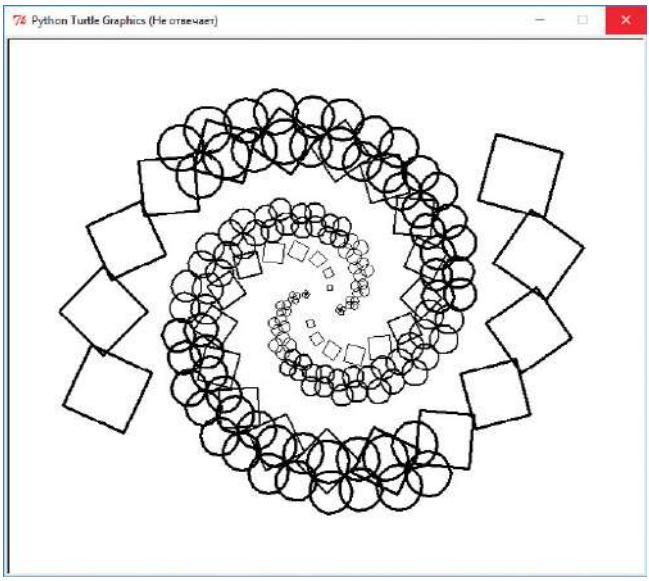
RosettesAndPolygons.py

```
# RosettesAndPolygons.py — спираль из многоугольников
# И розеток!
import turtle
t = turtle.Pen()
# Запросить у пользователя количество сторон, по умолчанию 4
sides = int(turtle.numinput("Количество сторон", ↵
   "Сколько сторон будет у вашей спирали?", 4))
# Внешний цикл для многоугольников и розеток, размер от 5 до 75
❶ for m in range(5,75):
      t.left(360/sides + 5)
❷     t.width(m//25+1)
❸     t.penup() # Не рисовать линии спирали
      t.forward(m*4) # Перейти к следующему углу
❹     t.pendown() # Приготовиться к рисованию
      # Нарисовать небольшую розетку на каждом ЧЕТНОМ углу
      # спирали
❺     if (m % 2 == 0):
❻          for m in range(sides):
                t.circle(m/3)
                t.right(360/sides)
                # ИЛИ, нарисовать небольшой многоугольник
                # на каждом НЕЧЕТНОМ углу
❼     else:
❽          for m in range(sides):
                t.forward(m)
                t.right(360/sides)
```

Давайте посмотрим, как работает эта программа. В строке ❶ мы создали цикл, работающий в диапазоне от 5 до 75. Мы пропускаем значения от 0 до 4, так как очень сложно разглядеть на экране фигуры, диаметр которых меньше 4 пикселей. Мы выполняем поворот для спирали, затем в строке ❷ мы используем целочисленное деление для увеличения ширины (толщины) ручки при отрисовке каждой 25-й фигуры. Линии на рис. 5.6 становятся толще по мере увеличения размера фигур.

В строке ❸ мы поднимаем ручку черепашки над экраном с тем, чтобы не рисовать линии между розетками и многоугольниками. В строке ❹ мы опускаем ручку, этим подготавливая ее к рисованию фигуры в углу большой спирали. В строке ❺

мы проверяем переменную-счетчик `m`, чтобы понять, рисуем ли мы в четном углу. Если угол четный (`m % 2 == 0`), мы рисуем розетку с помощью цикла `for` в строке ❻, в противном случае выражение `else` в строке ❼ говорит нам нарисовать многоугольник с помощью цикла `for`, начинающегося в строке ❽.



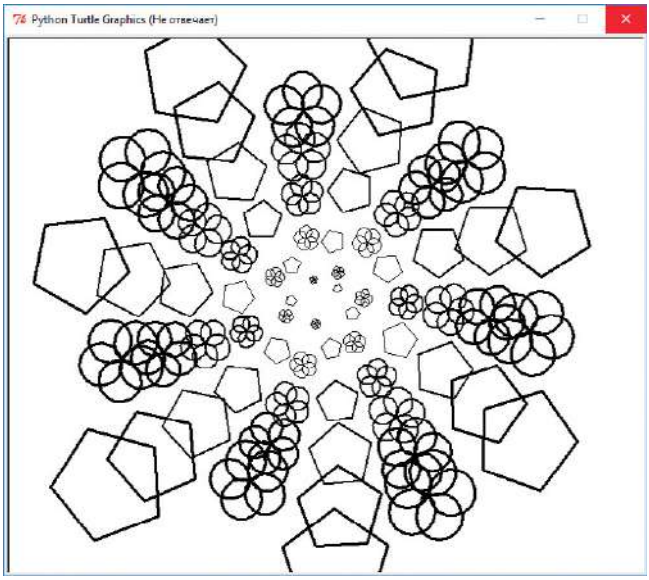


Рис. 5.6. *Два запуска программы RosettesAndPolygons.py, во время которых пользователь задает 4 стороны (сверху) и 5 сторон (снизу)*

Обратите внимание, что когда мы вводим четное количество сторон, то чередующиеся фигуры формируют отдельные участки спирали, как показано в верхней части рис. 5.6. Однако если количество сторон нечетное, то в каждом участке спирали происходит чередование четной (розетки) и нечетной (многоугольник) фигур. Добавив цвет и немного размышлений, вы можете заставить программу создать рисунок, подобный показанному на рис. 5.1. Выражения `if-else` добавляют еще одно измерение к нашему набору инструментов программирования.

Выражения `elif`

К выражению `if` предусмотрено еще одно полезное дополнение: условие `elif`. Нет, это не один из помощников Санты! Выражение `elif` — это способ совместить выражения `if-else`, если вам необходимо проверить условие с более чем двумя возможными результатами. Ключевое слово `elif` — это сокращение от «`else if`». Давайте поразмыслим о школьных оценках. Если на экзамене вы получили 98 баллов, то преподаватель может поставить вам оценку «5» или «5−», в зависимости от шкалы оценок. Но если же вы набрали меньшее количество баллов, то преподаватель может поставить вам не только какую-то одну оценку (слава богу, вариантов больше, чем «5» или «2»). На самом деле, в распоряжении учителя есть сразу несколько оценок, которые он может поставить: «5», «4», «3», «2» или «1».

Это именно тот случай, когда на помощь приходит выражение `elif` (или целый набор выражений `elif`). В качестве примера давайте возьмем шкалу оценок с десятибалльным интервалом, где 90 и более баллов — это оценка «5», 80–89 — оценка «4» и так далее. Если вы набрали 95 баллов, то можете распечатать оценку «5» и пропустить все остальные варианты. По аналогии, если ваш балл 85, то вам нет необходимости проверять условия далее, чем оценка «4». Конструкция `if-elif-else` позволяет нам осуществлять прямолинейную проверку оценок. Попробуйте запустить следующую программу, *WhatsMyGrade.py*, и ввести разное количество баллов в диапазоне от 0 до 100.

WhatsMyGrade.py

```
❶ grade = eval(input("Введите количество баллов (0-100): "))
❷ if grade >= 90:
      print("У вас 5!:) ")
```

```
❸ elif grade >= 80:
      print("У вас 4!")
❹ elif grade >= 70:
      print("У вас 3.")
❺ elif grade >= 60:
      print("У вас 2…")
❻ else:
      print("У вас 1. :(")
```

В строке (1) с помощью приглашения `input()` мы запрашиваем у пользователя количество баллов диапазоне от 0 до 100, конвертируем введенные данные в число с помощью функции `eval()` и сохраняем это число в переменной `grade`. В строке (2) мы сравниваем баллы пользователя со значением 90, порогом для оценки «5». Если пользователь ввел 90 баллов и больше, то Python выведет на экран сообщение `У вас 5! :)`, пропустит остальные выражения `elif` и `else` и продолжит выполнять оставшуюся часть программы. Если введенное количество баллов не 90 и больше, то мы переходим к проверке на соответствие оценке «4» в строке (3). Опять же, если количество баллов 80 или больше, программа распечатывает правильную оценку и пропускает оставшиеся условия, в том числе и `else`. В противном случае выражение `elif` в строке (4) проверяет на соответствие оценке «3», выражение `elif` в строке (5) — на соответствие оценке «2», и, наконец, если количество баллов меньше 60, то программа спускается до выражения `else` в строке (6) и выводит на экран сообщение `У вас 1. :(`.

Мы можем использовать выражения `if-elif-else` для проверки переменной на соответствие большому количеству значений. Однако иногда нам необходимо проверить сразу несколько переменных. Например, решая, что надеть сегодня, нам нужно знать температуру воздуха (тепло или холодно) и погоду (солнечно или же идет дождь). Чтобы сочетать несколько выражений ветвления, нам нужно изучить еще несколько новых трюков.

Сложные условия: `if`, `and`, `or`, `not`

Иногда единственного выражения ветвления может оказаться недостаточно. Что если нам нужно было бы знать, какая на улице погода: тепло там *и* солнечно или же холодно *и* идет дождь?

Вернемся к нашей первой программе в этой главе. В той программе мы отвечали `д`, если хотели отрисовать на экране спираль. Первые две строки программы запрашивали ввод и проверяли, ввел ли пользователь ответ `д`:

```
answer = input("Хотите увидеть спираль? д/н:")
if answer == 'д':
```

Чтобы увидеть спираль, пользователь должен был ввести именно ответ д, программой принимался только этот ответ. Даже нечто очень похожее, например заглавная буква Д или слово «да», не срабатывало, так как выражение `if` проверяло только соответствие букве д.

Есть простой способ решить проблему букв Д и д, который заключается в использовании функции `lower()`, переводящей все строки в нижний регистр. Вы можете опробовать эту функцию в оболочке IDLE:

```
>>> 'Да, Сэр'.lower()
'да, сэр'
```

Функция `lower()` заменила прописные буквы Д и С во фразе «Да, Сэр» на строчные, оставив при этом все остальные буквы неизменными.

Мы можем использовать функцию `lower()` при обработке пользовательского ввода, таким образом, вне зависимости от того, что введет пользователь, Д или д, выражение `if` всегда будет истинным (`True`):

```
if answer.lower() == 'д':
```

Теперь, если пользователь вводит Д или д, программа проверяет, не равняется ли соответствующая буква нижнего регистра д введенной пользователем букве. Однако если нам необходимо проверить вариант ввода полного слова Да, то нам потребуется воспользоваться *сложным выражением* `if`.

Сложные выражения `if` похожи на сложные предложения: «Я собираюсь в магазин, и я куплю что-нибудь к чаю». Сложные выражения `if` используются при необходимости сделать нечто большее, чем простая проверка истинности одного условия. Например, нам может потребоваться проверить истинность одного *и* другого условия. Мы можем также проверить истинность одного *или* другого условия. Мы можем удостовериться в том, что условие *не*истинно. Мы делаем это и в повседневной жизни тоже. Мы говорим: «Если на улице холодно *и* идет дождь, я надену теплый плащ», «Если на улице ветрено *или* холодно, я надену пальто» или «Если дождь *не* идет, я надену любимые ботинки».

При составлении сложного выражения `if` мы должны воспользоваться одним из *логических операторов*, приведенных в табл. 5.2.

Таблица 5.2. *Логические операторы*

Логический оператор	Использование	Результат
`and` (И)	`if (`*условие1* `и` *условие2*`):`	Истинно, только если оба условия 1 и 2 истинны (`True`)
`or` (ИЛИ)	`if (`*условие1* `и` *условие2*`):`	Истинно, если хотя бы одно из двух условий 1 и 2 истинно (`True`)
`not` (НЕ)	`if not (`*условие*`):`	Истинно, только если *условие* ложно (`False`)

Мы можем воспользоваться оператором `or`, чтобы проверить ввел ли пользователь букву `д` или слово `Да` целиком, оба варианта будут правильными.

```
answer = input("Хотите увидеть спираль? д/н:").lower()
if answer == 'д' or answer == 'да': # Проверяет
                                    # на соответствие 'д' или 'Да'
```

В этот раз мы проверяем, не является ли хотя бы одно из двух условий истинным (`True`). Если любое из условий истинно, пользователь увидит спираль. Обратите внимание, что мы вводим условное выражение полностью по обеим сторонам ключевого слова `or`: `if answer == 'д' or answer == 'да': #Проверяет на соответствие 'д' или 'Да'`. Начинающие программисты часто допускают ошибку, когда пытаются сократить условия `or`, убирая второе выражение `answer ==`. Чтобы запомнить правильное использование выражения `or`, всегда воспринимайте оба условия по отдельности. Если любое из условий, соединенных ключевым словом `or`, расценивается как истинное, то все выражение истинно, но для того, чтобы выражение работало правильно, все условия должны быть полными.

Сложное условие с ключевым словом `and` выглядит похоже, но ключевое слово `and` требует, чтобы *каждое* условие в выражении было истинным: только в этом случае все выражение может быть расценено как истинное. Для примера давайте напишем программу, решающую, что нам

надеть, в зависимости от погоды. Введите код программы *WhatToWear.py* в новом окне или скачайте эту программу по адресу **https://eksmo.ru/files/Python_deti.zip** и запустите ее.

WhatToWear.py

```
❶ rainy = input("Как погодка? Идет дождь? (д/н) ").lower()
❷ cold = input("На улице холодно? (д/н) ").lower()
❸ if (rainy == 'д' and cold == 'д'): # Дождливо и холодно,
                                     # блин!
      print("Лучше наденьте плащ.")
❹ elif (rainy == 'д' and cold!= 'д'): # Дождливо, но тепло
      print("Возьмите с собой зонт.")
❺ elif (rainy!= 'д' and cold == 'д'): # Сухо, но холодно
      print("Наденьте пальто: на улице холодно!")
❻ elif (rainy!= 'д' and cold!= 'д'): # Тепло и солнечно, ура!
      print("Надевайте, что хотите: на улице прекрасно!")
```

В строке ❶ мы спрашиваем пользователя, не идет ли дождь, а в строке ❷ мы уточняем, холодно ли на улице. Мы также удостоверяемся в том, что ответы, сохраняемые в переменных `rainy` и `cold`, представлены буквами нижнего регистра. Для этого мы добавляем в конце функции `input()` вызов функции `lower()` на обеих строках. Используя эти два новых условия (идет ли дождь, холодно ли на улице), мы можем пользователю помочь решить, что надеть. В строке ❸ сложное выражение `if` проверяет справедливость утверждения, что на улице и дождливо и холодно, если это так, то программа советует надеть плащ. В строке ❹ программа проверяет справедливость утверждения, что на улице идет дождь, но при этом не холодно. Для дождливой, но не холодной погоды программа рекомендует взять зонт. В строке ❺ мы проверяем утверждение, что дождь *не* идет (ответ `идет` *не эквивалентен* ответу `'д'`), но все же холодно, поэтому требуется пальто. Наконец, в строке ❻, если на улице не идет дождь *и* не холодно, предлагается надеть любую одежду!

Секретные послания

Теперь, когда мы понимаем, как использовать условия, мы научимся кодировать и декодировать секретные сообщения с помощью шифра Цезаря. *Шифр* — это секретный код, или способ изменения сообщений так, чтобы их было сложно прочесть. *Шифр Цезаря* назван в честь Юлия Цезаря,

который, по преданиям, любил отправлять личные послания, изменяя порядок букв в алфавите:

```
SECRET MESSAGES ARE SO COOL*! -> FRPERG ZRFFNTRF NER FB PBBY!
```

Мы можем создать простой шифр Цезаря с помощью кодировочного кольца, как, например, то, что показано на рис. 5.7. Чтобы создать закодированное сообщение, мы должны выбрать *ключ*, то есть выбрать число, на которое мы хотим передвинуть каждую букву. В закодированном сообщении и на рис. 5.7 каждая буква передвинута на ключевое значение `13`, это значит, что мы должны взять нужную букву и отсчитать по алфавиту 13 позиций после нее, чтобы получить закодированную букву. Так, английская буква *A* становится *N*, а буква *B* — *O* и так далее.

Рис. 5.7. *Шифр Цезаря*

Иногда такое перемещение называют *ротацией*, так как к тому времени, как мы дойдем до *M* (которая становится *Z*), мы находимся в конце алфавита. Чтобы иметь возможность закодировать букву *N*, мы должны будем досчитать до конца алфавита и вернуться к букве *A*. Буква *O* возвращается к букве *B*, проходя весь алфавит через букву *Z*, которая, в свою очередь, становится буквой *M*. Ниже приведен пример справочной таблицы для шифра Цезаря с ключевым значением `13`, где каждая буква передвинута на 13 позиций для кодирования и декодирования:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
N O P Q R S T U V W X Y Z→A B C D E F G H I J K L M
```

Заметили закономерность? Латинская буква *A* закодирована как буква *N*, в свою очередь буква *N* кодируется буквой *A*. Это называется *симметричным шифрованием* или *симметричным кодированием*, так как такой код или шифр обладает свойством *симметрии* — он одинаков в обоих

* Секретные сообщения — это очень круто! (*англ.*)

направлениях. Мы можем кодировать и декодировать англоязычные сообщения, используя один и тот же ключ `13`, потому как в английском (латинском) алфавите 26 букв, а ключ `13` означает, что мы передвигаем каждую букву точно на половину алфавита. Вы можете опробовать этот шифр со своим собственным сообщением: `HELLO`* `->` `URYYB` `->` `HELLO`.

Если мы можем написать программу, которая будет просматривать каждую букву в секретном послании, затем кодировать эту букву, передвигая ее на 13 позиций вперед, мы сможем отправлять закодированные сообщения любому человеку, имеющему такую программу или способному разгадать наш шифр. Для написания программы, работающей с отдельными буквами строки, нам необходимо получить еще несколько навыков по работе со строками на языке Python.

Игра со строками

Язык Python поставляется с мощными функциями для работы со строками. В языке реализованы функции, которые могут преобразовать всю строку в символы верхнего регистра, функции, которые могут конвертировать отдельные символы в числовые эквиваленты, а также функции, которые могут сказать, является ли отдельный символ буквой, числом или иным символом.

Давайте начнем с функции, которая может преобразовать строку в символы верхнего регистра. Чтобы упростить понимание нашей программы-кодировщика/декодера, мы будем преобразовывать весь текст послания в символы верхнего регистра, это позволит нам кодировать только один набор из 26 символов (от A до Z) вместо обработки двух наборов (от *A* до *Z* и от *a* до *z*). Функция, преобразовывающая строку в символы верхнего регистра, — это `upper()`. Любая строка, после которой ставится точка и указывается имя функции `upper()`, вернет ту же строку текста, но только записанную символами верхнего регистра, остальные символы останутся без изменений. Чтобы увидеть эту функцию в действии, в оболочке Python попробуйте ввести в кавычках свое имя или любую другую строку, после которой укажите команду.`upper()`:

```
>>> 'Брайсон'.upper()
'БРАЙСОН'
>>> 'Вау, это прикольно!'.upper()
'ВАУ, ЭТО ПРИКОЛЬНО!'
```

* ПРИВЕТ (*англ.*).

Как уже было сказано ранее, функция `lower()` выполняет противоположное действие:

```
>>> 'Брайсон'.lower()
'брайсон'
```

С помощью функции `isupper()` вы можете проверить, не является ли тот или иной отдельный символ буквой верхнего регистра:

```
>>> 'B'.isupper()
True
>>> 'b'.isupper()
False
>>> '3'.isupper()
False
```

С помощью функции `islower()` вы можете проверить, не является ли тот или иной отдельный символ буквой нижнего регистра:

```
>>> 'P'.islower()
False
>>> 'p'.islower()
True
```

Строка — это набор символов, поэтому циклический проход по строке с помощью цикла `for` языка Python разобьет строку на отдельные символы. В этом примере переменная `letter` пройдет по всем символам в строковой переменной `message`:

```
for letter in message:
```

Наконец, мы можем использовать обычный оператор сложения + (плюс) для объединения двух строк или для добавления букв к строке:

```
>>> 'Брай' + 'сон'
'Брайсон'
>>> 'Пэй' + 'н'
'Пэйн'
```

В данном случае мы добавляем вторую строку в конец первой. Сложение строк называется *подстановкой*. Сложение строк именуется еще *конкатенацией*, просто запомните, что это модное словечко обозначает объединение двух и более строк в одну.

Значение символа(ов)

Последний инструмент, который нам понадобится для создания программы-кодировщика/декодера, — это возможность выполнять математические операции с отдельными символами, как, например, прибавление 13 к значению буквы *A* для получения буквы *N*. В языке Python есть несколько функций, которые могут помочь нам в этом.

Каждая буква, число и символ при сохранении в памяти компьютера преобразуется в числовое значение. *ASCII (American Standard Code for Information Interchange — Американский стандартный код для обмена информацией)* — одна из наиболее популярных систем нумерации. В табл. 5.3 приведены значения ASCII для некоторых символов клавиатуры.

Таблица 5.3. *Числовые значения стандартных символов кодировки ASCII*

Значение	Символ	Описание	Значение	Символ	Описание
32		Пробел	65	A	Заглавная A
33	!	Восклицательный знак	66	B	Заглавная B
34	"	Двойные кавычки	67	C	Заглавная C
35	#	Символ решетка	68	D	Заглавная D
36	$	Знак доллара	69	E	Заглавная E
37	%	Процент	70	F	Заглавная F
38	&	Амперсанд	71	G	Заглавная G
39	'	Одинарная кавычка	72	H	Заглавная H
40	(	Левая скобка	73	I	Заглавная I

Значение	Символ	Описание	Значение	Символ	Описание
41	)	Правая скобка	74	J	Заглавная J
42	*	Звездочка	75	K	Заглавная K
43	+	Плюс	76	L	Заглавная L
44	,	Запятая	77	M	Заглавная M
45	-	Тире	78	N	Заглавная N
46	.	Точка	79	O	Заглавная O
47	/	Косая черта, слеш	80	P	Заглавная P
48	0	Ноль	81	Q	Заглавная Q
49	1	Один	82	R	Заглавная R
50	2	Два	83	S	Заглавная S
51	3	Три	84	T	Заглавная T
52	4	Четыре	85	U	Заглавная U
53	5	Пять	86	V	Заглавная V
54	6	Шесть	87	W	Заглавная W
55	7	Семь	88	X	Заглавная X
56	8	Восемь	89	Y	Заглавная Y
57	9	Девять	90	Z	Заглавная Z
58	:	Двоеточие	91	[	Левая квадратная скобка
59	;	Точка с запятой	92	\	Обратный слеш

Значение	Символ	Описание	Значение	Символ	Описание
60	<	Меньше	93	]	Правая квадратная скобка
61	=	Равно	94	^	Степень, циркумфлекс
62	>	Больше	95	_	Подчеркивание
63	?	Вопросительный знак	96	`	Гравис
64	@	Собака	97	a	Строчная a

Функция языка Python, позволяющая преобразовать символ ASCII в числовое значение, — `ord()`:

```
>>> ord('A')
65
>>> ord('Z')
90
```

Обратная функция — `chr()`:

```
>>> chr(65)
'A'
>>> chr(90)
'Z'
```

Эта функция преобразует числовое значение в соответствующий символ.

Программа-кодировщик/декодер

Используя все эти детали, мы можем составить программу, которая принимает сообщения и преобразовывает все буквы в нем в символы верхнего регистра. После этого программа циклически проходит по всем символам в сообщении и, если символ является буквой, передвигает его на 13 позиций для кодирования или раскодирования, добавляет полученную букву к выводимому сообщению — и печатает выводимое сообщение.

EncoderDecoder.py

```
#EncoderDecoderEN — кодирование/раскодирование сообщений
#на английском языке
message = input("Введите сообщение для кодирования или ↵
  раскодирования: ")          # Вывести сообщение
❶ message = message.upper()   # Перевести В ВЕРХНИЙ РЕГИСТР:)
❷ output = ""                 # Создать пустую строку для
                              # вывода
❸ for letter in message:      # Пройти циклом по всем буквам
                              # в сообщении
❹ if letter.isupper():        # Если буква в алфавите (A-Z),
❺ value = ord(letter) + 13    # передвинуть значение буквы на 13,
❻ letter = chr(value)         # преобразовать значение обратно
                              # в букву,
❼ if not letter.isupper():    # проверить, не передвинули ли
                              # мы значение слишком далеко
❽ value -= 26                 # Если передвинули, то вернуться
                              # обратно Z->A
❾ letter = chr(value)         # вычтя 26 из значения буквы
❿ output += letter            # Добавить букву к строке вывода
print("Выходное сообщение: ", output) # Вывести
                              # закодированное декодированное
                              # сообщение
```

Первая строка приглашает пользователя ввести сообщение для кодирования или раскодирования. В строке ❶ функция `upper()` преобразует сообщение в буквы верхнего регистра, чтобы упростить чтение букв программой и написание кода кодировщика. В строке ❷ мы создаем пустую строку с именем `output` (между двойными кавычками `""` ничего нет), в этой строке мы буква за буквой сохраним закодированное сообщение. В цикле `for` в строке ❸ используется тот факт, что Python воспринимает строки как наборы символов; переменная `letter` побуквенно пройдет вместе с циклом по строке `message`.

В строке ❹ функция `isupprer()` проверяет каждый символ в сообщении, чтобы удостовериться в том, что проверяемый символ является буквой верхнего регистра (от `A` до `Z`). Если перед нами буква, то в строке ❺ мы получаем числовое значение этой буквы в кодировке ASCII или Unicode с помощью функции `ord()` и прибавляем к полученному значению 13, чтобы закодировать эту букву. В строке ❻ мы преобразовываем новое, закодированное значение обратно в символ с помощью функции `chr()`, а в строке ❼ мы проверяем, является ли возвращенный символ буквой в диапазоне от `A` до `Z`. Если рассматриваемый символ не попадает в обозначенный диапазон,

мы возвращаем букву в начало алфавита в строке ❽ путем вычитания 26 из закодированного значения (именно так `Z` становится `M`), в строке ❾ мы превращаем новое значение в соответствующую букву.

В строке ❿ мы прибавляем обработанную букву к концу строки `output` (подстановка символа к концу строки) с помощью оператора +=. Оператор += — один из нескольких сокращенных операторов, объединяющих в себе математику (+) и присваивание (=), таким образом, выражение `output += letter` означает, что к строке `output` прибавляется строка `letter`. Это последняя строка цикла `for`, а значит, весь процесс повторяется для каждого символа во введенном сообщении, до тех пор пока программа полностью не составит побуквенно кодированное сообщение в переменной `output`. По завершении цикла последняя строка программы распечатывает выводимое сообщение.

Вы можете использовать эту программу для отправки закодированных сообщений друзьям, но вы должны помнить, что эта программа не предоставляет такой же уровень безопасности, каким располагают современные методы шифрования сообщений. Любой человек, разгадывающий кроссворды в воскресных газетах, сможет прочитать ваше закодированное сообщение, поэтому используйте эту программу только для развлечения с друзьями.

Поищите в Интернете статьи по теме *шифрование* и *криптография*, чтобы узнать о достижениях науки, позволяющих обезопасить секретные послания.

Что вы узнали

В этой главе вы узнали, как запрограммировать компьютер на принятие решений в зависимости от условий, прописанных в коде. Мы увидели, что выражения `if` позволяют программе выполнить набор инструкций только в том случае, если условие истинно (`if age >= 18`). Мы использовали булевы выражения (Истина/Ложь) для представления проверяемых условий,

мы также составляли выражения с помощью условных операторов, таких как >, <, <= и так далее.

Мы объединяли выражения `if` и `else` для выполнения того или иного блока кода таким образом, что если выражение `if` не выполняется, выполняется блок `else`. Мы расширили возможности выбора благодаря использованию выражений `if-elif-else`, как в программе со школьными оценками, где мы ставили ученику оценку «5», «4», «3», «2» или «1» в зависимости от количества баллов, введенных пользователем.

Мы научились проверять несколько условий одновременно, используя для этого логические операторы `and` и `or`, позволяющие сочетать условия (как, например, в выражении `rainy == 'y' and cold == 'y'`). Мы использовали оператор `not` для проверки того, является ли выражение ложным (`False`).

В программе для создания секретных посланий, приводимой в конце этой главы, вы узнали, что все буквы и символы преобразуются в числовые значения при сохранении в памяти компьютера, а также и то, что ASCII — это один из методов сохранения текста в виде числовых значений. Мы использовали функции `chr()` и `ord()` для преобразования символов в код ASCII, а также для обратного преобразования кода в символы. С помощью функций `upper()` и `lower()` мы преобразовывали все буквы строк в буквы верхнего или нижнего регистра, а также проверяли, записана ли строка символами верхнего или нижнего регистра, с помощью функций `isupper()` и `islower()` соответственно. Мы составляли строку путем подстановки букв к концу строки с помощью оператора +, мы также узнали, что сложение нескольких строк в одну называется *подстановкой* или *конкатенацией*.

На данный момент вы должны уметь делать следующее.

- Использовать выражения `if` для принятия решения с помощью условий.
- Использовать условные операторы и булевы выражения для управления ходом выполнения программы.
- Описывать, как булево выражение может быть расценено как истинное (`True`) или ложное (`False`).
- Писать выражения ветвления с использованием операторов сравнения (<, >, ==, !=, <=, =>).
- Использовать сочетания выражений `if-else` для выбора одного из двух альтернативных путей выполнения программы.

- Проверять переменную на четность и нечетность с помощью оператора взятия по модулю %.
- Писать выражения `if-elif-else`, позволяющие осуществлять выбор из нескольких опций.
- Использовать ключевые слова `and` и `or` для одновременной проверки нескольких условий.
- Использовать оператор `not` для проверки того, является ли выражение ложным (`False`).
- Объяснять, как буквы и другие символы сохраняются в виде числовых значений в памяти компьютера.
- Использовать функции `chr()` и `ord()` для преобразования символов в код ASCII, а также для обратного преобразования кода в символы.
- Изменять строки, используя различные строковые функции, например `lower()`, `upper()` и `isupper()`.
- Складывать строки и символы с помощью оператора +.

Задачи по программированию

Чтобы попрактиковаться в том, что вы изучили в этой главе, попробуйте решить эти задачи. (Если вы зашли в тупик, перейдите на сайт **https://eksmo.ru/files/Python_deti.zip** для загрузки готовых решений.)

#1. Цветные розетки и спирали

Для более наглядного решения задач вернитесь на с. 95 и взгляните на рис. 5.1 (спираль из розеток и маленьких спиралей). Попробуйте отредактировать программу *RosettesAndPolygons.py* на с. 108, чтобы сделать рисуемую ею спираль разноцветной, и, если хотите, попробуйте заменить многоугольники на маленькие спирали, чтобы результат был больше похож на рис. 5.1.

#2. Пользовательские ключи

Для тренировки работы с текстом создайте усовершенствованную версию программы *EncoderDecoderEN.py*, которая позволяла бы пользователю ввести собственное ключевое значение в диапазоне от 1 до 25, определяющее, на сколько позиций требуется передвинуть каждую букву в сообщении. Затем, в строке с маркером ❺ программы *EncoderDecoderEN.py* (с. 121), вместо того чтобы передвигать букву на 13 позиций, передвиньте букву на введенное пользователем ключевое значение.

Чтобы раскодировать сообщение, переданное в программу с другим ключом (давайте для примера будем использовать ключевое значение `5`, чтобы буква *A* стала *F*, *B* стала *G* и так далее), человек, получающий сообщение, должен знать ключ. Получатель сможет расшифровать сообщение с помощью обратного ключа (26 минус ключевое значение, 26 – 5 = 21), таким образом, *F* возвращается к букве *A*, тогда как *G* опять становится *B* и так далее.

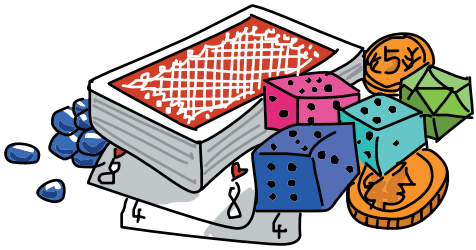
Если вы хотите упростить использование программы, для начала спросите, желает ли пользователь закодировать или раскодировать сообщение (`з` или `р`), а затем запросите у пользователя ключевое значение, которое вы сохраните в переменной `key` (количество мест, на которое нужно передвинуть буквы). Если пользователю нужно закодировать сообщение, в строке ❺ прибавьте введенное ключевое значение к каждой букве, но, если пользователю нужно раскодировать сообщение, к каждой букве прибавьте `26 - key`. Отправьте эту программу другу — и обменяйтесь сообщениями!

Глава 6

СЛУЧАЙНОЕ ВЕСЕЛЬЕ И ИГРЫ: НА УДАЧУ!

В главе 5 мы программировали компьютер на принятие решений на основе условий. В этой главе мы будем программировать компьютер на выбор случайного числа в диапазоне от 1 до 10, играть в «Камень, ножницы, бумагу» и даже бросать игральные кости и вытаскивать карту из колоды!

Все эти игры объединяет одна общая идея, а именно — *случайность*. Мы хотим, чтобы компьютер выбирал случайное число в диапазоне от 1 до 10, а мы будем угадывать это число. Мы хотим, чтобы компьютер случайным образом выбирал камень, бумагу или ножницы, а затем мы будем выбирать, чем играть, и смотреть, кто победил. Все эти примеры плюс игры в кости и карты называются *азартными играми*. Когда, играя в кости, мы бросаем пять кубиков, каждый раз мы получаем отличный от предыдущего результат. Именно этот элемент случайности делает такие игры интересными.



Можно запрограммировать компьютер на случайное поведение. В языке Python есть особый модуль под названием random, позволяющий нам симулировать случайный выбор. Используйте модуль random для отрисовки на экране случайных фигур или для программирования азартных игр. Давайте начнем с игры на угадывание.

Игра на угадывание

Мы можем использовать случайные числа в классической игре на угадывание «Больше-Меньше». Один игрок выбирает число в диапазоне от 1 до 10 (или от 1 до 100), а второй — пытается его угадать. Если названное им число слишком большое, игрок пытается назвать меньшее. Если же названное игроком число оказывается слишком маленьким, то игрок пытается назвать большее число. Если игрок угадывает число — он победил!

Мы уже знаем, как сравнивать числа с помощью выражения `if`, мы также знаем, как пользоваться циклом `while` и как сохранять попытки с помощью `input()`. Единственный новый навык, который мы должны изучить, — это генерация случайных чисел. Мы можем делать это с помощью модуля `random`.

Сначала нам необходимо импортировать модуль `random` с помощью команды `import random`. Вы можете попробовать эту команду в оболочке Python, для этого вам нужно ввести с клавиатуры `import random` и нажать клавишу **Enter**. В модуле реализовано несколько разных функций для генерации случайного числа. Мы будем использовать функцию `randint()` для генерации *случайного целого числа*. Функция `randint()` ожидает от нас получения двух аргументов, то есть двух элементов данных, которые мы будем указывать между скобок после имени функции: наименьшее и наибольшее числа диапазона. Указание в скобках наименьшего и наибольшего числа сообщит функции `randint()` границы диапазона, из которого мы хотим выбирать случайные числа. Введите следующие команды в IDLE:

```
>>> import random
>>> random.randint(1, 10)
```

На что Python ответит выводом случайного числа от 1 до 10 *включительно* (что означает, что случайное число будет включать числа 1 и 10). Попробуйте ввести команду `random.randint(1, 10)` несколько раз

и убедитесь, что программа каждый раз выдает новое случайное число. (Подсказка: вы можете использовать сочетание клавиш **Alt+P** или **Command+P** в macOS, чтобы повторить последнюю введенную команду без необходимости повторно набирать ее с клавиатуры.)

Если вы повторите эту команду много раз (хотя бы 10 раз), то заметите, что числа иногда повторяются, однако никакой отслеживаемой последовательности в числах, на первый взгляд, нет. Такие числа называют *псевдослучайными* числами, потому что они не есть *действительно* случайные (функция `randint` сообщает компьютеру, какое число «выбрать» следующим, основываясь на сложной математической формуле), однако такие числа *кажутся* случайными.

Давайте опробуем работу модуля `random` в программе *GuessingGame.py*. Введите следующий код в новое окно IDLE или скачайте его с сайта **https://eksmo.ru/files/Python_deti.zip**.

GuessingGame.py

```
❶ import random
❷ the_number = random.randint(1, 10)
❸ guess = int(input("Угадайте число от 1 до 10: "))
❹ while guess!= the_number:
❺     if guess > the_number:
            print(guess, "Слишком велико. Попробуйте снова.")
❻     if guess < the_number:
            print(guess, "Слишком мало. Попробуйте снова.")
❼     guess = int(input("Еще одна попытка: "))
❽ print(guess, "Правильное число! Вы победили!")
```

В строке ❶ мы импортируем модуль `random`, который предоставляет нам доступ ко всем функциям, определенным в классе `random`, в том числе и к функции `randint()`. В строке ❷ мы указываем имя модуля `random`, ставим точку — и вписываем имя функции, которую мы хотим использовать, то есть `randint()`. Мы передаем функции `randint()` аргументы `1` и `10` с тем, чтобы данная функция сгенерировала псевдослучайное число от 1 до 10, мы также сохраняем это число в переменной `the_number`. Это и будет то самое секретное число, которое попытается отгадать пользователь.

В строке (3) мы просим пользователя ввести наугад число от 1 до 10, оцениваем число и сохраняем его в переменной с именем `guess`. Наш игровой цикл начинается с выражения `while` в строке ❹. Мы используем

оператор != (не равно), чтобы понять, не равно ли введенное пользователем число секретному. Если пользователь угадывает число с первой попытки, выражение `guess != the_number` расценивается как ложное (`False`) — и цикл `while` не выполняется.

Если введенное наугад пользователем число не равно секретному числу, то с помощью выражений `if` в строках ❺ и ❻ мы проверяем, не было ли введенное число слишком большим (`guess > the_number`) или слишком маленьким (`guess < the_number`), после чего выводим сообщение, просящее пользователя попытаться отгадать еще раз. В строке ❼ мы принимаем еще одну попытку пользователя и повторно запускаем выполнение цикла. Цикл будет повторяться до тех пор, пока пользователь не угадает число.

Строка ❽ выполняется, когда пользователь угадал число, мы сообщаем ему правильное число — и программа завершается. На рис. 6.1 показаны результаты нескольких тестовых запусков программы.

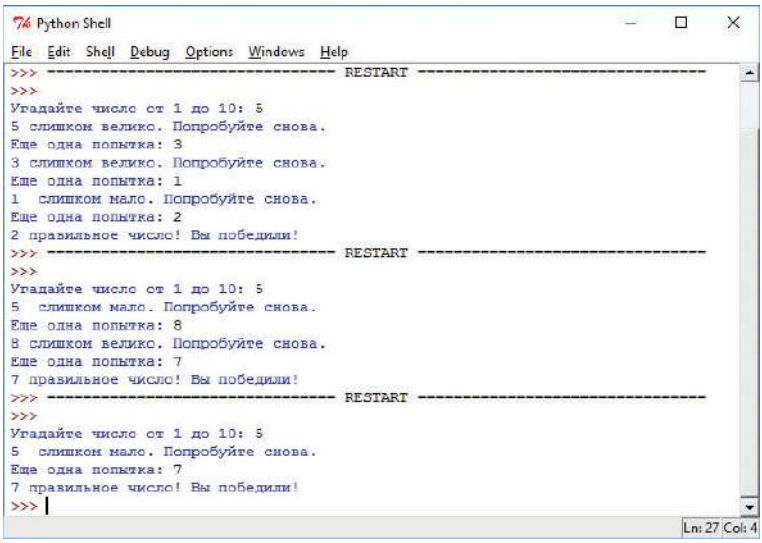


Рис. 6.1. *Программа GuessingGame.py, просящая пользователя угадать число больше или меньше введенного для трех разных случайных чисел*

При первом запуске на рис. 6.1 пользователь ввел наугад число 5, на что компьютер ответил, что 5 — слишком большое число. Тогда пользователь ввел наугад число 2, но 2 оказалось слишком маленьким числом. Тогда пользователь попробовал число 3 — и оказался прав! Последовательное называние наугад чисел в два раза больше или меньше возможных вариантов, как на рис. 6.1, называется *бинарным поиском*. Если игроки обучатся

этой стратегии, то они смогут угадать число от 1 до 10 за четыре попытки или даже быстрее. Каждый раз! Попробуйте!

Чтобы сделать программу более интересной, вы могли бы изменить значения передаваемых функции `randint()` аргументов, например чтобы сгенерировать число от 1 до 100 или до еще бóльшего числа (не забудьте изменить и приглашения к вводу). Вы также могли бы создать переменную `number_of_tries`, к значению которой прибавлялась бы `1` при каждой новой попытке угадать число, чтобы отслеживать количество попыток пользователя. В конце работы программы распечатайте количество попыток, чтобы пользователь выяснил, как хорошо он справился с заданием. В качестве дополнительного задания вы можете создать новый внешний цикл, который спрашивал бы у пользователя, хочет ли он сыграть еще раз после угадывания числа. Попробуйте создать такую программу самостоятельно и перейдите на сайт **https://eksmo.ru/files/Python_deti.zip** для загрузки примеров решения.

Цветные случайные спирали

В модуле `random` есть и другие полезные функции на основе `randint()`. Давайте воспользуемся ими для создания интересной визуализации: заполним экран спиралями случайных размеров и цветов, например, как показано на рис. 6.2.

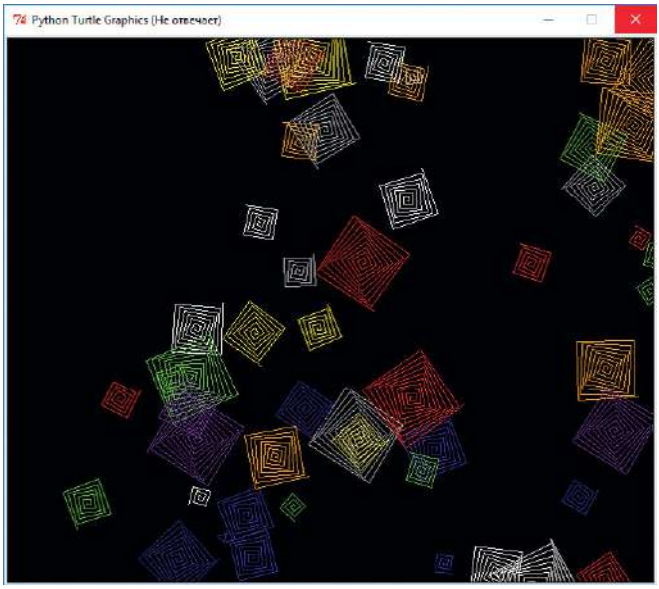


Рис. 6.2. *Спирали случайных размеров и цветов, отрисованные программой RandomSpirals.py в случайных участках экрана*

Подумайте над тем, как бы вы написали программу-аналог той, что создала фигуру на рис. 6.2. Вы уже знаете *почти* все трюки, необходимые для отрисовки таких случайных спиралей. Во-первых, с помощью циклов вы умеете рисовать спирали разных цветов. Вы умеете генерировать случайные числа и использовать их для установки количества повторений цикла `for` каждой спирали. Это позволит изменять размер спирали: чем больше итераций цикла, тем больше спираль, соответственно, меньшее количество итераций создаст меньшую спираль. Давайте посмотрим, что еще нам потребуется для поэтапного построения программы (окончательная версия программы *RandomSpirals.py* представлена на с. 136).

Выбери цвет, любой цвет

Одним из новых инструментов, которыми нам будет необходимо воспользоваться, является возможность выбора случайного цвета. Мы запросто можем это осуществить с помощью другого метода модуля `random`, а именно, `random.choice()`. Функция `random.choice()` принимает список или другую коллекцию в качестве аргумента (указывается в скобках) и возвращает случайным образом выбранный элемент этой коллекции. В нашем случае мы создадим список цветов и затем передадим этот список методу `random.choice()`, чтобы получить случайный цвет для своей спирали.

Вы можете опробовать это в командной оболочке IDLE:

```
>>> # Получение случайного цвета
>>> colors = ["red", "yellow", "blue", "green", "orange", ↵
     "purple", "white", "gray"]
>>> random.choice(colors)
'orange'
>>> random.choice(colors)
'blue'
>>> random.choice(colors)
'white'
>>> random.choice(colors)
'purple'
>>>
```

В этом коде мы вновь создали нашего старого друга — переменную `colors`, которую приравняли к списку названий цветов. Затем мы воспользовались функцией `random.choice()`, передав ей переменную `colors` в качестве аргумента. Данная функция случайным образом выбирает цвет из списка. При первом вызове мы получили оранжевый цвет, при втором голубой, при третьем белый и так далее. Данная функция может вернуть нам случайным образом выбранный цвет, который мы используем для настройки ручки черепашки перед началом рисования каждой спирали.

Учимся координации

Нам осталось решить одну проблему, а именно — как заставить спирали распределиться по всему экрану, в том числе в правом верхнем и левом нижнем углах. Для того чтобы иметь возможность случайным образом помещать спирали на экране черепашки, нам необходимо научиться понимать систему координат *x* и *y* используемой нами среды Turtle.

Картезианские координаты

Если вы изучали геометрию, то вы уже видели координаты (*x*, *y*) на линованной бумаге в клетку, как на рис. 6.3. Это так называемые *картезианские* координаты, получившие свое название в честь французского математика Рене Декарта. Он первым подписал точки на шкале парными цифрами, которые мы теперь называем *координатами x и y*.

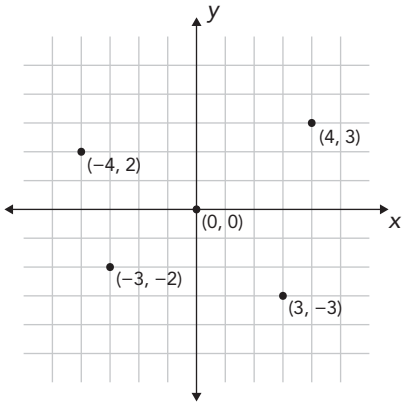


Рис. 6.3. *График с четырьмя точками и их картезианскими (x, y) координатами*

На графике на рис. 6.3 черная горизонтальная линия называется *осью x*, эта ось направлена слева направо. Черная вертикальная линия — это *ось y*, она направлена снизу вверх. Точка пересечения осей (0, 0) называется *центром координат*, так как все остальные точки на шкале подписываются координатами, отмеряемыми от этой точки. Воспринимайте центр координат как центр вашего экрана. Любая точка, которую вам требуется найти, может быть обозначена координатами *x* и *y*, начиная от центра координат и перемещаясь вправо или влево, вверх или вниз.

На графике мы подписываем эти точки двумя координатами в скобках, разделенными запятой: (*x*, *y*). Первое число, координата *x*, говорит нам, как далеко влево или вправо мы переместились, тогда как второе число, координата *y*, указывает на наше перемещение вверх или вниз. Положительные значения *x* говорят о перемещении вправо относительно центра координат, а отрицательные значения *x* — влево. Положительные значения *y* говорят нам переместиться вверх относительно центра координат, а отрицательные значения *y* — вниз.

Взгляните на точки, обозначенные на рис. 6.3. Точка в правом верхнем углу подписана координатами *x* и *y* (4, 3). Чтобы найти местоположение этой точки, нам нужно начать с центра координат (0, 0), переместиться на 4 клетки вправо (так как координата *x* — положительное число 4), а затем на три клетки вверх (так как координата *y* — положительное число 3).

Для того чтобы дойти до точки в правом нижнем углу (3, –3), нам необходимо вернуться к центру координат и отсчитать вправо три клетки, или деления. В этот раз координата *y* — отрицательное число –3, поэтому мы перемещаемся на 3 деления *вниз*. Перемещения на три деления вправо и на три деления вниз приводит нас к точке (3, –3). Чтобы прийти к точке (–4, 2), мы перемещаемся на 4 деления *влево* от центра координат, а затем вверх на 2 деления к точке в левом верхнем углу. Наконец, чтобы попасть к точке (–3, –2), мы перемешаемся влево на 3 деления, а затем на 2 деления вниз — и доходим до нужной точки в левом нижнем углу.

Установка случайной позиции черепашки

При работе с графикой Turtle Graphics мы можем перемещать черепашку от центра координат (0, 0) в любую другую точку на экране, просто сообщив компьютеру координаты *x* и *y* новой точки с помощью команды `turtle.setpos(x, y)`. Название функции `setpos()` — это сокращение от английского «set position» — установить позицию. Данная функция устанавливает черепашку в позицию в соответствии с переданными нами координатами *x* и *y*. Например, команда `turtle.setpos(10, 10)` переместит черепашку на 10 делений вправо и на 10 делений вверх относительно центра экрана.

На компьютере в качестве деления шкалы используется наш старый друг — *пиксель*. Таким образом, команда `turtle.setpos(10, 10)` переместит черепашку на 10 пикселей вправо и на 10 пикселей вверх относительно центра экрана. Так как пиксели очень маленькие — около

1/70 дюйма (0,3 миллиметра) или на большинстве дисплеев еще меньше, то нам, возможно, потребуется перемещаться на 100 пикселей за раз. Функция `setpos()` может обрабатывать любые передаваемые ей координаты.

Для того чтобы переместить черепашку в случайное положение на экране, мы сгенерируем пару случайных чисел, *x* и *y*, затем воспользуемся командой `turtle.setpos(x, y)` для того, чтобы перенести черепашку в точку с этими координатами. Однако, прежде чем перемещать черепашку, нам необходимо отнять ее ручку от экрана с помощью функции `turtle.penup()`. После переноса черепашки на новую позицию мы вызываем функцию `turtle.pendown()`, чтобы обратно опустить ручку на экран и тем самым позволить рисование. Если мы забудем поднять ручку, то черепашка будет рисовать прямую линию по мере перемещения по экрану при выполнении функции `setpos()`. Как вы можете видеть на рис. 6.2, нам не нужны лишние линии между спиралями. Наш код будет выглядеть следующим образом:

```
t.penup()
t.setpos(x, y)
t.pendown()
```

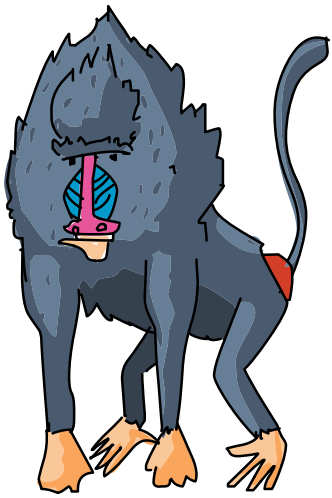
Функция `setpos()` в сочетании с парой случайных чисел, используемых в качестве координат (*x*, *y*), позволит нам размещать спирали в разных позициях на экране. Но как нам узнать, какой диапазон использовать для генерации случайных чисел? Этот вопрос приводит нас к последней проблеме, которую нам необходимо решить на пути создания программы рисования случайных спиралей.

Каков размер холста?

Теперь, когда мы знаем, как установить черепашку в случайную позицию в окне, или на холсте, у нас осталась нерешенной только одна проблема: как нам узнать размер холста? Мы сможем сгенерировать случайное число для установки координат *x* и *y* и нарисовать в этой точке спираль, но откуда нам знать, что выбранная точка находится в видимой части окна, а не вне окна: правее, левее, выше или ниже его?

Чтобы ответить на вопрос о размере холста, нам нужно воспользоваться еще двумя функциями: `turtle.window_width()` и `turtle.window_height()`. Первая из этих функций, `turtle.window_width()`, сообщает нам ширину окна Turtle в пикселях, тогда как функция `turtle.window_height()`

возвращает высоту окна в пикселях, то есть количество пикселей от нижней кромки окна до верхней. Например, окно Turtle на рис. 6.2 имеет ширину 960 пикселей и высоту — 810 пикселей.



Функции `turtle.window_width()` и `turtle.window_height()` помогут нам со случайными координатами *x* и *y*, однако нам нужно преодолеть еще одно препятствие. Как вы помните, при работе с графикой Turtle центр окна является центром координат, или (0, 0). Если мы просто сгенерируем случайное число между 0 и `turtle.window_width()`, во-первых, мы никогда ничего не нарисуем в левом нижнем углу окна, так как для этого сектора обе координаты по осям *x* и *y* всегда отрицательные, однако число между 0 и `turtle.window_width()` всегда положительное. Во-вторых, если мы начнем с центра и дойдем до точки `turtle.window_width()`, то окажемся далеко за пределами правой кромки окна.

Нам нужно не только понять ширину и высоту окна, в котором нам предстоит рисовать, но также определить диапазон координат. Например, если ширина нашего окна 960 пикселей и центр координат находится в центре окна, нам нужно знать, на сколько пикселей мы можем переместиться вправо без покидания видимой области окна. Так как точка (0, 0) находится в середине окна, то есть делит его пополам, все, что нам нужно сделать, — разделить ширину на два. Если центр координат находится в центре окна шириной 960 пикселей, то от центра координат вправо и влево помещается 480 пикселей. Диапазон координаты *x* будет от –480 (левые 480 пикселей от центра координат) до +480 (480 пикселей справа от центра координат), или, иными словами, от –960/2 до +960/2.

Для того чтобы наш диапазон подходил для окна любого размера, мы зададим диапазон координат *x* следующим образом: от `-turtle.window_width()//2` до `+turtle.window_width()//2`. Центр координат также находится по центру окна относительно его верхней и нижней кромки, поэтому над и под центром есть `turtle.window_height()//2` пикселей. В обоих случаях мы используем оператор целочисленного деления `//`, чтобы всегда получать только целочисленные результаты деления на 2.

Ширина и высота окна могут выражаться нечетными числами, а нам нужно, чтобы количество пикселей всегда было представлено целым числом.

Теперь, зная, как вычислить размер холста, мы можем воспользоваться этими выражениями для ограничения пределов случайных координат. В этом случае мы можем быть уверены, что все генерируемые нами случайные координаты попадают в видимую область окна. В модуле `random` языка Python реализована функция, позволяющая нам сгенерировать случайное число, попадающее в указанный диапазон: `randrange()`. Все, что нам нужно, — проинструктировать функцию `randrange()` использовать отрицательную половину ширины окна в качестве начального значения диапазона и положительную половину ширины окна для задания конечного значения диапазона (нам потребуется импортировать оба модуля: turtle и random, чтобы следующие строки кода заработали):

```
x = random.randrange(-turtle.window_width()//2,
                     turtle.window_width()//2)
y = random.randrange(-turtle.window_height()//2,
                     turtle.window_height()//2)
```

В этих строках кода используется функция `randrange()` для генерации пары координат (x, y), которые всегда находятся в видимой области окна и покрывают всю площадь этой области слева направо и снизу вверх.

Соберем все вместе

Теперь у нас есть все элементы этой головоломки, нам лишь нужно соединить их вместе, чтобы построить программу, рисующую случайные спирали разных цветов, разных размеров в разных участках экрана. Далее приведена законченная программа *RandomSpirals.py*: всего 20 строк кода создают графический калейдоскоп, как на рис. 6.2.

RandomSpirals.py

```
import random
import turtle
t = turtle.Pen()
turtle.bgcolor("black")
colors = ["red", "yellow", "blue", "green", "orange", ↵
          "purple", "white", "gray"]
for n in range(50):
    # Генерация спиралей случайных цветов/размеров
    # в случайных местах
```

```
❶     t.pencolor(random.choice(colors)) # Выбрать случайный цвет
❷     size = random.randint(10,40) # Задать случайный размер
      # Сгенерировать случайную (x, y) координату на экране
❸ x = random.randrange(-turtle.window_width()//2,
                       turtle.window_width()//2)
❹ y = random.randrange(-turtle.window_height()//2,
                       turtle.window_height()//2)

❺     t.penup()
❻     t.setpos(x, y)
❼     t.pendown()
❽     for m in range(size):
          t.forward(m*2)
          t.left(91)
```

Первое, что нам нужно сделать, — импортировать модули `random` и `turtle`, настроить окно Turtle и создать список цветов. В цикле `for` (переменная `n` увеличивается от `0` до `49`, что позволит нам отрисовать 50 спиралей) все становится гораздо интересней. В строке ❶ мы передаем переменную-список `colors` функции `random.choice()`, чтобы эта функция выбрала случайный цвет из списка. Затем мы передаем случайно выбранный цвет в функцию `t.pencolor()`, задающую цвет ручки. В строке ❷ функция `random.randint(10,40)` выбирает случайное число от 10 до 40. Мы сохраняем это число в переменной `size`, которой мы воспользуемся в строке ❽, чтобы сообщить Python, спираль из скольких линий мы хотели бы отрисовать. Строки ❸ и ❹ — это именно те команды, которые мы использовали ранее для генерации случайной пары координат (x, y), которые предоставляют нам случайную позицию в видимой области окна просмотра.

В строке ❺ мы поднимаем ручку черепашки над виртуальным листом бумаги, прежде чем перенести черепашку на новую случайную точку. В строке ❻ мы перемещаем черепашку на новую случайную точку, установив ее положение равным значению переменных `x` и `y`, то есть случайным координатам, выбранным ранее с помощью функции `randint()`. Теперь, когда черепашка заняла новое положение, мы возвращаем ручку на экран в строке ❼, чтобы иметь возможность видеть отрисовываемую спираль. В строке ❽ мы реализовали цикл `for`, ответственный за отрисовку каждой линии спирали. Для `m` в диапазоне `range(size)` черепашка переместится вперед на расстояние `m*2`, рисуя при этом отрезок прямой линии с длиной `m*2` (переменная `m` последовательно имеет значения `0`, `1`, `2`, `3` и так далее).

После этого черепашка повернет влево на 91 градус и приготовится рисовать новый отрезок.

Черепашка начинает отрисовку с центра спирали, рисует отрезок (длина 0), поворачивает влево — это первая линия, рисуемая циклом. При следующем проходе по циклу переменная `m` уже равна `1`, поэтому черепашка рисует отрезок длиной 2, а затем поворачивает. По мере выполнения итераций цикла черепашка будет все больше и больше отдаляться от центра спирали, рисуя все более и более длинные отрезки. Мы используем случайным образом сгенерированный размер `size`, целое число от 10 до 40, как количество линий отрисовываемой спирали.

Когда завершается рисование текущей спирали, мы возвращаемся к верхней строке цикла `for`. Программа выбирает новый случайный цвет, размер и местоположение, поднимает ручку, переносит ее на новую точку, вновь опускает ручку и проходит по внутреннему циклу `for` для рисования новой спирали некоего нового размера. Когда и эта спираль нарисована, мы вновь возвращаемся к внешнему циклу и повторяем весь процесс заново. Мы делаем все это 50 раз, что дает нам 50 спиралей разных цветов и размеров, разбросанных случайным образом по всему экрану.

Камень, ножницы, бумага

У нас уже достаточно навыков, чтобы запрограммировать игру «Камень, ножницы, бумага». В этой игре два игрока (или один игрок и компьютер) выбирают по одному из трех возможных предметов (камень, ножницы или бумагу), оба показывают, что они выбрали. Победитель в этой игре определяется по трем правилам: камень давит ножницы, ножницы режут бумагу, бумага накрывает камень.

Для того чтобы симулировать эту игру, мы создадим список опций (наподобие списка цветов `colors` из программы *RandomSpirals.py*). Мы также воспользуемся функцией `random.choice()`, чтобы выбрать один из трех предметов из списка на усмотрение компьютера. Затем мы попросим пользователя также сделать свой выбор — и воспользуемся несколькими выражениями `if`, чтобы определить победителя. Пользователь будет играть против компьютера!

Давайте перейдем к написанию кода. Введите код программы *RockPaperScissors.py* в новое окно IDLE или скачайте его с сайта **https://eksmo.ru/files/Python_deti.zip**.

RockPaperScissors.py

```
❶ import random
❷ choices = ["камень", "ножницы", "бумага"]
  print("Камень давит ножницы. Ножницы режут бумагу. Бумага ↵
        накрывает камень.")
❸ player = input("Выберите: камень, ножницы, бумага или ↵
                 (выйти)? ")
❹ while player!= "выйти": # Играть, пока пользователь не выйдет
      player = player.lower() # Поменять ввод пользователя
                              # на нижний регистр
❺     computer = random.choice(choices) # Выбрать один
                                        # из предметов
      print("Твой выбор " +player+ ", компьютер выбрал " ↵
        +computer+ ".")
❻     if player == computer:
            print("Ничья!")
❼     elif player == "камень":
            if computer == "ножницы":
                  print("Вы победили!")
            else:
                  print("Победил компьютер!")
❽     elif player == "бумага":
            if computer == "камень":
                  print("Вы победили!")
            else:
                  print("Победил компьютер!")
❾     elif player == "ножницы":
            if computer == "бумага":
                  print("Вы победили!")
            else:
                  print("Победил компьютер!")
      else:
            print("По-моему, произошла ошибка…")
      print() # Пропустить строку
❿     player = input("Выберите: камень, ножницы, бумага или ↵
                     выйти)? ")
```

В строке ❶ мы импортируем модуль random, чтобы получить доступ к функциям, помогающим нам сделать случайный выбор. В строке ❷ мы создаем список из трех предметов: камень, ножницы и бумага — и называем этот список `choices`. В строке ❸ мы приглашаем пользователя ввести свой выбор: `камень`, `ножницы`, `бумага` или `выйти` — и сохраняем выбор в переменной `player`. В строке ❹ мы запускаем игровой цикл с проверки,

не ввел ли пользователь команду выйти в ответ на приглашение программы. Если пользователь ввел эту команду, игра прекращается.

Если же пользователь не ввел команду выйти, начинается игра. После преобразования введенного пользователем текста в символы нижнего регистра для облегчения операций сравнения в выражениях `if` мы говорим компьютеру выбрать предмет. В строке ❺ мы говорим компьютеру случайным образом выбрать один из элементов списка `choices` и сохранить свой выбор в переменной `computer`. Как только выбор компьютера сохранен, наступает время финальной проверки на определение победителя. В строке ❻ мы проверяем, не получилось ли так, что пользователь и компьютер выбрали один и тот же предмет. Если это так, то мы сообщаем пользователю о ничьей. В противном случае в строке ❼ мы проверяем, не выбрал ли пользователь камень. В выражении `elif`, также в позиции ❼, мы помещаем вложенное выражение `if`, чтобы проверить, не выбрал ли компьютер ножницы. Если наш игрок выбрал камень, а компьютер — ножницы, то камень давит ножницы — и пользователь выигрывает! Если же это не камень и компьютер не выбрал ножницы, значит, компьютер должен был выбрать бумагу, соответственно, мы выводим сообщение, что компьютер победил.

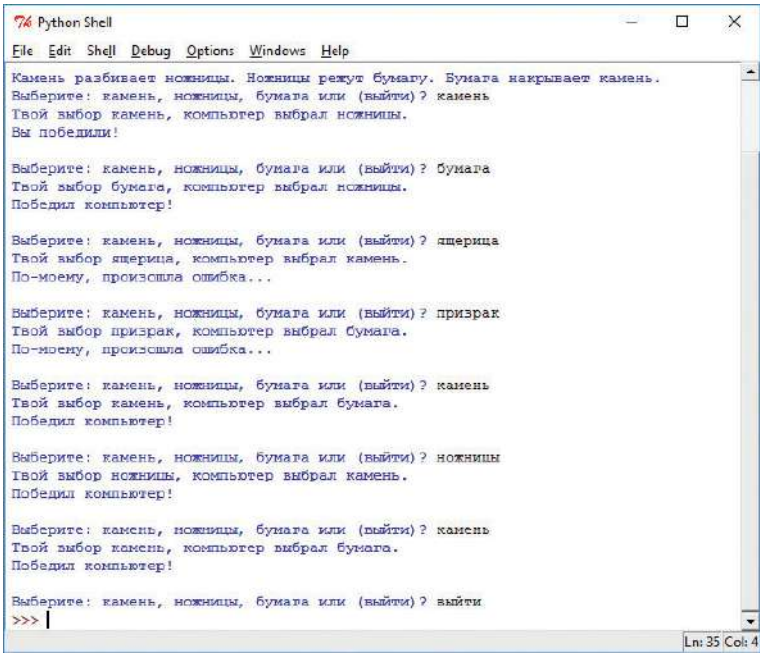


Рис. 6.4. *Благодаря возможности компьютера делать случайный выбор игра «Камень ножницы, бумага» RockPapersScissors.py может быть интересной!*

С помощью оставшихся двух выражений `elif` в строках ❽ и ❾ мы делаем ту же самую проверку, не выбрал ли пользователь бумагу или ножницы. Если ни одно из выражений не вернуло значение `True` (Истина), то мы сообщаем пользователю, что введенная им команда не может быть обработана, то есть пользователь либо выбрал несуществующую опцию или допустил опечатку. Наконец, в строке ❿ мы вновь просим пользователя сделать выбор перед повторением игрового цикла (нового раунда). На рис. 6.4 показан результат тестового запуска программы.

Иногда побеждает пользователь, иногда компьютер, а иногда у них ничья. Так как результат игры в достаточной степени непредсказуем, она некоторое время может быть довольно увлекательным занятием. Теперь, когда мы знаем, как можно использовать возможности компьютера делать случайный выбор в играх, рассчитанных на двух игроков, давайте попробуем создать карточную игру.

Выберите карту, любую карту

Случайность — это то, что делает карточные игры интересными. Ни один раунд не может быть повторен полностью (конечно, если вы умеете тасовать карты), поэтому вы можете продолжать играть снова и снова, и это вам не наскучит.

Используя изученные нами навыки, мы можем создать простую карточную игру. Наша первая попытка создания такой игры не будет отображать графические карты, так как для этого нам потребуется изучить еще несколько трюков, но мы можем сгенерировать случайный номинал карты («двойка бубны», «король крести», например), просто создав *массив*, или список, строк, наподобие тех, что мы создавали для программ по отрисовке спиралей. Мы можем создать игру «Пьяница», в которой пользователи должны по очереди вынимать карты из колоды. Победит тот, чья карта будет иметь больший номинал. Все, что нам нужно, — это понять, каким образом мы можем сравнивать карты пользователей. Давайте пошагово рассмотрим, как такая схема может работать. (Окончательная версия программы *HighCard.py* представлена на с. 149.)

Тасуем колоду

Во-первых нам нужно подумать над тем, как создать виртуальную колоду карт. Как я уже говорил ранее, на данном этапе мы не будем рисовать карты, однако для симуляции колоды нам нужны хотя бы номиналы и масти

карт. К счастью, карточные игры — это лишь строки (`"двойка бубны"`, `"король крести"`), а мы с вами уже знаем, как создавать массивы строк: мы делали массивы имен цветов с самой первой главы!

Массив — это упорядоченный или пронумерованный набор однотипных элементов. Во многих языках программирования массив — это особый тип коллекции. Однако в языке Python списки могут использоваться как массивы. В этой главе мы рассмотрим использование списка в качестве массива и то, как получить доступ к его отдельным элементам.

Мы могли бы создать список всех карт, просто создав массив, имя массива (`cards`) и приравняв его к списку 52 карт:

```
cards = ["двойка бубны", "тройка бубны", ↵
         "четверка бубны",
# Это займет у нас целую вечность…
```

Но о ужас! Нам нужно ввести 52 длинные строки из названий карт! Наш код уже будет иметь 52 строки, а мы еще даже не начали программировать сам игровой процесс, мы настолько устанем вводить код, что на игру энергии у нас просто не останется. Должен быть способ получше… Давайте думать как программисты! Ввод с клавиатуры — это повторяющаяся задача, а всю работу, связанную с повторениями, мы хотим переложить на компьютер. Названия мастей (*бубны*, *черви*, *пики*, *крести*) повторятся 13 раз, так как в масти 13 карт. Номиналы карт (от *двойки* до *туза*) повторятся 4 раза, так как мастей всего 4. Хуже всего то, что нам нужно будет ставить кавычки 104 раза!

Ранее, когда мы сталкивались с повторением, мы создавали циклы для упрощения решения проблем. Если бы мы хотели сгенерировать целую колоду карт, то цикл бы отлично справился с этой работой, однако нам не нужна вся колода карт, которой нам пришлось бы играть в одиночку: нам нужно только две карты: карта компьютера и карта пользователя. Если цикл не может нам помочь избежать повторения перечня мастей и номиналов карт, значит, нам нужно разбить эту проблему на еще более мелкие задачи.

В игре «Пьяница» каждый игрок показывает одну карту, игрок с картой большего номинала выигрывает. Поэтому нам нужно только 2 карты, а не 52. Давайте начнем с одной карты. Название карты состоит из номинала (от двойки до туза) и названия масти (от пик до крестей). Это похоже

на отличную возможность для создания двух списков строк: один список для номиналов и один для мастей. Вместо использования 52 повторяющихся записей для каждой карты мы просто случайным образом выбираем номинал из списка с 13 вариантами, а затем так же случайно выбираем название масти из 4 возможных вариантов.

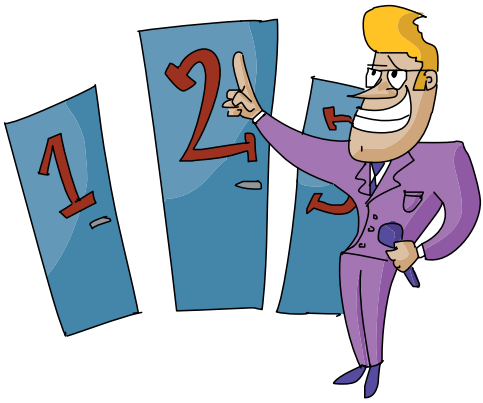
Мы заменим наш длинный массив `cards` двумя значительно более короткими `suits` и `faces`:

```
suits = ["пики", "бубны", "черви", "крести"]
faces = ["двойка", "тройка", "четверка", "пятерка", ↵
         "шестерка", "семерка", "восьмерка", "девятка", ↵
         "десятка", "валет", "дама", "король", "туз"]
```

Мы сократили 52 строки кода до 3–4! Вот это и есть разумное программирование. Давайте теперь посмотрим, как пользоваться этими массивами карт.

Раздаем карты

Мы уже знаем, как пользоваться функций `random.choice()` для выбора случайного элемента списка. Поэтому, для того чтобы выдать пользователю карту, мы просто воспользуемся функцией `random.choice()`, чтобы выбрать номинал из списка номиналов, а затем и название масти из списка мастей.



Обратите внимание, что при таком использовании функции `random.choice()` мы можем выдать одну и ту же карту два раза подряд. Мы не заставляем программу проверять, не была ли карта уже выдана, поэтому вы можете получить подряд два туза крести, например. Компьютер не мухлюет: просто мы не говорим ему сдавать карты из одной колоды. Программа как будто бы сдает карты из *бесконечной колоды*, то есть компьютер может вечно выдавать карты, и они у него никогда не закончатся.

```
import random
suits = ["пики", "бубны", "черви", "крести"]
faces = ["двойка", "тройка", "четверка", "пятерка", ↵
         "шестерка", "семерка", "восьмерка", "девятка", ↵
         "десятка", "валет", "дама", "король", "туз"]
my_face = random.choice(faces)
my_suit = random.choice(suits)
print ("У меня ", my_face, " ", my_suit)
```

При каждой попытке запуска этого кода компьютер будет выдавать вам новую случайную карту. Чтобы сдать вторую карту, вы воспользуетесь аналогичным блоком кода, но сохраните случайный выбор в переменных `your_face` и `your_suit`. Вам также нужно будет изменить выражение `print` так, чтобы оно выводило название новой карты. Вы уже вплотную приблизились к игре «Пьяница», но нам нужно придумать какой-то способ сравнения карты компьютера и карты пользователя, чтобы понять, кто же выиграл.

Подсчет карт

В том, что мы перечислили номиналы карт по возрастанию, есть определенный смысл. Нам нужно, чтобы список `faces` был пронумерован по возрастанию с тем, чтобы мы могли сравнить карты друг с другом и понять, которая карта из двух больше. Нам важно определить, какая из карт больше, так как в игре «Пьяница» в каждом туре побеждает наибольшая из карт.

Поиск элемента списка

К счастью, благодаря особенностям работы списков и массивов в языке Python мы можем определить, где именно в списке появляется то или иное значение, и можем использовать эту информацию для определения того, которая из двух карт больше. Порядковый номер элемента списка или массива называется *индексом* этого элемента. Как правило, мы обращаемся к элементам массивов по индексам.

В табл. 6.1 приведено визуальное представление массива `suits` и индекс каждой масти.

Таблица 6.1. *Массив* `suits`

значение	"пики"	"бубны"	"черви"	"крести"
индекс	0	1	2	3

Когда мы создаем список suits, Python автоматически присваивает индекс всем значениям этого списка. Компьютер начинает отсчет с нуля, поэтому индекс элемента "пики" 0, масть "бубны" имеет индекс 1 и так далее. Функция, позволяющая найти индекс элемента, — .index(), данная функция может быть использована с любым списком или массивом языка Python.

Чтобы найти индекс названия масти "пики", мы вызываем функцию suits.index("пики"). Иными словами, мы запрашиваем у массива suits, какой индекс соответствует значению "пики". Давайте испробуем это в оболочке Python. Введите следующие команды:

```
>>> suits = ["пики", " бубны", " черви", " крести"]
>>> suits.index("пики")
0
>>> suits.index("крести")
3
>>>
```

После того как мы создали массив значений мастей, suits, мы спрашиваем у Python, каков индекс значения "пики", — и компьютер выдает нам правильный индекс: 0. Аналогичным образом, индекс масти "крести" 3, а масти бубны и червы находятся в положениях 1 и 2 соответственно.

Какая карта больше?

Мы создали массив faces так, что номиналы карт расположены в порядке увеличения: от двойки до туза, таким образом, значение двойка, первый элемент массива faces, получит индекс 0, и так по всему списку до туза, которому будет присвоен индекс 12 (13-я позиция, начиная с 0). Мы можем использовать индекс, чтобы определить, значение какой из двух карт больше, иными словами, индекс номинала которой карты больше. Самая меньшая карта двойка имеет самый меньший индекс 0. Тогда как туз — самая большая карта с самым большим индексом 12.

Если мы сгенерируем два случайных значения номинала карт (my_face и your_face), то мы сможем сравнить индекс значения my_face с индексом значения your_face, чтобы понять, какая из двух карт больше, следующим образом.

```
import random
faces = ["двойка", "тройка", "четверка", "пятерка", ↵
         "шестерка", "семерка", "восьмерка", "девятка", ↵
         "десятка", "валет", "дама", "король", "туз"]
my_face = random.choice(faces)
your_face = random.choice(faces)
if faces.index(my_face) > faces.index(your_face):
    print ("Я победил!")
elif faces.index (my_face) < faces.index(your_face):
    print ("Вы победили!")
```

Мы дважды используем функцию `random.choice()`, чтобы извлечь случайные значения из массива `faces`, а затем сохранить их в переменных `my_face` и `your_face`. Мы используем команду `faces.index(my_face)`, чтобы узнать индекс элемента `my_face` в массиве `faces`, а также команду `faces.index(your_face)`, чтобы узнать индекс элемента `your_face`. Если индекс элемента `my_face` больше, значит, и номинал этой карты выше, соответственно, программа распечатает сообщение `Я победил!`. В противном случае, если индекс `my_face` меньше индекса `your_face`, значит, номинал вашей карты больше, поэтому программа выведет сообщение `Вы победили!`. Благодаря тому, в каком порядке был создан список, старшей карте всегда будет соответствовать больший индекс. Благодаря этому полезному инструменту мы имеем практически все необходимое для создания простой карточной игры, такой как «Пьяница». (Мы еще не добавили возможность проверки на ничью, однако мы добавим эту часть в законченную версию программы, представленную в разделе «Соберем все вместе» на с. 148.)

Выполнение программы без остановки

Последний инструмент, который нам понадобится, — это цикл, который позволит пользователю играть так долго, как он этого захочет. Однако этот цикл мы создадим несколько иначе: так, чтобы у нас была возможность повторно его использовать в других играх.

В первую очередь нам нужно решить, цикл какого типа нам нужно использовать. Вы, наверное, помните: цикл `for`, как правило, означает, что мы точно знаем требуемое количество повторений некоего блока кода. Так как мы не всегда можем предсказать, сколько раз пользователь захочет сыграть в нашу игру, цикл `for` нам не подходит. Цикл `while` может выполняться до тех пор, пока некое условие не станет ложным, например, когда

пользователь нажимает клавишу для выхода из программы. Для создания игрового цикла мы воспользуемся именно циклом `while`.

Циклу `while` требуется передать проверяемое на истинность условие, поэтому мы создадим переменную, которую будем использовать в качестве *флага*, или сигнала окончания программы. Давайте назовем переменную-флаг `keep_going` и с самого начала присвоим ей значение `True`:

```
keep_going = True
```

Так как мы начинаем с команды `keep_going = True`, программа войдет в игровой цикл хотя бы один раз.

Затем мы спросим пользователя, хочет ли он продолжать играть. Вместо того чтобы заставлять пользователя вводить `Д` или `да` каждый раз, когда он хочет продолжить игру, ограничимся тем, что просто попросим его нажать клавишу **Enter**.

```
answer = input("Нажмите [ENTER], чтобы продолжить, или ↵
               любую клавишу, чтобы выйти: ")
if answer == "":
    keep_going = True
else:
    keep_going = False
```

В приведенном выше участке кода мы приравниваем переменную `answer` к функции. Затем мы используем выражение `if`, чтобы проверить равенство `answer == ""`, чтобы увидеть, нажал ли пользователь клавишу **Enter**, или перед нажатием были нажаты другие клавиши. (Пустая строка `""` говорит нам, что пользователь не вводил никаких символов перед нажатием **Enter**.) Если пользователь хочет выйти из игры, все, что ему нужно сделать, — приравнять переменную `answer` к любой непустой строке `""`. Иными словами, перед нажатием **Enter** пользователю нужно нажать любую клавишу или несколько клавиш, в результате чего логическое выражение `answer == ""` вернет результат Ложь (`False`).

Выражение `if` проверяет истинность выражения `answer == ""`. Если выражение истинно, то значение `True` сохраняется в переменной-флаге `keep_going`. Не заметили ли вы некую повторяемость выполняемых операций? Если выражение `answer == ""` истинно (`True`), мы присваиваем значение `True` переменной `keep_going`, если выражение `answer == ""`

расценивается как ложное (`False`), нам необходимо присвоить значение `False` переменной `keep_going`.

Было бы гораздо проще приравнять переменную `keep_going` к любому результату оценки выражения `answer == ""`. Мы можем заменить вышеприведенный код следующим, более содержательным и изящным аналогом.

```
answer = input("Нажмите [ENTER], чтобы продолжить, или ↵
               любую клавишу, чтобы выйти: ")
keep_going = (answer == "")
```

Первая строка не изменилась. Вторая строка приравнивает переменную `keep_going` к результату логического выражения `answer == ""`. Если это выражение истинно (`True`), переменной `keep_going` будет присвоено значение `True`, и программа продолжит выполнение цикла. Если же выражение ложно (`False`), переменной `keep_going` будет присвоено значение `False` — и выполнение цикла будет прекращено.

Давайте посмотрим, как выглядит цикл полностью.

```
keep_going = True
while keep_going:
    answer = input("Нажмите [ENTER], чтобы продолжить, или ↵
                   любую клавишу, чтобы выйти: ")
    keep_going = (answer == "")
```

В вышеприведенном коде мы добавили выражение `while`, таким образом, наш цикл будет продолжать выполняться, пока переменная `keep_going` истинна (`True`). В окончательной версии программы мы «обернем» этот цикл вокруг кода, разыгрывающего один раунд. Для осуществления этого мы поместим выражение `while` перед участком кода, выбирающим карты, а приглашение нажать на клавишу — после кода, сообщающего, кто победил. Не забудьте отформатировать отступами код внутри цикла!

Соберем все вместе

Собрав вместе все компоненты, мы можем создать игру, наподобие карточной игры «Пьяница», которую мы назовем *HighCard.py*. Компьютер выдает карту себе и игроку, проверяет, какая из двух карт больше, и объявляет победителя. Введите код программы *HighCard.py* в новое окно IDLE или перейдите на сайт **https://eksmo.ru/files/Python_deti.zip**, чтобы загрузить код себе на компьютер и сыграть в игру.

HighCard.py

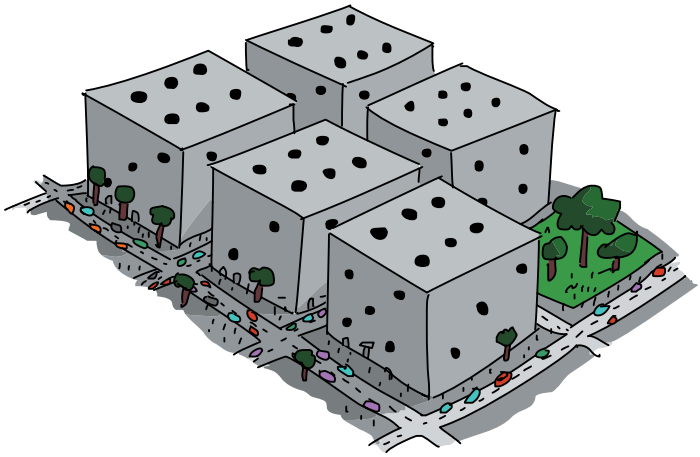
```
import random
suits = ["пики", "бубны", "черви", "крести"]
faces = ["двойка", "тройка", "четверка", "пятерка", ↵
         "шестерка", "семерка", "восьмерка", "девятка", ↵
         "десятка", "валет", "дама", "король", "туз"]
keep_going = True
while keep_going:
    my_face = random.choice(faces)
    my_suit = random.choice(suits)
    your_face = random.choice(faces)
    your_suit = random.choice(suits)
    print("У меня", my_face, " ", my_suit)
    print("У вас", your_face, " ", your_suit)
    if faces.index(my_face) > faces.index(your_face):
    print("Я победил!")
    elif faces.index(my_face) < faces.index(your_face):
         print("Вы победили!")
    else:
         print("У нас ничья!")
    answer = input("Нажмите [Enter], чтобы продолжить, или ↵
                    любую клавишу, чтоб выйти: ")
    keep_going = (answer == "")
```

Запустите игру, на экран будет выведена карта компьютера и ваша карта, а также сообщение о том, кто выиграл, и приглашение, предлагающее вам сыграть еще или выйти из игры. Сыграйте несколько раундов — и вы заметите, что выбор карт достаточно случаен для того, чтобы сделать результат интересным: иногда побеждает компьютер, иногда вы. Игра интересна благодаря элементу неожиданности.

Кидаем кубики: игра в кости в стиле яцзы

При создании карточной игры мы использовали массивы, чтобы упростить код, необходимый для раздачи карт и проверки того, какая из карт имеет более высокий номинал, основываясь на положении карты в списке карт. В этом разделе мы воспользуемся концепцией массива для генерации пяти случайных игральных кубиков, а также для проверки того, выпали ли три, четыре или пять одинаковых граней. Все это — упрощенная версия игры яцзы (покер на костях).

В игре яцзы у игрока пять игральных кубиков. У каждого кубика шесть граней, на каждой грани с помощью точек нанесено число от 1 до 6. В полной версии игры пользователь бросает пять кубиков, стараясь получить очки при выпадении трех граней с одинаковым значением (что мы называем *одинаковыми гранями*), а также при выпадении других комбинаций, почти как в карточном покере. Выпадение пяти одинаковых граней (скажем, все шесть граней с шестью точками обращены вверх) называется комбинацией яцзы. При выпадении такой комбинации игрок получает самое большое из возможных количество очков. В нашей упрощенной версии игры мы просто симулируем бросок пяти кубиков, после чего проверим, выпали ли три или четыре одинаковые грани, а также не выпала ли комбинация яцзы, а также сообщим пользователю результат.



Настройка игры

Теперь, когда мы понимаем цель игры, давайте поговорим о том, как мы будем писать для нее программный код. Во-первых, нам потребуется создать игровой цикл, чтобы пользователь мог повторно кидать кубики, пока не захочет выйти. Во-вторых, нам понадобится создать набор из пяти симулированных игральных кубиков в качестве массива, который может содержать пять случайных значений от 1 до 6, представляющих номинал каждого брошенного кубика. В-третьих, мы симулируем выброс кубиков путем присвоения случайного значения в диапазоне от 1 до 6 в каждый слот массива. Наконец, нам потребуется сравнить пять брошенных кубиков и проверить, не выпало ли три, четыре или пять одинаковых значений, а затем сообщить пользователю результат.

Последняя часть, возможно, самая сложная. Мы могли бы проверить выпадение комбинации яцзы, проведя проверку того, что номинал всех кубиков равен 1 или 2 и так далее, но это обозначало бы создание длинной вереницы сложных выражений `if`. Так как нам не важно, выпало ли у нас пять единиц, двоек или шестерок, нам важно лишь то, что выпало пять одинаковых граней, мы можем несколько упростить процесс проверки до выяснения того, не равно ли значение первого кубика значению второго, а значение второго — значению третьего и так далее до пятого. Затем, вне зависимости от того, каков номинал пяти одинаковых кубиков, мы знаем, что все пять кубиков одинаковые, то есть у нас выпала комбинация яцзы.

Кажется, что выпадение пяти одинаковых граней очень легко проверить, но давайте попробуем представить сам алгоритм проверки. Возможная комбинация четырех одинаковых граней может быть представлена массивом значений, например `[1, 1, 1, 1, 2]` (в этом случае выпало четыре единицы и одна двойка). Однако массив `[2, 1, 1, 1, 1]` также будет представлять комбинацию из четырех единиц, как, собственно, и массивы `[1, 1, 2, 1, 1]`, `[1, 2, 1, 1, 1]` и `[1, 1, 1, 2, 1]`. И это пять возможных конфигураций только для проверки четырех единиц! Выглядит так, будто нам потребуется очень длинная вереница условий `if`...

К счастью, будучи умелым программистом, вы знаете, что, обычно, существует более простой способ решения подобных задач. У всех массивов из предыдущего абзаца есть одна общая черта — четыре единицы в списке значений. Однако загвоздка в том, что пятое значение, `2`, может находиться в любой позиции в массиве. Гораздо проще было бы проверить, если все четыре единицы следовали друг за другом, а двойка отстояла отдельно. Если бы мы могли отсортировать массив, например, по возрастанию или по убыванию, то все единицы были бы сгруппированы вместе, уменьшая пять возможных комбинаций до всего лишь двух: `[1, 1, 1, 1, 2]` или `[2, 1, 1, 1, 1]`.

Сортировка кубиков

В языке Python списки, коллекции и массивы обладают встроенной функцией `sort()`, которая позволяет нам отсортировать элементы массива по значению от наименьшего до наибольшего и наоборот. Например, если мы назовем массив кубиков `dice`, то сможем отсортировать хранящиеся в нем значения с помощью команды `dice.sort()`. По умолчанию функция `sort()` отсортирует элементы массива `dice` от меньшего к большему, то есть *по возрастанию*.

Для нашей проверки массива на предмет содержания в нем четырех одинаковых значений сортировка массива означает, что нам нужно лишь проверить два условия: четыре одинаковых низких значения и одно высокое значение (как в случае с массивом `[1, 1, 1, 1, 2]`) или одно низкое значение и четыре одинаковых высоких значения (например, `[1, 3, 3, 3, 3]`). В первом случае мы знаем, что если кубики отсортированы и первый с четвертым элементом равны по значению, то перед нами комбинация из четырех одинаковых граней или лучше. Во втором же случае мы опять же имеем дело с отсортированными кубиками, и если второй и пятый кубик равны по значению, то перед нами комбинация из четырех одинаковых граней или лучше.

Мы говорим «комбинация из четырех одинаковых граней или *лучше*», так как равенство первого и четвертого кубика также верно при выпадении комбинации из пяти одинаковых кубиков. Это приводит нас к первой логической задаче: если пользователю выпадает комбинация из пяти одинаковых кубиков, это также значит, что им выпала и комбинация из четырех одинаковых кубиков, но нам нужно присвоить пользователю очки только за лучшую комбинацию. Мы решим эту задачу с помощью цепочки выражений `if-elif` таким образом, что, если пользователю выпадает комбинация яцзы, он не получает очки за выпадение также комбинации из четырех и трех одинаковых граней. Сочетание последовательности выражений `if-elif` с тем, что мы узнали о сортировке кубиков для проверки на выпадение четырех одинаковых граней, приведет нас следующему программному коду:

```
if dice[0] == dice[4]:
    print("Яцзы!")
elif (dice[0] == dice[3]) or (dice[1] == dice[4]):
    print("Четыре одинаковых!")
```

Во-первых, если мы уже отсортировали массив с кубиками, мы отмечаем сокращение вариантов: если первый и последний кубик имеют одинаковый номинал (`if dice[0] == dice[4]`), мы точно знаем, что нам выпало сочетание яцзы! Помните, что первые пять элементов массива нумеруются с 0 до 4. Если пять одинаковых кубиков не выпало, то нам нужно проверить оба варианта выпадения четырех одинаковых граней (первые четыре кубика одинаковы `dice[0] == dice[3]`, или одинаковы последние четыре кубика `(dice[1] == dice[4])`). В данном случае мы используем

логический оператор `or`, чтобы распознать комбинацию из четырех одинаковых кубиков, если истинно (`True`) *хотя бы одно* из двух выражений (первые четыре *или* последние четыре).

Проверка кубиков

Мы ссылаемся на каждый элемент массива по отдельности по его индексу, или позиции: `dice[0]` означает первый элемент массива кубиков, а `dice[4]` — ссылается на пятый элемент, так как мы начинаем счет с нуля. Именно таким образом мы можем проверить значение каждого отдельно взятого кубика и сравнить один кубик с другим. Как и в случае с массивом `suits[]`, проиллюстрированном в табл. 6.1, каждая запись массива `dice` представляет собой отдельное значение. При вызове элемента `dice[0]` для проверки его равенства элементу `dice[3]` мы смотрим на значение первого элемента массива `dice` и сравниваем его с четвертым элементом массива `dice`. Если массив отсортирован и эти элементы равны, значит, перед нами комбинация из четырех одинаковых кубиков.

Чтобы проверить выпадение комбинации из трех одинаковых кубиков, нам необходимо добавить еще одно выражение `elif`, кроме того, мы помещаем проверку трех кубиков после проверки четырех кубиков, с тем чтобы проводить проверку трех одинаковых граней только в том случае, если комбинация из пяти или четырех одинаковых кубиков не выпала. Нам также необходимо сообщить о выпадении только наивысшей комбинации. В случае с сортированными кубиками возможны три варианта выпадения комбинации из трех одинаковых кубиков: совпадают первые три кубика, три кубика посередине либо последние три. В коде это будет выглядеть следующим образом:

```
elif (dice[0] == dice[2]) or (dice[1] == dice[3]) or ↵
(dice[2] == dice[4]):
    print("Три одинаковых")
```

Теперь, когда мы научились проверять выпадение различных выигрышных комбинаций игры в кубики, давайте добавим игровой цикл и массив `dice`.

Соберем все вместе

Ниже приведена завершенная программа *FiveDice.py*. Введите код в новое окно или скачайте его по адресу: **https://eksmo.ru/files/Python_deti.zip**.

FiveDice.py

```
import random
# Игровой цикл
keep_going = True
while keep_going:
    # "Выброс" пяти случайных костей
❶   dice = [0,0,0,0,0] # Создание массива для пяти значений
                       # dice[0]-dice[4]
❷   for i in range(5): # "Бросить" случайное число от 1-6
                       # для всех 5 костей
❸       dice[i] = random.randint(1,6)
    print("Вам выпало:", dice) # Напечатать значения пяти
                               # костей
    # Отсортировать их
❹   # Проверка комбинации пяти, четырех или трех одинаковых
    # костей
    # Яцзы — все пять костей одинаковые
    if dice[0] == dice[4]:
        print("Яцзы!")
    # ЧетыреОдинаковых — первые или последние четыре
    # одинаковые
    elif (dice[0] == dice[3]) or (dice[1] == dice[4]):
        print("Четыре одинаковых!")
    # ТриОдинаковых — первые три, три посередине или
    # последние три одинаковые
    elif (dice[0] == dice[2]) or (dice[1] == dice[3]) or ↵
         dice[2] == dice[4]):
        print("Три одинаковые")
    keep_going = (input("Нажмите [Enter] для продолжения, ↵
                        любую клавишу, чтобы выйти: ") == "")
```

Строки программного кода, следующие после импорта модуля `random` и начала игрового цикла, следует, пожалуй, пояснить. В строке ❶ мы создаем массив `dice`, в котором хранится пять значений, инициализируемых нулями. Для массивов используются те же самые квадратные скобки `[и]`, что мы использовали и для самых первых списков цветов и массивов номиналов и мастей карт чуть ранее в этой главе. В строке ❷ мы настраиваем цикл `for` на пятикратное повторение для пяти кубиков, используя для этого диапазон от 0 до 4. Это и будет позициями массива, или индексами, для пяти кубиков.

В строке ❸ мы настраиваем все кубики по отдельности, от `dice[0]` до `dice[4]`, приравнивая их к случайному целому числу от 1 до 6,

представляющему пять кубиков и номинал случайно выпавших граней. В строке ❹ мы показываем пользователю выпавшие кубики, распечатав для этого значения массива `dice`. Результат выполнения этого выражения `print` показан на рис. 6.5.

```
*Python Shell*
File  Edit  Shell  Debug  Options  Windows  Help
Вам выпало: [4, 4, 4, 2, 3]
Три одинаковые
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [2, 3, 6, 4, 6]
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [4, 5, 3, 5, 4]
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [3, 3, 6, 1, 3]
Три одинаковые
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [2, 3, 1, 6, 2]
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [3, 2, 1, 6, 1]
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [2, 2, 1, 3, 6]
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [5, 3, 5, 4, 1]
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [3, 5, 5, 3, 4]
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [2, 2, 3, 2, 4]
Три одинаковые
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [6, 3, 6, 2, 3]
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [3, 2, 4, 4, 4]
Три одинаковые
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [5, 2, 2, 3, 4]
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [4, 3, 3, 4, 5]
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
Вам выпало: [4, 6, 1, 3, 6]
Нажмите [Enter] для продолжения, любую клавишу, чтобы выйти:
                                                    Ln: 1  Col: 1
```

Рис. 6.5. *Тестовый запуск программы с игральными кубиками. Обратите внимание, что нам выпало несколько комбинаций с тремя и одна с четырьмя одинаковыми кубиками*

В строке ❺ мы вызываем функцию `.sort()` массива `dice`, что упрощает проверку выпадения различных комбинаций, таких как пять, четыре одинаковые грани и так далее, путем упорядочивания значений элементов массива от меньшего к большему. Так, при выпадении комбинации `[3, 6, 3, 5, 3]` функция `dice.sort()` переупорядочит массив следующим образом: `[3, 3, 3, 5, 6]`. Выражение `if` проверяет равенство первого значения пятому. В нашем случае, так как первое и пятое значения не равны (3 и 6), мы знаем, что не все пять кубиков легли одинаковыми гранями вверх, и перед нами нет комбинации из пяти одинаковых номиналов. Первое выражение `elif` проверяет выпадение комбинации из четырех одинаковых кубиков, сравнивая значения первого и четвертого элемента массива (3 и 5) и значения второго и пятого (3 и 6). В значениях

этих элементов совпадений нет, поэтому программа делает вывод, что комбинация из четырех кубиков не выпала. Второе выражение `elif` проверяет выпадение комбинации из трех одинаковых кубиков. Так как значения первого и третьего элементов массива равны (`3` и `3`), мы знаем, что первые три значения равны. Мы информируем пользователя о выпадении трех одинаковых кубиков и предлагаем ему нажать нужную клавишу в зависимости от того, хочет ли он продолжать играть или выйти, как показано на рис. 6.5.

Запустите программу и нажмите клавишу **Enter** несколько раз, чтобы посмотреть, какие кубики вам выпадут.

Вы можете заметить, что комбинация из трех одинаковых граней выпадает достаточно часто: каждые пять-шесть выбросов. Комбинация из четырех одинаковых граней встречается гораздо реже, примерно раз в пятьдесят выбросов. На рис. 6.5 вы можете видеть, что, несмотря на целый экран попыток, комбинация из четырех кубиков выпала нам лишь однажды. Комбинация яцзы встречается еще реже: прежде чем выпадет яцзы, возможно, вам придется бросить кубики несколько сотен раз. Однако благодаря особенностям генератора случайных чисел вы можете получить такую комбинацию уже за первые несколько попыток. Несмотря на то что наша упрощенная версия игры яцзы не столь захватывающая, как настоящая игра, в нашу игру все равно интересно играть благодаря ее случайности и непредсказуемости.

Добавив элемент случайности в игру с кубиками и картами, в игру «Камень, ножницы, бумага», а также в игру на угадывание, мы увидели, как случайность может сделать любую игру интересной и веселой. Мы также получили удовольствие от графики, напоминающей калейдоскоп, которую мы создавали, используя случайное число для отрисовки цветных спиралей по всему экрану. В следующем разделе мы соединим все, что вы узнали о случайных числах и циклах с геометрией, для того чтобы превратить программу по отрисовке случайных спиралей в настоящий виртуальный калейдоскоп, генерирующий при каждом запуске отличающийся набор отраженных изображений!

Математика чисел яцзы

Если вам интересна математика, стоящая за игрой яцзы, а также почему комбинация из пяти одинаковых кубиков настолько редка, прочитайте это краткое пояснение. Во-первых, в игре участвуют пять кубиков, у каждого из которых шесть граней. Таким образом, количество возможных комбинаций равняется $6 \times 6 \times 6 \times 6 \times 6 = 6^5 = 7776$. То есть существует 7776 способов выбросить пять нормальных шестигранных кубиков. Для того чтобы вычислить вероятность выпадения комбинации из пяти граней одинаковых номиналов, нам необходимо выяснить количество возможных вариантов комбинации яцзы: пять единиц, пять двоек и так далее до пяти шестерок. Итак, существует шесть возможных комбинаций яцзы, которые могут выпасть при игре с пятью кубиками. Разделим 6 на 7776 вариантов комбинаций — и получим вероятность выпадения комбинации из пяти одинаковых кубиков: 6/7776 или 1/1296.

Все правильно: вероятность того, что вам выпадет комбинация из пяти одинаковых кубиков, всего лишь 1 к 1296, поэтому не отчаивайтесь, если вам придется долгое время бросать кубики, прежде чем вы получите свою первую комбинацию из пяти одинаковых кубиков. В среднем, такая комбинация будет выпадать вам каждые 1300 бросков или около того. Неудивительно, почему за комбинацию яцзы пользователь получает сразу 50 очков!

Калейдоскоп

Изображение со случайными цветными спиралями на рис. 6.2 немного напоминает калейдоскоп. Чтобы сделать это изображение еще больше похожим на настоящий калейдоскоп, давайте добавим одну важную функцию, которой так не хватает в нашей программе по отрисовке спиралей: отражение.

В калейдоскопе положение зеркал превращает случайные цвета и фигуры в красивые узоры. В примере, завершающем эту главу, мы воссоздадим эффект зеркального отражения, несколько модифицировав созданную ранее программу *RandomSpiral.py* так, чтобы она могла четыре раза «отразить» спирали на экране.

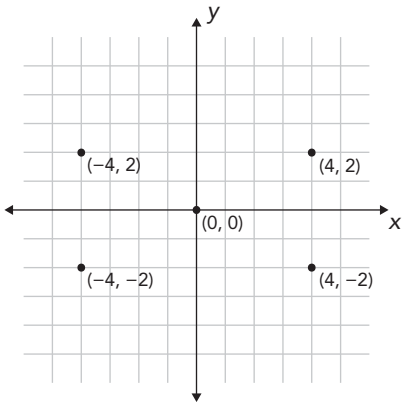


Рис. 6.6. *Отражения четырех точек по осям x и y с началом в позиции (4, 2)*

Чтобы понять, каким образом можно добиться этого эффекта отражения, нам необходимо поговорить о картезианских координатах. Давайте взглянем на четыре точки на рис. 6.6: (4, 2), (–4, 2), (–4, –2) и (4, –2).

Сравним две верхние точки: (4, 2) и (–4, 2). Если вертикальная ось *y* была бы зеркалом, то эти две точки были бы зеркальным отражением друг друга. Мы называем точку (4, 2) зеркальным отражением точки (–4, 2) *относительно* оси *y*. Нечто похожее происходит и с точками (4, 2) и (4, –2), только в качестве воображаемого зеркала в данном случае выступает ось *x*: точка (4, –2) есть отражение точки (4, 2) по оси *x*.

Если вы еще раз посмотрите на каждую пару координат (*x*, *y*) на рис. 6.6, то увидите, что для задания этих координат используются одинаковые числа, 4 и 2, отличающиеся только по знаку, + или –, в зависимости от местоположения точки. Мы можем создать отражение любой точки относительно осей *x* и *y*, изменяя знаки числовых значений координат следующим образом: (*x*, *y*), (–*x*, *y*), (–*x*, –*y*), (*x*, –*y*). Если хотите, можете начертить это на листе бумаги в клетку, используя любую пару координат (*x*, *y*). Например, попробуйте координаты (2, 3) — и получите четыре точки, отраженные по обеим сторонам оси *y*, а также сверху и снизу оси *x*.

Имея эти знания, мы можем написать план программы калейдоскопа следующим образом.

1. Выбрать случайную координату (*x*, *y*) в верхней правой части экрана и отрисовать спираль.
2. Отрисовать ту же самую спираль в точке (–*x*, *y*) в верхней левой части экрана.

3. Отрисовать ту же самую спираль в точке $(-x, -y)$ в нижней левой части экрана.
4. Отрисовать ту же самую спираль в точке $(x, -y)$ в нижней правой части экрана.

Если мы повторим эти шаги несколько раз, то на экране мы получим красивый калейдоскоп из случайных спиралей.

Давайте шаг за шагом пройдемся по законченному коду программы *Kaleidoscope.py*, чтобы понять, как он работает.

Kaleidoscope.py

```
import random
import turtle
t = turtle.Pen()
❶ t.speed(0)
turtle.bgcolor("black")
colors = ["red", "yellow", "blue", "green", "orange", ↵
          "purple", "white", "gray"]
for n in range(50):
    # Генерация спиралей случайных цветов/размеров
    # в случайных местах
    t.pencolor(random.choice(colors)) # Выбрать случайный
    # цвет из colors[]
    size = random.randint(10,40) # Задать случайный размер
    # от 10 до 40
    # Сгенерировать случайную (x, y) координату на экране
❷   x = random.randrange(0, turtle.window_width()//2)
❸   y = random.randrange(0, turtle.window_height()//2)
    # Первая спираль
    t.penup()
❹   t.setpos(x, y)
    t.pendown()
    for m in range(size):
         t.forward(m*2)
         t.left(91)
    # Вторая спираль
    t.penup()
❺   t.setpos(-x, y)
    t.pendown()
    for m in range(size):
    t.forward(m*2)
    t.left(91)
```

```
    # Третья спираль
    t.penup()
❻   t.setpos(-x, -y)
    t.pendown()
    for m in range(size):
    t.forward(m*2)
    t.left(91)
    # Четвертая спираль
    t.penup()
❼   t.setpos(x, -y)
    t.pendown()
    for m in range(size):
        t.forward(m*2)
        t.left(91)
```

Наша программа начинается как обычно — с импорта модулей `turtle` и `random`, но в строке ❶ мы делаем что-то новое: с помощью функции `t.speed(0)` мы увеличиваем до предела скорость черепашки. Функция `speed()` графики Turtle принимает аргумент-число от 0 до 10, при этом число 1 задает медленный темп анимации, `10` — быстрый темп. Аргумент `0` отключает анимацию (отрисовка выполняется с максимальной скоростью, которую может развить компьютер). Это странная шкала от 1 до 10, так как 0 следует после 10. Просто запомните, что если вам нужно максимально ускорить черепашку, установите скорость на `0`. Запустив программу, вы заметите, что спирали появляются практически мгновенно. Если хотите, чтобы черепашка двигалась быстрее, вы можете внести это изменение в любую из наших предыдущих программ.

Цикл соответствует циклу из программы *RandomSpirals.py* до того, как мы доходим до строк ❷ и ❸. В строке ❷ мы в два раза уменьшаем горизонтальный диапазон случайного числа для задания только положительных координат *x* (правая часть экрана от `x = 0` до `x = turtle.window_width()//2`), а в строке ❸ мы ограничиваем вертикальный диапазон до размеров верхней половины экрана: от `y = 0` до `y = turtle.window_height()//2`. Не забудьте, что целочисленное деление осуществляется с помощью оператора `//`, это необходимо, чтобы все пиксельные измерения оставались целыми числами.

Эти две строки кода каждый раз дают нам пару случайных координат (*x*,*y*) в верхней правой части экрана. В строке ❹ мы устанавливаем ручку черепашки в эту точку и сразу же отрисовываем первую спираль с помощью цикла `for`. Затем мы изменяем знаки каждой из координат так же, как мы

делали на рис. 6.6, это позволит создать три отражения точки в верхней левой (–*x*, *y*), строка ❺, нижней левой (–*x*, –*y*), строка ❻, и нижней правой (*x*, –*y*), строка (7), частях экрана. На рис. 6.7 изображен пример узора, создаваемого программой *Kaleidoscope.py*.

Вы можете найти отражения каждой спирали, просто взглянув на три другие части экрана. Это не истинные отражения: мы не начинаем с одинакового угла для каждой спирали, и мы не поворачиваем вправо вместо поворота влево при отрисовке отраженных спиралей. Однако при желании вы можете внести эти изменения в программу. В разделе «Задачи по программированию» в конце главы предложено несколько идей для улучшения программы калейдоскопа.

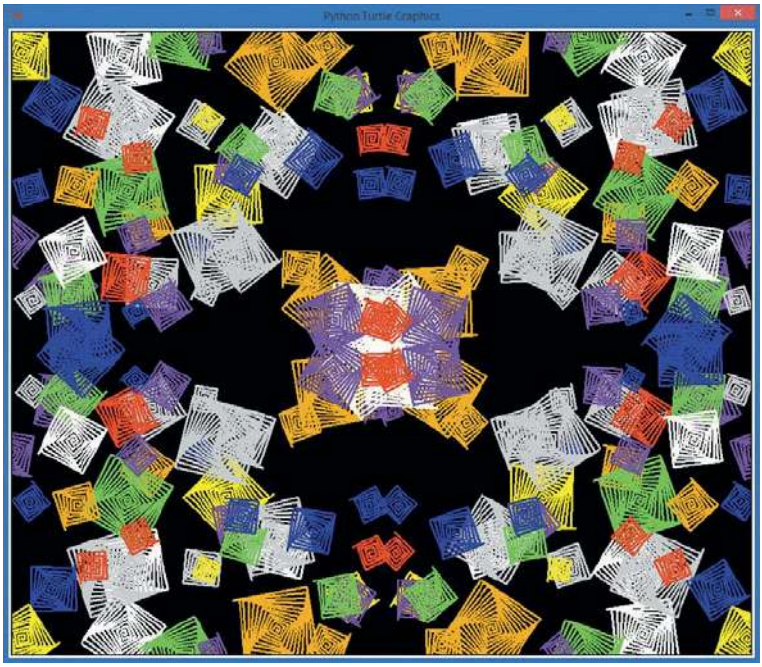


Рис. 6.7. *Эффект отражения/повторения программы Kaleidoscope.py*

Что вы узнали

До этой главы мы не знали, как заставить компьютер выполнять случайные действия. Теперь мы можем заставить компьютер сбросить игральные кости, выдать карты из колоды, нарисовать спирали случайного цвета, формы, размера в случайных частях экрана. Кроме того, компьютер теперь может нас иногда обыграть в «Камень, ножницы, бумагу».

Инструмент, который сделал возможным создание всех этих программ, — модуль `random`. В игре на угадывание мы использовали команду `random.randint(1, 10)` для генерации случайного числа в диапазоне от 1 до 10. В программах, рисующих случайные спирали, мы добавили функцию `random.choice()` для выбора случайного цвета из списка. Вы узнали, как использовать функции `turtle.window_width()` и `turtle.window_height()` для выяснения ширины и высоты экрана Turtle.

Вы также узнали, как пользоваться картезианскими координатами для поиска местоположения (x, y) на экране, вы пользовались функцией `random.randrange()` для генерации случайного числа в диапазоне между значениями правой и левой координаты x и верхней и нижней координаты y. Затем мы использовали функцию `turtle.setpos(x, y)` для перемещения черепашки в любую точку на графическом экране.

Мы объединили возможность выбора случайного элемента из списка с помощью функции `random.choice()` с умением проверять и сравнивать значения с помощью выражений `if-elif` для создания версии игры «Камень, ножницы, бумага» «пользователь против компьютера».

Вы изучили концепцию массива, и мы упростили код карточной игры благодаря созданию одного массива для хранения названия мастей и еще одного — для хранения номиналов карт. Для симуляции процесса сдачи карт мы применили функцию `random.choice()` к обоим массивам. Мы упорядочили номиналы карт от меньшего к большему, а затем воспользовались функцией.`index()` для поиска позиции нужного элемента в массиве. Мы использовали индекс номинала каждой из двух карт для выяснения, какая карта больше и, соответственно, кто из пользователей выиграл партию в игре «Пьяница». Мы создали повторно используемый игровой цикл с пользовательским вводом, переменной-флагом `keep_going` и выражением `while`. Мы можем использовать этот цикл в любом приложении или игре, которые пользователь пожелает повторно запустить несколько раз подряд.

Мы углубили понимание массивов, создав упрощенную версию игры яцзы. Для симуляции пяти кубиков мы создали массив с пятью значениями от 1 до 6, воспользовались функцией `randint()` для симуляции выброса кубиков, а также применили к массиву кубиков метод `sort()` для упрощения проверки выпадения выигрышных комбинаций. Мы увидели, что, если в сортированном массиве первый и последний элемент равны, значит, и все элементы этого массива равны. В нашей игре это означало выпадение

комбинации из пяти одинаковых кубиков. Для проверки выпадения комбинаций из четырех и трех одинаковых кубиков мы использовали сложный оператор `if`, соответственно две и три части которого объединялись оператором `or`. Мы использовали выражения `if-elif` для управления логикой нашей программы, чтобы комбинация из пяти одинаковых кубиков не зачлась так же, как комбинация из четырех и трех.

В программе калейдоскопа мы еще немного поработали с картезианскими координатами и симулировали эффект отражения, изменяя знаки значений координат (x, y). Мы четырежды повторяли каждую спираль случайного размера, цвета с центром в случайной точке экрана для создания эффекта калейдоскопа. Вы узнали о том, как увеличить скорость отрисовки черепашки с помощью команды `t.speed(0)`.

Случайные числа и выбор случайного элемента добавляют в игру неожиданности, что делает ее интереснее. Почти в каждой игре есть этот элемент случайности. Теперь, когда вы можете создавать случайность в программе, вы способны программировать более увлекательные игры. На данном этапе вы должны уметь делать следующее.

- Импортировать в свои программы модуль `random`.
- Использовать функцию `random.randint()` для генерации случайного числа в заданном диапазоне.
- Использовать функцию `random.choice()` для случайного выбора элемента списка или массива.
- Использовать функцию `random.choice()` для генерации 52 карт из двух массивов строк, хранящих только номиналы и масти.
- Определять размер окна для рисования с помощью функций `turtle.window_width()` и `turtle.window_height()`.
- Перемещать черепашку в любую точку экрана для рисования с помощью функции `turle.setpos(x, y)`.
- Использовать функцию `random.randrange()` для генерации случайного числа в любом диапазоне.
- Находить индекс элемента списка или массива с помощью функции. `index()`.
- Создавать игровой цикл `while` с использованием логической булевой переменной-флага, такой как `keep_going`.
- Создать массив значений одинакового типа, присвоить элементам массива значения с помощью индексов этих элементов (например,

`dice[0] = 2`) и использовать элементы массива как обычные переменные.

- Сортировать списки или массивы с помощью функции `.sort()`.
- Отражать точки относительно осей *x* и *y*, изменяя знаки числовых значений координат (*x*, *y*) точек.
- Изменять скорость отрисовки черепашки с помощью функции `.speed()`.

Задачи по программированию

В задачах по программированию в этой главе мы расширим программы *Kaleidoscope.py* и *HighCard.py*. (Если вы зашли в тупик, перейдите на сайт **https://eksmo.ru/files/Python_deti.zip** для загрузки готовых решений.)

#1. Случайное количество сторон и толщина

Добавим еще больше случайности в программу *Kaleidoscope.py*, включив две новые случайные переменные. Добавьте переменную `sides` для задания количества сторон, а затем воспользуйтесь этой переменной для изменения угла поворота при каждой итерации цикла отрисовки спирали (и, соответственно, количества сторон спирали). Для задания угла поворота вместо числа `91` воспользуйтесь выражением `360/sides + 1`. Затем создайте переменную с именем `thick`, которая будет хранить случайное число в диапазоне от 1 до 6, задающее толщину ручки черепашки. Добавьте строку `t.width(thick)` в нужный участок программного кода, чтобы изменить толщину линий каждой спирали случайного калейдоскопа.

#2. Реалистичные отраженные спирали

Если вы немного знакомы с геометрией, два новых улучшения сделают наш калейдоскоп еще более реалистичным. Во-первых, отслеживайте направление (между 0 и 360 градусами), в котором указывает черепашка перед отрисовкой первой спирали. Для этого воспользуйтесь функцией `t.heading()` и сохраните результат в переменной `angle`. Затем перед отрисовкой каждой отраженной спирали

с помощью функции `t.setheading()` изменяйте угол так, чтобы направление, на которое указывает черепашка, соответствовало корректно отраженному углу. Подсказка: второй угол должен быть задан выражением `180 – angle`, угол третьей спирали будет представлен выражением `angle – 180`, а четвертый — `360 – angle`.

Затем попробуйте поворачивать влево после отрисовки каждой линии первой и третьей спиралей и вправо — после отрисовки каждой линии второй и четвертой спиралей. Если вы примените эти изменения, то спирали, рисуемые вашей программой, будут выглядеть как зеркальные отражения друг друга в размерах, формах, цветах, толщине и направленности. Если хотите, вы также можете предотвратить столь сильное наложение фигур, изменив диапазоны координат *x* и *y* до `random.randrange(size, turtle.window_width()//2)` и `random.randrange(size, turtle.window_height()//2)`.

#3. «Пьяница»

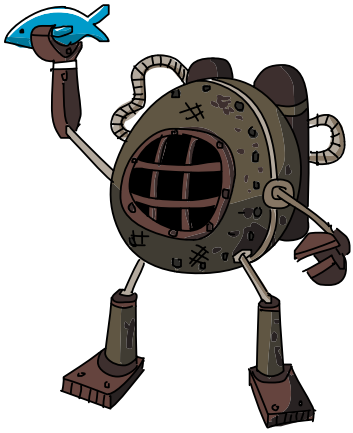
Превратите программу *HighCard.py* в полноценную версию игры «Пьяница». Во-первых, ведите учет очкам: создайте две переменные, которые сохраняли бы количество партий, выигранных компьютером и выигранных пользователем. Во-вторых, симулируйте игру одной полной колодой карт, сыграв только 26 партий. (Возможно, для этого потребуется использовать цикл `for`, а не `while`, так как это позволит вам вести учет количества сыгранных партий.) Затем объявите победителя, у которого оказалось больше очков. В-третьих, обрабатывайте ничьи, запоминая, сколько патовых ситуаций возникло подряд. Затем, когда победит один из пользователей, добавьте количество патов к текущему количеству очков победителя и вновь обнулите количество патов перед началом следующего раунда.

Глава 7

ФУНКЦИИ: ДА, У ЭТОГО ЕСТЬ НАЗВАНИЕ

Мы уже успели воспользоваться большим количеством *функций* — от `print()` до `input()`, до `turtle.forward()`. Но все эти функции были либо встроенными, либо импортированными из модулей и библиотек языка Python. В этой главе мы будем создавать наши *собственные* функции. Они будут делать все, что мы захотим, включая реагирование на действия пользователей, такие как щелчок мышью или нажатие клавиш.

Функции очень полезны, они дают нам возможность упорядочивать блоки повторно используемого кода, а затем ссылаться на эти блоки в дальнейшем в наших программах с помощью одного короткого имени или короткой команды. Рассмотрим в качестве примера функцию `input()`. Данная функция выводит на экран строку текста, приглашающую пользователя ввести данные, собирает то, что ввел пользователь, передает ввод в программу в качестве строки, которую можно сохранить в переменной. Мы повторно используем функцию `input()` всякий раз, когда необходимо узнать у пользователя какую-то дополнительную информацию. Если бы у нас не было этой функции, то нам, возможно, пришлось бы делать всю работу самостоятельно каждый раз, желая запросить у пользователя информацию.



Функция `turtle.forward()` — еще один замечательный визуальный пример: каждый раз, когда перемещаем черепашку вперед, чтобы нарисовать одну из сторон наших спиралей, Python выполняет попиксельную отрисовку линии строго нужной длины и в направлении, куда указывает черепашка. Не будь у нас функции `turtle.forward()`, нам пришлось бы самостоятельно искать способ окрашивания пикселей на экране, вести учет местоположений и углов, а также выполнять достаточно сложные вычисления каждый раз при отрисовке отрезка заданной длины.

Без этих функций наши программы были бы значительно длиннее, неудобнее при чтении и сложнее в написании. Функции дают нам возможность воспользоваться работой, которую уже проделали большое количество наших соратников-программистов. Хорошие новости также в том, что мы можем писать наши собственные функции: они сделают наш программный код более коротким, удобочитаемым и пригодным для повторного использования.

В главе 6 мы создали программы, которые отрисовывали на экране случайные спирали и узор наподобие калейдоскопа. Мы можем использовать функции для упрощения кода этих программ, а также сделать участки программного кода более пригодными для повторного использования.

Соберем все вместе с функциями

Взгляните еще раз на пример *RandomSpirals.py* на с. 136. Все, что находится в первом цикле `for`, отвечает за создание всего одной случайной спирали. Сам цикл `for` использует этот код для рисования 50 спиралей случайного цвета, размера и в случайных участках экрана.

Представим, что нам нужно воспользоваться кодом для рисования случайной спирали в другой программе, например в игре или приложении-заставке. В программе *RandomSpirals.py* очень сложно сказать, где начинается и заканчивается отрисовка спирали, а мы написали этот код всего несколько страниц назад. Представьте, если вам потребуется вернуться к коду этой программы через несколько месяцев! В этом случае мы потратим много сил и времени, чтобы понять, какие действия выполняет программа и какие именно строки кода нужно скопировать в новую программу, если опять потребуется нарисовать несколько случайных спиралей.

Чтобы сделать участок кода пригодным для использования в дальнейшем или просто более удобочитаемым, мы можем *определить функцию*

и присвоить ей легкое для восприятия имя, такое, как, например, имя функции `input()` или `turtle.forward()`. Определение функции также называется *объявлением* функции. Это значит, что мы просто сообщаем компьютеру, что должна делать данная функция. Давайте создадим функцию, которая будет рисовать случайную спираль на экране, и назовем ее `random_spiral()`. Мы сможем повторно использовать эту функцию в любое время и в любой программе, если нам потребуется создать случайные спирали.

Объявление `random_spiral()`

Откройте программу *RandomSpirals.py* (глава 6), сохраните ее в новом файле с именем *RandomSpiralsFunction.py* и начните определять функцию *после* настройки ручки черепашки, ее скорости и цвета, но *перед* циклом `for`. (Для справки вы можете использовать законченную версию программы на с. 168, чтобы представлять, как должен выглядеть код программы.) Определение функции `random_spiral()` следует поместить после настройки черепашки, так как функции потребуется воспользоваться ручкой черепашки `t` и списком цветов. Определение функции следует поместить перед циклом `for`, так как мы будем использовать функцию `random_spiral()` в цикле `for`, а перед использованием функцию необходимо определить. Теперь, когда мы нашли подходящий участок кода, куда нужно поместить функцию `random_spiral()`, давайте ее определим.

Для определения функций в языке Python используется ключевое слово `def` (сокращение от английского слова *definition* — определение), после которого указывается имя новой функции, скобки `()` и двоеточие (`:`). Далее приведена первая строка создаваемой функции `random_spiral()`:

```
def random_spiral():
```

Оставшаяся часть определения функции будет представлена одним или несколькими выражениями, сдвинутыми вправо, аналогично тому, как мы поступали при группировке выражений для циклов `for`. Для создания случайной спирали необходимо установить случайный цвет, случайный размер и случайную точку (x, y) на экране, а затем переместить ручку в эту точку — и нарисовать спираль. Далее приведен код законченной версии функции `random_spiral()`:

```
def random_spiral():
    t.pencolor(random.choice(colors))
    size = random.randint(10,40)
```

```
    x = random.randrange(-turtle.window_width()//2,
                         turtle.window_width()//2)
    y = random.randrange(-turtle.window_height()//2,
                         turtle.window_height()//2)
    t.penup()
    t.setpos(x, y)
    t.pendown()
    for m in range(size):
          t.forward(m*2)
          t.left(91)
```

Обратите внимание, что при объявлении функции компьютер на самом деле не запускает выполнение определяемой функции. Если мы введем определение функции в оболочку IDLE, компьютер не выдаст спираль (пока что). Чтобы отрисовать спираль, нам необходимо вызвать функции `random_spiral()`.

Вызов `random_spiral()`

Определение функции сообщает компьютеру, что необходимо сделать, когда кто-либо вызывает эту функцию. После определения функции мы *вызываем* эту функцию в программе, указывая имя функции и скобки после него:

```
random_spiral()
```

Не забывайте про скобки, так как именно они говорят компьютеру, что вы хотите запустить выполнение функции. Теперь, когда мы определили команду `random_spiral()` как функцию, при вызове функции `random_spiral()` в программе компьютер нарисует на экране Turtle случайную спираль.

Теперь, чтобы нарисовать 50 случайных спиралей, вместо использования всего кода программы *RandomSpirals.py* мы можем сократить код цикла `for` до следующего состояния:

```
for n in range(50):
    random_spiral()
```

Благодаря использованию созданной нами функции этот код гораздо более удобочитаем. Мы также можем перенести код для отрисовки случайной спирали в другую программу, просто скопировав и вставив определение функции.

Далее приведена законченная версия программы. Введите этот код в IDLE и сохраните его в файл с именем *RandomSpiralsFunction.py* или скачайте файл по адресу **https://eksmo.ru/files/Python_deti.zip**.

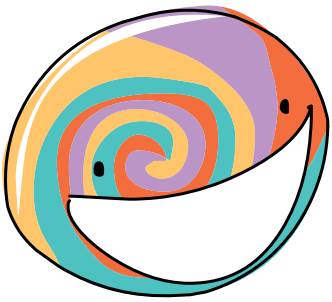
RandomSpiralsFunction.py

```
import random
import turtle
t = turtle.Pen()
t.speed(0)
turtle.bgcolor("black")
colors = ["red", "yellow", "blue", "green", "orange", ↵
          "purple", "white", "gray"]
def random_spiral():
    t.pencolor(random.choice(colors))
    size = random.randint(10,40)
    x = random.randrange(-turtle.window_width()//2,
                         turtle.window_width()//2)
    y = random.randrange(-turtle.window_height()//2,
                         turtle.window_height()//2)
    t.penup()
    t.setpos(x, y)
    t.pendown()
    for m in range(size):
         t.forward(m*2)
         t.left(91)

for n in range(50):
    random_spiral()
```

В дополнение к удобству чтения программы мы также получаем функцию `random_spiral()`, пригодную для повторного использования, которую мы можем копировать, изменять и с легкостью использовать в других программах.

Если вы вдруг обнаружите, что регулярно используете один и тот же кусок кода, преобразуйте его в функцию, как сделали мы на примере функции `random_spiral()`, с помощью ключевого слова `def` — и вы обнаружите, что такой код гораздо проще *портировать*, то есть переносить для использования в новых приложениях.



Вы даже можете создать собственный модуль с несколькими функциями и импортировать его так же, как мы импортировали в наши программы модули turtle и random (см. Приложение B для получения более подробной информации о том, как создавать модули на языке Python). Таким образом, вы сможете поделиться созданным вами кодом с друзьями.

Параметры: покормите свою функцию

При создании функции мы можем определить *параметры* этой функции. Параметры позволяют нам отправлять информацию в функцию путем передачи значений в виде *аргументов* внутри скобок. Мы передавали аргументы функциям с первого использования выражения `print()`. Когда мы пишем код `print("Привет")`, слово `"Привет"` является аргументом, представляющим строковое значение, которое мы хотим вывести на экран. При вызове функции черепашки `tutrle.left(90)` мы передаем значение `90` как количество градусов, на которое мы хотим развернуть черепашку влево.

Функция `random_spiral()` не требовала параметров. Вся необходимая информация находилась в коде внутри функции. Но, если нужно, создаваемые функции могут принимать информацию в форме параметров. Давайте определим функцию `draw_smiley()`, которая нарисует небольшой смайлик в случайном участке экрана. Эта функция будет принимать пару случайных координат и рисовать смайлик в этих координатах. Мы определим функцию и присвоим ей имя `random_spiral()` в программе *RandomSmileys.py*, завершенный код которой приведен на с. 177 Давайте изучим процесс создания этой программы шаг за шагом.

Смайлики в случайных участках экрана

Мы хотим написать программу, которая вместо случайных спиралей рисовала бы смайлики. Чтобы нарисовать смайлик, потребуется потратить чуть больше времени на планирование, чем на планирование алгоритма выбора случайного цвета и размера и рисования спирали. Давайте вернемся к нашему другу из главы 6: листку бумаги в клетку. Так как мы никогда раньше в наших программах не рисовали ничего равного по сложности смайлику, то желательно сначала нарисовать его на листке бумаги, а затем поэтапно перевести рисунок в код. На рис. 7.1 изображен смайлик, нарисованный на листке бумаги в клетку, который мы можем использовать для планирования отрисовки.

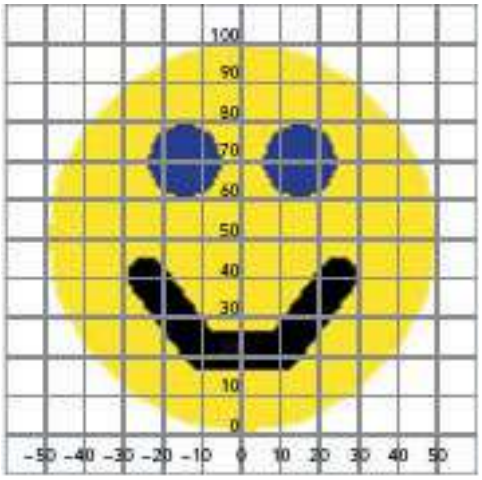


Рис. 7.1. *Предполагается, что программа первым делом будет рисовать смайлик на бумаге в клетку*

Наша программа будет рисовать смайлики, подобные этому, по всему экрану в случайных координатах (*x*, *y*). Определяемая функция `draw_smiley()` будет принимать два параметра, x и y, задающие точку, в которой будет рисоваться смайлик. Как показано на рис. 7.1, мы нарисуем смайлик так, словно он находится в точке (*x*, *y*), поэтому представьте, что мы перемещаем этот шаблон смайлика по экрану, помещая его центр координат в любую точку (*x*, *y*) на экране. Давайте поразмыслим над тем, как отрисовать каждый смайлик, начиная с заданной точки.

Рисование головы

У всех смайликов голова представлена желтым кругом, глаза — двумя черными кругами, а рот — несколькими черными линиями. После получения координат точки на экране наша функция `draw_smiley()` должна будет нарисовать правильно расположенные относительно заданной точки голову, глаза и рот. Чтобы понять, какой код будет составлять определение нашей функции `draw_smiley()`, необходимо по отдельности распланировать отрисовку головы, глаз и рта. Начнем с головы. Сначала мы отрисуем голову, чтобы она не перекрывала глаза и рот, которые нарисуем следом.

Каждое деление на рис. 7.1 мы будем воспринимать как 10 пикселей, таким образом, высота нарисованного нами смайлика будет составлять 100 пикселей, что будет равняться примерно дюйму при отображении

на большинстве мониторов. Так как *диаметр*, или высота и ширина, круга составляет 100 пикселей, это значит, что *радиус* (половина диаметра) круга равняется 50 пикселям. Нам необходим именно радиус, так как команда `circle()` модуля `turtle` принимает в качестве параметра радиус окружности. Команда, отрисовывающая окружность с радиусом 50 (то есть, диаметром 100), имеет вид `t.circle(50)`. Функция `circle()` отрисовывает окружность прямо над точкой текущего положения черепашки (x, y). Нам потребуется эта информация, чтобы корректно разместить глаза и рот, поэтому я нарисовал смайлик так, чтобы его нижняя точка совпадала с точкой начала координат (0, 0). Чтобы понять, где необходимо рисовать остальные элементы, нам нужно прибавить координаты каждого из элементов к начальной позиции в точке начала координат (0, 0).

Для рисования большой желтой головы мы зададим желтый цвет ручки, а также установим цветом заливки желтый. Включим заливку фигуры, нарисуем окружность (которая будет залита желтым цветом, так как мы включили заливку фигур), по окончании отрисовки отключим заливку. Предположив, что ручке черепашки, ранее определенной в программе, было присвоено имя `t`, код для отрисовки желтого круга, представляющего голову смайлика, в текущей точке (x, y) будет выглядеть следующим образом:

```
# Голова
t.pencolor("yellow")
t.fillcolor("yellow")
t.begin_fill()
t.circle(50)
t.end_fill()
```

Чтобы залить окружность желтым цветом, мы окружили команду `t.circle(50)` четырьмя строками кода. Во-первых, мы устанавливаем цвет ручки в желтый с помощью команды `t.pencolor("yellow")`, во-вторых, мы устанавливаем желтый цвет заливки с помощью команды `t.fillcolor("yellow")`. В-третьих, перед вызовом функции `t.circle(50)` мы сообщаем компьютеру, что хотели бы залить рисуемую окружность желтым цветом. Мы передаем компьютеру эту информацию с помощью функции `t.begin_fill()`. Наконец, мы рисуем окружность и сообщаем компьютеру, что мы закончили рисовать фигуру, которую теперь нужно залить цветом.

Рисование глаз

В первую очередь необходимо понять, где следует разместить черепашку для корректной отрисовки левого глаза, затем установить голубой цвет заливки и, наконец, отрисовать окружность правильного размера. Высота глаз примерно 20 пикселей (два деления), мы также знаем, что диаметр 20 означает, что нам нужен радиус вполовину меньше этого значения, то есть 10, следовательно, для отрисовки каждого глаза мы будем использовать команду `t.circle(10)`. Сложность заключается в определении места рисования глаз.

Нашей начальной точкой (x, y) будет локальная точка начала координат каждого смайлика, вы можете посмотреть местоположение левого глаза на рис. 7.1. Судя по рисунку, отрисовка глаза начинается в точке на 6 делений выше точки начала координат (60 пикселей вверх в положительном направлении оси y), при этом точка начала отрисовки сдвинута на 1,5 деления влево от оси y (то есть на приблизительно 15 пикселей влево, в отрицательном направлении оси x).

Дабы сообщить программе, как попасть в нужную точку для рисования левого глаза, начиная с переданной в качестве пары аргументов точки (x, y) в нижней части большого желтого круга, мы должны начать движение в точке x и переместиться влево на 15 пикселей, начать движение в точке y и переместиться вверх на 60 пикселей, иными словами, мы должны переместиться в точку (`x-15, y+60`). Таким образом, вызов функции `t.setpos(x-15, y+60)` должен переместить черепашку в точку начала отрисовки левого глаза. Далее приведен код для отрисовки левого глаза:

```
# Левый глаз
t.setpos(x-15, y+60)
t.fillcolor("blue")
t.begin_fill()
t.circle(10)
t.end_fill
```

Очень легко допустить ошибку, введя команду `setpos` и указав в качестве аргументов просто координаты (–15, 60), однако стоит помнить, что нам необходимо будет нарисовать большое количество смайликов в различных точках (x, y) на экране, при этом далеко не все смайлики будут отрисовываться в точке (0, 0). Команда `t.setpos(x-15, y+60)` позволит

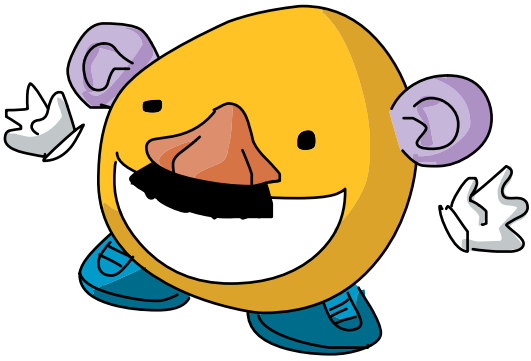
нам быть уверенными в том, что, где бы мы ни рисовали желтое лицо, левый глаз всегда будет находиться в его левой верхней части.

Код для отрисовки правого глаза почти идентичен коду для отрисовки левого. Мы можем видеть, что правый глаз находится в 15 пикселах (1,5 деления) от точки (*x*, *y*) и по-прежнему в 60 пикселах кверху. Команда `t.setpos(x-15, y+60)` расположит правый глаз симметрично левому. Далее приведен код для отрисовки правого глаза:

```
# Правый глаз
t.setpos(x+15, y+60)
t.begin_fill()
t.circle(10)
t.end_fill
```

Так как мы не изменяли цвет заливки после отрисовки левого глаза, он все еще остался голубым, поэтому нам достаточно лишь перенести черепашку в нужную точку `(x+15, y+60)`, включить заливку, отрисовать глаз и закончить заливку.

Рисование рта



Теперь давайте распланируем самую важную часть смайлика: улыбку. Чтобы упростить улыбку, мы нарисуем рот из трех толстых черных линий. Судя по рисунку, левая часть улыбки начинается в точке примерно 2,5 деления влево и 4 деления вверх относительно точки (*x*, *y*), поэтому для начала отрисовки улыбки мы поместим черепашку в точке `(x-25, y+40)`. Для того чтобы улыбку было лучше видно, мы изменим цвет ручки на черный и увеличим толщину до 10. С левого верхнего уголка улыбки мы должны перейти к точке `(x-10, y+20)`, затем к `(x+10, y+20)` — и, наконец, прийти к верхнему правому уголку улыбки в точке `(x+25, y+40)`. Обратите внимание, что эти точки являются зеркальными отражениями друг друга относительно оси *y*, это позволяет сделать лицо смайлика симметричным.

Далее приведен код для отрисовки рта:

```
  # Рот
  t.setpos(x-25, y+40)
  t.pencolor("black")
  t.width(10)
  t.goto(x-10, y+20)
  t.goto(x+10, y+20)
  t.goto(x+25, y+40)
❶ t.width(1)
```

Установив черепашку в левый верхний уголок рта, мы изменяем цвет ручки на черный и устанавливаем толщину 10 пикселей. Мы начинаем отрисовку, инструктируя черепашку пройти по всем трем точкам улыбки. Функция `goto()` модуля `turtle` делает то же самое, что и функция `setpos()`, а именно, перемещает черепашку в нужную точку. Я использовал ее в этой программе, показывая, что для функции `setpos()` существует альтернатива. Наконец, в строке ❶ команда `t.width(1)` обратно устанавливает значение толщины ручки в 1 пиксель. Это сделано для того, чтобы следующие отрисовываемые нами фигуры (смайлики) не были слишком толстыми.

Определение и вызов функции `draw_smiley()`

Теперь нам остается лишь определить функцию `draw_smiley()`, которая будет включать в себя весь код, рисующий смайлик, настроить цикл для генерации 50 случайных точек (x, y) на экране и вызвать функцию `draw_smiley(x, y)` для отрисовки смайликов во всех 50 точках.

Определение функции `draw_smiley()` должно будет принять два параметра, `x` и `y`, определяющие место смайлика на экране. Кроме того, функция должна будет поднять ручку черепашки над экраном, перенести черепашку в заданную точку (x, y), а затем вернуть ручку на экран и приготовиться к рисованию. После этогонужно лишь добавить наши заготовки кода для отрисовки большого желтого лица, левого и правого глаз и рта.

```
def draw_smiley(x, y):
    t.penup()
    t.setpos(x, y)
    t.pendown()
    # Весь код отрисовки будет помещен сюда…
```

Последним блоком кода будет цикл for, генерирующий 50 случайных точек для отрисовки смайликов и вызывающий функцию `draw_smiley()` для отрисовки каждого смайлика. Цикл будет выглядеть следующим образом:

```
for n in range(50):
    x = random.randrange(-turtle.window_width()//2,
                         turtle.window_width()//2)
    y = random.randrange(-turtle.window_height()//2,
                         turtle.window_height()//2)
    draw_smiley(x, y)
```

Значения координат *x* и *y* аналогичны тем, что мы видели в главе 6, когда генерировали случайные точки в левой и нижней частях экрана, а затем отражали их в правой и верхней частях. Выполнив команду `draw_smiley(x, y)`, мы передаем функции `draw_smiley(x, y)` эти случайные координаты в качестве аргументов, которые позволят функции отрисовать смайлик в заданной случайной точке.

Соберем все вместе

Собранная воедино программа должна выглядеть примерно следующим образом.

RandomSmileys.py

```
import random

import turtle

t = turtle.Pen()

t.speed(0)

t.hideturtle()

turtle.bgcolor("black")
❶ def draw_smiley(x, y):
    t.penup()
    t.setpos(x, y)
    t.pendown()
    # Голова
    t.pencolor("yellow")
    t.fillcolor("yellow")
    t.begin_fill()
    t.circle(50)
```

```
    t.end_fill()
    # Левый глаз
    t.setpos(x-15, y+60)
    t.fillcolor("blue")
    t.begin_fill()
    t.circle(10)
    t.end_fill()
    # Правый глаз
    t.setpos(x+15, y+60)
    t.begin_fill()
    t.circle(10)
    t.end_fill()
    # Рот
    t.setpos(x-25, y+40)
    t.pencolor("black")
    t.width(10)
    t.goto(x-10, y+20)
    t.goto(x+10, y+20)
    t.goto(x+25, y+40)
    t.width(1)
❷ for n in range(50):
    x = random.randrange(-turtle.window_width()//2,
                         turtle.window_width()//2)
    y = random.randrange(-turtle.window_height()//2,
                         turtle.window_height()//2)
    draw_smiley(x, y)
```

Как обычно, мы импортируем модули, которые потребуются нам для настройки черепашки, а также устанавливаем скорость черепашки на `0` (самая быстрая). Мы использовали команду `hideturtle()` для того, чтобы сама черепашка не отображалась на экране: это также позволяет увеличить скорость отрисовки.

В строке ❶ мы определяем функцию `draw_smiley()` таким образом, чтобы она действительно выполняла работу по отрисовке лица смайлика, левого и правого глаза, а также улыбки с помощью заранее заготовленного нами кода. Все, что нужно функции для выполнения этой задачи, — координата *x* и координата *y*.

В цикле `for` в строке ❷ выбираются случайные координаты `x` и `y`, а затем передаются в функцию `draw_smiley()`, которая отрисовывает смайлик со всеми чертами лица, корректно расположенными относительно этой случайной точки.

Программа *RandomSmileys.py* нарисует в случайных участках экрана 50 смайликов, как показано на рис. 7.2.

Вы можете изменить программу так, чтобы она рисовала любую нужную вам фигуру: все, что вам для этого понадобится, — создать функцию, которая отрисовывала бы любую фигуру, начиная с любой точки (x, y). Начните работу с листочка в клетку, как мы сделали при объяснении этого примера, такой подход упростит поиск важных точек. Если вам не нравится, что некоторые смайлики наполовину выходят за пределы экрана слева или справа или что некоторые из них находятся почти вне видимой области экрана сверху, вы можете воспользоваться небольшим количеством математики при задании значений `x` и `y` в выражениях с вызовом функции `randrange()` так, чтобы все смайлики оставались в пределах видимой области экрана. Перейдите на сайт **https://eksmo.ru/files/Python_deti.zip** для получения примера решения этой задачи.

Рис. 7.2. *Программа RandomSmileys.py производит счастливый результат*

`Return`: важно не то, что ты получаешь, важно то, что ты возвращаешь

С помощью аргументов мы можем передавать информацию в функцию, но что если нам нужно *получить* информацию *из* функции? Например, предположим, что мы создали функцию, которая конвертирует дюймы

в сантиметры, и нужно сохранить конвертированное значение для использования в последующих вычислениях, а не просто отобразить его напрямую на экране. Чтобы передать информацию из функции обратно в программу, мы будем использовать выражение `return`.

Возврат значения из функции

Во многих случаях нам потребуется вернуть информацию из функции. Например, давайте на самом деле создадим функцию, конвертирующую дюймы в сантиметры, и назовем ее `convert_in2cm()`. Мы можем представить себе параметр, который нам потребуется передать в такую функцию, а именно — измерение в дюймах. Однако эта функция — идеальный кандидат для возврата информации обратно в программу, а именно — конвертированное измерение в сантиметрах.

Для преобразования длины в дюймах в сантиметры мы должны умножить количество дюймов на 2,54 — примерное количество сантиметров в одном дюйме. Чтобы вернуть результат вычисления обратно в программу, мы воспользуемся выражением `return`. Так как функция *возвращает значение*, или результат, значение, указанное после ключевого слова `return`, будет возвращено в программу. Давайте определим нашу функцию:

```
def convert_in2cm(inches):
    return inches * 2.54
```

Если вы введете эти две строки в оболочку Python, а затем введете команду `convert_in2cm(72)` и нажмете клавишу **Enter**, Python выведет ответ `182.88`. 72 дюйма равняются приблизительно 182,88 сантиметра (или 6 футам — мой рост). Значение 182.88 возвращается функцией, и в оболочке командной строки мы видим возвращаемое значение распечатанным на следующей после вызова функции строке.

Можно выполнить еще одно полезное преобразование: фунты в килограммы. Чтобы преобразовать фунты в килограммы, мы должны разделить вес в фунтах на 2,2 — примерное количество фунтов в 1 килограмме. Давайте создадим функцию `convert_lb2kg()`, которая будет принимать в качестве параметра значение веса в фунтах и возвращать преобразованное значение в килограммах:

```
def convert_lb2kg(pounds):
    return pounds / 2.2
```

Принцип действия выражения `return` напоминает использование параметров, но в обратном порядке, за исключением того, что мы можем вернуть только *одно* значение, а не набор значений, как в случае с установкой принимаемых параметров. (Впрочем, это единственное значение может быть и списком, поэтому, проделав немного работы, вы сможете возвращать в программу несколько значений с помощью одного выражения `return`.)

Использование возвращенных значений в программе

Используя две функции преобразования, давайте создадим глупую, но забавную программу: калькулятор веса и роста в шариках для пинг-понга. Эта программа будет отвечать на вопросы: «Каков мой рост в шариках для пинг-понга?» и «Каков мой вес в шариках для пинг-понга?»

В соответствии со стандартами, шарик для пинг-понга весит 2,7 грамма (0,095 унций) и составляет 40 миллиметров (4 сантиметра или 1,57 дюйма) в диаметре. Чтобы подсчитать, каков наш рост и вес в шариках для пинг-понга, мы должны разделить рост на 4, а вес — на 2,7. Однако не все знают свой вес в граммах и рост в килограммах, так в Соединенных Штатах мы обычно измеряем вес в фунтах, а рост — в футах и дюймах. К счастью, две уже созданные нами функции помогут нам конвертировать эти измерения в метрическую систему. После этого мы сможем воспользоваться полученными числами для конвертирования в шарики.

В нашей программе мы определим две функции преобразования: `convert_in2cm` и `convert_lb2kg`. Затем мы попросим пользователей ввести свой рост и вес, вычислим рост и вес в шариках для пинг-понга — и отобразим результат на экране. Введите и запустите выполнение следующего кода.

PingPongCalculator.py

```
❶ def convert_in2cm(inches):
      return inches * 2.54

  def convert_lb2kg(pounds):
      return pounds / 2.2

❷ height_in = int(input("Введите свой рост в дюймах: "))
  weight_lb = int(input("Введите свой вес в фунтах: "))
```

```
❸ height_cm = convert_in2cm(height_in)
❹ weight_kg = convert_lb2kg(weight_lb)

❺ ping_pong_tall = round(height_cm / 4)
❻ ping_pong_heavy = round(weight_kg * 1000 / 2.7)

❼ feet = height_in // 12
❽ inch = height_in % 12

❾ print("При росте", feet, "футов и", inch, "дюймах и весе",↵
    weight_lb, "фунтов")
  print("ваш рост", ping_pong_tall, "шариков для пинг-понга, а ")
  print("вес сопоставим с", ping_pong_heavy, "шариков для ↵
        пинг-понга!")
```

В строке ❶ мы вводим две созданные нами функции преобразования. Обе функции принимают входящий параметр (`inches` или `pounds`), каждая из функций возвращает значение. В строке ❷ мы просим пользователя ввести его рост и вес и сохраняем введенные значения в переменных `height_cm` и `weight_kg` соответственно. В строке ❸ мы вызываем функцию `convert_in2cm`, передаем ей переменную в `height_in` качестве значения, которое мы хотели бы конвертировать, и сохраняем конвертированное значение в переменной `height_cm`. В строке ❹ мы производим еще одну операцию преобразования и вычисления, конвертируя вес человека в фунтах (сокращенно на английском языке записывается как *lbs*) в килограммы (*кг*) с помощью функции `convert_lb2kg`.

Уравнение в строке ❺ выполняет две операции: во-первых, рост человека в сантиметрах делится на 4 для вычисления роста в шариках для пинг-понга, а затем с помощью функции `round()` выполняется округление ответа до ближайшего целого числа. В строке ❻ мы выполняем аналогичные операции, конвертируя вес человека в килограммах в граммы, умножив его на 1000, а затем делим результат на 2,7 — вес в граммах шарика для пинг-понга. Получившееся число округляется до ближайшего целого числа и сохраняется в переменной `ping_pong_heavy`.

В строках ❼ и ❽ мы также занимаемся математикой, дополнительно вычисляя рост человека в футах и дюймах. Как я уже упоминал ранее, именно таким образом мы обычно указываем рост в Соединенных Штатах Америки, поэтому такое преобразование было бы приятным завершающим жестом программы, а также возможностью человеку проверить

корректность введенной информации. Оператор `//` осуществляет целочисленное деление, то есть 66 дюймов, или 5,5 фута, будет преобразовано просто в `5` и сохранено в переменной `feet`, а оператор % (взятие по модулю) сохранит остаток от деления, `6` дюймов. Выражения `print` в строке (9) выводят на экран рост и вес пользователя, выраженные в стандартных шариках для пинг-понга.

Далее приведены результаты выполнения программы-калькулятора пинг-понга с параметрами в шариках для пинг-понга моих сыновей, Макса и Алекса, и моих. (Единственный минус этой программы заключается в том, что теперь мои дети хотят весить 31 000 шариков для пинг-понга.)

```
>>> ====================RESTART ==========================
>>>
Введите свой рост в дюймах: 42
Введите свой вес в фунтах: 45
При росте 3 футов и 6 дюймах и весе 45 фунтов
ваш рост 27 шариков для пинг-понга, а
вес сопоставим с 7576 шариков для пинг-понга!
>>> ====================RESTART ==========================
>>>
Введите свой рост в дюймах: 47
Введите свой вес в фунтах: 55
При росте 3 футов и 11 дюймах и весе 55 фунтов
ваш рост 30 шариков для пинг-понга, а
вес сопоставим с 9259 шариков для пинг-понга!
>>> ====================RESTART ==========================
>>>
Введите свой рост в дюймах: 72
Введите свой вес в фунтах: 185
При росте 6 футов и 0 дюймах и весе 185 фунтов
ваш рост 46 шариков для пинг-понга, а
вес сопоставим с 31145 шариков для пинг-понга!
>>>
```

Любая создаваемая нами функция может возвращать значение, точно так же как любая определяемая нами функция может принимать параметры в качестве ввода. В зависимости от того, что вы хотите поручить выполнять своей функции, воспользуйтесь одной или обеими возможностями для создания кода, необходимого вашей функции.

Прикосновение интерактивности

Мы уже запрограммировали большое количество красивых графических приложений, однако мы все еще в шаге или даже двух от создания следующей видеоигры либо мобильного приложения. Один из навыков, который нам нужно изучить, — это программирование интерактивного взаимодействия с пользователем, иными словами, того, что позволит нашим программам отвечать на щелчки мышью, нажатия клавиш и т.п.

Большинство приложений *интерактивны* — то есть они позволяют пользователю касаться, щелкать, перетаскивать, нажимать кнопки и в целом иметь ощущение полного контроля над программой. Мы называем такие программы *событийно управляемыми приложениями*, так как эти приложения ожидают действия пользователя, то есть *события*. Код, который отвечает на пользовательское событие, такое как открытие окна, когда пользователь щелкает по значку, или запуск игры при касании кнопки, называется *обработчиком событий*, потому что он обрабатывает и отвечает на событие, созданное пользователем. Такой код также называется *слушателем* событий, так как принцип его работы напоминает ситуацию, когда компьютер терпеливо сидит и слушает указания пользователя. В ближайшее время мы научимся обрабатывать события, чтобы сделать наши приложения более захватывающими и интерактивными.

Обработка событий: `TurtleDraw`

Существует множество способов заставить приложения обрабатывать пользовательские события. В модуле `turtle` языка Python реализованы функции, обрабатывающие пользовательские события, в том числе щелчки мышью и нажатия клавиш. Первой мы протестируем функцию `turtle.onscreenclick()`. Как можно предположить из названия, эта функция позволяет нам обрабатывать события, генерируемые при щелчке мышью по экрану Turtle.

Эта функция, впрочем, несколько отличается от тех, что мы использовали и создавали ранее: аргумент, передаваемый нами в функцию `turtle.onscreenclick()`, — это не значение, а имя другой функции:

```
turtle.onscreenclick(t.setpos)
```

Помните функцию `setpos()`, которую мы использовали для переноса указателя мыши на определенную точку экрана (x, y)? Сейчас мы говорим компьютеру, что, когда пользователь щелкает мышью по экрану Turtle,

черепашка должна быть перенесена в ту точку экрана, по которой был сделан щелчок мышью. Функция, передаваемая другой функции в качестве аргумента, иногда называется *функцией обратного вызова* (так как другая функция *обратно вызывает* эту функцию). Обратите внимание, что при передаче одной функции в качестве аргумента другой после имени передаваемой функции не нужно вводить скобки.

Передавая имя функции `t.setpos` в функцию `turtle.onscreenclick()`, мы сообщаем компьютеру, что в ответ на щелчок мышью по экрану черепашка должна устанавливаться в ту позицию, по которой пользователь щелкнул мышью. Давайте испробуем это на примере короткой программы.

TurtleDraw.py

```
import turtle
t = turtle.Pen()
t.speed(0)
turtle.onclick(t.setpos)
```

Введите этот код в IDLE, запустите программу, а затем щелкните мышью в разных участках экрана. Вы только что создали графический редактор с помощью всего лишь четырех строк кода! На рис. 7.3 показан простой набросок, который я нарисовал.

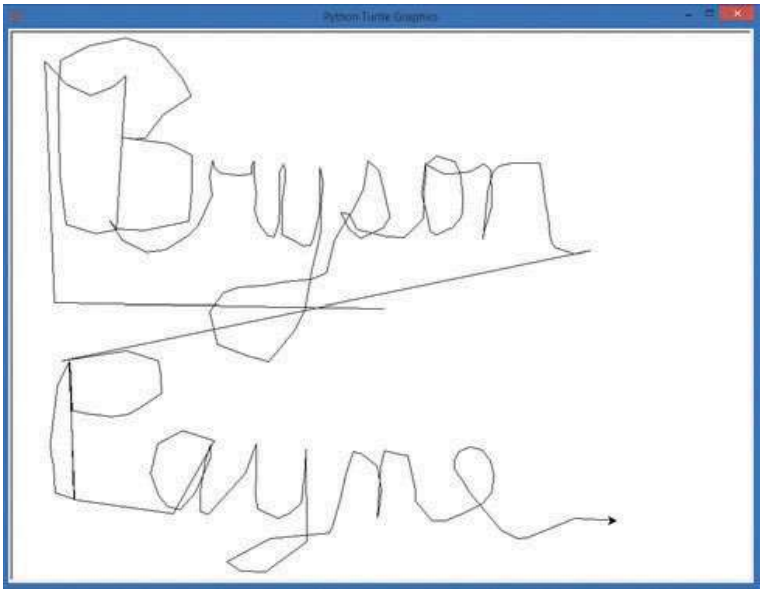


Рис. 7.3. *Набросок TurtleDraw.py*
(видимо, я действительно писатель, а не художник — и это не просто так)

Все это работает потому, что мы сказали компьютеру произвести некое действие, когда пользователь щелкает мышью по экрану, а именно — установить черепашку в ту точку экрана, по которой был сделан щелчок. Ручка черепашки прижата к экрану по умолчанию, поэтому, когда пользователь щелкает мышью по экрану для рисования, черепашка перемещается туда и рисует линию от исходного положения до точки, по которой пользователь щелкнул мышью.

Вы можете настроить программу *TurtleDraw.py* под себя, изменив цвет фона, цвет ручки, ее ширину и другие параметры. Протестируйте версию программы, созданную моим четырехлетним сыном (с небольшой помощью его папы).

TurtleDrawMax.py

```
import turtle
t = turtle.Pen()
t.speed(0)
turtle.onscreenclick(t.setpos)
turtle.bgcolor("blue")
t.pencolor("green")
t.width(99)
```

Максу очень понравилась рисующая программа, но он захотел, чтобы экран был голубым, а ручка — зеленой и очень толстой, поэтому мы установили значения параметров `bgcolor()`, `pencolor()` и `width()` равными `blue`, `green` и `99` соответственно. Мы также решили попробовать наудачу установить эти параметры *после* того, как скажем компьютеру, что именно нужно сделать при щелчках мышью по экрану (`t.setpos`). И все заработало как нужно, потому что программа продолжает выполняться все время, даже во время ожидания щелчков мышью. Поэтому, когда пользователь щелкнет по экрану в первый раз, экран и ручка уже окрашены нужным цветом, для ручки также задана нужная толщина, как показано на рис. 7.4.

Использовав функцию `setpos()` в качестве функции обратного вызова для функции `turtle.onscreenclick()`, мы создали интересную программу, которая взаимодействует с пользователем, рисуя линии в ответ на щелчок мышью в любом участке окна. Попробуйте настроить приложение в соответствии со своими желаниями, изменив цвета, толщину или любые другие параметры, какие только можете себе представить.

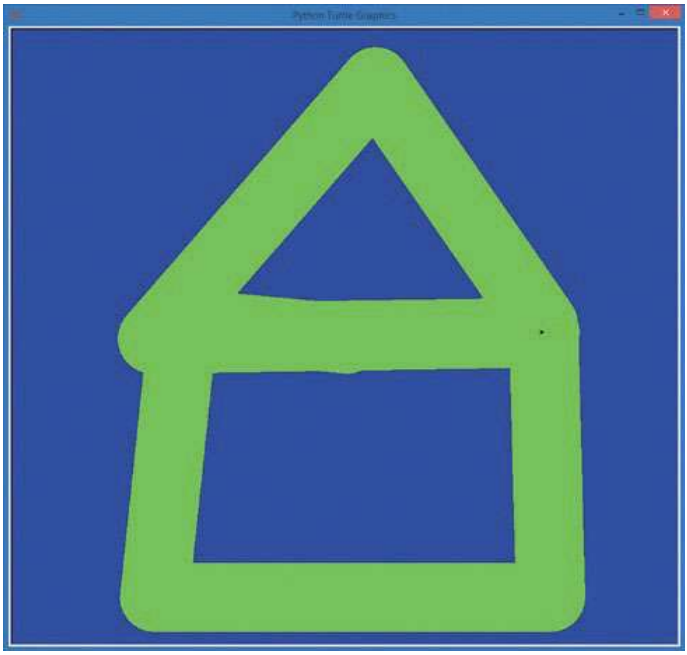


Рис. 7.4. *Рисунок, созданный несколькими щелчками мыши в программе TurtleDrawMax.py*

Прослушивание событий клавиатуры: `ArrowDraw`

Наша программа, рисовавшая с помощью черепашки, показала нам, как прослушивание щелчков мыши может дать пользователю ощущение более полного контроля над программой. В этом разделе мы узнаем, как использовать механизмы интерактивного взаимодействия с клавиатурой для добавления дополнительных опций в программы. Мы также определим функции, которые будем использовать в качестве обработчиков событий.

В программе *TurtleDraw.py* мы передавали в функцию `turtle.onscreenclick()` параметр `t.setpos` в качестве функции обратного вызова, чтобы сказать компьютеру, что именно мы хотим сделать, когда происходит событие `onscreenclick()`: мы хотели перенести черепашку в точку экрана, по которой был совершен щелчок мышью. Функция `setpos()` уже определена для нас в модуле `turtle`, но если мы захотим создать собственные функции для обработки событий? Предположим, мы хотим создать программу, которая позволяет пользователю перемещать черепашку по экрану нажатием клавиш со стрелками вместо щелчка левой кнопкой мыши. Как нам решить эту задачу?

Во-первых, нам необходимо создать функции для перемещения черепашки при каждом нажатии на клавишу со стрелкой, а затем мы должны проинструктировать компьютер отслеживать нажатия этих клавиш. Давайте напишем программу, которая будет прослушивать клавиши со стрелками вверх (↑), влево (←) и вправо (→) на клавиатуре и позволять пользователю перемещать или поворачивать черепашку с помощью одной из этих клавиш.

Давайте определим несколько функций — `up()`, `left()` и `right()`, — которые будут перемещать и поворачивать черепашку:

```
def up():
    t.forward(50)
def left():
    t.left(90)
def right():
    t.right(90)
```

Наша первая функция, `up()`, перемещает черепашку вперед на 50 пикселей. Вторая, `left()`, поворачивает черепашку влево на 90 градусов. Наконец, функция `right()` поворачивает черепашку вправо на 90 градусов.

Для того чтобы запустить каждую из этих функций, когда пользователь нажимает на правильную клавишу со стрелкой, мы должны сообщить компьютеру, какая функция соответствует какой клавише, а также скомандовать компьютеру начать прослушивание нажатий клавиш. Для установки функции обратного вызова для события нажатия клавиши мы используем функцию `turtle.onkeypress()`. Обычно эта функция принимает два параметра: имя функции обратного вызова (созданный нами обработчик события) и конкретная клавиша, нажатие на которую нужно прослушивать. Чтобы соединить все три функции с соответствующими клавишами, нам нужно написать следующий код:

```
turtle.onkeypress(up, "Up")
turtle.onkeypress(left, "Left")
turtle.onkeypress(right, "Right")
```

Первая строка настраивает функцию `up()` в качестве обработчика события нажатия на клавишу `"Up"`. Первый параметр — имя функции (`up`), за которым указывается английское название клавиши **Стрелка вверх** (↑) — `"Up"`. Тот же принцип работает и для нажатий на клавиши

Стрелка влево (←) и **Стрелка вправо** (→). Последним шагом будет инструкция о начале прослушивания нажатий клавиш, передаваемая с помощью следующей команды:

```
turtle.listen()
```

Эта последняя строка нам нужна по нескольким причинам. Во-первых, в отличие от щелчков мышью, простое нажатие на клавишу не дает гарантии, что именно окно `turtle` получит это нажатие. Когда вы щелкаете мышью по окну на рабочем столе, это окно перемещается вперед и получает *фокус*, иными словами, это значит, что именно это окно будет получать пользовательский ввод. Когда вы щелкаете мышью по окну `turtle`, это автоматически приводит к фокусировке любых событий мыши на этом окне. Однако в случае с клавиатурой простое нажатие клавиш не приводит к получению этих нажатий нужным окном. Команда `turtle.listen()` гарантирует, что окно `turtle` будет находиться в фокусе рабочего стола и, соответственно, сможет услышать нажатия клавиш. Во-вторых, команда `listen()` указывает компьютеру начать обрабатывать события нажатия всех клавиш, которые мы подключили к функциям с помощью функции `onkeypress()`.

Ниже приведена полная версия программы *ArrowDraw.py*.

ArrowDraw.py

```
  import turtle
  t = turtle.Pen()
  t.speed(0)
❶ t.turtlesize(2,2,2)
  def up():
      t.forward(50)
  def left():
      t.left(90)
  def right():
      t.right(90)
  turtle.onkeypress(up, "Up")
  turtle.onkeypress(left, "Left")
  turtle.onkeypress(right, "Right")
  turtle.listen()
```

В программе *ArrowDraw.py* встречается единственная новая команда в строке (1), которая в два раза увеличивает размер и контур стрелки черепашки с помощью вызова функции `t.turtlesize(2,2,2)`. Три

параметра данной функции — это: горизонтальное растяжение (2 означает увеличение ширины в 2 раза), вертикальное растяжение (в 2 раза выше) и толщина контура (2 пиксела в толщину). На рис. 7.5 показан результат.

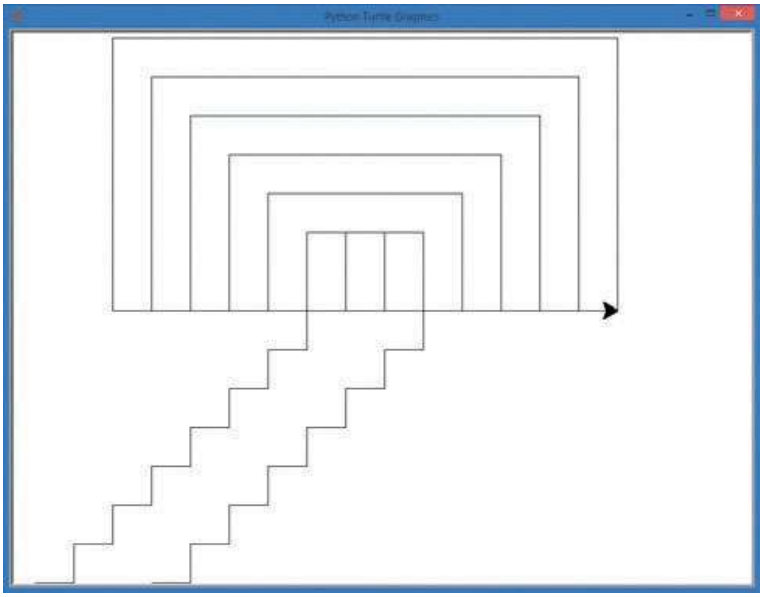


Рис. 7.5. *Программа ArrowDraw.py позволяет пользователю рисовать с помощью клавиш ←, ↑ и →. Увеличенный размер стрелки черепашки упрощает понимание того, куда направлена стрелка в данный момент*

Эта программа напоминает старую игрушку «Волшебный экран»: вы можете нарисовать интересные фигуры, используя только три клавиши со стрелками, а также отследить свои шаги. Не стесняйтесь персонализировать это приложение, выбирайте свои цвета, толщину ручки или добавляйте любые параметры, какие только захотите. Еще одна функция, которую можно добавить, и она включена в задачу в конце главы, заключается в возможности перенести черепашку в новую точку щелчком мыши. Придумывайте новые функции — и испытывайте их, ведь это самый лучший способ изучить что-то новое!

Обработка событий с параметрами: `ClickSpiral`

В программе *TurtleDraw.py* мы дали возможность пользователям рисовать щелчками мыши, проинструктировав обработчик `turtle.onscreenclick()` вызывать функцию `t.setpos` при каждом щелчке мышью по экрану. Давайте создадим новую программу: *ClickSpiral.py*, которая будет рисовать спирали в любой точке экрана, по которой щелкнет пользователь, как показано на рис. 7.6.

Обработчик события `onscreenclick()` передает в качестве аргументов координаты *x* и *y* при каждом щелчке мышью в указанную нами функцию обратного вызова. Если мы хотим обрабатывать щелчки мышью с помощью собственной функции, нам нужно лишь написать такую функцию, которая принимала бы эти значения — координаты *x* и *y* каждого щелчка мышью — в качестве пары параметров.

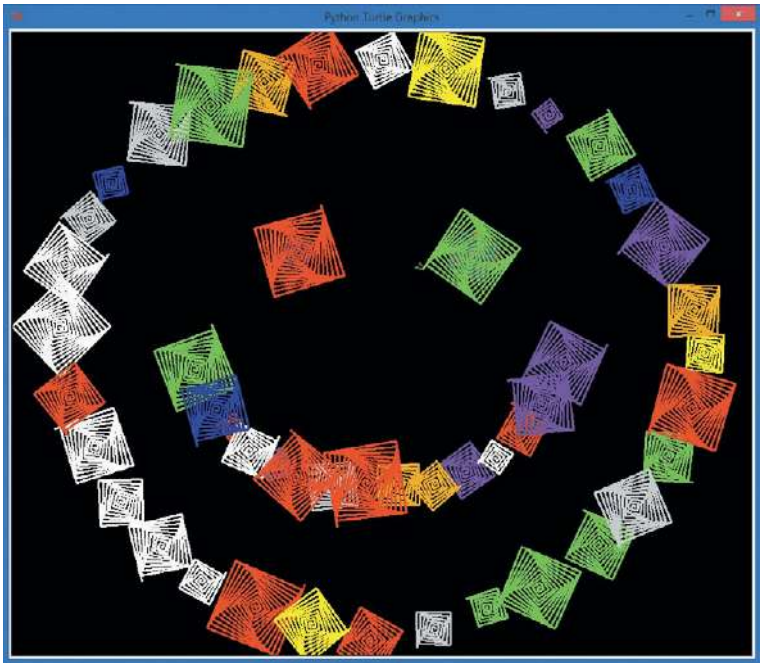


Рис. 7.6. *Смайлик, нарисованный приложением ClickSpiral.py*

Программа *RandomSpiralsFunction.py* (с. 170) содержала функцию `random_spiral()`, которая рисовала случайные спирали в случайных участках экрана. Сейчас вместо того, чтобы рисовать спирали где попало, мы хотим, чтобы спирали появлялись там, где пользователь щелкнет мышью. Чтобы сделать это, мы можем переписать функцию `random_spiral()` так, чтобы она принимала два параметра, `x` и `y`, от обработчика события `onscreenclick()`. Мы переименуем функцию в `spiral(x, y)`.

```
def spiral(x, y):
    t.pencolor(random.choice(colors))
    size = random.randint(10,40)
    t.penup()
```

```
    t.setpos(x, y)
    t.pendown()
    for m in range(size):
          t.forward(m*2)
          t.left(91)
```

В данной версии мы изменили определение функции так, чтобы оно отражало новое имя и чтобы функция принимала два получаемых нами параметра для отрисовки спирали в выбранной точке экрана: `spiral(x, y)`. Для каждой спирали мы по-прежнему выбираем случайный цвет и размер. Однако мы убрали две строки, которые генерировали случайные координаты `x` и `y`, так как мы получим координаты `x` и `y` в качестве аргументов от обработчика событий `onscreenclick()`. Как и в случае с функцией `random_spiral()`, мы перемещаем ручку в нужную точку (x, y), а затем отрисовываем спираль.

Единственный шаг, который нам еще нужно сделать, — это настроить окно `Turtle` и список цветов, а затем проинструктировать обработчик событий `turtle.onscreenclick()` вызывать функцию спирали каждый раз, когда пользователь щелкает мышью по окну для рисования. Далее приведена завершенная программа.

ClickSpiral.py

```
import random
import turtle
t = turtle.Pen()
t.speed(0)
turtle.bgcolor("black")
colors = ["red", "yellow", "blue", "green", "orange", ↵
          "purple", "white", "gray"]
def spiral(x, y):
    t.pencolor(random.choice(colors))
    size = random.randint(10,40)
    t.penup()
    t.setpos(x, y)
    t.pendown()
    for m in range(size):
          t.forward(m*2)
          t.left(91)
❶ turtle.onscreenclick(spiral)
```

Как и в программе *TurtleDraw.py*, мы не используем скобки после имени функции обратного вызова, а также не указываем никакие параметры.

Команда (1): `turtle.onscreenclick(spiral)` сообщает нашей программе о необходимости вызывать функцию `spiral(x, y)` каждый раз, когда пользователь щелкает мышью по экрану. При этом обработчик событий будет автоматически посылать два параметра — координаты точки (*x*, *y*), по которой был сделан щелчок мышью, — в функцию обратного вызова `spiral`. То же самое происходило и в программе *TurtleDraw.py*, но в этот раз мы создали собственную функцию для отрисовки спирали случайного цвета и размера в точке, куда щелкнули мышью.

Еще один шаг вперед: `ClickAndSmile`

Давайте расширим возможности этого интерактивного приложения, внеся еще одно изменение. Представим, что вместо спирали мы хотим создать смайлик при каждом щелчке мышью по экрану для рисования. Код этой программы будет очень похож на код программы *RandomSmileys.py* на с. 177, однако вместо цикла, отрисовывающего 50 смайликов в случайных участках экрана, эта программа будет обрабатывать щелчок мышью, отрисовывая смайлик в любом участке экрана по выбору пользователя, как только пользователь щелкает мышью по экрану.

На самом деле, так как созданная нами функция `draw_smiley()` уже принимает два параметра (координаты точки *x* и *y*, в которой мы хотим отрисовать смайлик), код программы *ClickAndSmile.py* идентичен коду программы *RandomSmileys.py* за исключением, разве что, последнего блока. Нужно лишь заменить цикл, отрисовывающий 50 случайных смайликов, на вызов функции `turtle.onscreenclick(draw_smiley)`. Вы помните, что функция `turtle.onscreenclick(draw_smiley)` позволяет нам передавать имя функции (как, например, `setpos`) в качестве обработчика щелчков мышью? Мы можем передать этой функции имя функции `draw_smiley` с тем, чтобы в каждой точке, где сделан щелчок мышью, функция `draw_smiley()` выполнила свою работу. Внутри скобок функции `turtle.onscreenclick()` мы не используем скобки после имени функции `draw_smiley`, а также не указываем никакие параметры.

ClickAndSmile.py

```
import random
import turtle
t = turtle.Pen()
t.speed(0)
t.hideturtle()
turtle.bgcolor("black")
```

```
def draw_smiley(x, y):
    t.penup()
    t.setpos(x, y)
    t.pendown()
    # Лицо
    t.pencolor("yellow")
    t.fillcolor("yellow")
    t.begin_fill()
    t.circle(50)
    t.end_fill()
    # Левый глаз
    t.setpos(x-15, y+60)
    t.fillcolor("blue")
    t.begin_fill()
    t.circle(10)
    t.end_fill()
    # Правый глаз
    t.setpos(x+15, y+60)
    t.begin_fill()
    t.circle(10)
    t.end_fill()
    # Рот
    t.setpos(x-25, y+40)
    t.pencolor("black")
    t.width(10)
    t.goto(x-10, y+20)
    t.goto(x+10, y+20)
    t.goto(x+25, y+40)
    t.width(1)
turtle.onscreenclick(draw_smiley)
```

Теперь вместо рисования случайных смайликов по всему экрану пользователь может отрисовать смайлик в любом участке окна, по которому щелкнет мышью. Пользователи могут даже нарисовать большой смайлик, состоящий из множества маленьких смайликов, как показано на рис. 7.7.

Какие бы приложения вы ни желали создать, скорее всего, интерактивность — то их свойство, которое будет непосредственно влиять на ощущения пользователей от использования приложения. Вспомните игры, за которыми вы проводите бóльшую часть времени. У всех них есть одна общая черта, а именно — пользователи имеют определенный контроль над тем, что происходит. Перемещаете ли вы весло, чтобы ударить по мячу, нажимаете ли вы кнопку мыши или касаетесь экрана и проводите пальцем по нему, чтобы выстрелить чем-то в воздух, — вы генерируете события, а программы, которые вы любите, обрабатывают эти события, чтобы

сделать нечто крутое. Давайте создадим еще одно интерактивное приложение для практики, а затем мы напишем еще больше приложений, похожих на те, что используются каждый день.

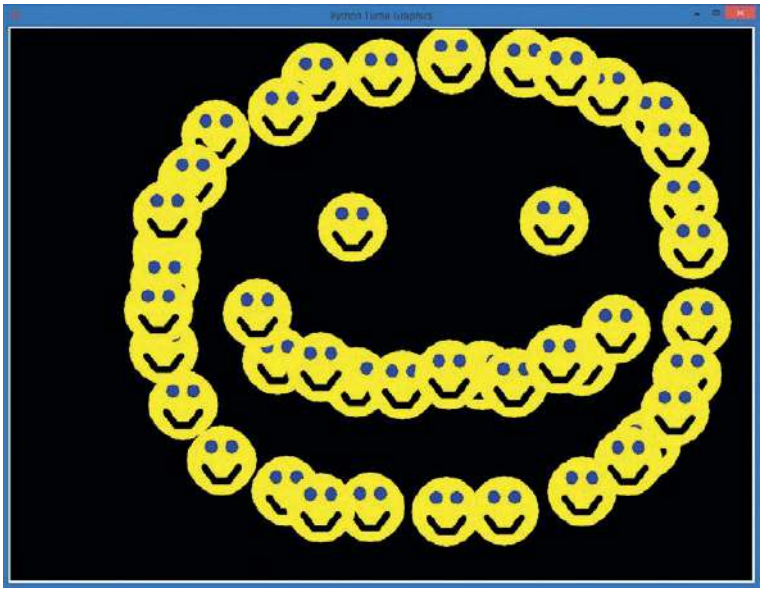


Рис. 7.7. *Мы сделали программу со смайликами более интерактивной, рисующей смайлик в любом участке окна, по которому пользователь щелкнет мышью*

ClickKaleidoscope

Объединим наше умение создавать функции с умением обрабатывать интерактивные щелчки мышью для создания интерактивного калейдоскопа. Пользователь сможет щелкнуть по любому участку экрана — и в ответ программа нарисует четыре отраженные спирали случайной формы и цвета, начиная с точки, по которой пользователь щелкнул мышью. Результат будет выглядеть как программа *Kaleidoscope.py* со с. 159, с той разницей, что пользователи смогут создавать с помощью калейдоскопа собственные уникальные орнаменты.

Функция `draw_kaleido()`

Давайте поговорим о трудностях, с которыми мы столкнемся при создании программы персонализированного калейдоскопа. Мы знаем, что хотим позволить пользователю щелкнуть мышью по экрану для начала процесса рисования, поэтому мы воспользуемся функцией `turtle.onscreenclick()`

из предыдущего раздела. Мы знаем, что эта функция даст нам точку на экране (x, y), которую мы сможем использовать в функции обратного вызова. Можно еще раз просмотреть нашу исходную программу калейдоскопа, и увидеть: все, что нам нужно для создания желаемого эффекта отражения, — нарисовать по одной спирали в каждой из четырех точек: (x, y), $(-x, y)$, $(-x, -y)$ и $(x, -y)$.

Для создания иллюзии зеркала все четыре отраженные спирали должны быть одинакового цвета и одинакового размера. Мы назовем нашу функцию `draw_kaleido()` и определим ее следующим образом:

```
❶ def draw_kaleido(x, y):
❷ t.pencolor(random.choice(colors))
❸ size = random.randint(10,40)
  draw_spiral(x, y, size)
      draw_spiral(-x, y, size)
      draw_spiral(-x, -y, size)
```

В строке ❶ мы называем нашу функцию `draw_kaleido` и позволяем ей принимать два параметра, `x` и `y` из обработчика событий `turtle.onscreenclick()`, с тем чтобы отрисовка четырех отраженных спиралей начиналась с точки (x, y), по которой пользователь щелкнул мышью. Затем в строке ❷ мы случайным образом выбираем цвет ручки для всех четырех отраженных спиралей из обычного набора цветов — списка `colors`.

В строке ❸ мы выбираем случайный размер для всех четырех отраженных спиралей и сохраняем его в переменной `size`. Наконец, мы рисуем все четыре спирали в точках (x, y), $(-x, y)$, $(-x, -y)$ и $(x, -y)$ с помощью функции с именем `draw_spiral()`, которую нам только предстоит еще написать.

Функция draw_spiral()

Функции draw_spiral() потребуется отрисовать спираль, начиная с определенной пользователем точки экрана (*x*, *y*). Ручка черепашки Python запомнит цвет после его установки, поэтому у нас нет необходимости передавать эту информацию в качестве параметра функции draw_spiral(), однако местоположение (*x*, *y*) и размер (size) спирали, которую мы хотим нарисовать, нам нужны в любом случае. Поэтому мы определим нашу функцию draw_spiral() так, чтобы она принимала три параметра:

```
def draw_spiral(x, y, size):
    t.penup()
    t.setpos(x, y)
    t.pendown()
    for m in range(size):
         t.forward(m*2)
         t.left(92)
```

Эта функция принимает в качестве параметров местоположение отрисовываемой спирали, передаваемое в переменных x и y, а также параметр size, сообщающий нам о размере будущей спирали. Внутри функции мы поднимаем ручку черепашки над экраном, чтобы при ее перемещении на экране не оставался след, мы перемещаем ручку в заданную точку (*x*, *y*), а затем возвращаем ручку на экран, тем самым подготавливаясь к началу рисования. Цикл for выполнит итерации и увеличит значение переменной m от 0 до size, рисуя при этом спираль до достижения длины стороны указанного значения.

Все, что нам нужно сделать в нашей программе, кроме импортирования модулей turtle и random и настройки экрана и списка цветов, — это проинструктировать компьютер начать прослушивание щелчков мышью по экрану Turtle и вызывать функцию draw_spiral() при каждом щелчке мышью. Все это мы можем осуществить с помощью команды turtle.onscreenclick(draw_kaleido).

Соберем все вместе

Далее приведена полная версия программы *ClickKaleidoscope.py*. Введите код программы в редактор IDLE или загрузите его с сайта **https://eksmo.ru/files/Python_deti.zip** и запустите программу.

ClickKaleidoscope.py

```
import random
import turtle
t = turtle.Pen()
t.speed(0)
t.hideturtle()
turtle.bgcolor("black")
colors = ["red", "yellow", "blue", "green", "orange", ↵
          "purple", "white", "gray"]
def draw_kaleido(x, y):
    t.pencolor(random.choice(colors))
    size = random.randint(10,40)
    draw_spiral(x, y, size)
    draw_spiral(-x, y, size)
    draw_spiral(-x, — y, size)
    draw_spiral(x, — y, size)
def draw_spiral(x, y, size):
    t.penup()
    t.setpos(x, y)
    t.pendown()
    for m in range(size):
         t.forward(m*2)
         t.left(92)
turtle.onscreenclick(draw_kaleido)
```

Наша программа начинается, как обычно, с выражений `import`, после которых мы настраиваем среду Turtle и создаем список цветов. Далее мы определяем функции `draw_spiral()` и `draw_kaleido()`. Программный код завершается командой, инструктирующей компьютер начать прослушивание щелчков мышью по экрану Turtle и вызывать функцию `draw_spiral()` при каждом щелчке мышью. Теперь, когда бы пользователь ни щелкнул мышью по точке на экране для рисования, программа нарисует спираль и отразит ее относительно осей x и y так, что на экране появятся четыре спирали одинаковой случайной формы и размера.

Результат — полностью интерактивная версия программы спирального калейдоскопа, позволяющая пользователю контролировать создаваемый отраженный орнамент, щелкая мышью по тем участкам экрана, в которых необходимо рисовать спирали. На рис. 7.8 показан результат тестового запуска программы и отраженные орнаменты, созданные из спиралей.

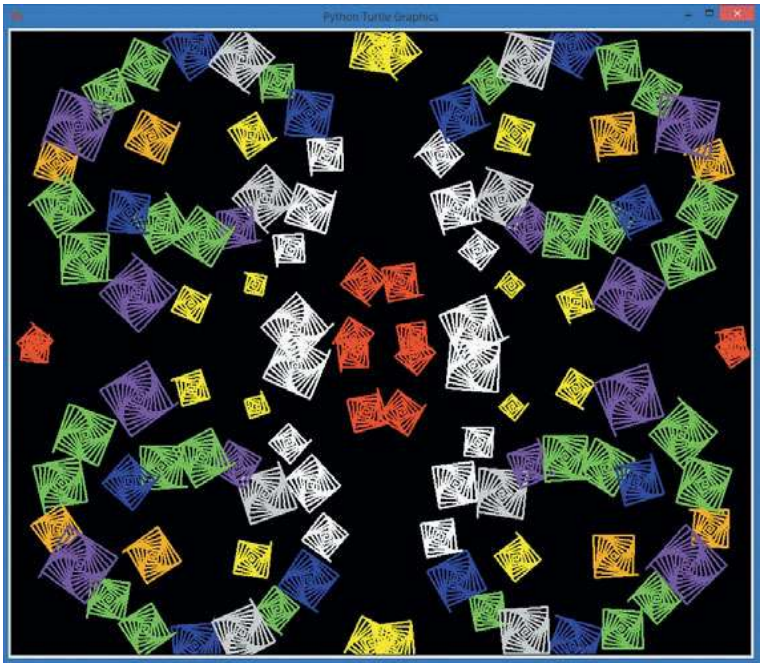


Рис. 7.8. *Наша интерактивная программа калейдоскопа позволяет создавать вам любые отраженные орнаменты!*

Попробуйте создать собственные орнаменты (как, например, первая буква вашего имени или фамилии!), создайте снимок экрана (в Windows нажав и удерживая клавишу **Alt,** нажмите клавишу **Print Screen**, чтобы скопировать окно Turtle, а затем вставьте снимок в Word или ваш любимый графический редактор; на компьютере Mac нажмите и удерживайте клавиши Windows [⊞] (или **Command** [⌘], если у вас на компьютере операционная система MacOS), **Shift** и **4**, нажмите клавишу **Пробел,** а затем щелкните мышью по окну Turtle, чтобы сохранить окно для рисования как изображение на рабочем столе с именем **Screenshot <date and time>.png**).

Что вы узнали

В этой главе вы научились организовывать блоки повторно используемого кода в функции, вызывать свои собственные функции из любого участка программы, передавать таким функциям информацию в виде параметров, а также получать информацию из функций в качестве возвращаемых значений. Мы написали наши первые событийно управляемые программы,

инструктируя компьютер о начале прослушивания щелчков мышью и нажатий на клавиши клавиатуры. Вы также научились писать собственные функции обратного вызова для реакции на эти пользовательские события.

Мы разработали наши первые полностью интерактивные программы. Благодаря навыкам, полученным в этой главе, вы готовы начать писать еще более продвинутые приложения. Приложения, которые мы часто и с удовольствием используем, дают пользователю ощущение контроля над программой за счет того, что программа отвечает на щелчки мышью, прикосновения, нажатия клавиш и т.д.

Изучив концепции, изложенные в этой главе, вы должны уметь делать следующее.

- Делать код более пригодным для повторного использования благодаря функциям.
- Организовывать и группировать код в функции.
- Определять функции на языке Python с помощью ключевого слова `def`.
- Вызывать собственные функции в создаваемых программах.
- Определять и использовать функции, принимающие параметры в качестве входящих значений.
- Писать функции, возвращающие значения при вызове.
- Преобразовывать математическую формулу в функцию, возвращающую значение.
- Объяснять некоторые особенности событийно управляемых программ.
- Писать простую событийно управляемую программу с применением обработчика события.
- Писать приложение, принимающее щелчки мышью и рисующее на экране.
- Писать код обработчиков событий клавиатуры.
- Программировать обработчики событий, принимающие параметры.
- Использовать координаты экрана x и y для отрисовки требующихся орнаментов, например калейдоскопов.

Задачи по программированию

Далее приведено несколько задач, которые позволят вам расширить изученное в этой главе. Для загрузки готовых примеров решений перейдите на сайт **https://eksmo.ru/files/Python_deti.zip**.

#1. Отраженные смайлики

Создайте гибрид программ *ClickAndSmile.py* и *ClickKaleidoscope.py*, рисующий смайлики, отраженные в четырех углах экрана при щелчке мыши, по аналогии со спиралями программы-калейдоскопа. Если вы хотите усложнить задачу, сделайте так, чтобы два смайлика отрисовывались перевернутыми, как будто они действительно отражены в зеркале относительно оси *x*.

#2. Еще немного пинг-понговой математики

Измените калькулятор шариков для пинг-понга так, чтобы он принимал в качестве ввода количество шариков. Программа должна сообщать пользователю высоту введенного количества шариков, сложенных друг на друга, а также каков вес этого количества шариков.

#3. Улученный графический редактор

Измените программу *ArrowDraw.py* так, чтобы она позволяла пользователю поворачивать черепашку на меньшие углы, например 45 градусов (или даже 30 или 15), чтобы пользователи получили более точный контроль над черепашкой. Затем добавьте дополнительные опции клавиш, например возможность нажатия клавиши с символом больше (**>**), чтобы увеличить длину рисуемой линии, или клавиши с символом меньше (**<**), чтобы уменьшить длину линии, нажатие на клавишу **W** должно увеличивать толщину линии, а нажатие на клавишу **T** — уменьшать. Чтобы превратить эту программу в отличный графический редактор, добавьте обратную связь в форме вывода на экран информации о толщине ручки, длине отрезка и направлении черепашки при внесении каждого изменения в рисунок.

Наконец, в виде завершающего штриха добавьте возможность менять местоположение черепашки. (Подсказка: создайте функцию, принимающую два параметра (*x*,*y*); поднимать черепашку, переносить ее в точку (*x*,*y*), а потом помещать ее обратно на экран. Затем передайте имя этой функции методу `turtle.onscreenclick()` для завершения программы.)

Глава 8

ТАЙМЕРЫ И АНИМАЦИЯ: КАК ПОСТУПИЛ БЫ ДИСНЕЙ?

Один из подходов, благодаря которому я научился программировать, будучи еще подростком, заключался в создании коротких игр и анимаций с последующим изменением кода для выполнения новых операций. Я был потрясен, видя что благодаря моему коду на экране сразу появлялась графика. Думаю, что вам это понравится не меньше, чем мне.

У игр и анимации есть несколько схожих черт. Во-первых, это весело! Во-вторых, и игры и анимация подразумевают отрисовку графики на экране, а также изменение этой графики со временем для создания иллюзии движения. Мы могли рисовать графику с самого начала книги, однако библиотека Turtle слишком медленная и не очень хорошо подходит для использования с большим количеством анимации и движущихся объектов. В этой главе мы установим и начнем работать с новым модулем под названием *Pygame*. Этот модуль даст нам возможность рисовать, создавать анимации и даже создавать игры жанра аркада, используя уже полученные вами навыки.

Использование графического интерфейса Pygame

Графический интерфейс пользователя (*GUI*, от английского «*graphical user interface*», аббревиатура читается как «гу-и») включает в себя все кнопки, иконки, меню и окна, которые вы видите на экране компьютера.

Именно через элементы графического интерфейса вы взаимодействуете с компьютером. Когда вы перетаскиваете файл или щелкаете мышью по значку, чтобы открыть программу, вы работаете с GUI. В играх, когда вы нажимаете на клавиши, передвигаете мышь или щелкаете ей, единственная причина, по которой вам стоит ожидать какой-либо реакции (например, бег, прыжок, поворот вида и так далее), заключается в том, что программист настроил GUI.

Как и библиотека Turtle, библиотека Pygame очень визуальная и идеально подходит для создания GUI для игр, анимаций и еще много чего. Эта библиотека переносимая и совместима практически со всеми операционными системами от Windows до macOS и до Linux и далее, поэтому игры, создаваемые с помощью Pygame, могут быть запущены почти на любом компьютере. На рис. 8.1 показано, как выглядит сайт Pygame, на который вам нужно перейти, чтобы загрузить на компьютер библиотеку Pygame.

Рис. 8.1. *Библиотека Pygame бесплатная, как и учебники и примеры игр, размещенные на сайте*

Для начала установите модуль `pygame`, загрузив установщик со страницы **Downloads** (Загрузки) по адресу **www.pygame.org/download.shtml**. Для Windows вам, скорее всего, понадобится загрузить файл *pygame-1.9.1.win32-py3.1.msi*,

однако, если вы столкнетесь с какими-либо трудностями, обратитесь к Приложению Б для получения справки. Для macOS и Linux процесс установки несколько более сложный, см. Приложение Б или перейдите на сайт **https://eksmo.ru/files/Python_deti.zip**, чтобы получить пошаговые инструкции.

Вы можете проверить правильность установки библиотеки Pygame, введя следующую команду в оболочку Python:

```
>>> import pygame
```

Если в ответ вы получаете обычное приглашение >>>, знайте, что Python смог безошибочно найти модуль `pygame`, а библиотека Pygame готова к использованию.

Рисование точки с помощью Pygame

После установки Pygame вы можете запустить короткую программу для отрисовки точки на экране, как показано на рис. 8.2.

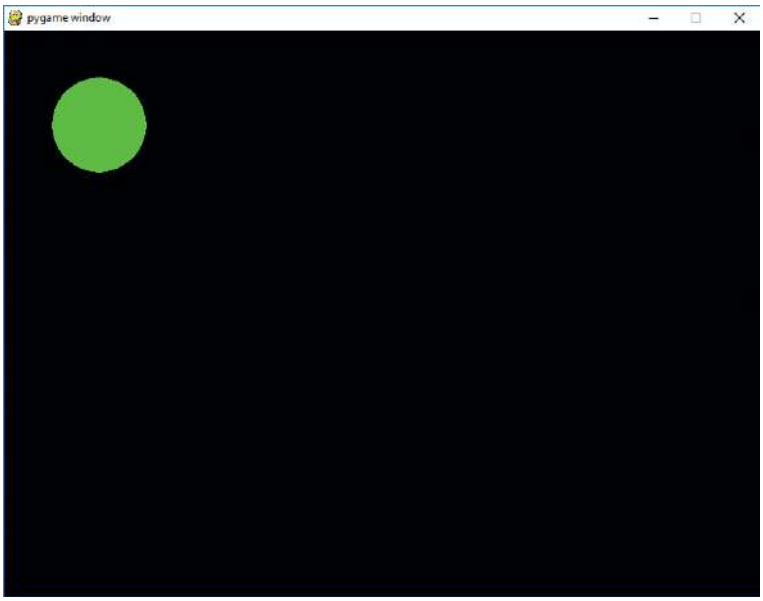


Рис. 8.2. *Программа ShowDot.py в действии*

Введите следующий программный код в новое окно IDLE или скачайте его по адресу: **https://eksmo.ru/files/Python_deti.zip.**

ShowDot.py

```
   import pygame

❶ pygame.init()
❷ screen = pygame.display.set_mode([800,600])

❸ keep_going = True
❹ GREEN = (0,255,0) # Три цвета RGB для зеленого (GREEN)
                    # цвета
   radius = 50

❺ while keep_going:
❻     for event in pygame.event.get():
❼         if event.type == pygame.QUIT:
                  keep_going = False
❽     pygame.draw.circle(screen, GREEN, (100,100), radius)
❾     pygame.display.update()

❿ pygame.quit()
```

Давайте построчно проанализируем эту программу. В первую очередь мы импортируем модуль `pygame` для получения доступа к его функциям. В строке ❶ мы *инициализируем*, то есть настраиваем для дальнейшего использования, модуль Pygame. Каждый раз, когда вы захотите воспользоваться модулем Pygame, потребуется вызывать команду `import pygame` перед вызовом остальных функций Pygame.

Строка ❷ `pygame.display.set_mode([800,600])` создает дисплей шириной 800 и высотой 600 пикселей. Мы сохраним это в переменной `screen`. В Pygame окна и графику принято называть *поверхностями*, а поверхность дисплея `screen` — это наше основное окно, в котором будет отрисовываться вся наша графика.

В строке ❸ вы, возможно, узнали нашу переменную цикла `keep_going`, которую мы использовали в игровых циклах программ *HighCard.py* и *FiveDice.py* в главе 6 в качестве булевого (логического) флага, сообщающего программе продолжать играть. В этом примере Pygame мы используем игровой цикл для продолжения рисования графики на экране до тех пор, пока пользователь не закроет окно.

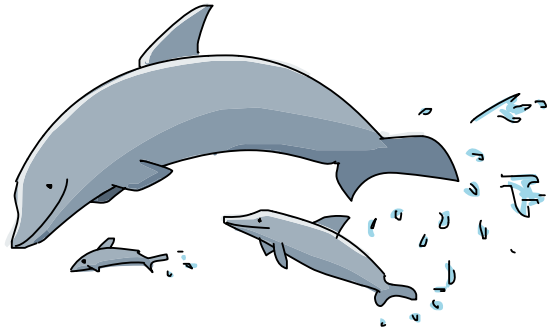
В строке ❹ мы объявляем две переменные: `GREEN` и `radius`, которые будут использоваться при отрисовке круга. Переменной `GREEN`

присваивается значение тройки цветов RGB `(0, 255, 0)`, что соответствует ярко-зеленому цвету.

RGB, или *Red Green Blue* (Красный, Зеленый, Голубой), — это один из многих способов задания цвета. Для выбора цвета необходимо выбрать три числа, каждое должно находиться в диапазоне от 0 до 255. Первое число определяет количество красного в выбранном цвете, второе число — количество зеленого, а третье — количество голубого. Мы выбрали число `255` для задания зеленого цвета и `0` — для красного и голубого, поэтому наш цвет RGB состоит только из зеленого и не содержит примеси ни красного, ни голубого цветов.

Наша переменная `GREEN` — это константа. Иногда мы записываем имена *констант*, то есть переменных, значения которых не могут изменяться, заглавными буквами. Так как цвет должен оставаться неизменным на протяжении всей программы, мы записали имя переменной `GREEN` заглавными буквами. Переменной `radius` мы присвоили значение 50 пикселей, что позволит нам отрисовать круг 100 пикселей в диаметре.

Цикл `while` в строке ❺ — это наш игровой цикл. Благодаря этому циклу программа в окне Pygame будет продолжать выполняться до тех пор, пока пользователь не решит выйти из нее. Цикл `for` в строке ❻ — это тот участок кода, в котором мы обрабатываем все интерактивные события, создаваемые пользователем при работе с нашей программой. В этом примере единственное событие, наступление которого мы проверяем, — это нажатие пользователем красной кнопки X, которое приводит к закрытию окна и выходу из программы ❼. Если событие произошло, значение переменной `keep_going` устанавливается в `False` — и игровой цикл завершается.



В строке ❽ мы рисуем зеленый круг с радиусом `50` в окне `screen` в точке `(100, 100)`: 100 пикселей вправо и 100 пикселей вниз от верхнего левого

угла окна (см. раздел «Что нового в Pygame» на с. 207 для получения более подробной информации о том, чем отличается система координат Pygame от Turtle). Для отрисовки таких фигур, как круг, прямоугольник и отрезок, мы используем модуль `pygame.draw` библиотеки Pygame. Мы передаем функции `pygame.draw.circle()` четыре аргумента: поверхность, на которой мы хотим отрисовать круг (`screen`), цвет круга (`GREEN`), координаты центральной точки, а также радиус. Функция `update()` в строке ❾ говорит Pygame обновить экран и отобразить изменения.

Наконец, когда пользователь выходит из игрового цикла, команда `pygame.quit()` в строке ❿ очищает модуль `pygame` (то есть отменяет все настройки, сделанные, начиная со строки ❶) и закрывает окно `screen` так, чтобы программа могла нормально завершиться.

При запуске программы *ShowDot.py* вы должны увидеть на экране изображение как на рис. 8.2. Уделите некоторое время игре с этой программой: создайте другой триплет цветов RGB, нарисуйте точку в другом участке экрана или нарисуйте еще одну точку. Так вы сможете увидеть мощность и легкость рисования графики с помощью библиотеки Pygame, более того, вы сможете получить удовольствие.

Эта первая программа содержит базу, над которой мы будем надстраивать код для создания более сложной графики, анимаций и со временем игр.

Что нового в Pygame

Прежде чем мы погрузимся еще глубже в восхитительный мир Pygame, нам следует уделить внимание некоторым важнейшим отличиям Pygame от нашего старого доброго друга — графики Turtle.

- Как показано на рис. 8.3, у нас новая система координат. В графике Turtle центр координат совпадал с центром экрана, координата *y* увеличивалась по мере нашего перемещения к верхней кромке экрана. В Pygame используется более часто встречающаяся координатная система, ориентированная на окно (мы можем видеть ее во многих других языках программирования GUI, в том числе Java, C++ и так далее). *Верхний левый угол* окна Pygame является точкой начала координат (`0`, `0`). Координаты по оси *x* также продолжают увеличиваться по мере перемещения вправо (однако отрицательных координат по оси *x* в этой системе нет, так как они находились бы слева за пределами экрана). Координаты по оси *y* увеличиваются при перемещении

вниз (отрицательные координаты по оси *y* находились бы сверху за пределами экрана).

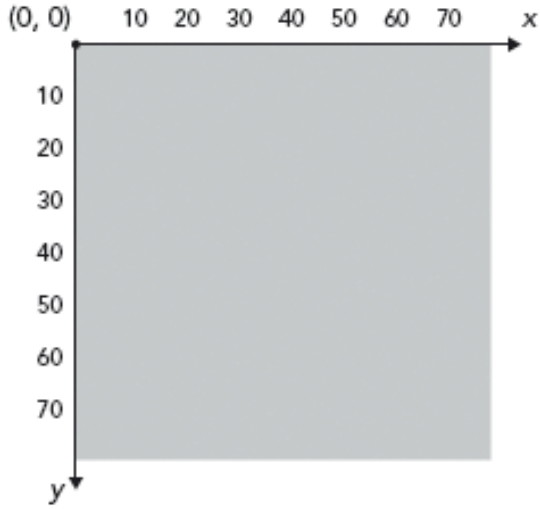


Рис. 8.3. *В Pygame используется система координат, ориентированная на окно*

- В Pygame игровой цикл используется всегда. В программах, которые мы создавали ранее, мы использовали такой цикл, только если хотели продолжать играть или возвращаться и делать что-то снова. Однако Pygame требуется игровой цикл, который обновлял бы экран и обрабатывал события (даже если единственное событие, которое обрабатывает программа, — это закрытие окна).
- В Pygame мы обрабатываем события, вызывая функцию `pygame.event.get()` для вызова списка событий, которые сгенерировал пользователь. Этими событиями могут быть щелчки мышью, нажатия клавиш или события окна, например закрытие окна пользователем. Для обработки всего содержимого списка событий `pygame.event.get()` используется цикл `for`. В программах Turtle для обработки событий мы использовали функции обратного вызова. В Pygame можно создавать функции и вызывать их в коде обработчика событий, но мы также можем обработать события просто с помощью выражений `if` для тех событий, которые нужно прослушивать.

Эти отличия делают Pygame новым способом решения задач, а это — именно то, что мы постоянно ищем! Чем больше инструментов у нас под рукой, тем больше задач мы можем решить.

Части игры

В этом разделе мы изменим нашу программу *ShowDot.py* так, чтобы она отображала изображение-смайлик вместо зеленого круга, как показано на рис. 8.4.

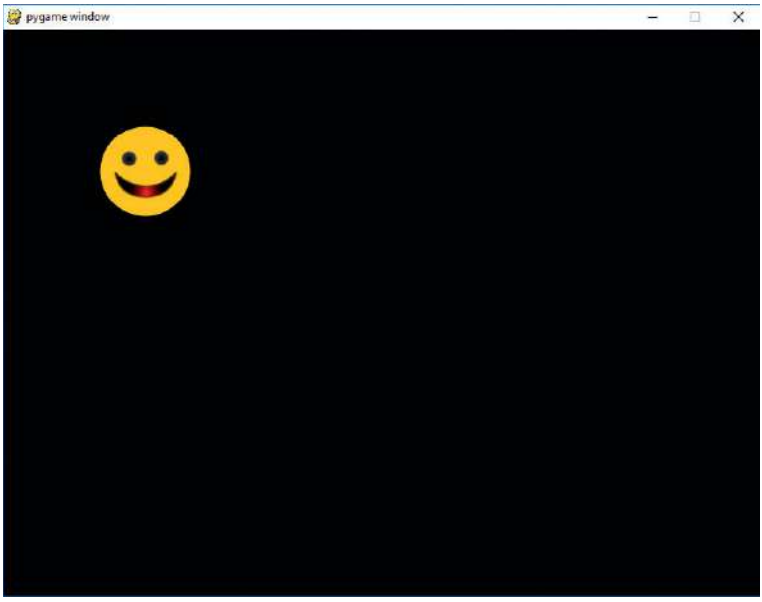


Рис. 8.4. *Программа ShowPic.py рисует на экране изображение CrazySmile.bmp*

По мере создания программы *ShowPic.py* мы изучим три основные части любой игры или анимации Pygame. Во-первых, в программе есть часть с настройками, где мы импортируем необходимые модули, создаем экран и инициализируем некоторые важные переменные. Затем идет игровой цикл, который обрабатывает события, рисует графику и обновляет дисплей. Как правило, игровой цикл — это цикл `while`, который продолжает выполняться до тех пор, пока пользователь не выйдет из игры. Наконец, нам необходимо каким-то образом завершить программу, когда пользователь выходит из игры.

Настройка

В первую очередь скачайте и сохраните изображение смайлика в той же папке, куда вы сохраняете программы Python. Перейдите на сайт **https://eksmo.ru/files/Python_deti.zip**, найдите страницу загрузки исходных кодов, скачайте и сохраните изображение *CrazySmile.bmp* в той же

папке, в которую вы сохраняете программы *.py*. Вообще не имеет значения, где именно вы храните свои файлы *.py*, просто удостоверьтесь, что вы сохранили файл изображения BMP (сокращение от английского слова *bitmap — битовая карта*, часто используемый формат файла) в ту же самую папку.

Далее давайте займемся настройками.

```
import pygame #Настройка
pygame.init()
screen = pygame.display.set_mode([800,600])
keep_going = True
❶ pic = pygame.image.load("CrazySmile.bmp")
```

Как всегда, мы сначала импортируем модуль `pygame`, а затем инициализируем его с помощью функции `pygame.init()`. Далее мы настраиваем переменную `screen`, чтобы она представляла новое окно Pygame размером 800×600 пикселей. Мы создаем наш логический флаг `keep_going` для управления игровым циклом и устанавливаем его равным `True`. Наконец, мы делаем что-то новое в строке ❶: мы используем функцию `pygame.image.load()`, позволяющую загрузить изображение из файла. Для файла изображения мы создаем переменную и загружаем в нее файл *CrazySmile.bmp*, к которому будем по ходу программы обращаться по имени переменной — `pic`.

Создание игрового цикла

К этому моменту мы еще ничего не нарисовали, но уже настроили библиотеку Pygame и загрузили изображение. Игровой цикл — это тот участок кода, в котором мы отобразим на экране графический смайлик. В этом участке кода мы также будем обрабатывать события, генерируемые пользователем. Давайте начнем с обработки одного важного события: пользователь решает выйти из игры:

```
while keep_going:     #Игровой цикл
    for event in pygame.event.get():
❶           if event.type == pygame.QUIT:
                  keep_going = False
```

Наш игровой цикл будет продолжать выполняться до тех пор, пока значение переменной `keep_going` равняется `True`. Внутри цикла мы сразу же

проверяем наступление событий пользователя. В более сложных играх пользователь может генерировать несколько событий одновременно, как, например, нажатие и удерживание клавиши со стрелкой на клавиатуре при одновременном перемещении мыши влево или прокручивании колесика мыши.

В этой простой программе единственное событие, наступление которого мы прослушиваем, — это нажатие на кнопку закрытия окна для выхода из программы. Мы проверяем наступление этого события в строке ❶. Если пользователь сгенерировал событие `pygame.QUIT`, попытавшись закрыть окно, нужно сообщить нашему игровому циклу совершить выход. Мы делаем это, устанавливая значение переменной `keep_going` равным `False`.

Нам все еще нужно нарисовать изображение и обновить экран, чтобы проверить, появляется ли графика на экране. Поэтому в завершение цикла мы добавим следующие две строки:

```
screen.blit(pic, (100,100))
pygame.display.update()
```

Метод `blit()` рисует на экране изображение `pic`, это то самое изображение, которое мы загрузили с диска (смайлик), на поверхности дисплея `screen`. Мы будем использовать функцию `blit()`, когда нам потребуется скопировать пиксели с одной поверхности (как, например, изображение, загруженное с диска) на другую (например, окно для рисования). В данном случае нам необходимо воспользоваться методом `blit()`, так как функция `pygame.image.load()` работает иначе, чем функция `pygame.draw.circle()`, использованная ранее для отрисовки точки. Все функции `pygame.draw` принимают в качестве аргумента поверхности, таким образом, передав функции `pygame.draw.circle()` поверхность `screen`, мы смогли нарисовать в окне дисплея с помощью функции `pygame.draw.circle()`. Однако функция `pygame.image.load()` не принимает поверхность в качестве аргумента, вместо этого данная функция создает новую, отдельную поверхность для рисунка.

Изображение не появится на исходной поверхности экрана для рисования, если вы не воспользуетесь функцией `blit()`. В этом случае мы сообщили методу

`blit()`, что мы хотим нарисовать изображение `pic` в точке `(100,100)`, то есть, 100 пикселей вправо и 100 пикселей вниз от верхнего левого угла экрана (напомним, в системе координат Pygame точка начала координат находится в верхнем левом углу экрана, см. рис. 8.3 на с. 208).

Последняя строка нашего игрового цикла — это вызов функции `pygame.display.update()`. Эта команда сообщает Pygame показать окно для рисования со всеми изменениями, внесенными при данном проходе цикла. В том числе и со смайликом. При выполнении функции `update()` окно будет обновлено и отобразит все изменения, внесенные в поверхность `screen` при данном проходе цикла.

К настоящему моменту мы уже позаботились о создании кода настройки, а также создали игровой цикл с обработчиком событий, который прослушивает нажатие пользователем кнопки закрытия окна. Если пользователь нажимает кнопку закрытия окна, программа обновляет окно и выходит из цикла. Далее мы позаботимся о завершении программы.

Выход из программы

Последний раздел нашего кода осуществит выход из программы, когда пользователь решит выйти из игрового цикла:

```
pygame.quit()   #Выход
```

Если вы упустите эту строку в программе, окно будет оставаться открытым, даже если пользователь попытается закрыть его. Вызов функции `pygame.quit()` закроет окно и освободит память, которая использовалась для хранения нашего изображения `pic`.

Соберем все вместе

Соберите всю программу вместе — и вы увидите наш файл с изображением *CrazySmile.bmp*, конечно, если вы сохранили этот файл в той же папке, что и программу *ShowPic.py*. Далее приведен полный листинг программного кода.

ShowPic.py

```
import pygame # Настройка
pygame.init()
screen = pygame.display.set_mode([800,600])
```

```
keep_going = True
pic = pygame.image.load("CrazySmile.bmp")
while keep_going: # Игровой цикл
    for event in pygame.event.get():
         if event.type == pygame.QUIT:
               keep_going = False
    screen.blit(pic, (100,100))
    pygame.display.update()

pygame.quit() # Выход
```

При щелчке мышью по кнопке, закрывающей окно, окно программы, в котором отображается рисунок, должно закрыться.

Этот код содержит все компоненты, на которые мы будем надстраивать другие блоки кода, делая наши программы еще более интерактивными. В оставшейся части этой главы и в главе 9 мы добавим код в наш игровой цикл, чтобы программа могла отвечать на различные события (например, заставляя изображения на экране перемещаться, когда пользователь двигает мышью). Теперь давайте посмотрим, как создавать программу, рисующую анимированный скачущий мяч!

Правильный тайминг: двигайся и прыгай

У нас уже есть все необходимые навыки для создания анимации, или иллюзии движения, с помощью лишь небольшого изменения в программе *ShowPic.py*. Что если вместо того, чтобы выводить изображение смайлика в фиксированной точке экрана при каждом проходе игрового цикла, мы будем незначительно менять местоположение смайлика при каждой смене кадра? Под *кадром* я подразумеваю каждый проход по игровому циклу. Термин кадр пришел от способа создания классической анимации: люди рисуют тысячи отдельных картинок, каждая из которых лишь незначительно отличается от предыдущих. Одна картинка — один кадр. После этого аниматоры накладывают все картинки на полоску пленки и прогоняют ее через проектор. Когда картинки очень быстро сменяют друг друга, создается впечатление, что персонажи на этих картинках движутся.

С помощью компьютера мы также можем воссоздать такой эффект, нарисовав для начала картинку на экране, очистив экран, немного передвинув картинку и отрисовав ее снова. Этот эффект будет несколько похож на то, что изображено на рис. 8.5.

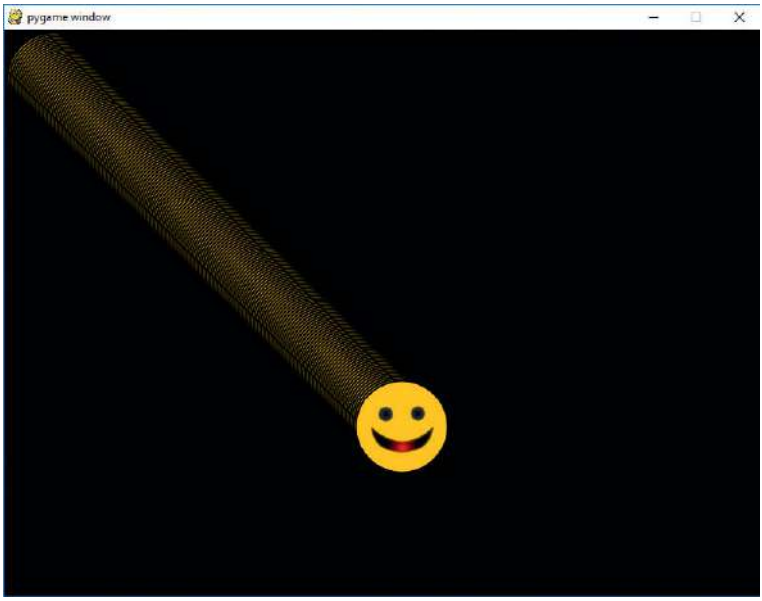


Рис. 8.5. *В этой первой попытке создания анимации смайлик выскочит за пределы экрана*

Если вы когда-либо видели или делали анимацию, созданную на страничках блокнота (когда вы рисуете в уголках страниц блокнота или тетради, а затем быстро пролистываете их, чтобы создать мини-мультфильм), то вы понимаете, что иллюзия движения может быть создана при различных частотах смены кадров. Наша цель — достичь частоты смены кадров 60 fps (кадров в секунду), что достаточно для создания плавной анимации.

Передвигаем смайлик

Мы можем создать простое движение в цикле `while`, изменяя с течением времени точку отрисовки изображения со смайликом. Иными словами, в нашем игровом цикле нам нужно лишь обновлять координаты (x, y) изображения, что позволит нам отрисовывать его в новой точке при каждом прохождении цикла.

В программу *ShowPic.py* мы добавим две переменные: `picx` и `picy`, в которых будем сохранять экранные координаты x и y изображения.

Мы добавим эти переменные в конец участка программного кода с настройками и сохраним новую версию программы как *SmileyMove.py* (полная версия программы приведена на с. 218).

```
  import pygame # Настройка
  pygame.init()
❶ screen = pygame.display.set_mode([600,600])
  keep_going = True
  pic = pygame.image.load("CrazySmile.bmp")
❷ colorkey = pic.get_at((0,0))
❸ pic.set_colorkey(colorkey)
  picx = 0
  picy = 0
```

Строки ❷ и ❸ необязательные и призваны исправить небольшую проблему. Если вам кажется, что изображение CrazySmile.bmp выглядит на экране так, будто у него есть черные углы, вы можете включить эти строки, чтобы быть уверенными в том, что уголки изображения выглядят прозрачными.

Обратите внимание также и на то, что в строке ❶ мы изменили размер окна экрана и сделали его квадратным, 600×600 пикселей. Игровой цикл начнется точно так же, как в программе *ShowPic.py*, но мы добавим в него код, который будет изменять значение переменных `picx` и `picy` при каждом прохождении цикла:

```
while keep_going: # Игровой цикл
    for event in pygame.event.get():
         if event.type == pygame.QUIT:
               keep_going = False

    picx += 1 # Передвинуть картинку
    picy += 1
```

Оператор += прибавляет к переменной в левой части (`picx` и `picy`) некое значение, таким образом, с помощью выражения `+= 1` мы сказали компьютеру, что хотели бы увеличить координаты рисунка *x* и *y* (`picx` и `picy`) на 1 пиксель при каждом прохождении цикла.

Наконец, нам также нужно скопировать рисунок в новое местоположение на экране, обновить дисплей и сообщить программе, что нужно сделать, чтобы выйти:

```
    screen.blit(pic, (picx, picy))
    pygame.display.update()
pygame.quit() # Выход
```

Если вы запустите выполнение этих строк кода, вы увидите, как взлетит наше изображение! Придется смотреть внимательно, так как мы моментально вынесем изображение за пределы экрана.

Посмотрите еще раз на рис. 8.5, чтобы увидеть смайлик до вылета за пределы экрана.

Первая версия может оставить на экране следы из окрашенных пикселей, даже если изображение со смайликом покинуло окно для рисования. Мы можем сделать анимацию более чистой, для этого нам потребуется очищать дисплей между кадрами. Окрашенные линии, которые мы видим на экране позади смайлика, — это верхние левые пиксели изображения. С каждым кадром перемещаясь вниз и перекладывая изображение поверх предыдущего, мы оставляем позади остаточные пиксели предыдущего изображения.

Мы можем исправить это, добавив в цикл отрисовки команду `screen.fill()`. Команда `screen.fill()` принимает в качестве аргумента цвет, поэтому нам нужно сообщить функции, какой цвет нужно использовать для заливки экрана. Давайте также добавим в программу переменную `BLACK` (в имени переменной `BLACK` нужно использовать все заглавные буквы, чтобы показать, что это константа) и установим ее значение равным триплету цветов RGB (`0, 0, 0`). В этом случае перед отрисовкой каждой новой смещенной копии анимированного изображения перед нами будет эффективно очищенный, заполненный черными пикселями экран.

Добавьте следующую строку в блок настройки сразу же после выражения `picy = 0`, чтобы создать черный цвет заливки:

```
BLACK = (0, 0, 0)
```

Добавьте эту строку сразу перед командой `screen.blit()`, рисующей на экране изображение `pic`:

```
screen.fill(BLACK)
```

Даже после внесения этих изменений наш смайлик по-прежнему очень быстро пропадает с экрана, но в этот раз на экране хотя бы не остаются следы от движущегося изображения. Залив экран черными пикселями, мы создали эффект «стирания» старого изображения с экрана при каждой смене кадра перед отрисовкой нового изображения в новой точке. Это позволяет создать иллюзию более плавной анимации. Однако на достаточно мощном компьютере наше изображение со смайликом улетает за пределы экрана слишком быстро. Чтобы исправить это, нам нужен новый инструмент: таймер или счетчик, который позволит сохранять стабильное и предсказуемое количество кадров в секунду.

Анимация смайлика с помощью класса `clock`

Чтобы заставить наше приложение *SmileyMove.py* вести себя как анимация в играх или фильмах, необходимо ограничить количество кадров, которое наша программа может отрисовывать в одну секунду. На данный момент мы перемещаем изображение со смайликом только на 1 пиксель вниз и на 1 пиксель вправо при каждом прохождении игрового цикла. Однако наш компьютер способен рисовать эту простую сцену очень быстро, производя сотни кадров в секунду, из-за чего смайлик моментально улетает с экрана.

Плавная анимация возможна при частоте 30 или 60 кадров анимации в секунду: нам не нужно, чтобы ежесекундно перед нашими глазами проносились сотни кадров.

В Pygame реализован инструмент, который поможет нам управлять скоростью анимации: класс счетчика `Clock`. *Класс* — это нечто наподобие шаблона, который может использоваться для создания *объектов* определенного типа, наделенных функциями и значениями, помогающими таким объектам вести себя определенным образом. Вы можете воспринимать класс как формочку для выпечки печенья, а объекты — как собственно печенье. Когда мы хотим испечь печенье определенной формы, мы создаем формочку, которую снова используем всякий раз, когда захотим испечь печенье этой формы. Как функции помогают нам запаковать повторно используемый код, так классы позволяют нам упаковать данные и функции

в повторно используемый шаблон, с помощью которого мы можем создавать объекты будущих программ.

С помощью следующей строки мы можем добавить объект класса `Clock` в программу:

```
timer = pygame.time.Clock()
```

Эта команда создает переменную `timer`, связанную с объектом `Clock`. Она позволит нам вставлять паузу после каждого прохода по игровому циклу и ожидать время, нужное для отрисовки строго определенного количества кадров в секунду.

Добавление следующей строки в игровой цикл будет сохранять частоту кадров на уровне 60 fps, инструктируя счетчик `Clock` с именем `timer` «тикать» 60 раз за секунду:

```
timer.tick(60)
```

В следующем листинге представлена полностью собранная программа *SmileyMove.py*. Этот код дает нам плавную и стабильную анимацию изображения со смайликом, медленно скользящим к нижнему правому углу экрана и уходящим за пределы видимой области.

SmileyMove.py

```
import pygame # Настройка
pygame.init()
screen = pygame.display.set_mode([600,600])
keep_going = True
pic = pygame.image.load("CrazySmile.bmp")
colorkey = pic.get_at((0,0))
pic.set_colorkey(colorkey)
picx = 0
picy = 0
BLACK = (0,0,0)
timer = pygame.time.Clock() # Таймер для анимации

while keep_going: # Игровой цикл
    for event in pygame.event.get():
         if event.type == pygame.QUIT:
               keep_going = False
```

```
        picx += 1 # Передвинуть картинку
        picy += 1

        screen.fill(BLACK) # Очистить экран
        screen.blit(pic, (picx, picy))
        pygame.display.update()
        timer.tick(60) # Ограничить до 60 кадров в секунду

    pygame.quit() # Выход
```

Оставшаяся проблема заключается в том, что смайлик уходит за пределы экрана в течение всего нескольких секунд, а это не слишком интересно. Давайте изменим нашу программу так, чтобы смайлик оставался на экране, но отскакивал от угла к углу.

Отскок смайлика от стены

Мы добавили движение от одного кадра к другому, перемещая рисуемое изображение при каждом прохождении по игровому циклу. Мы увидели, что можно регулировать скорость анимации путем добавления объекта `Clock` и информирования его о том, сколько раз в минуту вызывать функцию `tick()`. В этом разделе мы увидим, как сохранить смайлик на экране. Эффект будет похож на то, что изображено на рис. 8.6, где кажется, что смайлик отскакивает вперед-назад между двумя углами окна.

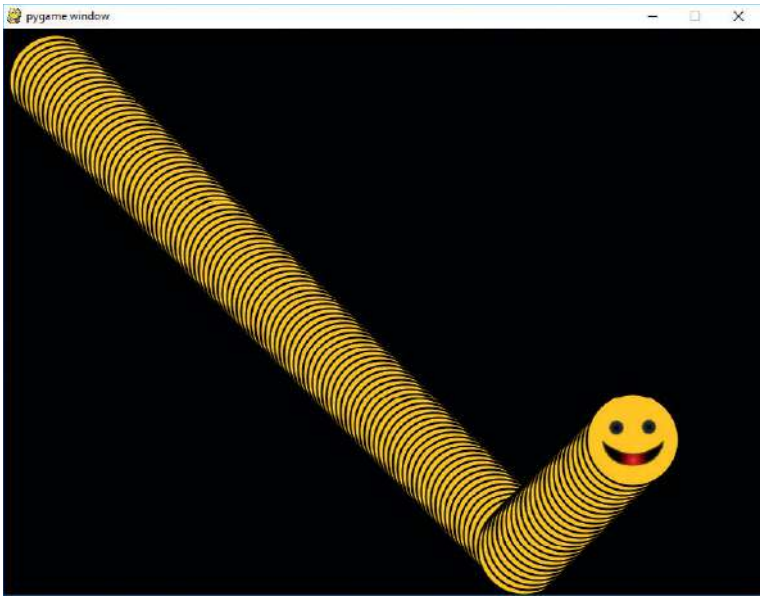


Рис. 8.6. *Наша цель состоит в том, чтобы смайлик «отскакивал» от углов экрана*

Причина, по которой наше изображение вылетало за пределы экрана, — в том, что мы не установили *рамки*, или границы, анимации. Все, что мы рисуем на экране, *виртуально*, то есть не существует в реальном мире, следовательно, нарисованные объекты не могут ударяться друг о друга. Если мы хотим, чтобы виртуальные объекты взаимодействовали, мы должны сами создать такие взаимодействия, используя программную логику.

Удар о стену

Когда я говорю, что нам нужно, чтобы смайлик «отскакивал» от края экрана, я имею в виду, что нам нужно поменять направление движения смайлика, когда он приблизится к краю экрана. Это и будет выглядеть так, словно смайлик отскакивает от твердого края. Чтобы сделать это, необходимо проверить, не достигло ли положение (`picx`, `picy`) смайлика воображаемой границы на краю экрана. Такая программная логика называется *обнаружением столкновений*, так как мы пытаемся *обнаружить*, когда произойдет *столкновение*, когда смайлик «ударится» о край окна для рисования.

Мы знаем, что можем проверить выполнение условий с помощью выражения `if`, то есть можно проверить, касается или ударяется ли наше изображение о правый край экрана, узнав, не является ли переменная `picx` больше некоторого значения.

Давайте выясним, что за значение это может быть. Мы знаем, что ширина нашего экрана 600 пикселей, так как мы создали экран с помощью функции `pygame.display.set_mode([600,600])`. В качестве рамки мы могли бы использовать значение 600, но наш смайлик все равно вышел бы за пределы экрана, так как координатная пара (`picx`, `picy`) соответствует верхнему левому углу изображения со смайликом.

Чтобы найти логическую границу, иными словами, вертикальную линию, которую должна достичь переменная `picx`, чтобы на экране отобразилось, будто смайлик ударился о правый угол окна экрана `screen`, нам нужно знать ширину картинки. Так как мы знаем, что `picx` соответствует верхнему левому углу изображения, а также что координаты увеличиваются вправо, мы просто можем прибавить ширину рисунка к значению `picx`. Когда данная сумма будет равняться 600, это будет означать, что правый край рисунка касается правого края окна.

Узнать ширину изображения можно, например, посмотрев эту информацию в свойствах файла. В Windows щелкните правой кнопкой мыши по файлу *CrazySmile.bmp*, затем из открывшегося контекстного меню выберите пункт **Свойства** (Properties), затем щелкните по вкладке **Подробно** (Details). В macOS щелкните мышью по файлу *CrazySmile.bmp* и выделите его, нажмите клавиши **i+I** (или **⌘+I**, если у вас операционная система MacOS), чтобы открыть окно информации о файле, выберите пункт **Подробнее** (More Info). Вы увидите ширину и высоту изображения, как показано на рис. 8.7.

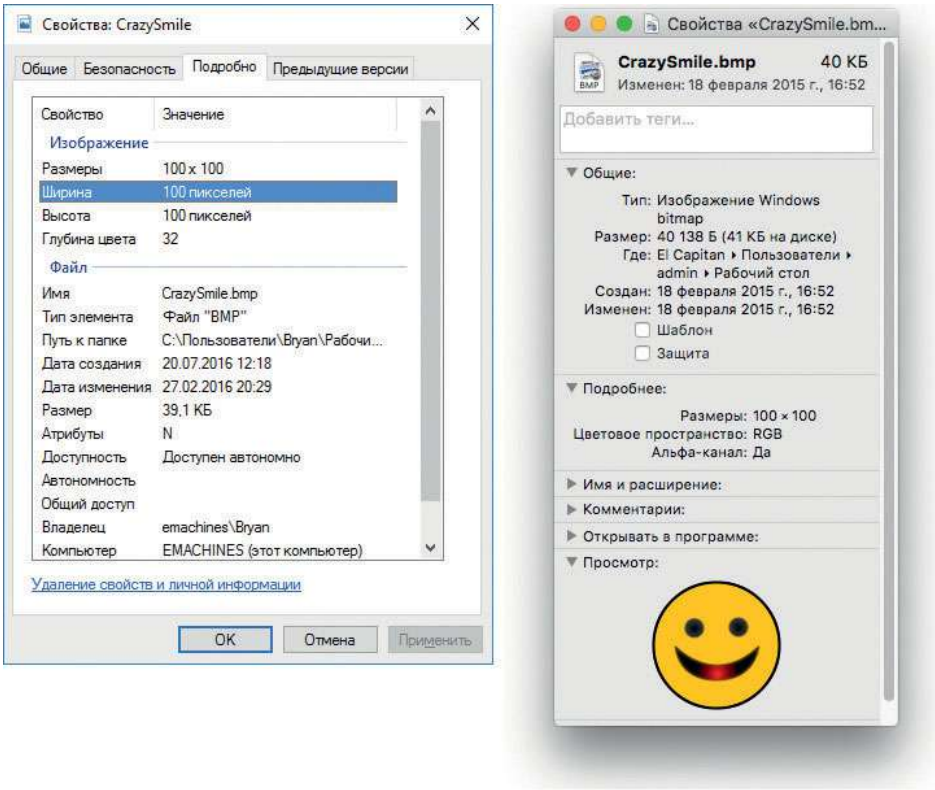


Рис. 8.7. *Чтобы определить виртуальные рамки, от которых будет отскакивать смайлик, нам необходимо знать ширину файла с изображением*

Ширина и высота файла **CrazySmile.bmp** составляет 100 пикселей. Таким образом, если ширина окна экрана `screen` составляет 600 пикселей, а изображению `pic` для полноценного отображения требуется 100 пикселей, то переменная `picx` должна быть левее отметки 500 пикселей по направлению оси *x*. Вышеприведенные измерения показаны на рис. 8.8.

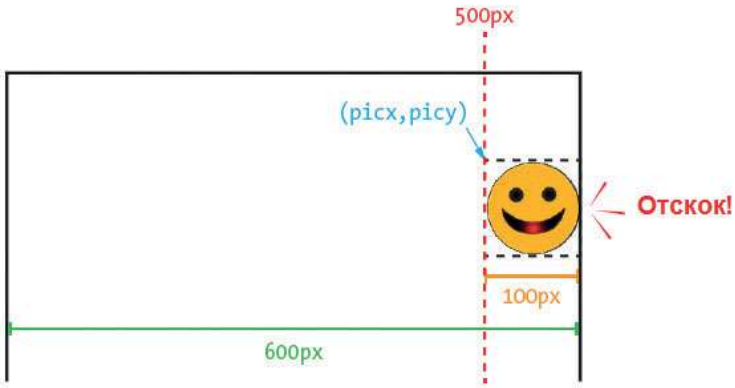
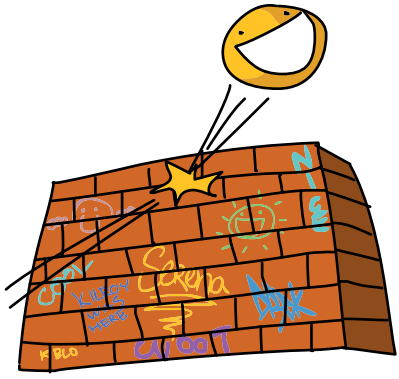


Рис. 8.8. *Вычисление отскока от правого края окна*

Что произойдет, если мы заменим наш файл или захотим иметь возможность обрабатывать изображения разной ширины и высоты? К счастью, в библиотеке Pygame реализована удобная функция в классе `pygame.image`, которую использует наша переменная с изображением `pic`. Функция `pic.get_width()` возвращает ширину в пикселях изображения, сохраненного в переменной `pic` типа `pygame.image`. Вместо жесткого кодирования программы на использование только изображений шириной 100 пикселей мы можем воспользоваться этой функцией. Точно так же функция `pic.get_height()` возвращает высоту в пикселях изображения, сохраненного в переменной `pic` типа `pygame.image`.

Выражение наподобие следующего позволит нам проверить, выходит ли изображение за пределы экрана:

```
if picx + pic.get_width() > 600:
```



Иными словами, если сумма начальной координаты *x* изображения и ширины изображения больше ширины экрана, мы будем точно знать, что изображение вышло за пределы экрана, и сможем изменить направление движения.

Изменение направления

«Отскок» от края экрана означает начало движения в противоположном направлении после удара о край. Направление движения изображения управляется обновлением значений переменных `picx` и `picy`. В нашей старой программе *SmileyMove.py* мы просто добавляли 1 пиксель к значениям переменных `picx` и `picy` при каждом проходе по циклу `while` с помощью следующих строк:

```
picx += 1
picy += 1
```

Однако эти строки перемещали наше изображение каждый раз на один пиксель вправо и вниз, никакого «отскока» и изменения направления не было, так как мы никогда не изменяли число, прибавляемое к переменным `picx` и `picy`. Эти две строки гарантировали нам, что изображение будет перемещаться вниз и вправо со скоростью 1 пиксель за кадр, каждый кадр, даже после того, как смайлик покинет экран.

Вместо этого мы можем заменить константное значение 1 на переменную, которая будет представлять *скорость*, или количество пикселей, на которое изображение должно перемещаться при каждой смене кадра. Скорость — это количество движения за определенный период времени. Например, автомобиль, значительно перемещающийся за небольшой промежуток времени, движется с *высокой скоростью*. Улитка, которая едва перемещается за то же время, движется на *низкой скорости*. Мы можем определить переменную `speed` в участке кода с настройками, чтобы данная переменная представляла количество движения в пикселях, осуществляемое каждый кадр.

```
speed = 5
```

Затем все, что нам останется делать в игровом цикле, — изменять значения переменных `picx` и `picy` на новое значение скорости (вместо константного значения 1) при каждом прохождении цикла:

```
    picx += speed
    picy += speed
```

При частоте смены кадров 60 в минуту один пиксель за кадр в *SmileyMove.py* показалось мне чересчур медленным, поэтому я увеличил скорость до 5, чтобы изображение двигалось быстрее. Но мы все еще не отскакиваем от правого края экрана: наше изображение просто пропадает с экрана быстрее, потому что переменная `speed` не изменяется, когда смайлик ударяется о край экрана.

Мы можем решить эту последнюю проблему, добавив в программу логику обнаружения столкновений, то есть тест, проверяющий, не ударилось ли изображение о воображаемую границу левого или правого краев экрана:

```
    if picx <= 0 or picx + pic.get_width() >= 600:
        speed = -speed
```

В первую очередь мы проверяем пересечение левой и правой границы экрана. Для этого мы смотрим, не пытается ли `picx` отрисовать в точке с отрицательной координатой *x* (за пределами левой границы экрана, где `x < 0`), или не превышает ли сумма выражения `picx + pic.get_width()` ширину экрана 600 пикселей (это значит, что начальная координата *x* изображения + его ширина вышли за правую границу экрана). Если верно хотя бы одно из предположений, мы знаем, что зашли слишком далеко, а значит, должны изменить направление движения.

Обратите внимание на трюк, который мы используем, если один из тестов на пересечение границ возвращает значение Истина (True). Устанавливая `speed = -speed`, мы изменяем *направление* движения в цикле `while`, умножая переменную `speed` на –1, то есть делая ее отрицательной по отношению к самой себе. Можно воспринимать это действие так: если мы будем продолжать выполнять цикл с переменной `speed` равной 5 до тех пор, пока `picx` плюс ширина изображения не ударится о правый

край экрана шириной 600 пикселей (picx + pic.get_width() >= 600), установка speed = -speed изменит значение переменной speed с 5 на -5 (минус пять). Затем при каждом изменении значения переменных picx и picy при следующих итерациях цикла мы будем добавлять -5 к текущему местоположению. Это аналогично *вычитанию* 5 из значений переменных picx и picy, то есть перемещая изображение по экрану вверх и влево. Если все сработает, наш смайлик теперь будет отскакивать от нижнего правого угла экрана и начнет путешествовать *обратно*, к точке (0,0) в верхнем левом углу экрана.

Но это еще не все! Так как выражение if проверяет также и пересечение левой границы экрана (picx <= 0), то в момент, когда смайлик как будто бы ударится о левый край экрана, программа вновь изменит скорость speed на -speed. Так, если скорость равна -5, то программа изменит ее на -(-5), то есть +5. Таким образом, отрицательная скорость speed заставляла наше изображение двигаться влево-вверх на 5 пикселей с каждым кадром. Как только мы ударимся о picx <= 0 на левой границе экрана, выражение speed = -speed вновь изменит значение переменной speed на плюс 5, а изображение со смайликом вновь начнет двигаться *вправо-вниз*, в положительных направлениях координат *x* и *y*.

Соберем все вместе

Попробуйте запустить версию 1.0 данного приложения, *SmileyBounce1.py*, чтобы посмотреть, как смайлик отскакивает от верхнего левого угла окна по направлению к нижнему правому и наоборот, при этом никогда не покидая экран.

SmileyBounce1.py

```
# SmileyBounce1.py
import pygame # Настройка
pygame.init()
screen = pygame.display.set_mode([600,600])
keep_going = True
pic = pygame.image.load("CrazySmile.bmp")
colorkey = pic.get_at((0,0))
pic.set_colorkey(colorkey)
picx = 0
picy = 0
BLACK = (0,0,0)
timer = pygame.time.Clock()
speed = 5
```

```
while keep_going: # Игровой цикл
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            keep_going = False
    picx += speed
    picy += speed

    if picx <= 0 or picx + pic.get_width() >= 600:
        speed = -speed

    screen.fill(BLACK)
    screen.blit(pic, (picx, picy))
    pygame.display.update()
    timer.tick(60)

pygame.quit() # Выход
```

В этой, первой, версии программы вы создали нечто, выглядящее как плавно анимированный смайлик, отскакивающий между двумя углами квадратного окна для рисования. Мы получили этот эффект с высокой точностью потому, что окно имеет форму идеального квадрата 600×600 пикселей. А благодаря тому что мы всегда изменяем значения переменных `picx` и `picy` на одинаковое значение (`speed`), наш смайлик перемещается только по одной диагонали, где $x = y$. Для сохранения такой простой траектории перемещения рисунка нам нужно лишь контролировать пересечение значением переменной `picx` пограничных значений, соответствующих левому и правому краям экрана.

Но что если нам захочется устроить отскок от всех четырех краев экрана (от верхнего, нижнего, левого и правого) в окне, по форме не являющемся идеальным квадратом, а скажем, имеющем пропорции 800×600 пикселей? Нам потребуется добавить еще программной логики, которая проверяла бы прохождение переменной `picy` верхней и нижней границы (верхний и нижний край экрана). Кроме того, нам потребуется по отдельности следить за горизонтальной и вертикальной скоростью. Этим мы и займемся далее.

Отскок смайлика от четырех стен

В программе *SmileyBounce1.py* мы держали в четкой связке движение по горизонтали (вправо-влево) и по вертикали (вверх-вниз). Таким образом, когда бы изображение ни двигалось вправо, оно также двигалось и вниз, а когда изображение двигалось влево, оно также поднималось вверх. Это хорошо срабатывало в случае с квадратным окном, так как ширина и высота такого окна были одинаковыми. Давайте возьмем эту программу

за основу и создадим реалистическую анимацию с отскоком от всех четырех сторон окна для рисования. С помощью инструкции `screen = pygame.display.set_mode([800,600])` мы создадим окно размером 800×600 пикселей, это позволит сделать нашу анимацию более интересной.

Горизонтальная и вертикальная скорость

В первую очередь давайте разделим горизонтальную и вертикальную компоненты скорости. Иными словами, давайте создадим одну переменную, `speedx`, для *горизонтальной* скорости (как быстро изображение перемещается вправо или влево) и еще одну, `speedy`, для вертикальной скорости (как быстро изображение перемещается вниз или вверх). Мы можем достичь этого, изменив строку `speed = 5` в разделе установок программного кода приложения на инициализацию двух переменных (`speedx` и `speedy`) следующим образом:

```
speedx = 5
speedy = 5
```

Затем мы также можем изменить логику обновления местоположения изображения в игровом цикле:

```
    picx += speedx
    picy += speedy
```

Мы изменяем `picx` (горизонтальное положение, или положение по оси *x*) на значение `speedx` (горизонтальная скорость), в то время как переменная `picy` (вертикальное положение, или положение по оси *y*) изменяется на значение переменной значение `speedy` (вертикальная скорость).

Удар о четыре стены

Последнее, о чем нам осталось подумать, — как обнаружить столкновение с границами для всех четырех краев окна (верхний и нижний в дополнение к правому и левому). Сначала давайте изменим установку правой и левой границ, чтобы они соответствовали новым размерам экрана (ширина экрана 800 пикселей), а затем воспользуемся новой горизонтальной скоростью `speedx`:

```
    if picx <= 0 or picx + pic.get_width() >= 800:
        speedx = -speedx
```

Обратите внимание, что установка левой границы остается неизменной: `picx <= 0`, так как 0 по-прежнему является значением левой границы, когда `picx` в левой части экрана. Однако в этот раз установка правой границы изменилась: `picx + pic.get_width() >= 800`, так как теперь ширина экрана 800, но программа по-прежнему начинает отрисовку изображения в точке `picx` и рисует его в полную ширину вправо. Таким образом, когда сумма `picx + pic.get_width()` равняется 800, мы видим, как смайлик касается правого края окна для рисования.

Мы немного изменили действие, запускаемое касанием правой или левой границы. Изначально выражение выглядело как `speed = -speed`, в этот раз мы заменили его выражением `speedx = -speedx`. Теперь у нас есть две компоненты скорости: переменная `speedx` будет контролировать правое и левое направление скорости (отрицательные значения переменной `speedx` будут перемещать смайлик влево, положительные — вправо). Таким образом, когда смайлик ударяется о правый край экрана, мы делаем значение переменной `speedx` отрицательным, чтобы изображение начало возвращаться влево. Когда же изображение ударяется о левый край экрана, мы вновь делаем значение переменной `speedx` положительным, что опять заставит изображение перемещаться вправо.

Давайте поступим аналогичным образом и с переменной `picy`:

```
    if picy <= 0 or picy + pic.get_height() >= 600:
        speedy = -speedy
```

Чтобы проверить, ударился ли смайлик о верхний край экрана, мы используем выражение `picy <= 0`, аналогичное выражению `picx <= 0` для левого края. Для выяснения того, ударилось ли изображение о нижний край экрана, нам необходимо знать высоту экрана для рисования (600 пикселей) и высоту изображения (`pic.get_height()`), нам также нужно проверить, не превышает ли сумма координаты верхней точки изображения `picy` и высоты изображения, `pic.get_height()`, высоту экрана 600 пикселей.

Если значение переменной `picy` выходит за пределы этих границ, нам необходимо изменить направление вертикальной

скорости (speedy = -speedy). Благодаря этой установке анимация выглядит так, будто смайлик отскакивает от нижнего края экрана и направляется вновь вверх либо отскакивает от верхнего края экрана и направляется вниз.

Соберем все вместе

Когда мы соберем всю программу вместе в приложение *SmileyBounce2.py*, то получим убедительную анимацию с мячиком, отскакивающим от всех четырех краев экрана, причем мячик будет отскакивать до тех пор, пока мы не решим выйти из приложения.

SmileyBounce2.py

```
# SmileyBounce2.py
import pygame # Настройка
pygame.init()
screen = pygame.display.set_mode([800,600])
keep_going = True
pic = pygame.image.load("CrazySmile.bmp")
colorkey = pic.get_at((0,0))
pic.set_colorkey(colorkey)
picx = 0
picy = 0
BLACK = (0,0,0)
timer = pygame.time.Clock()
speedx = 5
speedy = 5

while keep_going: # Игровой цикл
    for event in pygame.event.get():
         if event.type == pygame.QUIT:
               keep_going = False
    picx += speedx
    picy += speedy

    if picx <= 0 or picx + pic.get_width() >= 800:
         speedx = -speedx
    if picy <= 0 or picy + pic.get_height() >= 600:
         speedy = -speedy

    screen.fill(BLACK)
    screen.blit(pic, (picx, picy))
    pygame.display.update()
    timer.tick(60)

pygame.quit() # Выход
```

Отскоки выглядят реалистично. Если смайлик идет слева направо по направлению к нижнему краю под углом 45 градусов, он отскочит от края влево также с углом 45 градусов. Вы можете поэкспериментировать с другими значениями переменных `speedx` и `speedy` (например, `3` и `5`, `7` и `4`), чтобы пронаблюдать изменение углов при каждом отскоке.

Также просто для развлечения вы можете закомментировать строку `screen.fill(BLACK)` программы *SmileyBounce2.py*, чтобы увидеть путь, который проходит смайлик по мере отскоков от краев экрана. *Закомментировать* строку — значит превратить ее в комментарий, подставив символ «решетка» в начало строки следующим образом:

```
#screen.fill(BLACK)
```

Это проинструктирует программу игнорировать команду в этой строке. Теперь экран не очищается после рисования каждого смайлика — и вы можете увидеть узор, создаваемый следами, которая оставляет ваша анимация, как показано на рис. 8.9. Так как каждый новый смайлик рисуется поверх предыдущего, результат отрисовки выглядит как крутая ретро-3D-заставка.

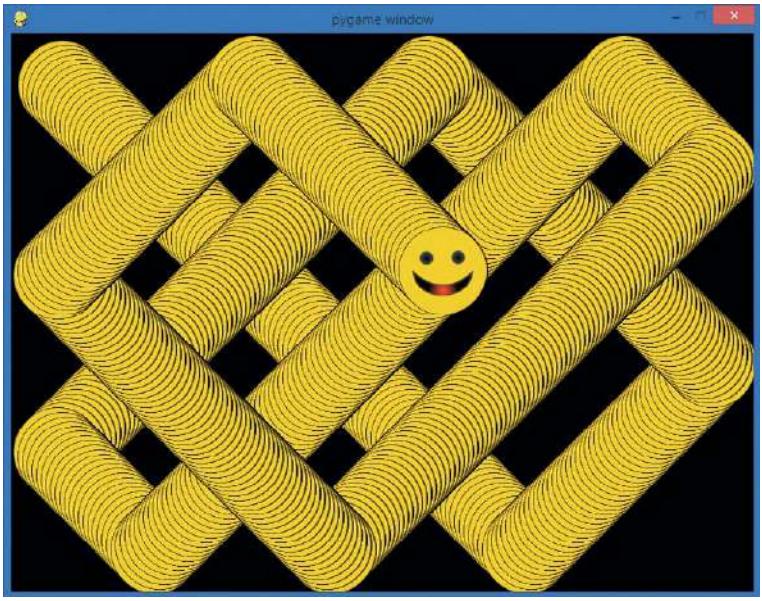


Рис. 8.9. *Если вы закомментируете строку, очищающую экран после каждого кадра, то смайлик будет отставлять следы отскоков, которые создадут крутой узор*

Наша логика обнаружения столкновений позволила нам создать иллюзию того, как твердый смайлик отскакивает от всех четырех краев твердого окна для рисования. Это улучшение изначальной версии программы, которая позволяла смайлику постепенно соскользнуть в небытие. При создании игр, которые позволяют пользователю взаимодействовать с элементами на экране, а также позволяют этим элементам взаимодействовать друг с другом, как, например, тетрис, мы используем методику обнаружения столкновений и проверки границ, аналогичную той, что мы использовали при создании этой программы.

Что вы узнали

В этой главе вы научились создавать иллюзию движения, которую мы называем *анимацией*. Иллюзия движения создается за счет отрисовки изображения в разных участках экрана. Мы увидели, каким образом модуль Pygame может помочь упростить и ускорить программирование игры или анимации, так как в этом модуле реализованы сотни функций, упрощающих решение практически всех задач игрового приложения, от рисования изображения до создания обусловленной таймером анимации, и даже обнаружение столкновений. Мы установили модуль Pygame, чтобы использовать его функции для создания собственных веселых приложений.

Вы узнали структуру игры или приложения, которые мы можем создавать с помощью модуля Pygame, а именно блок настроек, игровой цикл, обрабатывающий события, обновляющий и рисующий графику, а затем обновляющий дисплей, и, наконец, блок выхода.

Мы начали программировать с Pygame с рисования простой зеленой точки в указанном участке экрана, но быстро перешли к отрисовке на экране картинки из файла изображения на диске, хранящегося в той же папке, что и сама программа.

Вы узнали, что система координат модуля Pygame отличается от системы координат библиотеки Turtle. В Pygame точка начала координат `(0, 0)` находится в левом верхнем углу экрана, при этом положительные координаты оси *y* увеличиваются при движении вниз.

Вы узнали, как можно создавать анимацию путем рисования объектов на экране, очистки экрана и повторного рисования тех же самых объектов в незначительно отличающемся местоположении. Мы увидели, что объект `pygame.time.Clock()` может сделать наши анимации более

стабильными, ограничивая число рисований анимации в секунду. Это называется количеством *кадров в секунду*, или *fps*.

Мы создали собственную программу обнаружения столкновений, чтобы проверить, не «ударились» ли объекты о край экрана, а затем добавили логику для визуальной симуляции эффекта обратного отскока объектов путем изменения направления соответствующих переменных скорости (умножив эти переменные на –1).

Программирование крутых приложений из этой главы дало нам навыки, которые позволяют нам делать следующее.

1. Устанавливать и использовать модуль `pygame` в собственных программах на языке Python.
2. Объяснять структуру приложения Pygame, в том числе блоки настройки, игрового цикла и выхода.
3. Создавать игровой цикл, обрабатывающий события, обновляющий и рисующий графику, а затем обновляющий дисплей.
4. Рисовать фигуры на экране с помощью функций `pygame.draw`.
5. Загружать изображения с диска с помощью функции `pygame.image.load()`.
6. Рисовать изображения и объекты на экране с помощью функции `blit()`.
7. Создавать анимации путем повторной отрисовки объектов в разных участках экрана.
8. Делать анимации гладкими, чистыми и предсказуемыми с помощью функции `tick()` таймера `pygame.time.Clock()`, а также ограничивать количество кадров анимации в секунду.
9. Обнаруживать столкновения путем создания логики `if` для проверки условий пересечения границ, как, например, в случае с ударом графического объекта о край экрана.
10. Контролировать горизонтальную и вертикальную скорость движущихся на экране объектов путем изменения количества движения по осям x и y от одного кадра к следующему.

Задачи по программированию

Далее приведены три задачи, которые позволят вам расширить знания и навыки, полученные в этой главе. Вы можете скачать примеры решения задач по адресу **https://eksmo.ru/files/Python_deti.zip**.

#1. Точка, изменяющая цвет

Давайте более подробно изучим триплеты RGB. В этой главе мы уже поработали с несколькими цветами RGB. Если вы помните, то зеленый цвет задавался как `(0, 255, 0)`, черный — `(0, 0, 0)` и так далее. На сайте **colorschemer.com/online/** введите различные значения красного, зеленого и голубого цветов в диапазоне от 0 до 255, чтобы увидеть, какие оттенки могут отображать пиксели вашего экрана с разным количеством красного, зеленого и голубого цветов. Начните с выбора собственного триплета цвета для программы *ShowDot.py*. Затем измените программу так, чтобы она отрисовывала точку большего или меньшего размера в разных участках экрана. Наконец, попробуйте создать случайный триплет RGB с помощью функции `random.randint(0,255)` для каждого из трех компонентов цвета (не забудьте импортировать модуль `random` в верхней части программы), с тем чтобы точка изменяла свой цвет каждый раз при отрисовке на экране. Эффектом таких изменений будет точка, изменяющая цвет. Назовите свое детище *DiscoDot.py*.

#2. 100 случайных точек

В качестве второго задания давайте заменим единственную точку на 100 точек случайных цветов и размеров, рисуемых в случайных участках экрана. Чтобы сделать это, давайте создадим три массива, способных хранить по 100 значений для цветов, координат и размеров:

```
# массивы цветов, координат, размеров 100 случайных точек
colors = [0]*100
locations = [0]*100
sizes = [0]*100
```

Затем заполним эти три массива случайными триплетами цвета, парами координат и значениями размер/радиус для 100 случайных точек:

```
import random
# сохранить случайные значения в массивах colors,
# locations, sizes
for n in range(100):
    colors[n] = (random.randint(0,255),
                 random.randint(0,255),
                 random.randint(0,255))
    locations[n] = (random.randint(0,800),
                    random.randint(0,600))
    sizes[n] = random.randint(10, 100)
```

Наконец, вместо отрисовки одной точки в цикле `while` добавим цикл `for` для отрисовки 100 случайных точек с использованием массивов `colors`, `locations` и `sizes`:

```
    for n in range(100):
         pygame.draw.circle(screen, colors[n],
                            locations[n], sizes[n])
```

Назовите свое создание *RandomDots.py*. Результат выполнения приложения должен выглядеть примерно так, как показано на рис. 8.10.

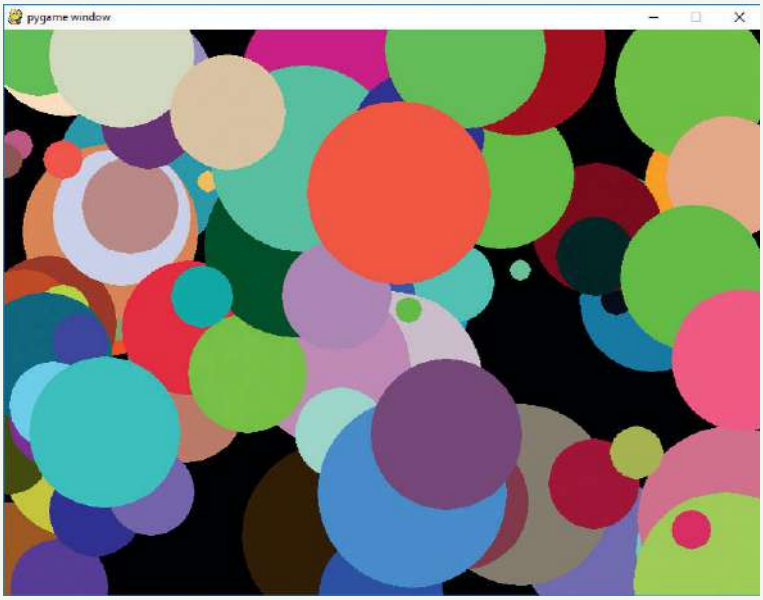


Рис. 8.10. *Улучшенная версия программы, рисующей точку, RandomDots.py, дает нам 100 точек случайного цвета и размера в случайных участках экрана*

#3. Дождь из точек

Наконец, давайте продвинем программу *RandomDots.py* еще на один шаг вперед, запрограммировав «дождь» из точек, которые сливались бы, как капли дождя, с правого нижнего угла и повторно появлялись в левом верхнем углу. В этой главе вы узнали, что мы можем создавать анимацию, изменяя с течением времени местоположение объекта. Местоположение каждой из точек записано у нас в массиве `locations`, таким образом, изменяя значения *x* и *y* координат, мы можем анимировать точки. Измените цикл `for` из программы *RandomDots.py* так, чтобы он вычислял новые координаты *x* и *y* каждой точки, основываясь на предыдущем значении, например, следующим образом:

```
for n in range(100):
     pygame.draw.circle(screen, colors[n], ↵
       locations[n], sizes[n])
new_x = locations[n][0] + 1
new_y = locations[n][1] + 1
locations[n] = (new_x, new_y)
```

Это изменение вычисляет новые координаты *x* и *y* (переменные `new_x` и `new_y`) каждой точки при каждом проходе по игровому циклу, однако это позволяет точкам спадать с нижнего правого угла экрана. Давайте исправим этот недостаток, добавив проверку, не находятся ли координаты каждой новой точки (переменные `new_x` и `new_y`) за пределами правой и нижней границы экрана. Если это так, вернем точку обратно в левую часть экрана перед сохранением новой координаты:

```
     if new_x > 800:
           new_x -= 800
     if new_y > 600:
           new_y -= 600
```

Комбинированный эффект этих изменений будет заключаться в стабильном потоке случайных точек, «идущих дождем» по направлению к нижней правой части окна, сходящих с экрана и появляющихся вновь у верхнего или левого угла экрана. На рис. 8.11

изображена последовательность из четырех кадров. На трех фотографиях вы можете проследить движение групп точек по мере их перемещения вниз и вправо.

Сохраните новое приложение в файл с именем *RainingDots.py*.

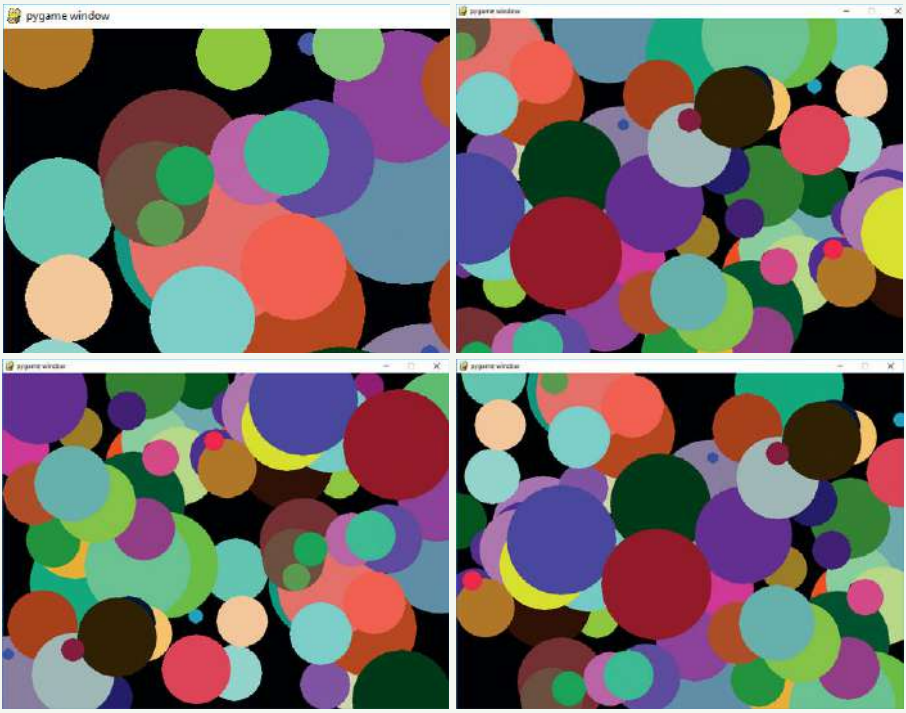


Рис. 8.11. Четыре кадра, показывающие 100 случайных точек, перемещающихся по экрану вправо-вниз

Глава 9

ВЗАИМОДЕЙСТВИЕ С ПОЛЬЗОВАТЕЛЕМ: ПОДКЛЮЧАЕМСЯ К ИГРЕ

В главе 8 мы использовали несколько функций библиотеки Pygame, чтобы рисовать на экране фигуры и изображения. Мы также научились создавать анимацию путем отрисовки фигур в разных участках экрана в разные моменты времени. К сожалению, мы не могли *взаимодействовать* с анимированными объектами, как, например, мы взаимодействуем с ними в игре. Запустив игру, мы ожидаем, что сможем щелкать по объектам, перетаскивать их, перемещать, бить по ним или выводить на экран, чтобы иметь возможность воздействовать на элементы игры.

Интерактивные программы дают нам ощущение контроля над приложением или игрой, так как мы можем взаимодействовать с персонажем игры или другим объектом программы. Это именно то, чему вы научитесь в этой главе: мы воспользуемся возможностями библиотеки Pygame обрабатывать взаимодействие с пользователем, происходящее с помощью компьютерной мыши. Это позволит сделать наши программы более интерактивными и интересными для пользователя.

Добавление интерактивности: щелкни и перетащи

Давайте добавим немного интерактивности, то есть взаимодействия с пользователем, создав две программы, которые позволят им рисовать на экране. В первую очередь мы расширим нашу базу знаний библиотеки

Pygame, научимся обрабатывать такие события, как щелчки мышью, и позволим пользователю рисовать точки на экране. Затем мы добавим программную логику, обрабатывающую по отдельности нажатие и отпуск кнопки мыши, а также дадим возможность пользователю перетаскивать мышь с зажатой кнопкой, рисуя при этом на экране, по аналогии с программой Paint.

Щелчки для отрисовки точек

Мы создадим программу *ClickDots.py*, используя те же самые шаги, что и при создании программы *ShowPic.py* (с. 212), выделив в коде блоки настройки, игрового цикла и выхода. Обратите особое внимание на ту часть кода игрового цикла, в которой обрабатываются события, так как именно в этот участок кода мы добавим выражение `if`, которое будет обрабатывать щелчки мышью.

Настройка

Далее приведено несколько первых строк блока настройки. Создайте новый файл и сохраните его под именем *ClickDots.py* (окончательная версия программы показана на рис. 210).

```
import pygame # Настройка
pygame.init()
screen = pygame.display.set_mode([800,600])
pygame.display.set_caption("Click and Draw")
```

Блок настройки начинается, как обычно, с команд `import pygame` и `pygame.init()`, затем мы создаем объект `screen` в качестве дисплея окна для рисования. Однако в этот раз, с помощью команды `pygame.display.set_caption()`, мы добавили окну заголовок, или *надпись*. Заголовок окна позволит пользователю понять, какую программу он запустил. Аргумент, который мы передаем функции `set_caption()`, — это строка текста, которая появится в строке заголовка окна, как показано на рис. 9.1*.

* На некоторых компьютерах заголовок, написанный русскими буквами, может отображаться некорректно, поэтому мы рекомендуем оставить англоязычный заголовок.

Рис. 9.1. *Строка заголовка вверху окна программы ClickDots.py говорит пользователю «Щелкай и рисуй»*

В оставшейся части блока настройки создается переменная игрового цикла, `keep_going`, устанавливается константа цвета (в этой программе мы будем рисовать красным цветом), а также создается радиус отрисовываемых точек.

```
keep_going = True
RED = (255,0,0) # триплет RGB для КРАСНОГО цвета
radius = 15
```

Теперь давайте перейдем к игровому циклу.

Игровой цикл: обработка щелчков мышью

В нашем игровом цикле мы должны сообщить программе, когда осуществить выход, а также — каким образом обрабатывать нажатия на кнопку мыши:

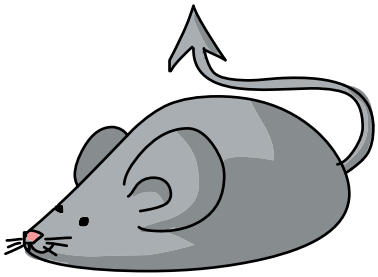
```
while keep_going: # Игровой цикл
    for event in pygame.event.get(): # Обработка событий
❶       if event.type == pygame.QUIT:
            keep_going = False
```

```
❷           if event.type == pygame.MOUSEBUTTONDOWN:
❸                 spot = event.pos
❹                 pygame.draw.circle(screen, RED, spot, radius)
```

В строке ❶ мы обрабатываем событие pygame.QUIT, устанавливая значение переменной цикла keep_going в False.

Второе выражение if, в строке ❷, обрабатывает событие нового типа: pygame.MOUSEBUTTONDOWN, событие, наступление которого говорит нам, что пользователь нажал одну из кнопок мыши. Когда бы пользователь ни нажал кнопку мыши, данное событие появится в списке событий, получаемых программой от функции pygame.event.get(). Мы, в свою очередь, можем использовать выражение if как для проверки наступления данного события, так и для информирования программы о том, какие действия необходимо выполнить, если событие все-таки произошло. В строке ❸ мы создаем переменную spot, которая будет хранить *x* и *y* координаты положения указателя мыши. С помощью event.pos мы можем получить координаты точки, в которой произошло событие щелчка мышью, при этом event — это текущее событие цикла for. Наше выражение if только что подтвердило, что данное конкретное событие event есть событие типа pygame.MOUSEBUTTONDOWN. Для событий мыши реализован атрибут pos (в данном случае — event.pos), хранящий пару координат (*x*, *y*), говорящую нам, где именно произошло данное событие мыши.

Узнав точку экрана, в которой пользователь осуществил щелчок мышью, в строке ❹ мы говорим программе нарисовать закрашенный круг на поверхности screen, данный круг должен быть красного цвета (RED), в соответствии с настройками, находиться в точке spot и иметь radius 15, который мы также задали в блоке настройки.



Соберем все вместе

Единственное, что нам осталось сделать, — обновить дисплей и сказать программе, что делать, когда настанет время выходить. Далее приведена полная версия программы *ClickDots.py*.

ClickDots.py

```
import pygame # Настройка
pygame.init()
screen = pygame.display.set_mode([800,600])
pygame.display.set_caption("Click and Draw")
keep_going = True
RED = (255,0,0) # Триплет RGB для КРАСНОГО цвета
radius = 15

while keep_going: # Игровой цикл
    for event in pygame.event.get(): # Обработка событий
        if event.type == pygame.QUIT:
            keep_going = False
        if event.type == pygame.MOUSEBUTTONDOWN:
            spot = event.pos
            pygame.draw.circle(screen, RED, spot, radius)
        pygame.display.update() # Обновить дисплей

pygame.quit() # Выход
```

Эта программа короткая, но она позволяет пользователям рисовать картинки по одной точке, как было показано на рис. 9.1. Если мы хотим рисовать непрерывно по мере перетаскивания мыши с зажатой кнопкой, то нам нужно научиться обрабатывать еще одно событие мыши `pygame.MOUSEBUTTONUP`. Давайте попробуем.

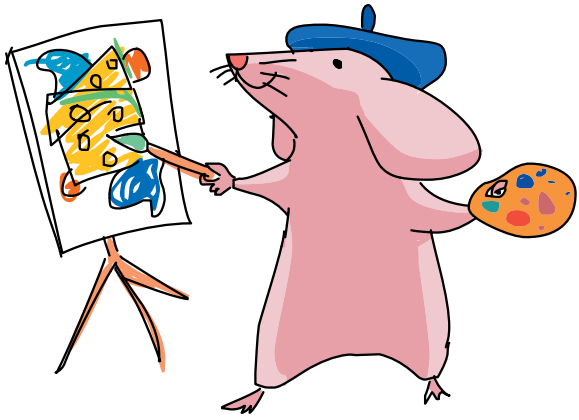
Перетаскивание для рисования

Давайте создадим более естественную программу для рисования, *DragDots.py*, которая будет позволять пользователям щелкать мышью и перетаскивать указатель для рисования плавных непрерывных линий, как при использовании кисточки. Мы получим плавное интерактивное приложение для рисования, как показано на рис. 9.2.

Чтобы создать этот эффект, необходимо изменить логику работы программы. В программе *ClickDots.py* мы обрабатывали события `MOUSEBUTTONDOWN`, просто рисуя круг в той точке, где произошло событие кнопки мыши. Для непрерывного рисования нам нужно распознавать два события: `MOUSEBUTTONDOWN` и `MOUSEBUTTONUP`. Иными словами, нам нужно разделить щелчки кнопкой мыши на *нажатие* и *отпуск* с тем, чтобы знать, когда пользователь *перетаскивает* мышь (с зажатой кнопкой), а не просто перемещает ее с отпущенной кнопкой.

Рис. 9.2. *Наше приложение для забавного рисования DragDots.py!*

Одним из способов выполнения этого задания является использование еще одной логической (булевой) переменной-флага. Мы можем устанавливать логическую переменную `mousedown` в `True` всякий раз, когда пользователь нажимает кнопку мыши, и устанавливать эту переменную в `False`, когда пользователь отпускает кнопку мыши. В нашем игровом цикле, если кнопка мыши нажата (то есть когда переменная `mousedown` равна `True`), мы можем получить координаты указателя мыши, чтобы нарисовать точку в этом участке экрана. Если программа достаточно быстра, то процесс рисования должен быть столь же плавным, как и в программе Paint.



Настройка

Код блока настройки должен выглядеть примерно следующим образом.

```
import pygame # Настройка
pygame.init()
screen = pygame.display.set_mode([800,600])
❶ pygame.display.set_caption("Click and drag to draw")
keep_going = True
❷ YELLOW = (255,255,0) # Триплет RGB для ЖЕЛТОГО цвета
radius = 15
❸ mousedown = False
```

Блок настройки нашего приложения выглядит так же, как соответствующий блок приложения *ClickDots.py*, с той разницей, что в строке ❶ мы задаем отличающуюся надпись, цвет рисования желтый (`YELLOW`), также отличается и последняя строка ❸. Логическая переменная `mousedown` будет для нас флагом, сигнализирующим программе, что пользователь опустил, то есть нажал, кнопку мыши.

Далее мы добавим в наш игровой цикл обработчики событий. Эти обработчики событий установят значение переменной `mousedown` в `True`, если пользователь нажал и не отпускает кнопку мыши, и в `False`, если этого не произошло.

Игровой цикл: обработка нажатия и отпуска кнопки мыши

Игровой цикл программы должен выглядеть следующим образом.

```
while keep_going: # Игровой цикл
    for event in pygame.event.get(): # Обработка событий
        if event.type == pygame.QUIT:
            keep_going = False
❶       if event.type == pygame.MOUSEBUTTONDOWN:
❷           mousedown = True
❸       if event.type == pygame.MOUSEBUTTONUP:
❹           mousedown = False
❺   if mousedown: # Рисование/обновление графики
❻       spot = pygame.mouse.get_pos()
❼       pygame.draw.circle(screen, YELLOW, spot, radius)
❽ pygame.display.update() # Обновить дисплей
```

Игровой цикл начинается точно так же, как и в других программах Pygame, однако в строке ❶ при проверке того, нажал ли пользователь одну из кнопок мыши, вместо начала отрисовки мы устанавливаем значение переменной `mousedown` в `True` ❷. Это будет сигналом нашей программе начать отрисовку.

Следующее выражение `if` в строке ❸ проверяет, не *отпустил* ли пользователь кнопку мыши. Если это произошло, команда в строке ❹ изменит значение переменной `mousedown` обратно на `False`. Это позволит нашему игровому циклу знать о необходимости прекратить рисование, когда кнопка мыши не нажата.

В строке ❺ завершается цикл `for` (как можно видеть из отступов), а игровой цикл продолжается проверкой, нажата ли в данный момент кнопка мыши (то есть верно ли утверждение `mousedown == True`). Если кнопка мыши нажата, значит, пользователь перетаскивает мышь, значит, нам нужно позволить пользователю рисовать на экране `screen`.

В строке ❻ мы напрямую получаем текущее положение мыши с помощью команды `spot = pygame.mouse.get_pos()`, а не передаем в программу координаты последнего щелчка мышью, так как мы хотим рисовать в любой части экрана, в которую пользователь перетащит мышь, а не только лишь в той точке, в которой была нажата кнопка. В строке ❼ мы рисуем текущий круг на поверхности `screen` с цветом, установленным константой `YELLOW`, в точке (x, y) `spot`, куда пользователь перетаскивает мышь в настоящий момент, с радиусом `radius` 15, указанным нами в блоке настройки. Наконец, мы заканчиваем цикл строкой ❽, обновляя окно дисплея с помощью команды `pygame.display.update()`.

Соберем все вместе

Последним шагом, завершающим программу, как обычно, станет команда `pygame.quit()`. Далее приведена полная версия программы.

DragDots.py

```
import pygame # Настройка
pygame.init()
screen = pygame.display.set_mode([800,600])
pygame.display.set_caption("Click and drag to draw")
keep_going = True
YELLOW = (255,255,0) # Триплет RGB для ЖЕЛТОГО цвета
radius = 15
mousedown = False
```

```
while keep_going: # Игровой цикл
    for event in pygame.event.get(): # Обработка событий
        if event.type == pygame.QUIT:
            keep_going = False
        if event.type == pygame.MOUSEBUTTONDOWN:
            mousedown = True
        if event.type == pygame.MOUSEBUTTONUP:
            mousedown = False
        if mousedown: # Рисование/обновление графики
            spot = pygame.mouse.get_pos()
    pygame.draw.circle(screen, YELLOW, spot, radius)
        pygame.display.update() # Обновить дисплей

pygame.quit() # Выход
```

Приложение *DragDots.py* так быстро отвечает на действия пользователя, что при работе с ним может создаться ощущение рисования непрерывной кистью, а не последовательностью точек. Чтобы увидеть отдельно отрисованные точки, придется перетаскивать мышь довольно быстро. Модуль Pygame позволяет создавать более быстрые игры и анимации с бóльшим потоком графики по сравнению с графикой Turtle, которую мы использовали в предыдущих главах.

Несмотря на то что цикл `for` обрабатывает каждое событие при каждом проходе по циклу `while`, который держит наше приложение открытым, модуль Pygame достаточно эффективен и оптимизирован для того, чтобы выполнять эти операции десятки или даже сотни раз в секунду. Это позволяет создать иллюзию моментального движения и незамедлительной реакции на каждое наше движение или команду — очень важный аспект при создании анимаций и интерактивных игр. Модуль Pygame готов принять брошенный вызов: это правильный набор инструментов для создания приложений с интенсивным использованием графических ресурсов.

Улучшенная интерактивность: взрыв из смайликов

Моим студентам и сыновьям нравится создавать одну анимацию — улучшенную версию программы *SmileyBounce2.py* под названием *SmileyExplosion.py*. Эта программа выводит отскакивающий смайлик на новый забавный уровень, позволяя пользователю щелкать и перетаскивать мышь для создания сотен отскакивающих смайликов случайного размера,

перемещающихся по случайной траектории со случайной скоростью. Эффект этих изменений выглядит как показано на рис. 9.3. Мы пошагово создадим эту программу. Окончательная версия приведена на с. 253.

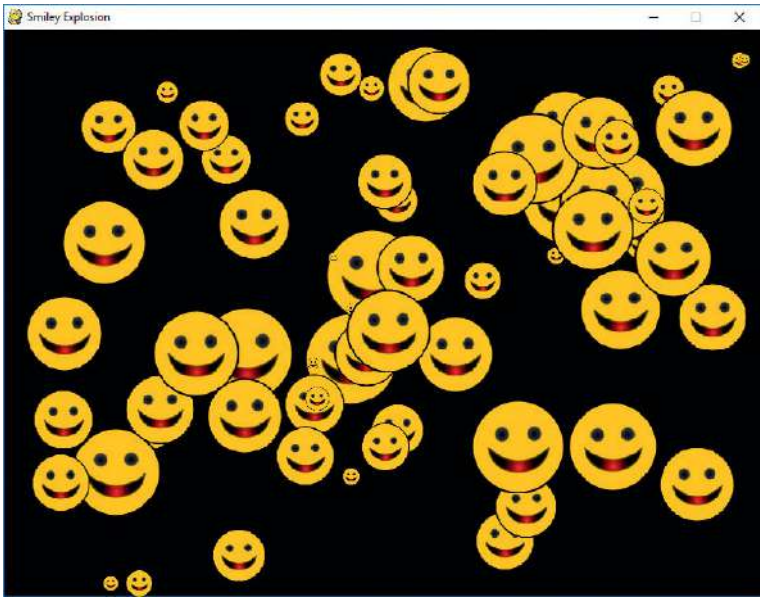


Рис. 9.3. *Наше следующее приложение выглядит как взрыв воздушных шаров со смайликами, разлетающихся по всему экрану*

Как вы можете видеть, у нас на экране окажутся сотни воздушных шаров со смайликами, скачущих по всему экрану, поэтому нам потребуется отрисовывать графику быстро и плавно для сотен объектов в каждом кадре. Чтобы добиться этого, мы воспользуемся следующим инструментом: спрайтовая графика.

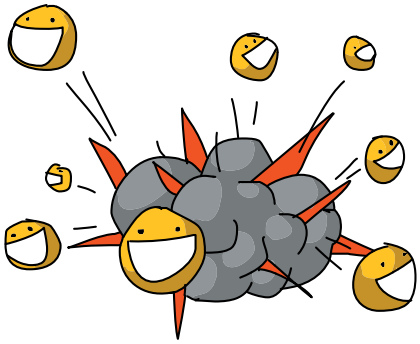
Спрайты смайликов

Термин *спрайт* появился в эпоху зарождения видеоигр. Движущиеся графические объекты на экране назывались спрайтами, так как эти объекты словно плавали на определенном фоне, подобно вымышленным феям, от которых и получили свое название*. Эти легкие объекты спрайтовой графики позволили создавать быструю и плавную анимацию, сделавшую видеоигры столь популярными.

В модуле Pygame реализована поддержка спрайтовой графики через класс `pygame.sprite.Sprite`. Из главы 8 вы помните, что класс подобен

* Слово «спрайт» восходит к английскому *sprite* — «фея», от латинского *spiritus* — «дух». *Прим. пер.*

шаблону, который может использоваться для создания пригодных для многократного использования объектов, и каждый из них будет обладать всей полнотой функций и свойств. В программе *SmileyMove.py* на с. 218 мы использовали класс `Clock`, а также метод `tick()`, чтобы сделать наши анимации более плавными и предсказуемыми. В приложении со взрывом смайликов мы воспользуемся несколькими полезными классами Pygame, а также создадим собственный класс, который будет отслеживать перемещение по экрану каждого отдельно взятого смайлика.



Еще немного информации о классах и объектах

В главе 8 вы узнали, что классы похожи на формочки для печенья, а объекты — на печенье, приготовленное с использованием некой определенной формочки. Когда нам необходимо создать несколько сущностей с похожими функциями и характеристиками (например, движущиеся изображения со смайликами разного размера и в разных участках экрана) и, особенно, когда нам необходимо, чтобы каждая такая сущность содержала некую информацию (например, размер, местоположение и скорость), класс может предоставить необходимый шаблон для создания любого нужного количества объектов данного типа. Мы говорим, что объекты — это *экземпляры* какого-то конкретного класса.

В библиотеке Pygame — десятки классов, предназначенных для использования. Для каждого класса реализованы собственные *методы* (то, что мы называем функциями класса), а также *атрибуты* или *данные*, то есть переменные и их значения, сохраненные в каждом таком объекте. В классе `Clock` из главы 8 метод `tick()` представлял собой функцию, устанавливающую количество повторений анимации при заданной частоте смене кадров. Что касается объектов класса `Sprite` — плавающих смайликов, то в этой главе нас будут интересовать такие их атрибуты, как положение на экране, размер и скорость движения по осям *x* и *y*. Иными словами, мы создадим класс `Smiley` с этими атрибутами. Если нам необходим шаблон для повторного использования, мы всегда можем создать собственный класс.

Разбиение проблемы или программы на объекты, а затем конструирование классов для последующего воссоздания этих объектов является основой *объектно-ориентированного программирования*. Объектно-ориентированное программирование — это способ решения задач с помощью объектов. Это один из самых популярных подходов, используемых в разработке программного обеспечения. Одна из причин популярности этого подхода заключается в концепции повторного использования программного кода. *Пригодность к повторному использованию* означает, что, создав полезный класс однажды, мы можем снова использовать этот класс в другой программе, а не писать код класса заново. Например, компания, разрабатывающая игры, может создать класс `Card`, который представлял бы карты в стандартной колоде. Каждый раз при программировании новой игры, как, например, блекджек, «Пьяница», покер, Go Fish и т.п., компания может повторно использовать класс `Card`, что значительно сэкономит время и деньги. Отличным примером применения этой концепции является класс `Sprite` модуля Pygame. Команда разработчиков Pygame создала класс `Sprite` со многими функциями, необходимыми нам при программировании игрового объекта, от бегущего персонажа до космического корабля или плавающих смайликов. Благодаря использованию класса `Sprite` программистам вроде нас с вами больше нет необходимости писать весь базовый код для отрисовки объекта на экране, обнаружения столкновения объектов и так далее. Класс `Sprite` обрабатывает за нас многие из этих функций, а мы можем сконцентрироваться на создании с его помощью уникального приложения.

Класс `Group` — это другой полезный класс модуля Pygame. `Group` — это класс-*контейнер*, который позволяет нам хранить несколько объектов `Sprite` вместе в виде группы. Класс `Group` помогает нам хранить в одном месте все наши спрайты (доступ к которым производится через единый объект `Group`). Это особенно важно, когда у нас на экране будет несколько десятков, а возможно, и сотен плавающих спрайтов. В классе `Group` также реализованы удобные методы для обновления всех спрайтов в группе (например, перемещение всех объектов `Sprite` в новое местоположение при смене кадра), а также для добавления новых объектов `Sprite`, удаления объектов `Sprite` из группы и так далее. Давайте посмотрим, каким образом мы можем использовать эти классы для создания приложения со взрывом из смайликов.

Использование классов для создания приложения

Сейчас мы создадим объекты `Sprite` для шариков со смайликами. Эти объекты будут использовать преимущества свойств класса `Sprite` производить быструю анимацию по всей поверхности экрана, даже когда в одном кадре перемещаются сотни спрайтов. Я также упомянул, что модуль Pygame поддерживает группы спрайтов, которые могут быть нарисованы и обработаны в виде коллекции. Группе спрайтов будет соответствовать тип `pygame.sprite.Group()`. Давайте взглянем на блок настройки этого приложения.

```
import pygame
import random

BLACK = (0,0,0)
pygame.init()
screen = pygame.display.set_mode([800,600])
pygame.display.set_caption("Smiley Explosion")
mousedown = False
keep_going = True
clock = pygame.time.Clock()
pic = pygame.image.load("CrazySmile.bmp")
colorkey = pic.get_at((0,0))
pic.set_colorkey(colorkey)
❶ sprite_list = pygame.sprite.Group()
```

Блок настройки выглядит так же, как и соответствующий блок программы *SmileyBounce2.py*, за исключением того, что в строке ❶ мы добавили переменную `sprite_list`. Эта переменная будет содержать нашу группу спрайтов-смайликов. Сохранение спрайтов в объекте `Group` упростит и ускорит выполнение таких операций, как отрисовка смайликов на экране при каждой смене кадров, перемещение смайликов для каждого последующего шага анимации и даже проверка и обнаружение столкновений смайликов друг с другом и с другими объектами на экране.

Для создания объектов-спрайтов для сложных анимаций и игр мы создадим собственный класс `Sprite`, который будет *расширять*, или надстраивать новые возможности, класса `Sprite` модуля Pygame, добавляя новые переменные и функции, которые мы хотим реализовать для наших собственных спрайтов. Мы назовем класс наших спрайтов `Smiley`, добавим переменные для хранения местоположения каждого смайлика (`pos`), скорость по осям x и y (`xvel` и `yvel`), а также масштаб, или размер, каждого смайлика (`scale`):

```
class Smiley(pygame.sprite.Sprite):
    pos = (0,0)
    xvel = 1
    yvel = 1
    scale = 100
```

Определение нашего класса `Smiley` начинается с ключевого слова `class`, после которого указывается имя, присвоенное создаваемому классу, а также тип, который мы расширяем (`pygame.sprite.Sprite`).

Настройка спрайтов

После создания класса `Smiley` и перечисления всех переменных, в которые мы хотим записать данные для всех смайликов — объектов класса, следует шаг под названием *инициализация*, иногда этот этап также именуется *конструктором* класса. Это будет специальная функция, вызываемая каждый раз при создании, или *конструировании*, нового объекта класса `Smiley`. Точно так же как инициализация переменной присваивает этой переменной начальное значение, *функция инициализации*, `__init__()`, нашего класса `Smiley` будет инициатором необходимых нам начальных значений объекта-спрайта. Два символа подчеркивания по обеим сторонам имени функции `__init__()` в языке Python имеют особое значение. В данном случае, `__init__()` — это имя специальной функции, использующейся для инициализации класса. В этой функции мы расскажем языку Python, каким образом следует инициализировать каждый объект класса `Smiley`. Каждый раз при создании объекта класса `Smiley` функция `__init__()` будет делать свою работу за кулисами основного действия, настраивая переменные и выполняя другие задачи для каждого объекта класса `Smiley`.

В функции `__init__()` нам потребуется настроить целый набор программных сущностей. Во-первых, мы определим, какие параметры требуется передавать функции `__init__()`. Для настройки случайных смайликов мы можем передать местоположение и начальные скорости по осям *x* и *y*. Поскольку `Smiley` — это класс, а спрайты со смайликами будут объектами типа `Smiley`, первым параметром всех

функций класса будет сам объект (спрайт). Мы обозначим этот объект как self (сам), так как данный объект содержит функцию __init__(), а также другие функции, предназначенные для обработки собственных данных объекта. Давайте взглянем на код функции __init__().

```
    def __init__(self, pos, xvel, yvel):
❶       pygame.sprite.Sprite.__init__(self)
❷       self.image = pic
        self.rect = self.image.get_rect()
❸       self.pos = pos
❹       self.rect.x = pos[0] - self.scale/2
        self.rect.y = pos[1] - self.scale/2
❺       self.xvel = xvel
        self.yvel = yvel
```

Четыре параметра функции __init__() это: сам объект (self), местоположение на экране, где мы хотим отобразить смайлик (pos), а также xvel и yvel, задающие значения горизонтальной и вертикальной скорости. Далее в строке ❶ мы вызываем функцию инициализации основного класса Sprite, чтобы наш объект мог воспользоваться преимуществами свойств спрайтовой графики без необходимости писать для них программный код с нуля. В строке ❷ мы присваиваем объекту (self.image) графику pic, загруженную ранее с диска (*CrazySmile.bmp* — убедитесь, что этот файл по-прежнему находится в той же папке, что и программа), и получаем размеры прямоугольника, содержащего изображение 100×100.

В строке ❸ выражение self.pos = pos сохраняет местоположение, переданное нами в функцию __init__(), в собственную переменную pos объекта. Затем, в строке ❹, мы настраиваем координаты *x* и *y* прямоугольника отрисовки спрайта, приравнивая их к координатам *x* и *y*, сохраненным в pos, сдвинутым на половину размера изображения (self.scale/2) для того, чтобы центр смайлика совпадал с точкой щелчка мышью. Наконец, в строке ❺ мы сохраняем вертикальную и горизонтальную скорости, переданные в функцию __init__() в собственных переменных xvel и yvel объекта (self.xvel и self.yvel).

Эта функция-конструктор __init__() настроит все необходимые параметры для рисования каждого смайлика на экране, однако она не обрабатывает анимацию, необходимую для перемещения смайликов по экрану. Для этого нашим спрайтам мы добавим еще одну полезную функцию, update().

Обновление спрайтов

Спрайты созданы для анимации, а мы уже знаем, что анимация — это покадровое обновление местоположения графического элемента (при каждом проходе по игровому циклу). Для спрайтов Pygame реализована встроенная функция `update()`, и мы можем *переопределить* эту функцию — настроить ее в соответствии с нашими потребностями так, чтобы запрограммировать поведение созданными нами спрайтов нужным образом.

Наша функция `update()` довольно проста. Обновления скачущих спрайтов со смайликами лишь покадрово изменяют местоположение каждого спрайта в соответствии со скоростью движения каждого конкретного смайлика, а также перехватывают столкновения смайликов с границей экрана.

```
    def update(self):
        self.rect.x += self.xvel
        self.rect.y += self.yvel
        if self.rect.x <= 0 or self.rect.x > ↵
           screen.get_width() - self.scale:
              self.xvel = -self.xvel
        if self.rect.y <= 0 or self.rect.y > ↵
           screen.get_height() - self.scale:
              self.yvel = -self.yvel
```

Функция `update()` принимает только один параметр — сам объект-спрайт (`self`), а код, перемещающий наш спрайт, очень похож на код анимации из программы *SmileyBounce2.py*. Единственное заметное отличие состоит в том, что мы ссылаемся на местоположение спрайта (x, y) через `self.rect.x` и `self.rect.y`, а на скорость — посредством `self.xvel` и `self.yvel`. При перехвате столкновения с границами экрана используются функции `screen.get_width()` и `screen.get_height()`, таким образом, перехват столкновения может работать при любом размере экрана.

Большие и маленькие смайлики

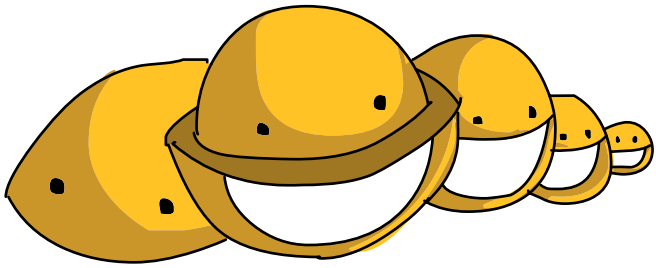
Последняя функция, которую мы добавим в первую версию приложения, — изменение *масштаба*, или размера изображения. Мы внесем это изменение в функцию `__init__()` сразу после кода, присваивающего `pic` переменной `self.image`. Сначала мы изменим значение переменной `scale` объекта на случайное число в диапазоне от 10 до 100 (чтобы размер законченного смайлика варьировался между 10×10 и 100×100). Мы применим это изменение в размере, также называемое *трансформацией*,

воспользовавшись функцией pygame.transform.scale() следующим образом.

```
    self.scale = random.randrange(10,100)
    self.image = pygame.transform.scale(self.image, ↵
                 (self.scale, self.scale))
```

Функция Pygame pygame.transform.scale() принимает изображение (наша переменная self.image с изображением смайлика) и размеры (наше новое случайное значение переменной self.scale в качестве высоты и ширины трансформированного изображения) и возвращает масштабированное (уменьшенное или увеличенное) изображение, которое мы сохраним как новое значение переменной self.image.

Внеся последнее изменение, мы должны получить возможность использовать наш класс спрайта Smiley для рисования смайликов случайного размера и со случайными скоростями по всему экрану. При этом сам код отрисовки аналогичен коду приложения для рисования *DragDots.py*, но будет иметь новые возможности.



Соберем все вместе

Далее приведена полная версия программного кода приложения *SmileyExplosion.py*.

SmileyExplosion.py

```
import pygame
import random

BLACK = (0,0,0)
pygame.init()
screen = pygame.display.set_mode([800,600])
pygame.display.set_caption("Smiley Explosion")
```

```
mousedown = False
keep_going = True
clock = pygame.time.Clock()
pic = pygame.image.load("CrazySmile.bmp")
colorkey = pic.get_at((0,0))
pic.set_colorkey(colorkey)
sprite_list = pygame.sprite.Group()

class Smiley(pygame.sprite.Sprite):
    pos = (0,0)
    xvel = 1
    yvel = 1
    scale = 100

    def __init__(self, pos, xvel, yvel):
        pygame.sprite.Sprite.__init__(self)
        self.image = pic
        self.scale = random.randrange(10,100)
        self.image = pygame.transform.scale(self.image, ↵
          (self.scale, self.scale))
        self.rect = self.image.get_rect()
        self.pos = pos
        self.rect.x = pos[0] - self.scale/2
        self.rect.y = pos[1] - self.scale/2
        self.xvel = xvel
        self.yvel = yvel

    def update(self):
        self.rect.x += self.xvel
        self.rect.y += self.yvel
        if self.rect.x <= 0 or self.rect.x > ↵
           screen.get_width() - self.scale:
              self.xvel = -self.xvel
        if self.rect.y <= 0 or self.rect.y > ↵
           screen.get_height() - self.scale:
              self.yvel = -self.yvel

while keep_going:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
              keep_going = False
        if event.type == pygame.MOUSEBUTTONDOWN:
              mousedown = True
        if event.type == pygame.MOUSEBUTTONUP:
              mousedown = False
    screen.fill(BLACK)
❶   sprite_list.update()
❷   sprite_list.draw(screen)
```

```
    clock.tick(60)
    pygame.display.update()
    if mousedown:
        speedx = random.randint(-5, 5)
        speedy = random.randint(-5, 5)
❸       newSmiley = Smiley(pygame.mouse.get_pos(), ↵
          speedx, speedy)
❹       sprite_list.add(newSmiley)

pygame.quit()
```

Код игрового цикла программы *SmileyExplosion.py* похож на код приложения для рисования *DragDots.py*, но с заметными изменениями. В строке ❶ мы вызываем функцию `update()` применительно к списку спрайтов со смайликами, сохраненному в `sprite_list`. Эта единственная строка вызовет функцию обновления, которая переместит каждый смайлик на экране и проверит столкновения с границами. Точно так же в строке ❷ мы отрисуем каждый смайлик в правильном местоположении на экране. Всего с помощью двух строк кода можно нарисовать и оживить несколько сотен смайликов — это огромная экономия времени и лишь часть всей мощи спрайтовой графики модуля Pygame.

В коде рисования `mousedown` мы генерируем случайные значения переменных `speedx` и `speedy` для задания горизонтальной и вертикальной скорости каждого смайлика, а в строке ❸ мы создаем новый смайлик, `newSmiley`, вызвав конструктор класса `Smiley`. Обратите внимание, что нет необходимости использовать имя функции `__init__()`, вместо этого мы используем лишь имя класса, `Smiley`, каждый раз при конструировании или создании нового объекта класса или типа `Smiley`. Функции-конструктору мы передаем положение указателя мыши, а также только что сгенерированное случайное значение скорости. Наконец, в строке ❹ мы добавляем вновь созданный спрайт со смайликом `newSmiley` и включаем его в группу `Group` с именем `sprite_list`.

Мы только что создали быструю, плавную и интерактивную анимацию для отображения множества спрайтов с графическими смайликами, плавающих по всему экрану, подобно воздушным шарикам, движущихся со случайными скоростями во всех направлениях. В окончательном обновлении этой программы мы увидим еще более захватывающие и мощные функции спрайтовой графики, перехватывающие и обрабатывающие столкновения.

SmileyPop, версия 1.0

В качестве завершающего примера мы добавим еще один важный аспект веселья в программу *SmileyExplosion.py*, а именно — возможность «лопать» шарики/пузырьки со смайликами при щелчке правой кнопкой мыши (или щелчком мыши с зажатой клавишей **CONTROL** на компьютерах Mac). Эффект этого нововведения будет выглядеть как в других играх с лопающимися шариками, таких как Ant Smasher, Whack-a-Mole и так далее. Мы сможем создавать шарики со смайликами, перетаскивая мышь с зажатой левой клавишей, и лопать их (то есть убирать с экрана) щелчком правой клавишей мыши над одним или несколькими спрайтами со смайликами.

Перехват столкновений и удаление спрайтов

У нас отличные новости: класс `Sprite` модуля Pygame поставляется со встроенной возможностью перехвата столкновений. Для проверки столкновения двух прямоугольников, содержащих спрайты, можно использовать функцию `pygame.collide_rect()`. Для проверки столкновения двух круглых спрайтов мы можем воспользоваться функцией `collide_circle()`. Если же надо проверить столкновение спрайта с единичной точкой (например, с пикселем, по которому пользователь щелкнул мышью), мы можем воспользоваться функцией `rect.collidepoint()` для проверки наложения, или столкновения, спрайта с этой точкой экрана.

Если мы определили, что пользователь щелкнул мышью по точке, касающейся одного или нескольких спрайтов, мы можем удалить каждый из этих спрайтов из группы `sprite_list`, вызвав функцию `remove()`. Мы можем обработать всю логику лопанья шариков со смайликами в коде обработчика события `MOUSEDOWN`. Для того чтобы превратить программу *SmileyExplosion.py* в *SmileyPop.py*, мы заменим следующие две строки кода:

```
        if event.type == pygame.MOUSEBUTTONDOWN:
            mousedown = True
```

на следующие семь строк кода:

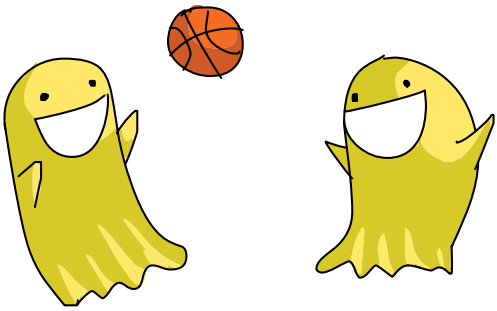
```
    if event.type == pygame.MOUSEBUTTONDOWN:
❶       if pygame.mouse.get_pressed()[0]: # Левая кнопка
                                          # мыши — рисовать
            mousedown = True
❷       elif pygame.mouse.get_pressed()[2]: # Правая
                                  # кнопка мыши — лопать
```

```
❸                 pos = pygame.mouse.get_pos()
❹                 clicked_smileys = [s for s in sprite_list ↵
                    if s.rect.collidepoint(pos)]
❺                 sprite_list.remove(clicked_smileys)
```

Выражение `if` обработчика события `MOUSEBUTTONDOWN` остается тем же самым, с той разницей, что теперь мы вставили код для определения, *которая* из кнопок мыши была нажата. В строке ❶ мы проверяем нажатие *левой* кнопки мыши (первая кнопка, индекс `[0]`). Если это так, мы включаем логический флаг `mousedown` — и игровой цикл отрисует новые смайлики. В строке ❷ мы проверяем нажатие *правой* кнопки мыши, эта проверка является началом проверки того, не была ли нажата правая кнопка мыши над одним из смайликов из группы `sprite_list`.

Сначала, в строке ❸, мы получаем местоположение указателя мыши и сохраняем его в переменной `pos`. В строке ❹ мы используем программное сокращение для генерации списка спрайтов из группы `sprite_list`, которые накладываются, или сталкиваются, с точкой `pos`, по которой пользователь щелкнул мышью. Если прямоугольник спрайта `s` из группы `sprite_list` сталкивается с точкой `pos`, записать этот спрайт в список `[s]` и сохранить список как `clicked_smileys`. Возможность создать один список, коллекцию или массив из другого списка, массива или коллекции — это мощная особенность языка Python, которая сильно сокращает объем кода в приложении.

Наконец, в строке ❺ мы вызываем полезную функцию `remove()` для нашей группы спрайтов `Group` с именем `sprite_list`. Эта функция `remove()` отличается от стандартной функции `remove()` языка Python, удаляющей один элемент из списка или коллекции. Функция `pygame.sprite.Group.remove()` удалит любое количество спрайтов из списка. В данном случае эта функция удалит все спрайты из нашей группы `sprite_list`, которые сталкиваются с точкой экрана, где пользователь щелкнул мышью. После удаления спрайтов из группы `sprite_list` при повторной отрисовке `sprite_list` на экране при следующем проходе по игровому циклу щелкнутые мышью спрайты уже отсутствуют



в списке, следовательно, программа их не отрисует. Выглядит это так, словно они исчезли или мы лопнули их, как шарики или пузыри.

Соберем все вместе

Далее приведена полная версия кода программы *SmileyPop.py*.

SmileyPop.py

```
import pygame
import random

BLACK = (0,0,0)
pygame.init()
screen = pygame.display.set_mode([800,600])
pygame.display.set_caption("Pop a Smiley")
mousedown = False
keep_going = True
clock = pygame.time.Clock()
pic = pygame.image.load("CrazySmile.bmp")
colorkey = pic.get_at((0,0))
pic.set_colorkey(colorkey)
sprite_list = pygame.sprite.Group()

class Smiley(pygame.sprite.Sprite):
    pos = (0,0)
    xvel = 1
    yvel = 1
    scale = 100

    def __init__(self, pos, xvel, yvel):
        pygame.sprite.Sprite.__init__(self)
        self.image = pic
        self.scale = random.randrange(10,100)
        self.image = pygame.transform.scale(self.image, ↵
          (self.scale, self.scale))
        self.rect = self.image.get_rect()
        self.pos = pos
        self.rect.x = pos[0] - self.scale/2
        self.rect.y = pos[1] - self.scale/2
        self.xvel = xvel
        self.yvel = yvel

    def update(self):
        self.rect.x += self.xvel
```

```
            self.rect.y += self.yvel
            if self.rect.x <= 0 or self.rect.x > ↵
               screen.get_width() - self.scale:
                  self.xvel = -self.xvel
        if self.rect.y <= 0 or self.rect.y > ↵
           screen.get_height() - self.scale:
                  self.yvel = -self.yvel

    while keep_going:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                  keep_going = False
            if event.type == pygame.MOUSEBUTTONDOWN:
                  if pygame.mouse.get_pressed()[0]: # Левая
                                     # кнопка мыши - рисовать
                        mousedown = True
                  elif pygame.mouse.get_pressed()[2]: # Правая
                                    # кнопка мыши - лопать
                        pos = pygame.mouse.get_pos()
                        clicked_smileys = [s for s in ↵
                          sprite_list if ↵
                          s.rect.collidepoint(pos)] ↵
                          sprite_list.remove(clicked_smileys)
            if event.type == pygame.MOUSEBUTTONUP:
                  mousedown = False
        screen.fill(BLACK)
        sprite_list.update()
        sprite_list.draw(screen)
        clock.tick(60)
        pygame.display.update()
        if mousedown:
            speedx = random.randint(-5, 5)
            speedy = random.randint(-5, 5)
            newSmiley = Smiley(pygame.mouse.get_pos(), ↵
              speedx, speedy)
            sprite_list.add(newSmiley)
    pygame.quit()
```

Помните — для корректной работы программы вам требуется, чтобы файл с изображением *CrazySmile.bmp* находился той же папке, что и код. Если все заработало — это программа настолько веселая и увлекательная, что в нее можно играть практически часами! В следующей главе мы изучим элементы дизайна компьютерных игр, которые делают их более интересными, а также создадим работающую игру с нуля!

Что вы узнали

В этой главе мы объединили взаимодействие с пользователем и анимацию для создания взрывов смайликов на экране, а также использовали спрайтовую графику для упрощения и ускорения анимации из сотен смайликов. Мы узнали, как создавать собственный класс `Sprite` таким образом, чтобы иметь возможность настраивать его особенности и поведение в соответствии со своими потребностями, в том числе переменные для хранения данных, функцию инициализации и собственную функцию обновления. Мы также узнали, как масштабировать изображения с помощью модуля Pygame, чтобы наши смайлики могли иметь разные формы и размеры. Мы изучили преимущества использования группы `pygame.sprite.Group()` для сохранения всех спрайтов ради последующего быстрого обновления и отрисовки на экране.

В завершающем примере мы добавили логику перехвата столкновений, основанную на самих спрайтах, чтобы проверить, не щелкнул ли пользователь правой кнопкой мыши над одним из спрайтов со смайликом. Мы увидели, как можно проверять наступление событий левой кнопки мыши отдельно от событий правой кнопки. Узнали, что в языке Python реализован мощный механизм для выделения элементов из списка на основе условия `if`, а также увидели, каким образом удалять спрайты из группы `Group` с использованием функции `remove()`.

В этой главе мы создали несколько интересных и веселых приложений, венцом которых стала программа *SmileyPop*, которую мы сделаем еще больше похожей на игру в главе 10. Pygame дал нам все навыки, необходимые для программирования замечательных игр!

Программирование забавных приложений в этой главе дало нам навыки для решения следующих задач.

- Использовать спрайтовую графику, настраивая в соответствии со своими потребностями класс `pygame.sprite.Sprite()`.
- Изменять, обновлять, рисовать, получать доступ к списку спрайтов с помощью класса `pygame.sprite.Group()` и его функций.
- Трансформировать изображение, применяя функцию `pygame.transform.scale()` для увеличения или уменьшения размера изображения в пикселях.
- Обнаруживать столкновения с помощью функции `rect.collidepoint()` и подобных ей в классе `Sprite`.

- Удалять спрайты из группы `Group` с использованием функции `remove()`.

Задачи по программированию

Вот три проблемные задачи для расширения навыков, полученных при изучении этой главы. Для загрузки примеров решения этих задач перейдите на сайт **https://eksmo.ru/files/Python_deti.zip**.

#1. Точки случайного цвета

Начните с выбора собственного триплета цвета для использования в программе *DragDots.py*. Затем измените программу так, чтобы она отрисовывала точки, окрашенные в случайные цвета, создавая триплет из трех случайных чисел в диапазоне от 0 до 255, который позже использовался бы для задания цвета. Назовите ваше новое приложение *RandomPaint.py*.

#2. Рисование в цвете

Дайте возможность пользователю рисовать двумя или большим количеством цветов, используя одну из следующих опций.

- Изменяйте текущий цвет рисования каждый раз, когда пользователь нажимает клавишу на клавиатуре, — либо на случайный цвет, либо на конкретный цвет, в зависимости от нажатой клавиши (например, красный для клавиши **R**, голубой для клавиши **B** и так далее).
- Рисуйте разными цветами для каждой из кнопок мыши (например, красный для левой клавиши, зеленый для средней и голубой — для правой).
- Добавьте несколько цветных прямоугольников внизу или сбоку экрана, а затем измените программу так, чтобы при щелчке мышью по одному из этих прямоугольников цвет рисования менялся на текущий цвет прямоугольника.

Попробуйте один или все три подхода сразу и сохраните свой новый файл как *ColorPaint.py*.

#3. Бросание смайликов

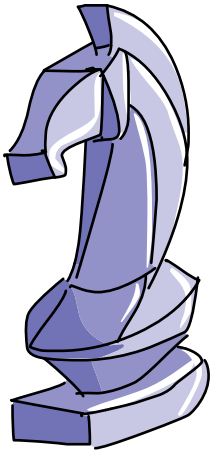
В модуле Pygame реализована функция `pygame.mouse.get_rel()`, возвращающая количество *относительного* движения, или величину, на которую изменилось положение указателя мыши по осям *x* и *y* в пикселях с момента последнего вызова функции `get_rel()`. Измените файл *SmileyExplosion.py* так, чтобы в программе использовалось относительное движение мыши по осям *x* и *y* в качестве горизонтальной и вертикальной скорости каждого смайлика (вместо генерации пары случайных значений для переменных `speedx` и `speedy`). Будет казаться, словно пользователь бросает смайлики, так как они будут постепенно терять скорость, перемещаясь в направлении перетаскивания мыши!

Добавьте еще один реалистичный эффект, несколько замедлив смайлики, умножив переменные `xvel` и `yvel` на число, меньше чем 1.0 (например, 0.95), в участке кода `update(self)` каждый раз, когда смайлики отскакивают от края экрана. С течением времени смайлики замедлятся, будто трение от каждого удара заставляет шарики все больше терять скорость. Сохраните свое новое приложение как *SmileyTrhow.py*.

Глава 10

ПРОГРАММИРОВАНИЕ ИГР: КОДИНГ ДЛЯ РАЗВЛЕЧЕНИЯ

В главе 9 мы объединили анимацию и взаимодействие с пользователем для создания интересного и веселого приложения. В этой главе мы сделаем надстройку на эти концепции и добавим некоторые элементы дизайна игр для создания своей игры с нуля. Мы объединим наше умение рисовать анимации на экране с возможностью обрабатывать взаимодействие с пользователем. Игру, напоминающую пинг-понг, мы назовем *Smiley Pong*.



Все игры, которые нам столь нравятся, состоят из определенных *элементов дизайна игр*. Далее приведена разбивка дизайна игры *Smiley Pong*.

Игровое поле или игровая доска. Черный экран представляет половину стола для пинг-понга.

Цели и достижения. Пользователь пытается набрать очки и избежать потери жизней.

Игровые фигурки (персонажи игры или объекты). У пользователя есть шарик и ракетка.

Механика. Мы заставим ракетку перемещаться вправо-влево с помощью мыши и защищать нижнюю границу экрана. Мячик может начать двигаться быстрее по мере усложнения игры.

Ресурсы. Игроку будет предоставлено пять жизней, или попыток, чтобы набрать как можно больше очков.

Вышеприведенные элементы используются в играх, чтобы завлечь пользователя. Эффективно созданная игра подразумевает наличие всех этих элементов, что упростит сам процесс игры, но в то же время усложнит задачу выиграть.

Создание каркаса игры: Smiley Pong, версия 1.0

Pong, изображенная на рис. 10.1, была одной из самых первых аркадных видеоигр, отсчитывающей свою историю еще с 1960-х годов. Даже спустя более 40 лет в нее все еще интересно играть.

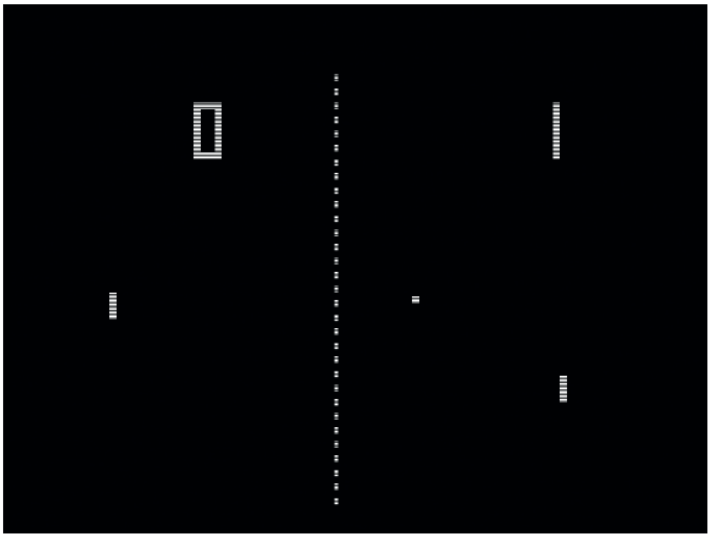


Рис. 10.1. *Знаменитая игра Pong для компьютера Atari 1972 года*

Геймплей (сюжет) версии игры Pong для одного игрока очень прост. Ракетка перемещается вдоль одного края экрана (мы поместим нашу ракетку вниз) и отталкивает шарик, в нашем случае — смайлик. Игроки получают

по одному очку каждый раз, когда им удается отбить шарик, а пропуская шарик, они теряют по одному очку (или жизни).

Основой для этой игры мы возьмем программу с отскакивающими смайликами из главы 8. Используя программу *SmileyBounce2.py* (с. 229), мы уже имеем плавно анимированный шарик со смайликом, отскакивающий от всех границ окна, а еще позаботились о цикле `while`, благодаря которому анимация будет выполняться до тех пор, пока пользователь не выйдет из программы. Для создания игры Smiley Pong мы добавим ракетку, которая будет следовать за перемещением мыши вдоль нижнего края экрана. Кроме того, мы включим сюда перехват столкновений, чтобы видеть, когда шарик со смайликом ударяется о ракетку. Последним штрихом будет начать с нуля очков и пяти жизней, давать пользователю одно очко, когда он ударяет по шарику, и отнимать одну жизнь, когда шарик отскакивает от экрана. На рис. 10.2 показано, в каком направлении мы будем трудиться. Когда мы закончим, окончательная версия программы будет выглядеть так, как показано на с. 276.

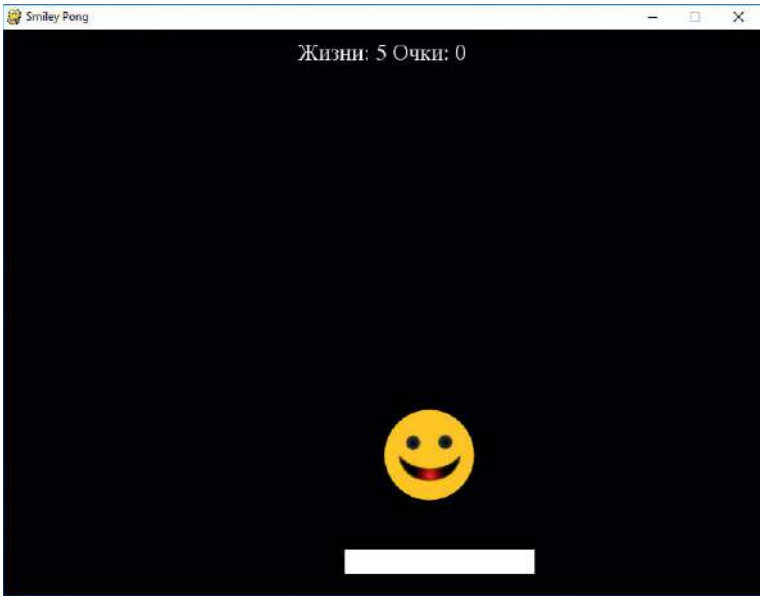


Рис. 10.2. *Игра Smiley Pong, которую мы создадим*

Первое, что мы добавим в бывшее приложение *SmuleyBounce2.py*, будет ракетка.

Отрисовка стола и игровых фигурок

В окончательной версии игры ракетка будет передвигаться вдоль нижнего края экрана вслед за движениями мыши, когда пользователь пытается отразить удар шарика.

Для создания ракетки в раздел кода «Настройка» нам потребуется ввести следующую информацию.

```
WHITE = (255, 255, 255)
paddlew = 200
paddleh = 25
paddlex = 300
paddley = 550
```

Эти переменные помогут нам создать ракетку, которая будет иметь вид простого белого прямоугольника шириной 200 и высотой 25. Нам потребуется, чтобы координаты верхнего левого угла ракетки начинались в точке (300, 550), это слегка приподнимет ракетку над нижней границей и отцентрирует по горизонтали на экране 800×600.

Однако мы не будем рисовать этот прямоугольник прямо сейчас. Этих переменных будет достаточно для отрисовки прямоугольника на экране в первый раз, однако наша ракетка должна будет следовать за движениями мыши. Нам нужно, чтобы ракетка была отцентрирована на экране относительно того, куда пользователь двигает указатель мыши по оси *x* (из стороны в сторону), при этом координата *y* должна быть фиксированной и соответствовать нижней части экрана. Для реализации этого требуется знать координаты мыши по оси *x*. Мы можем получить местоположение мыши с помощью функции `pygame.mouse.get_pos()`. В данном случае из координат, возвращаемых функцией `get_pos()`, нам нужна только координата *x*. Так как при определении местоположения мыши эта координата указывается первой, мы можем получить ее следующим образом:

```
paddlex = pygame.mouse.get_pos()[0]
```

Помните, что Pygame *начинает* отрисовывать прямоугольник (*x*, *y*) в предоставленной нами точке, оставшаяся часть прямоугольника рисуется правее и ниже этой точки. Для центровки ракетки относительно местоположения указателя мыши нам необходимо вычесть половину ширины

ракетки из ее координаты *x*, что поместит указатель мыши на середину ракетки:

```
paddlex -= paddlew/2
```

Теперь, когда нам известно, что центр ракетки всегда будет находиться там же, где и указатель мыши, в игровом цикле осталось только нарисовать прямоугольник ракетки на экране:

```
pygame.draw.rect(screen, WHITE, (paddlex, paddley, /
                 paddlew, paddelh))
```

Если вы добавите те три строки перед функцией `pygame.display.update()` в цикле `while` программы *SmileyBounce2.py*, а в раздел настройки добавите цвет ракетки, переменные `paddlew, paddelh, paddlex` и `paddley`, вы увидите, что ракетка будет следовать за перемещениями вашей мыши. Однако пока что шарик не будет отскакивать от ракетки, так как мы еще не ввели в программу логику перехвата столкновений шарика и ракетки. Мы сделаем это в следующем шаге.

Учет очков

Учет очков — это то, что делает игру интересной. Баллы, жизни, звезды — все, чтобы вы не использовали для учета очков, дает пользователю ощущение достижения по мере наблюдения за ростом очков. В нашей игре Smiley Pong мы хотим, чтобы пользователь получал одно очко каждый раз, когда шарик ударяется о ракетку, и терял одну жизнь всякий раз, когда не сумел поймать шарик и тот ударился о нижнюю границу экрана. Нашей следующей задачей будет добавить в программу логику, благодаря которой шарик отскакивал бы от ракетки, а пользователь получал очки, а также эта логика должна вычитать одну жизнь всякий раз, когда шарик ударяется о нижний край экрана. На рис. 10.3 показано, как может выглядеть ваша игра, когда пользователь набрал несколько очков. Обратите внимание, что дисплей очков был обновлен до 8.

Как мы уже упоминали ранее, мы начнем игру с нулевым количеством очков и пятью жизнями, установив эти параметры в разделе кода «Настройка».

```
points = 0
lives = 5
```

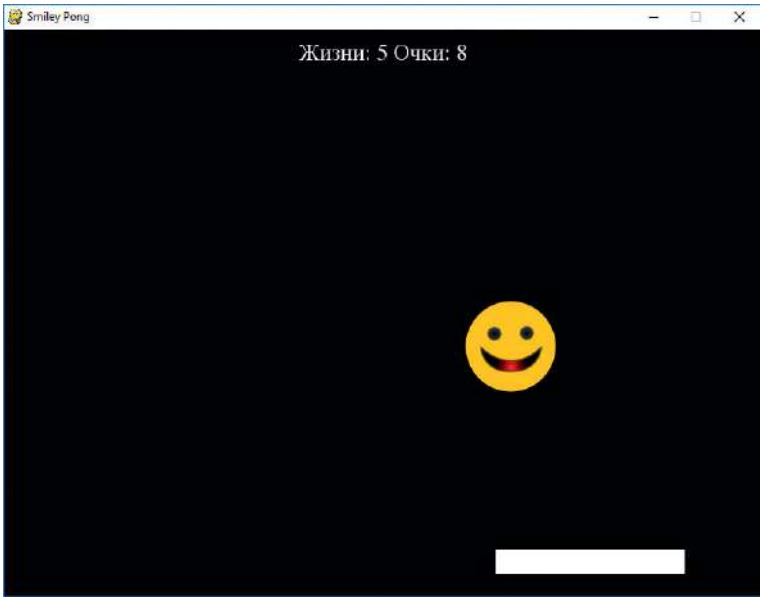


Рис. 10.3. *При отскоке шарика от ракетки мы будем увеличивать количество очков пользователя*

Далее нам необходимо придумать, каким образом увеличивать значение переменной `points` и уменьшать значение переменной `lives`.

Вычитание жизни

Давайте начнем с вычитания жизни. Мы знаем, что, если шарик ударяется о нижнюю границу экрана, значит, пользователь не смог его поймать ракеткой, соответственно, он должен потерять одну жизнь.

Для добавления логики, отнимающей одну жизнь, если шарик ударяется о нижнюю границу экрана, мы должны разбить наше выражение `if`, отвечающее за обнаружение столкновений с верхней и нижней границами экрана (`if picy <= 0 or picy >= 500`), на две части и обнаруживать столкновения с верхним и нижним краями отдельно. Если шарик ударяется о верхнюю границу экрана (`picy <= 0`), нам нужно, чтобы он просто отскакивал обратно, поэтому мы изменим направление скорости шарика по оси `y` с помощью выражения `-speedy`:

```
    if picy <= 0:
        speedy = -speedy
```

Если же шарик ударяется о нижний край экрана (`picy >= 500`), нам нужно вычесть одну жизнь из переменной `lives`, а затем выполнить отскок мяча:

```
    if picy >= 500:
        lives - = 1
        speedy = -speedy
```

Итак, проблема с вычитанием жизни решена, теперь нам нужно придумать, каким образом добавлять очки. В разделе «SmileyPop, версия 1.0» на с. 256, мы видели, что модуль Pygame содержит функции, упрощающие обнаружение столкновений. Но раз мы создаем игру Smiley Pong с нуля, давайте посмотрим, каким образом можно написать собственный код для обнаружения столкновений. Этот код может пригодиться в будущих приложениях, поэтому написать его — полезный опыт решения проблемы.

Бьем ракеткой по шарику

Для того чтобы проверить, не ударился ли шарик о ракетку, нам сперва необходимо представить, каким образом может такой контакт шарика и ракетки произойти. Шарик может удариться о верхний левый угол ракетки либо о правый верхний угол, или же шарик может отскочить прямо от верхней грани ракетки.

Когда вы разрабатываете логику обнаружения столкновений, очень полезно оказывается нарисовать все на бумаге и обозначить углы и грани, возможные столкновения с которыми необходимо проверять. На рис. 10.4 показан набросок, содержащий ракетку и два случая столкновения шарика с углами ракетки.

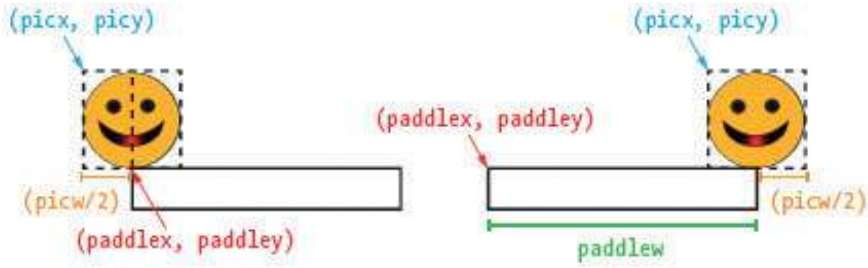
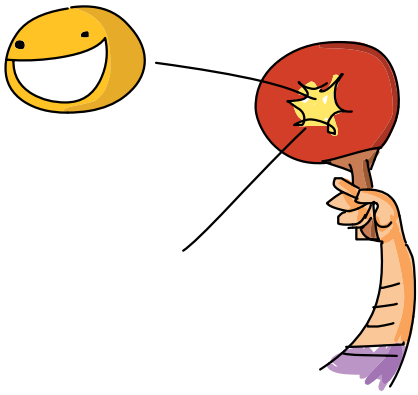


Рис. 10.4. *Два случая столкновения ракетки и шарика со смайликом*

Так как нам необходимо, чтобы шарик отскакивал от ракетки довольно реалистично, то нужно проверять случаи, когда центр нижней части шарика просто касается углов ракетки по краям слева и справа. Нам следует удостовериться в том, что пользователь получит очко, и когда шарик отскочит напрямую от верхней грани ракетки, и в том случае, когда шарик будет отбит углами ракетки. Для этого необходимо проверить, что местоположение шарика по вертикали соответствует нижней части экрана, где находится ракетка, и, если это так, мы должны выяснить, позволяет ли положение шарика по горизонтали отбить его ракеткой.



В первую очередь давайте выясним диапазон координат *x*, в рамках которого мы сможем говорить о том, что шарик ударился о ракетку. Так как середина шарика будет соответствовать половине его ширины, отсчитывая от верхнего левого угла (`picx`, `picy`), зададим ширину шарика в качестве переменной в разделе кода приложения «Настройки».

```
picw = 100
```

Как показано на рис. 10.4, шарик мог удариться о верхний левый угол ракетки, когда `picx` плюс половина ширины картинки (`picw/2`) касается `paddlex` — координаты *x* левого угла ракетки. В коде мы проверили бы выполнение этого условия, сделав его частью выражения `if`: `picx + picw/2 >= paddlex`.

Мы используем условие *больше или равно*, так как шарик может находиться правее (дальше, чем `paddlex` по оси *x*), но тем не менее касаться ракетки. Случай с углом — это всего лишь первый пиксель, за который игрок получает очко при ударе мячика о ракетку. Все координаты *x*, находящиеся между левым и правым углами, считаются действительными, поэтому эти координаты должны не только отбить шарик обратно, но и присвоить игроку одно очко.

Как мы можем видеть на рисунке, чтобы найти случай касания правого верхнего угла, нам необходимо, чтобы координата середины шарика,

вычисляемая как `picx + picw/2`, была меньше или равнялась координате правого верхнего угла ракетки. При этом координата правого верхнего угла ракетки вычисляется как `paddlex + paddlew` (начальная координата *x* ракетки плюс ширина ракетки). В коде это будет выглядеть как `picx + picw/2 <= paddlex + paddlew`.

Мы можем просто поместить эти два выражения в одно выражение `if`, однако этого не совсем достаточно. Эти координаты *x* покрывают весь экран от левого до правого угла ракетки, а также сверху вниз. Если мы определим только координаты *x*, наш шарик может находиться в любой точке по оси *y*, поэтому нам необходимо сузить область допустимых значений. Недостаточно знать, что шарик находится в *горизонтальных* пределах ракетки, нам также необходимо знать, что он находится в *вертикальном* диапазоне координат *y*, которые позволили бы ему ударяться о ракетку.

Мы знаем, что верхняя грань ракетки находится на уровне 550 пикселей в направлении оси *y*, рядом с нижней гранью экрана, так как раздел кода «Настройки» включает строку `paddley = 550`, соответственно, отрисовка прямоугольника начинается в этой координате *y* и продолжается вниз на 25 пикселей — высоту ракетки, сохраненную в переменной `paddleh`. Мы знаем, что высота нашей картинки составляет 100 пикселей, давайте сохраним эту информацию в переменной `pich` (от английского *picture height* — «высота рисунка»), которую включим в раздел с настройками: `pich = 100`.

Для того чтобы координата *y* нижней части нашего шарика касалась верхней грани ракетки, нам нужно, чтобы сумма положения `picy` и высоты рисунка `pich` (`picy + pich`) была больше или равнялась координате *y* ракетки (`paddley`). Часть выражения `if`, отвечающая за проверку удара шарика о ракетку в направлении оси *y*, выглядела бы следующим образом: `if picy + pich >= paddley`. Однако это условие само по себе позволит шарику быть в любой точке, координата *y* которой больше значения `paddley`, даже на нижней границе экрана. Мы не хотим, чтобы пользователь получал очки за удар ракеткой по шарику после того, как тот коснулся нижней границы экрана, поэтому нам необходимо еще одно выражение `if`: оно установит максимальное значение координаты *y*, за которое мы согласны давать пользователю очки.

Естественным выбором для задания максимального значения координаты *y* могла бы быть нижняя грань ракетки, или `paddley + paddleh` (координата *y* ракетки плюс ее высота). Однако, если нижняя точка шарика уже находится ниже нижней грани ракетки, пользователь не должен получать очки за удар по шарику. Таким образом, нам нужно, чтобы значение выражения `picy + pich` (нижняя точка шарика) было меньше или равнялось `paddley + paddleh`, иными словами — `picy + pich <= paddley + paddleh`.

Нам осталось проверить выполнение еще одного условия. Помните, что шарик и ракетка виртуальные, они не существуют в реальном мире, у них нет реальных краев и они не взаимодействуют друг с другом, как взаимодействовали бы реальные игровые фигурки. Мы могли бы провести ракеткой сквозь шарик, когда он отскакивает от нижнего края экрана. Однако мы не хотим присваивать пользователю очки, когда у нас есть все основания полагать, что он упустил шарик. Поэтому, перед тем как прибавлять одно очко, мы должны убедиться, что шарик направляется вниз, кроме того, что он должен находиться в вертикальном и горизонтальном диапазоне ракетки. Мы можем сказать, что шарик движется по направлению к нижней части экрана, если его скорость по оси *y* (`speedy`) больше нуля. Когда `speedy > 0`, шарик движется в положительном направлении по оси *y*.

Теперь у нас есть перечень всех условий, которые мы включим в два выражения `if`, чтобы проверить, ударил ли пользователь ракеткой по шарику:

```
if picy + pich >= paddley and picy + pich <= paddley + ↵
   paddleh  and speedy > 0:
if picx + picw / 2 >= paddlex and picx + ↵
   picw / 2 <= paddlex +  paddlew:
```

В первую очередь мы проверяем, находится ли шарик в вертикальном диапазоне, в котором он может коснуться ракетки, а также движется ли он вниз, а не наверх. Затем мы проверяем, находится ли шарик в горизонтальном диапазоне, в котором он также может коснуться ракетки.

В обоих выражениях `if` составные условия делают выражения слишком длинными — и они не могут уместиться на экране. Символ обратный слеш, ↵, позволяет нам продолжить длинную строку текста, перенеся ее на новую строку. Вы можете написать длинную строку кода на одной строке, либо вы можете перенести код так, чтобы он поместился на экране, для

этого в конце строке необходимо поставить символ обратный слеш, \, нажать клавишу **Enter** и продолжить код на следующей строке. В этой главе в коде игр есть несколько длинных строк, поэтому вы будете видеть обратные слеши в нескольких листингах кода. Просто помните, что Python прочитает все строки, разделенные обратным слешем, как одну строку кода.

Добавление точки

Давайте создадим программную логику для отскока шарика и присваивания очка. Чтобы завершить логику нашей ракетки, мы добавим еще две строки после двух выражений `if`:

```
    if picy + pich >= paddley and picy + pich <= paddley + ↵
       paddleh and speedy > 0:
          if picx + picw / 2 >= paddlex and picx + ↵
             picw / 2 <= paddlex + paddlew:
                points += 1
                speedy = -speedy
```

Добавить очки очень просто: `points += 1`. Изменить направление движения шарика так, чтобы это выглядело, словно он был отбит ракеткой, тоже несложно: нам нужно лишь реверсировать скорость по направлению оси *y* так, чтобы шарик опять начал подниматься вверх: `speedy = -speedy`.

Вы можете запустить программу с внесенными изменениями и пронаблюдать, как шарик отскакивает от ракетки. Каждый раз, когда ракетка бьет по шарику, вы зарабатываете одно очко; каждый раз, когда шарик попадает мимо ракетки, вы теряете одну жизнь; но на экране мы это еще не отображаем. Давайте сделаем это в следующем шаге.

Отображение очков

У нас есть необходимая программная логика для добавления очков и вычитания жизней, однако мы еще не видим очки на экране во время игры. В этом разделе мы нарисуем текст на экране, чтобы предоставить пользователям обратную связь во время игры, как показано на рис. 10.5.

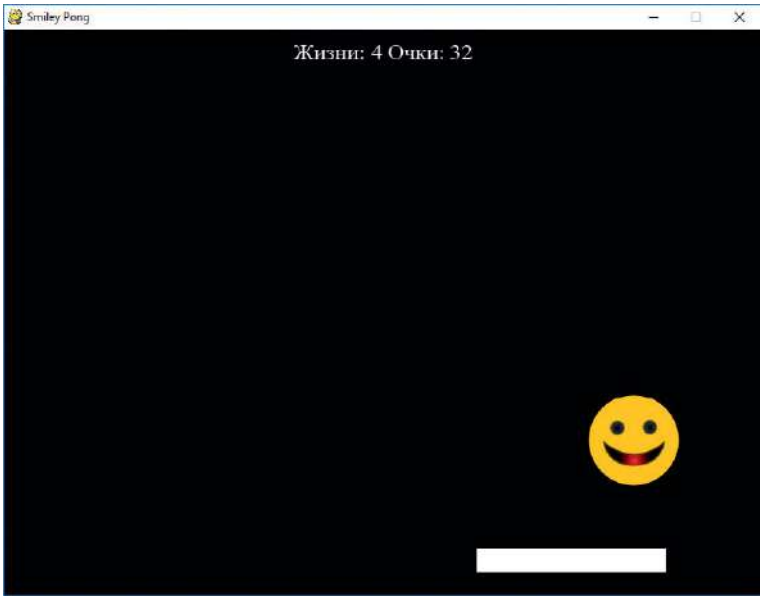


Рис. 10.5. *Smiley Pong, версия 1.0, становится настоящей игрой!*

Нашим первым шагом будет собрать строку текста для отображения. В обычной видеоигре на экране отображается количество очков и оставшихся жизней, например: *Жизни: 4*, *Очки: 32*. У нас уже есть переменные с количеством жизней (`lives`) и общим количеством очков (`points`). Все, что нам остается сделать, — воспользоваться функцией `str()` для преобразования этих чисел в их текстовые эквиваленты (5 становится `"5"`) и добавить текст, поясняющий, что означают цифры в каждом проходе по игровому циклу:

```
draw_string = "Жизни: " + str(lives) + "Очки: " + ↵
 str(points)
```

Наша строковая переменная будет именоваться `draw_string` и будет содержать текст, который мы хотим нарисовать на экране, отображая даные для пользователей во время игры. Для того чтобы нарисовать текст, у нас должна быть переменная или объект, подключенная к модулю отрисовки текста `pygame.font`. По-английски слово *font* означает «шрифт», или «гарнитура шрифта», иными словами — стиль рисуемых символов, например Arial или Times New Roman. В раздел «Настройки» кода добавьте следующую строку:

```
font = pygame.font.SysFont("Times", 24)
```

Это выражение создает переменную с именем `font`, которая позволит нам нарисовать на экране Pygame текст шрифтом Times размером 24 точки. Вы можете увеличить или уменьшить размер текста в вашей программе, однако сейчас размер 24 нам отлично подойдет. Далее мы нарисуем сам текст. Эта команда должна быть добавлена в игровой цикл сразу после объявления `draw_string`. Чтобы нарисовать текст на окне, мы сначала должны нарисовать строку теста на собственной поверхности с помощью функции `render()` созданного нами объекта `font`:

```
text = font.render(draw_string, True, WHITE)
```

Это выражение создает переменную с именем `text` для сохранения поверхности, содержащей белые пиксели, составляющие все буквы, цифры и символы нашей строки текста. Следующим шагом будет получение размеров (ширина и высота) этой поверхности. Более длинные строки будут обработаны так, что займут более широкие пространства экрана, для отрисовки же коротких строк потребуется меньше пикселей. Аналогичный принцип работает и для больших и меньших шрифтов. Строка текста будет обработана на прямоугольной поверхности, поэтому мы назовем переменную, хранящую прямоугольник, содержащий рисуемую строку, `text_rect`*:

```
text_rect = text.get_rect()
```

Команда `get_rect()` поверхности текста вернет размеры рисуемой строки. Далее мы отцентрируем по горизонтали текстовый прямоугольник `text_rect` на экране. Для этого воспользуемся атрибутом.`centerx`, кроме того, чтобы текст было лучше видно, мы поместим текстовый прямоугольник на 10 пикселей ниже верхней границы экрана. Далее приведены две команды для установки местоположения:

```
text_rect = screen.get_rect().centerx
 text_rect.y = 10
```

Теперь настало время нарисовать изображение `text_rect` на экране. Сделаем мы это с помощью функции `blit()`, которую мы уже использовали для изображения `pic`:

* От *англ.* rectangle — прямоугольник. *Прим. пер.*

```
    screen.blit(text, text_rect)
```

После этих изменений игра Smiley Pong стала похожа на классическую версию игры, только вместо шарика используется смайлик. Запустите приложение — и на экране вы увидите нечто похожее на то, что изображено на рис. 10.5. Мы на верном пути к созданию игры качества аркады!

Соберем все вместе

Для создания этой игры нам потребовалось использовать много навыков программирования. Переменные, циклы, условия, математика, графика, обработка событий — практически весь наш набор инструментов. Игры — это приключение и для игрока, и для программиста. Создание игры — процесс сложный, но благодарный: мы можем создавать геймплей в соответствии с нашими желаниями и затем делиться им с другими. Моим сыновьям понравилась версия 1.0, они также дали мне много идей для расширения игры и создания версии 2.0.

Далее приведена полная версия 1.0 *SmileyPong1.py*.

SmileyPong1.py

```
import pygame # Настройка
pygame.init()
screen = pygame.display.set_mode([800,600])
pygame.display.set_caption("Smiley Pong")
keepGoing = True
pic = pygame.image.load("CrazySmile.bmp")
colorkey = pic.get_at((0,0))
pic.set_colorkey(colorkey)
picx = 0
picy = 0
BLACK = (0,0,0)
WHITE = (255,255,255)
timer = pygame.time.Clock()
speedx = 5
speedy = 5
paddlew = 200
paddleh = 25
paddlex = 300
paddley = 550
picw = 100
pich = 100
points = 0
```

```
lives = 5
font = pygame.font.SysFont(«Times», 24)

while keepGoing: # Игровой цикл
        for event in pygame.event.get():
              if event.type == pygame.QUIT:
                    keepGoing = False
        picx += speedx
        picy += speedy

        if picx <= 0 or picx + pic.get_width() >= 800:
              speedx = -speedx
        if picy <= 0:
              speedy = -speedy
        if picy >= 500:
              lives -= 1
              speedy = -speedy

        screen.fill(BLACK)
        screen.blit(pic, (picx, picy))

        # Отрисовка ракетки
        paddlex = pygame.mouse.get_pos()[0]
        paddlex -= paddlew/2
        pygame.draw.rect(screen, WHITE, (paddlex, ↵
          paddley, paddlew, paddleh))

        # Проверка отскока от ракетки
        if picy + pich >= paddley and picy + pich <= ↵
           paddley + paddleh and speedy > 0:
    if picx + picw / 2 >= paddlex and picx + picw / 2 <= ↵
       paddlex + paddlew:
        points += 1
              speedy = -speedy

    # Отрисовка текста на экране
    draw_string = "Жизни: " + str(lives) + " Очки: " + ↵
      str(points)

    text = font.render(draw_string, True, WHITE)
    text_rect = text.get_rect()
    text_rect.centerx = screen.get_rect().centerx
    text_rect.y = 10
    screen.blit(text, text_rect)
    pygame.display.update()
    timer.tick(60)

pygame.quit() # Выход
```

Геймплей практически завершен: шарик отскакивает от ракетки, очки увеличиваются, игроки теряют жизни, если упускают шарик и тот ударяется о нижний край экрана. Теперь поразмыслите над тем, какие улучшения вы хотели бы внести, проработайте логику и попробуйте добавить код в версию 1.0, чтобы сделать игру еще интереснее. В следующем разделе вы добавите три новые функции для создания полностью интерактивной видеоигры, которой сможете поделиться с другими.

Усложнение и конец игры: Smiley Pong, версия 2.0

В версию 1.0 игры Smiley Pong вполне можно играть. Игроки могут набирать очки, терять жизни и видеть свой прогресс на экране. Однако у нас еще нет одного важного аспекта: конца игры. Еще один момент, которого не хватает нашей игре, — ощущение усложнения задачи по ходу игры. Мы добавим следующие функции в версию 1.0 игры Smiley Pong для создания версии 2.0: способ показать, что игра проиграна, когда пользователь теряет последнюю жизнь, способ начать игру повторно без необходимости закрывать программу, а также способ увеличить сложность по мере продолжения игры. Мы поочередно добавим эти функции, постепенно создавая интересную, захватывающую игру в жанре аркады! Окончательная версия приведена на с. 283.

Конец игры

Игра версии 1.0 никогда не прекращалась, так как мы не добавляли в программу логику, обрабатывающую окончание игры. Мы знаем условие, выполнение которого требуется проверить: считается, что игра закончилась, когда у пользователя не осталось жизней. Теперь нам нужно придумать, что делать, когда пользователь потерял последнюю жизнь.

Первое, что нужно сделать, — остановить игру. Мы не хотим закрывать программу, однако хотим прекратить движение мяча. Во-вторых, нам нужно изменить текст на экране, чтобы сообщить пользователям о завершении игры, и вывести количество набранных очков. Мы можем выполнить оба задания с помощью одного выражения `if` сразу после объявления `draw_string` для отрисовки текста с количеством жизней и очков:

```
if lives < 1:
        speedx = speedy = 0
```

```
        draw_string = "Игра окончена. Количество очков: " ↵
          + str(points)
        draw_string += ". Нажмите F1, чтобы сыграть еще ↵
          раз. "
```

Обнулив значения переменных `speedx` и `speedy` (соответственно горизонтальная и вертикальная скорость), мы прекратили движение шарика. Пользователь все еще может перемещать ракетку по экрану, однако мы визуально остановили ход игры, чтобы дать пользователю понять, что игра окончена. Сопроводительный текст делает все еще более понятным и к тому же сообщает пользователю о его успехах в этом раунде.

Прямо сейчас мы предлагаем пользователю нажать клавишу **F1**, чтобы сыграть снова, однако нажатие на клавишу пока что не приводит ни к каким действиям. Нам необходимо реализовать логику для обработки события нажатия на клавишу и повторного запуска игры.

Повторная игра

Мы хотим дать пользователю возможность начать новую игру по исчерпании всех жизней. Мы уже добавили на экран текст, предлагающий пользователю нажать **F1**, чтобы повторить игру, так давайте добавим код для обнаружения нажатия клавиши и повторного запуска игры. Во-первых, давайте проверим, нажата ли клавиша **F1**:

```
        if event.type == pygame.KEYDOWN:
              if event.key == pygame.K_F1: # F1 = Новая игра
```

В цикле `for` — обработчике событий внутри игрового цикла — мы добавили выражение `if`, проверяющее наступление события `KEYDOWN`. Если такое событие произошло, мы проверяем, какая клавиша была нажата в этом событии (`event.key`), чтобы понять, была ли это клавиша **F1** (`pygame.K_F1`). Код, следующий далее, будет нашим кодом *сыграть еще раз*, или кодом *новая игра*.

Вы можете получить полный список кодов клавиш Python, таких как `K_F1`, на сайте **www.pygame.org/docs/ref/key.html**.

«Сыграть еще раз» значит, что мы хотим начать игру с самого начала. Игра Smiley Pong начиналась с 0 очков, 5 жизней и шарика, приближающегося к нам со скоростью 5 пикселей в кадр из левого верхнего угла экрана, (0, 0).

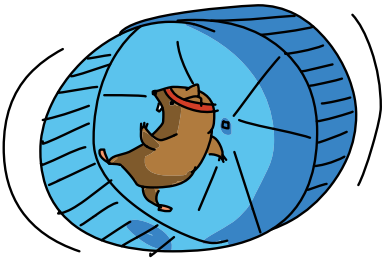
Сброс значений следующих переменных должен создать эффект начала новой игры:

```
points = 0
lives = 5
picx = 0
picy = 0
speedx = 5
speedy = 5
```

Добавьте эти строки к выражению `if` для события `KEYDOWN` клавиши **F1** — и вы сможете начать игру сначала в любое время. Если же вы хотите, чтобы запустить новую игру можно было, лишь потратив все жизни, добавьте условие `lives==0`. Однако в книге мы оставим выражения `if` как есть, чтобы пользователь мог повторно запустить игру в любое время.

Быстрее и быстрее

Нашей игре не хватает лишь одного элемента, свойственного играм: она не усложняется с течением времени. Можно подолгу в нее играть, при этом обращая на игру все меньше внимания. Давайте добавим усложнение по мере продолжения игры, чтобы увлечь пользователя и сделать игру более похожей на аркаду.



Нам нужно постепенно увеличивать скорость шарика по мере продвижения игрового процесса, однако скорость не следует увеличивать слишком сильно, иначе пользователь может быть сбит с толку. Нам нужно лишь незначительно ускорять игру после каждого отскока шарика. Логичным местом для реализации этой функции был бы участок кода, отвечающий за проверку отскоков. Увеличение скорости означает увеличение значений переменных `speedx` и `speedy`, чтобы шарик преодолевал большее расстояние с каждой сменой кадра. Попытайтесь изменить выражения `if`, ответственные за обнаружение столкновений (заставляющие шарик отскакивать от краев экрана), следующим образом:

```
if picx <= 0 or picx >= 700:
      speedx = -speedx * 1.1
if picy <= 0:
      speedy = -speedy + 1
```

В первом случае, когда шарик отскакивает от левой и правой сторон экрана в горизонтальном направлении, мы увеличиваем горизонтальную скорость (`speedx`), умножая ее на `1.1` (кроме того, мы также изменяем направление, добавляя знак минус). Это даст 10-процентное увеличение скорости после каждого отскока от левого или правого края.

Когда шарик отскакивает от верхней границы экрана (`if picy <= 0`), мы знаем, что скорость становится положительной, так как шарик отскакивает от верха и направляется обратно вниз в положительном направлении оси *y*, таким образом, мы можем прибавить 1 к переменной `speedy` после изменения направления знаком минус. Если шарик прилетел к верхней границе со скоростью 5 пикселей в кадр, то в обратном направлении он полетит со скоростью 6 пикселей в кадр, потом 7 и так далее.

Если вы внесете эти изменения, вы увидите, что шарик начнет двигаться все быстрее и быстрее и никогда не будет замедляться. Скоро шарик станет летать так быстро, что пользователь сможет потерять все пять жизней за одну секунду.

Мы сделаем нашу игру более приятной (и честной), сбрасывая скорость каждый раз, когда пользователь теряет жизнь. Если скорость становится настолько быстрой, что пользователь не может ударить ракеткой о шарик, возможно, пора сбросить скорость, если пользователь все еще может играть.

Код отскока от нижней границы экрана — это тот участок, в котором пользователи теряют жизни, поэтому давайте изменим скорость после вычитания жизни:

```
if picy >= 500:
      lives -= 1
      speedy = -5
      speedx = 5
```

Все это сделает игру более логичной, так как благодаря этим изменениям шар больше не выходит из-под контроля и не остается в таком состоянии до окончания игры. Когда пользователь теряет жизнь, шарик замедляется — и пользователь может ударить по нему ракеткой еще пару раз, прежде чем шарик вновь наберет скорость.

Однако у нас осталась еще одна проблема: шарик может начать перемещаться так быстро, что «застрянет» внизу. Сыграв несколько раз, пользователь столкнется с ситуацией, когда станет терять все оставшиеся жизни

в результате одного лишь отскока шарика от нижней границы экрана. Это произойдет потому, что шарик, перемещаясь очень быстро, может вылететь далеко за пределы нижней границы экрана, при этом, сбросив скорость, мы можем не успеть вернуть шарик на экран до начала следующего кадра.

Чтобы решить эту проблему, давайте добавим одну строку в выражение `if`:

```
        picy = 499
```

После потери пользователем жизни мы возвращаем шарик полностью на экран, присваивая переменной `picy` значение, например, `499`, помещающее шарик целиком над нижней границей экрана. Это поможет шарику полностью вернуться на экран вне зависимости от того, как быстро он перемещался до удара о нижний край экрана.

После внесения вышеописанных изменений версия 2.0 выглядит как на рис. 10.6.

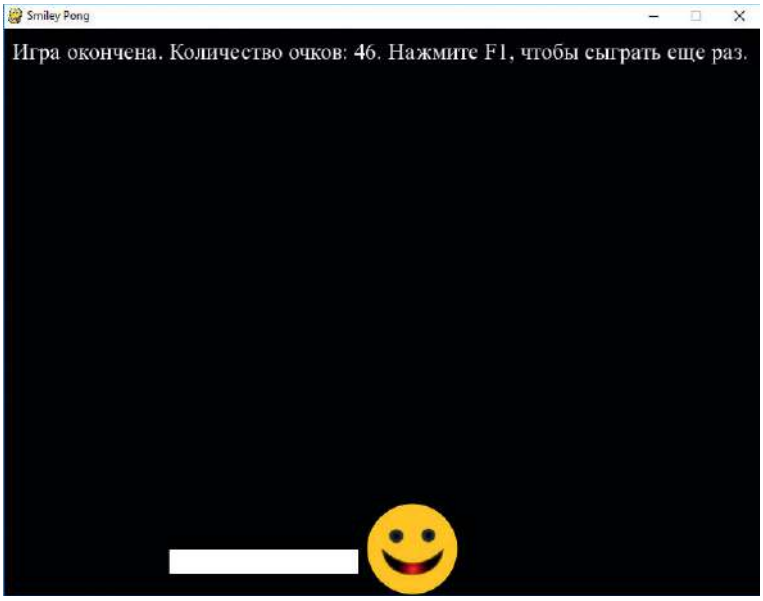


Рис. 10.6. *Версия 2.0 игры Smiley Pong отличается более скоростным геймплеем, концом игры и возможностью начать игру заново*

Версия 2.0, как настоящая аркада, завершается экраном «конец игры/сыграть еще раз».

Соберем все вместе

Здесь приведен полный код версии 2.0, *SmileyPong2.py*. Написав чуть меньше 80 строк кода, вы получите полноценную игру жанра аркады, которую можно показать родным и друзьям. А можно использовать ее как надстройку для развития ваших навыков программирования.

SmileyPong2.py

```
import pygame # Настройка
pygame.init()
screen = pygame.display.set_mode([800,600])
pygame.display.set_caption("Smiley Pong")
keepGoing = True
pic = pygame.image.load("CrazySmile.bmp")
colorkey = pic.get_at((0,0))
pic.set_colorkey(colorkey)
picx = 0
picy = 0
BLACK = (0,0,0)
WHITE = (255,255,255)
timer = pygame.time.Clock()
speedx = 5
speedy = 5
paddlew = 200
paddleh = 25
paddlex = 300
paddley = 550
picw = 100
pich = 100
points = 0
lives = 5
font = pygame.font.SysFont(«Times», 24)

while keepGoing: # Игровой цикл
   for event in pygame.event.get():
         if event.type == pygame.QUIT:
               keepGoing = False
         if event.type == pygame.KEYDOWN:
               if event.key == pygame.K_F1:  # F1 = Новая игра
                     points = 0
                     lives = 5
                     picx = 0
                     picy = 0
                     speedx = 5
                     speedy = 5
```

```
picx += speedx
picy += speedy

if picx <= 0 or picx >= 700:
      speedx = -speedx * 1.1
if picy <= 0:
      speedy = -speedy + 1
if picy >= 500:
      lives -= 1
      speedy = -5
      speedx = 5
      picy = 499

screen.fill(BLACK)
screen.blit(pic, (picx, picy))

# Отрисовка ракетки
paddlex = pygame.mouse.get_pos()[0]
paddlex -= paddlew/2
pygame.draw.rect(screen, WHITE, (paddlex, paddley, ↵
  paddlew, paddleh))

# Проверка отскока от ракетки
if picy + pich >= paddley and picy + pich <= paddley + ↵
   paddleh and speedy > 0:
      if picx + picw/2 >= paddlex and picx + picw/2 <= ↵
         paddlex + paddlew:
            speedy = -speedy
            points += 1

# Отрисовка текста на экране
draw_string = "Жизни: " + str(lives) + " Очки: " + ↵
str(points)

# Проверка конца игры
if lives < 1:
      speedx = speedy = 0
      draw_string = "Игра окончена. Количество очков: " ↵
        + str(points)
      draw_string += ". Нажмите F1, чтобы сыграть еще ↵
        раз. "

text = font.render(draw_string, True, WHITE)
text_rect = text.get_rect()
text_rect.centerx = screen.get_rect().centerx
text_rect.y = 10
screen.blit(text, text_rect)
```

```
    pygame.display.update()
    timer.tick(60)

pygame.quit() # Выход
```

Вы можете продолжить надстраивать игровые элементы этого примера (см. «Задачи по программированию» на с. 296) либо использовать эти «строительные элементы» для создания чего-то нового. Большинство игр и даже приложений другого рода обладают функциями, добавленными нами в этой главе, да и сам процесс создания игры, как правило, похож на то, как мы создавали игру Smiley Pong. Во-первых, распланируйте каркас игры, затем создайте рабочий *прототип*, или версию 1.0. Когда прототип заработал, добавляйте в него функции до тех пор, пока не создадите финальную версию, соответствующую вашим желаниям. Вы обнаружите, что *итеративное создание версий* приложения, то есть добавление по одной новой функции в каждую последующую версию, — очень полезная техника, подходящая даже при создании более сложных приложений.

Добавление новых функций: SmileyPop 2.0

Мы воспользуемся итеративным процессом создания версий приложения еще раз, добавив дополнительные функции, которые я и мой сын Макс захотели увидеть в приложении SmileyPop из главы 9. Во-первых, мой сын хотел добавить звуковой эффект каждый раз, когда пользователь лопает щелчком мыши шарик со смайликом. Во-вторых, мы оба хотели реализовать некую обратную связь, отображаемую на дисплее (например, сколько шариков было создано и сколько из них получилось лопнуть), я также хотел видеть показатель прогресса, например количество процентов лопнувших шариков. Приложение SmileyPop уже было забавным, однако эти эффекты улучшили бы его.

Взгляните снова на код приложения *SmileyPop.py* на с. 258. Мы начнем с этой версии приложения и создадим вторую версию (v2.0 — сокращение от «версия 2.0»), добавив код. Окончательная версия, *SmileyPop2.py*, приведена на с. 292.

Начнем мы с реализации просьбы Макса: звука лопающихся шариков.

Добавление звука с помощью Pygame

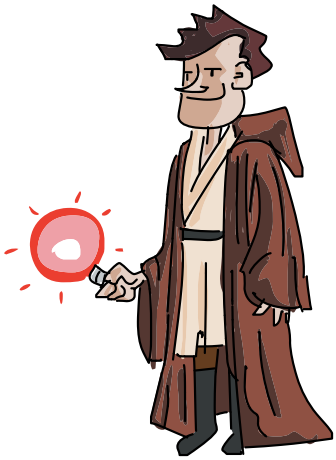
На сайте **www.pygame.org/docs** вы найдете модули, классы и функции, которые не только сделают вашу игру интереснее и веселее, но также упростят программирование. Модуль, который нужен для добавления звуковых эффектов, называется `pygame.mixer`. Для использования модуля микшера с целью добавления звука в ваши игры вам в первую очередь необходим звуковой файл. Для звукового эффекта лопающихся шариков загрузите файл **pop.wav** с сайта **https://eksmo.ru/files/Python_deti.zip** из раздела для загрузки исходных кодов главы 10.

Мы добавим следующие две строки в раздел настроек приложения *SmileyPop.py* сразу под строкой `sprite_list = pygame.sprite.Group()`:

```
pygame.mixer.init() # Добавление звуков
pop = pygame.mixer.Sound("pop.wav")
```

Мы начинаем с инициализации микшера (аналогично тому, как мы инициализируем Pygame с помощью команды `pygame.init()`). Затем мы загружаем звуковой эффект *pop.wav* в объект `Sound`, чтобы воспроизвести его в нашей программе.

Вторая строка загружает файл *pop.wav* как объект `pygame.mixer.Sound` и сохраняет его в переменной `pop`, которую мы будем использовать, когда захотим услышать звук лопающегося шарика. Как и в случае с файлами изображений, вам потребуется сохранить файл *pop.wav* в самой программе *SmileyPop.py*, чтобы код смог найти этот файл и воспользоваться им.



Далее мы добавим логику для проверки факта щелчка мышью по смайлику и воспроизведения звука `pop`, если пользователь лопнул смайлик. Это делается в разделе обработчика событий игрового цикла, в том же выражении `elif`, которое обрабатывает события правой кнопки мыши (`elif pygame.mouse.get_pressed()[2]`). После команды `sprite_list.remove(clicked_smileys)`, убирающей из списка `sprite_list` смайлики,

по которым пользователь щелкнул мышью, мы можем проверить, произошли ли столкновения смайликов, — и воспроизвести звук.

Пользователь мог щелкнуть мышью по участку экрана, в котором не находилось никаких смайликов, которые могли лопнуть, либо пользователь мог упустить смайлик, пытаясь совершить щелчок. Мы уточним, на самом ли деле пользователь щелкнул мышью по какому-нибудь смайлику, проверив выполнение условия `if len(clicked_smileys) > 0`. Функция `len()` возвращает длину списка или коллекции, и если длина этого списка больше нуля, значит, пользователь щелкнул по одному или нескольким смайликам. Вспомните, что `clicked_smileys` — это список спрайтов-смайликов, которые наложились при отрисовке на точку, по которой пользователь щелкнул мышью, или столкнулись с ней.

Если в списке `clicked_smileys` содержатся спрайты-смайлики, значит, пользователь корректно щелкнул правой кнопкой мыши по как минимум одному из них, соответственно, мы воспроизведем звук лопающегося шарика:

```
        if len(clicked_smileys) > 0:
            pop.play()
```

Обратите внимание, что обе строки должны быть корректно выровнены относительно кода выражения `elif`, обрабатывающего щелчки правой кнопкой мыши.

Вышеприведенные четыре строки кода — это все, что нужно для воспроизведения звука лопающихся шариков, когда пользователю удается щелкнуть правой кнопкой по смайлику. Для внесения этих изменений и прослушивания результата не забудьте скачать звуковой файл *pop.wav* и сохранить его в той же папке, что и программу *SmileyPop.py*. Выставьте на колонках достаточную громкость — и лопайте шарики в свое удовольствие!

Отслеживание и отображение прогресса пользователя

Следующая функция, которую нам нужно добавить, должна неким образом показывать пользователю его прогресс. Звуковые эффекты добавили своего рода развлекательную обратную связь (пользователи слышат звук лопающихся шариков, только если на самом деле попали по спрайту-смайлику). Однако давайте также отследим, сколько шариков пользователь создал и сколько лопнул, а также каков процент лопнутых шариков.

Создание логики по отслеживанию количества созданных и лопнутых шариков мы начнем, добавив в раздел кода с настройками переменную `font` и две переменные-счетчики: `count_smileys` и `count_popped`.

```
font = pygame.font.SysFont(«Arial», 24)
WHITE = (255,255,255)
count_smileys = 0
count_popped = 0
```

Мы настраиваем нашу переменную `font` на шрифт Arial размером 24 точки. Нам нужно отрисовать текст на экране белым цветом, поэтому мы создаем переменную для хранения цвета `WHITE` и приравниваем ее к триплету RGB, соответствующему белому цвету: `(255,255,255)`. Переменные `count_smileys` и `count_popped` будут хранить количество созданных и лопнутых смайликов. Начальное значение обеих переменных при запуске программы равняется нулю.

Созданные и лопнутые смайлики

В первую очередь давайте посчитаем смайлики по мере добавления их в список `sprite_list`. Чтобы сделать это, перейдем почти в самый конец кода программы *SmileyPop.py*, где выражение `if mousedown` проверяет факт перетаскивания мыши с зажатой кнопкой и добавляет смайлики в список `sprite_list`. Добавьте в это выражение `if` только последнюю строку:

```
if mousedown:
      speedx = random.randint(-5, 5)
      speedy = random.randint(-5, 5)
      newSmiley = Smiley(pygame.mouse.get_pos(), ↵
        speedx, speedy)
      sprite_list.add(newSmiley)
      count_smileys += 1
```

Прибавление 1 к текущему значению переменной `count_smileys` каждый раз при добавлении нового смайлика в список `sprite_list` поможет нам отслеживать общее количество отрисованных смайликов.

Мы добавим подобную логику и в выражение `if`, воспроизводящее звук лопающихся шариков, когда пользователь щелкает мышью по одному или нескольким смайликам, однако к значению переменной `count_popped` мы прибавим не единицу, а действительное количество лопнутых

шариков. Помните, что пользователь может щелкнуть мышью по одному или нескольким спрайтам-смайликам, накладывающимся на одну и ту же точку. В обработчике событий щелчка правой кнопки мыши мы собрали все столкнувшиеся смайлики в список `clicked_smileys`. Для выяснения того, сколько очков необходимо добавить к значению переменной `count_popped`, мы просто снова воспользуемся функцией `len()` для получения корректного количества смайликов, лопнутых пользователем щелчком правой кнопкой мыши. Добавьте следующие строки в выражение `if`, которое вы написали для воспроизведения звука лопающихся шариков:

```
        if len(clicked_smileys) > 0:
            pop.play()
            count_popped += len(clicked_smileys)
```

Прибавляя `len(clicked_smileys)` к переменной `count_popped`, мы всегда будем получать корректное количество лопнутых смайликов в любой момент времени. Теперь нам осталось лишь добавить в игровой цикл код, который будет отображать количество созданных, количество лопнутых смайликов, а также процентное соотношение лопнутых шариков к прогрессу пользователя.

Как и в случае с дисплеем игры Smiley Pong, мы создадим текстовую строку для отрисовки на экране, а также мы отобразим числа как строки с помощью функции `str()`. Добавьте эти строки в игровой цикл прямо перед вызовом функции `pygame.display.update()`:

```
    draw_string = "Шариков создано: " + str(count_smileys)
    draw_string += " — Шариков лопнуто: " + str(count_popped)
```

Эти две строки кода создадут переменную draw_string, которая будет отображать количество созданных и лопнутых шариков.

Процент лопнутых шариков

Добавьте следующие четыре строки сразу после выражений с переменными draw_string:

```
if (count_smileys > 0):
      draw_string += " — Процент: "
      draw_string += ↵
        str(round(count_popped/count_smileys*100, 1))
      draw_string += "%"
```

Для получения процентного соотношения лопнутых смайликов к общему количеству созданных мы делим переменную count_popped на переменную count_smileys (count_popped/count_smileys), затем умножаем полученный результат на 100, чтобы получить количество процентов (count_popped/count_smileys*100). Однако если мы попробуем отобразить это число, то столкнемся с двумя проблемами. Во-первых, при запуске программы значения обеих переменных равны нулю, следовательно, наша попытка вычислить количество процентов приведет к возникновению ошибки «деление на ноль». Чтобы исправить это, мы будем отображать количество лопнутых шариков, только если значение переменной count_smileys больше нуля.

Во-вторых, если пользователь создал три смайлика и лопнул только один из них, соотношение один к трем, 1/3, в процентном выражении будет равняться 33,33333333… Однако мы не хотим, чтобы отображаемая строка текста становилась очень длинной каждый раз, когда при вычислении процентов результат будет выражаться десятичной дробью с периодом, поэтому мы воспользуемся функцией round() для округления полученного числа до одного десятичного разряда.

Последним шагом будет отрисовка строки белыми пикселями, центровка этих пикселей около верхней границы экрана, а также вызов функции screen.blit() для копирования этих пикселей на экран для рисования:

```
text = font.render(draw_string, True, WHITE)
text_rect = text.get_rect()
text_rect.centerx = screen.get_rect().centerx
```

```
    text_rect.y = 10
    screen.blit (text, text_rect)
```

Результат внесения этих изменений вы можете видеть на рис. 10.7.

Маленькие смайлики гораздо сложнее поймать и лопнуть, особенно когда они двигаются быстро, поэтому уровня более 90 процентов достичь достаточно сложно. И это именно то, чего мы хотим. Мы использовали этот компонент обратной связи вместе с компонентом вызов/достижение, чтобы наша программа ощущалась пользователями как настоящая игра, в которую можно играть.

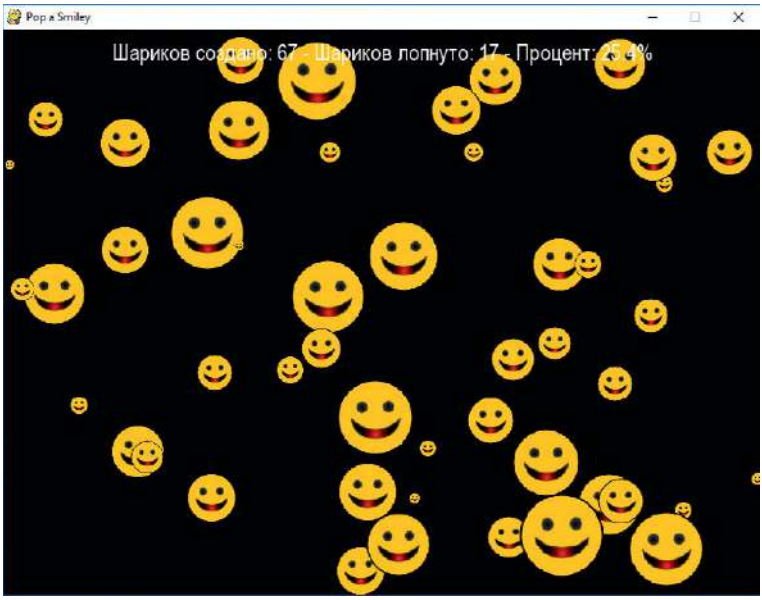


Рис. 10.7. *Приложение SmileyPop стало больше походить на игру, когда мы добавили в него звук и дисплей для отображения прогресса и обратной связи*

Звук лопающихся шариков и обратная связь через дисплей сделали приложение SmileyPop больше похожим на мобильное приложение. Щелкая правой кнопкой мыши по смайликам, вы вполне можете представить себя лопающим смайлики касанием пальца экрана мобильного устройства. (Чтобы узнать, как создавать приложения для мобильных устройств, посетите сайт MIT App Inventor по адресу **appinventor.mit.edu**.)

Соберем все вместе

Далее приведен полный листинг кода приложения SmileyPop версии 2.0. Помните о необходимости хранения файла исходного кода *.py*, файла изображения *CrazySmile.bmp* и звукового файла *pop.wav* в одной папке.

Так как код этого приложения занимает почти 90 строк, возможно, он слишком длинный для набора вручную. Посетите сайт **https://eksmo.ru/files/Python_deti.zip** для загрузки кода вместе со звуковым и графическим файлами.

SmileyPop2.py

```
import pygame
import random

BLACK = (0,0,0)
WHITE = (255,255,255)
pygame.init()
screen = pygame.display.set_mode([800,600])
pygame.display.set_caption("Pop a Smiley")
mousedown = False
keep_going = True
clock = pygame.time.Clock()
pic = pygame.image.load("CrazySmile.bmp")
colorkey = pic.get_at((0,0))
pic.set_colorkey(colorkey)
sprite_list = pygame.sprite.Group()
pygame.mixer.init() # Добавляем звуки
pop = pygame.mixer.Sound("pop.wav")
font = pygame.font.SysFont(«Arial», 24)
count_smileys = 0
count_popped = 0

class Smiley(pygame.sprite.Sprite):
    pos = (0,0)
    xvel = 1
    yvel = 1
    scale = 100

    def __init__(self, pos, xvel, yvel):
         pygame.sprite.Sprite.__init__(self)
         self.image = pic
         self.scale = random.randrange(10,100)
         self.image = pygame.transform.scale(self.image,↵
          (self.scale, self.scale))
         self.rect = self.image.get_rect()
```

```
        self.pos = pos
        self.rect.x = pos[0] - self.scale/2
        self.rect.y = pos[1] - self.scale/2
        self.xvel = xvel
        self.yvel = yvel
    def update(self):
        self.rect.x += self.xvel
        self.rect.y += self.yvel
        if self.rect.x <= 0 or self.rect.x > ↵
           screen.get_width() - self.scale:
              self.xvel = -self.xvel
        if self.rect.y <= 0 or self.rect.y > ↵
           screen.get_height() - self.scale:
              self.yvel = -self.yvel

while keep_going:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
              keep_going = False
        if event.type == pygame.MOUSEBUTTONDOWN:
              if pygame.mouse.get_pressed()[0]: # Левая
                                  # кнопка мыши - рисовать
                    mousedown = True
              elif pygame.mouse.get_pressed()[2]: # Правая
                                # кнопка мыши - лопать
                    pos = pygame.mouse.get_pos()
                    clicked_smileys = [s for s in ↵
                      sprite_list if ↵
                      s.rect.collidepoint(pos)]
                    sprite_list.remove(clicked_smileys)
                    if len(clicked_smileys) > 0:
                          pop.play()
                          count_popped += ↵
                            len(clicked_smileys)
              if event.type == pygame.MOUSEBUTTONUP:
                    mousedown = False
    screen.fill(BLACK)
    sprite_list.update()
    sprite_list.draw(screen)
    clock.tick(60)
    draw_string = "Шариков создано: " + str(count_smileys)
    draw_string += " - Шариков лопнуто: " + str(count_
    popped)
    if (count_smileys > 0):
        draw_string += " - Процент: "
        draw_string += ↵
          str(round(count_popped/count_smileys*100, 1))
        draw_string += "%"
```

```
    text = font.render(draw_string, True, WHITE)
    text_rect = text.get_rect()
    text_rect.centerx = screen.get_rect().centerx
    text_rect.y = 10
    screen.blit (text, text_rect)

    pygame.display.update()
    if mousedown:
          speedx = random.randint(-5, 5)
          speedy = random.randint(-5, 5)
          newSmiley = Smiley(pygame.mouse.get_pos(), ↵
            speedx, speedy)
          sprite_list.add(newSmiley)
          count_smileys += 1

pygame.quit()
```

Чем больше программ вы пишете, тем лучше у вас получается программировать. Вы можете начать писать с программного кода для игр, которые вы находите интересными, с создания приложения, решающего определенные важные для вас задачи, или создания приложений для других людей. Продолжайте программировать, решайте больше проблем, становитесь все лучшим и лучшим программистом, и в скором времени вы сможете создавать продукты, полезные для пользователей по всему миру.

Не важно, что вы программируете: будь то мобильные игры и приложения, программы, управляющие автомобилями, роботами или дронами, или же еще одно приложение для социальных сетей, написание кода — это навык, способный изменить всю вашу жизнь.

У вас есть нужные навыки. У вас есть способность. Продолжайте практиковаться, продолжайте программировать, выходите на рынок и измените что-то в своей жизни, в жизнях людей, дорогих вашему сердцу, и людях всего мира.

Что вы узнали

В этой главе вы узнали об элементах дизайна игр, от целей и достижений до правил и механики. Мы создали с нуля игру Smiley Pong, рассчитанную на одного игрока, а также превратили приложение SmileyPop в игру, которую легко можно представить запущенной на смартфоне или планшете.

Мы соединили анимацию, взаимодействие с пользователем и дизайн игр для создания двух версий игры Smiley Pong и второй версии приложения SmileyPop, постепенно прибавляя дополнительные функции.

В игре Smiley Pong мы нарисовали стол и игровые фигурки, добавили взаимодействие с пользователем для перемещения ракетки, а также добавили обнаружение столкновений и подсчет очков. Мы отобразили на экране текст, информирующий пользователей об их достижениях и состоянии игры. Вы научились обнаруживать в Pygame события нажатий на клавиши клавиатуры, добавили логику «конец игры» и «играть снова», а также закончили версию 2.0 увеличением скорости перемещения шарика по мере продвижения игры. Теперь у вас есть рамочные элементы, а также строительные блоки для создания более сложных игр.

В случае с SmileyPop мы начали с игры, в которую уже было интересно играть, и с помощью модуля `pygame.mixer` добавили обратную связь с пользователем в форме звука лопающихся шариков, а затем также добавили логику и дисплей для отслеживания прогресса пользователя по мере создания и уничтожения все большего количества шариков.

Приложения, которые вы будете создавать, используя ваши навыки программирования, также начнутся с простой версии, или *опытного образца*, который вы можете запускать и использовать как базу для новых версий. Вы можете начать с любой программы и добавлять по одной функции за раз, сохраняя каждую новую версию. Такой подход называется *итеративным созданием версий*. Он позволяет отладить каждую новую функцию и довести ее до безотказной работы, кроме того, данный подход позволяет вам сохранять последнюю работающую версию, даже если ваш код перестает нормально функционировать после внесения изменений.

Иногда новая функция хорошо вольется в приложение, и вы сохраните эту базу для создания новой версии. Иногда новый код работать не будет, либо новая функция будет не такой удачной, как ожидалось. В любом случае вы будете совершенствовать свои навыки программирования, испытывая новые функции и решая новые задачи.

Счастливого программирования!

После усвоения концепций, изложенных в этой главе, вы должны уметь делать следующее.

- Узнавать общие элементы дизайна игр в играх и приложениях, которые вы видите.

- Использовать элементы дизайна игр в создаваемых вами приложениях.
- Создавать каркас игры путем рисования стола и игровых фигурок, а также добавления взаимодействия с пользователем.
- Программировать обнаружение столкновений между игровыми фигурками и учет очков в приложении или игре.
- Отображать текстовую информацию на экране с помощью модуля `pygame.font`.
- Писать игровую логику для определения момента окончания игры.
- Обнаруживать и обрабатывать события нажатия клавиш в Pygame.
- Разрабатывать код для начала новой игры или повторного начала игры после окончания текущей игры.
- Использовать математику и логику для постепенного усложнения игр.
- Добавлять в ваши игры звуки с помощью модуля `pygame.mixer`.
- Отображать проценты и округленные числа для информирования пользователей об их прогрессе в игре.
- Понимать процесс итеративного создания версий: добавление по одной функции в приложение и сохранение измененного приложения в качестве новой версии (1.0, 2.0 и так далее).

Задачи по программированию

Чтобы увидеть примеры решения этих задач, а также для загрузки звуковых файлов для этой главы перейдите на сайт **https://eksmo.ru/files/Python_deti.zip**.

#1. Звуковые эффекты

В игру Smiley Pong мы могли бы добавить еще одну функцию, а именно — звуковые эффекты. В классической консольной и аркадной игре Pong шарик издавал высокотональный импульсный звук, когда пользователю прибавлялось очко, либо низкотональный импульсный звук или жужжание, когда пользователь упускал шарик. Для решения одной из последних задач воспользуйтесь навыками, изученными при создании версии 2.0 приложения Smiley Pop,

и обновите игру Smiley Pong v2.0 до v3.0, добавляя звуковые эффекты отскока от ракетки и нижнего края экрана. Сохраните новый файл как *SmileyPong3.py*.

#2. Попадания и пропуски

Чтобы сделать приложение Smiley Pop еще больше похожим на игру, добавьте логику для отслеживания количества попаданий и пропусков из общего количества щелчков мыши. Если при щелчке правой кнопкой мыши пользователь попадает по одному из спрайтов-смайликов, прибавьте 1 к количеству попаданий `hits` (1 попадание на щелчок — нам не нужно дублировать `count_popped`). Если пользователь щелкает кнопкой и не попадает ни по одному из смайликов, учтите это в переменной `miss`. Вы также можете запрограммировать логику для окончания игры после определенного количества пропусков, либо вы могли бы предоставить пользователю максимальное количество щелчков, чтобы пользователь пытался достичь максимального процентного соотношения. Либо вы даже можете добавить таймер и предложить пользователю создать и лопнуть как можно больше шариков, например, за 30 секунд. Сохраните эту новую версию как *SmileyPopHitCounter.py*.

#3. Зачистка шариков

Вы можете добавить функцию «очистки» (или кнопку-обман) для уничтожения разом всех шариков путем нажатия на функциональную кнопку, наподобие функции «играть снова» в приложении Smiley Pong. Вы также можете заставить смайлики замедляться с течением времени, умножая скорость на число меньше 1 (например, 0,95) каждый раз, когда шарик отскакивает от края экрана. Возможности, на самом деле, бесчисленны.

Приложение А

УСТАНОВКА PYTHON В СРЕДЕ WINDOWS, MACOS И LINUX

В этом приложении мы проведем вас по всем шагам установки среды Python для операционных систем Windows, macOS и Linux. В зависимости от версии используемой вами операционной системы то, что вы увидите на экране, может незначительно отличаться от того, что показано в книге, однако нижеприведенные шаги должны помочь вам установить и настроить среду Python.

Если вы устанавливаете Python на компьютер в школе или на работе, возможно, для установки вам потребуется разрешение отдела IT. Если при установке среды Python в школе вы столкнетесь с какими-либо трудностями, попросите помощи у специалистов по IT, чтобы они тоже знали, что вы изучаете программирование.

Python для Windows

Для Windows мы будем использовать язык Python версии 3.2.5, что упростит установку модуля Pygame (см. Приложение Б) для программ из глав 8–10.

Загрузка установщика

1. Зайдите на сайт **python.org** и установите указатель мыши поверх ссылки **Downloads** (Загрузки) — на экране появится перечень доступных опций, как показано на рис. А.1.

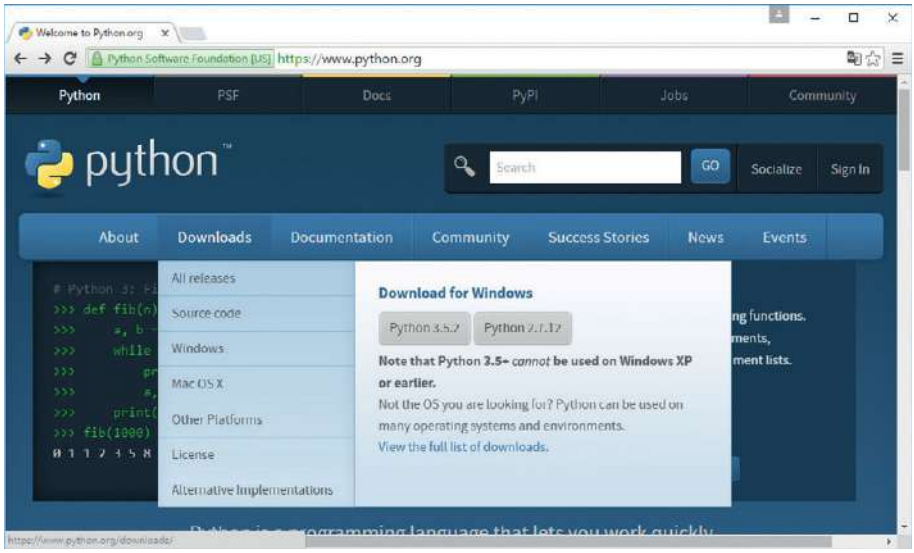


Рис. А.1. *Установите указатель мыши поверх ссылки* ***Downloads*** *для отображения перечня доступных опций*

2. В открывшемся списке щелкните мышью по ссылке **Windows**, это перенесет вас на страницу **Python Releases for Windows** (Выпуски Python для Windows), как показано на рис. А.2.

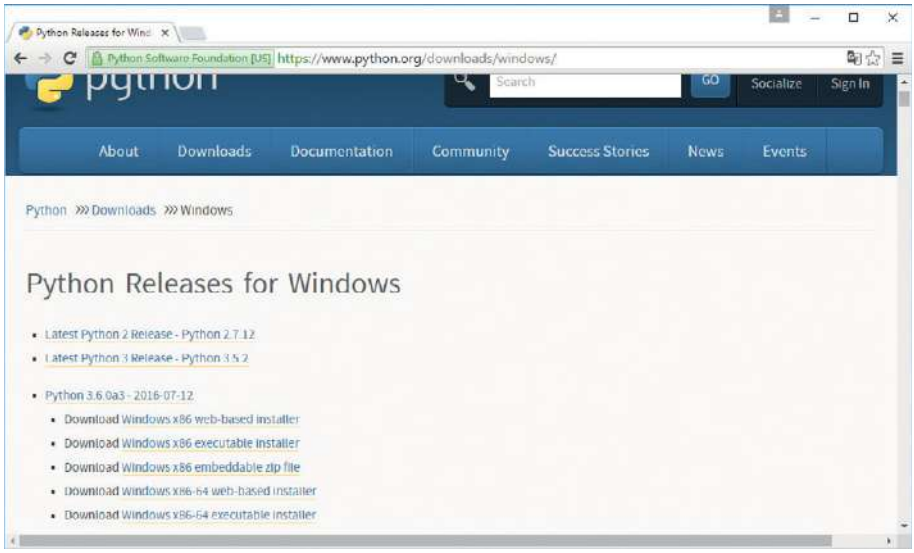


Рис. А.2. *Страница загрузок Python для Windows*

3. Пролистывайте вниз до тех пор, пока не увидите ссылку, начинающуюся с *Python 3.2.5*. Под этой ссылкой вы также увидите несколько опций, как показано на рис. А.3.

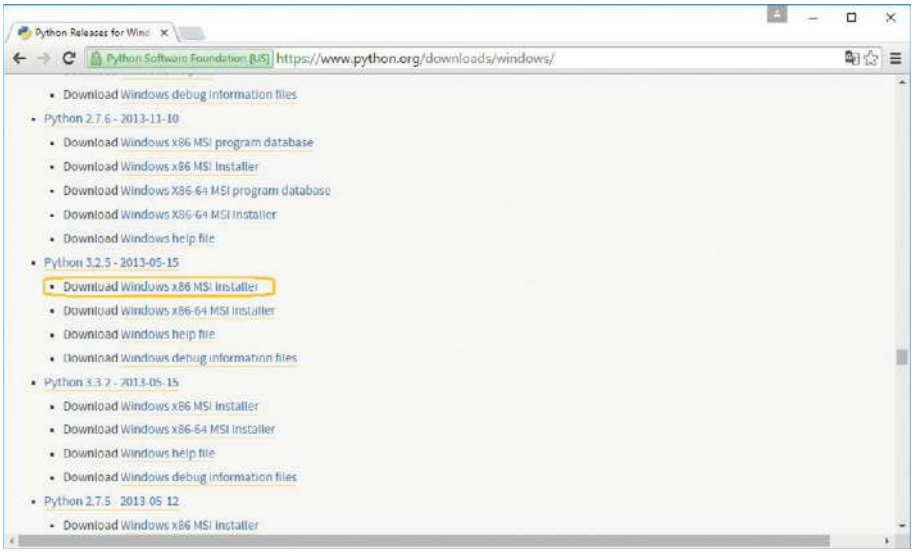


Рис. А.3. *В разделе* ***Python Releases for Windows*** *найдите установщик Python 3.2.5*

4. Под ссылкой Python 3.2.5 щелкните мышью по опции **Windows x86 MSI installer** (Установщик MSI для Windows x86), чтобы скачать на компьютер программу-установщик.

Запуск установщика

1. Дождитесь окончания загрузки — и откройте папку **Загрузки** (Downloads). В этой папке вы должны будете увидеть программный файл установщика Windows **python-3.2.5**, как показано на рис. А.4.
2. Для начала установки дважды щелкните мышью по файлу программы-установщика Windows **python-3.2.5**.
3. На экране может появиться диалоговое окно с предупреждением системы безопасности, как показано на рис. А.5. Если вы видите это окно, щелкните мышью по кнопке **Да** (Yes). Выводя это окно, Windows лишь информирует вас о том, что программное обеспечение пытается что-то установить на ваш компьютер.

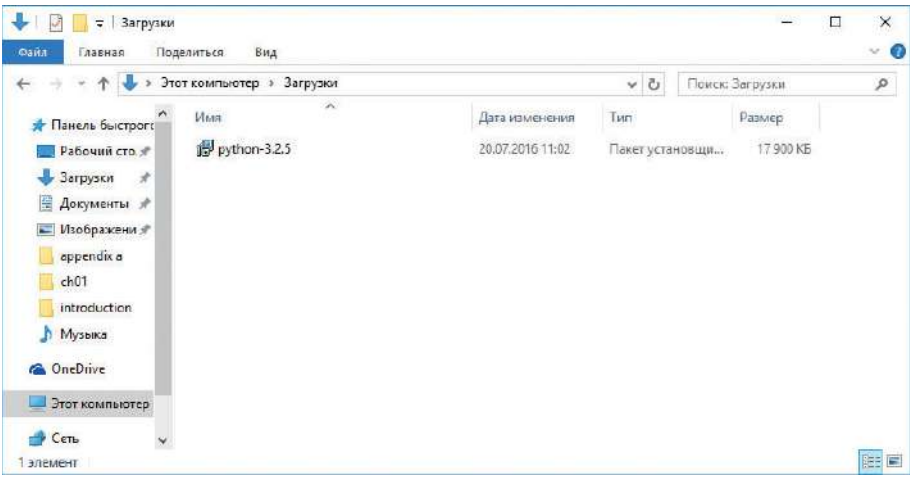


Рис. А.4. *Дважды щелкните мышью по файлу установщика в папке Загрузки*

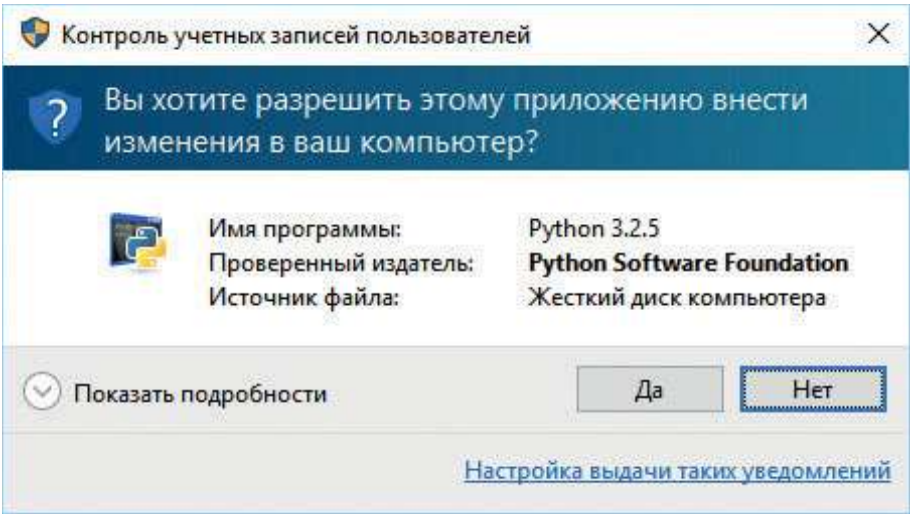


Рис. А.5. *Нажмите кнопку* ***Да****, чтобы разрешить установку*

4. Установщик может спросить вас, желаете ли вы установить Python для всех пользователей или только для себя, как показано на рис. А.6. Как правило, лучше выбирать опцию **Install for all users** (Установить для всех пользователей), однако, если это не разрешено в вашей школе или офисе или у вас не получается осуществить такую установку, попробуйте выбрать опцию **Install just for me** (Установить только для меня). Затем нажмите кнопку **Next** (Далее).

Рис. А.6. *Установка для всех пользователей*

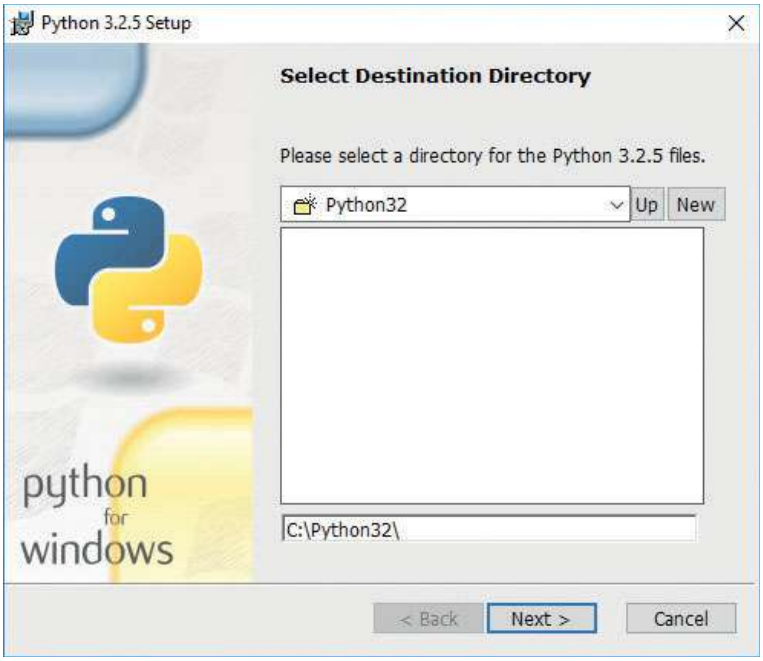


Рис. А.7. *Выберите папку для установки Python*

5. Далее вы увидите окно **Select Destination Directory** (Выберите конечную папку), наподобие изображенного на рис. А.7. В этом окне вы можете выбрать, в какую папку следует установить среду Python. Программа попытается произвести установку на диск **C:** в папку **Python32**, и это должно сработать на вашем ноутбуке или домашнем ПК. Нажмите кнопку **Next** (Далее), чтобы продолжить установку. (Если вы устанавливаете на компьютер в школе или на работе и столкнулись с трудностями, персонал IT может посоветовать вам установить в другую папку, например **User** (Пользователь) или **Desktop** (Рабочий стол).)

6. Теперь вы увидите окно, наподобие изображенного на рис. А.8, просящего вас персонализировать Python. Здесь ничего менять не нужно: просто нажмите кнопку **Next** (Далее).

Рис. А.8. *Ничего менять не нужно: просто нажмите кнопку* ***Next***

7. Теперь работа с установщиком завершена — и на экране вы должны увидеть окно, наподобие изображенного на рис. А.9. Нажмите кнопку **Finish** (Закончить).

Рис. А.9. *Нажмите кнопку* ***Finish****,*
чтобы выйти из программы-установщика

Вы установили Python! Теперь вы можете попробовать его, чтобы быть уверенным в том, что установка прошла успешно.

Тестирование Python

1. Выполните команду **Start ⇒ Programs ⇒ Python 3.2 ⇒ IDLE (Python GUI)** (Пуск ⇒ Программы ⇒ Python 3.2 ⇒ IDLE (Python GUI)), как показано на рис. А.10. (В Windows 8 или более поздних версиях вы можете нажать кнопку **Windows** или **Пуск** (Start), перейти к инструменту поиска и ввести с клавиатуры запрос `IDLE`.)

2. На экране должно появиться окно текстового редактора оболочки. Это программа Python, в окно которой вы можете вводить команды и моментально видеть результаты. Если вам любопытно, можете начать пробовать вводить какой-либо код. Введите команду `print("Здравствуй, Python!")` и нажмите клавишу **Enter**. Оболочка Python должна ответить фразой `Здравствуй, Python!`, как показано на рис. А.11. Также попробуйте ввести выражение сложения, например `2 + 3`. Нажмите клавишу **Enter** — и Python даст правильный ответ!

Рис. А.10. *Запуск IDLE из меню* ***Пуск***

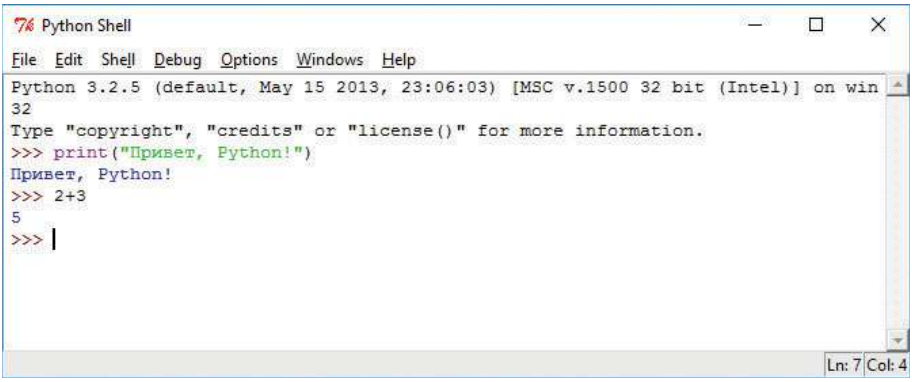


Рис. А.11. *Тестирование некоторых команд в оболочке Python*

3. Наконец, для упрощения чтения вы можете изменить размер текста в программе IDLE. Выполните команду **Options ⇒ Configure IDLE...** (Опции ⇒ Настроить IDLE...). На вкладке **Fonts/Tabs** (Шрифты/Отступы), изображенной на рис. A.12, измените параметр **Size** (Размер) на **18** или любой другой размер, который вам будет проще читать. Вы также можете установить флажок **Bold** (Жирный), чтобы сделать буквы толще. Выберите шрифт, который вам кажется более приятным для глаз.

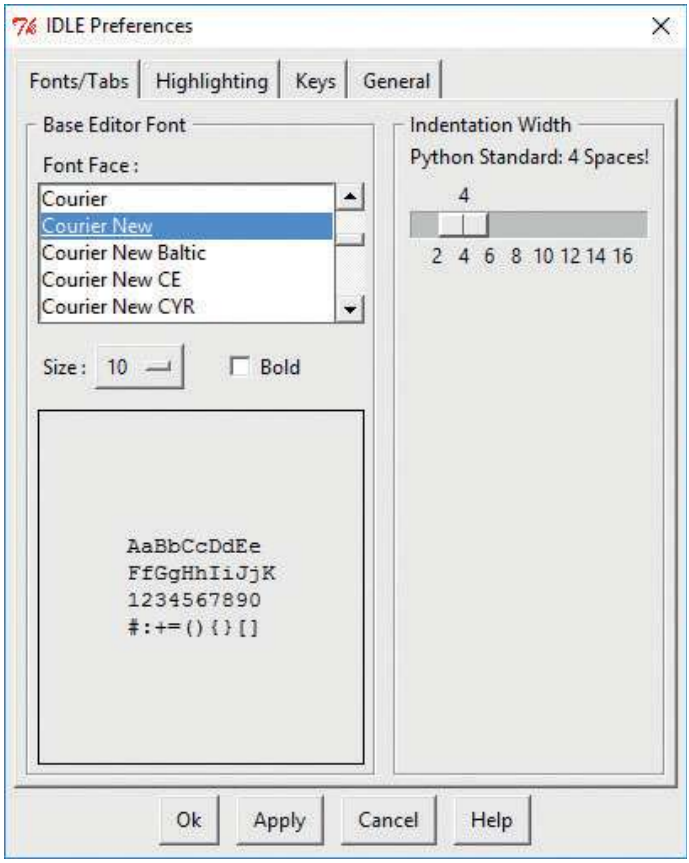


Рис. A.12. *Настройка предпочтений в программе IDLE*

4. После того как вы выбрали шрифт и установили размер, чтобы упростить чтение данных, вводимых в окно IDLE, нажмите кнопку **Apply** (Применить), а затем — кнопку **OK**, чтобы вернуться в окно оболочки Python IDLE. Теперь, выполняя ввод с клавиатуры, вы должны видеть на экране текст выбранного вами шрифта и размера.

Теперь все готово для изучения глав 1–7. Для использования программ из глав 8–10 перейдите к Приложению Б и выполните шаги для установки модуля Pygame. Счастливого программирования!

Python для macOS

Большинство компьютеров Apple поставляются с предустановленной более ранней версией языка Python, однако для использования функций языка Python 3 надо установить версию 3.4.2, что позволит нам запускать коды примеров из этой книги.

Загрузка установщика

1. Зайдите на сайт **python.org** и установите указатель мыши поверх ссылки **Downloads** (Загрузки) — на экране появится перечень доступных опций, среди которых вы увидите пункт **Mac OS X**, как показано на рис. А.13.

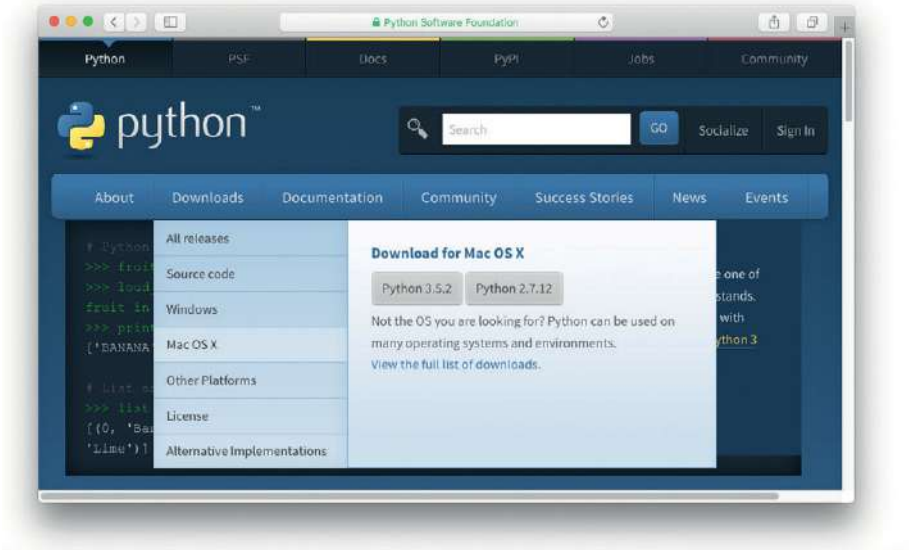


Рис. А.13. *Установите указатель мыши поверх ссылки* ***Downloads*** *для отображения перечня доступных дистрибутивов, среди которых вы должны увидеть пункт* ***Mac OS X***

2. В открывшемся списке щелкните мышью по ссылке **Mac OS X**, это перенесет вас на страницу **Python Releases for Mac OS X** (Выпуски Python для Mac OS X).

3. На данной странице найдите ссылку, начинающуюся с *Python 3.4.2*, и щелкните мышью по ней, чтобы скачать на компьютер программу-установщик.

Запуск установщика

1. Дождитесь окончания загрузки — и откройте папку **Загрузки** (Downloads). В этой папке вы должны будете увидеть программный файл установщика **python-3.4.2**, как показано на рис. А.14. Дважды щелкните мышью по этому файлу, чтобы начать установку.

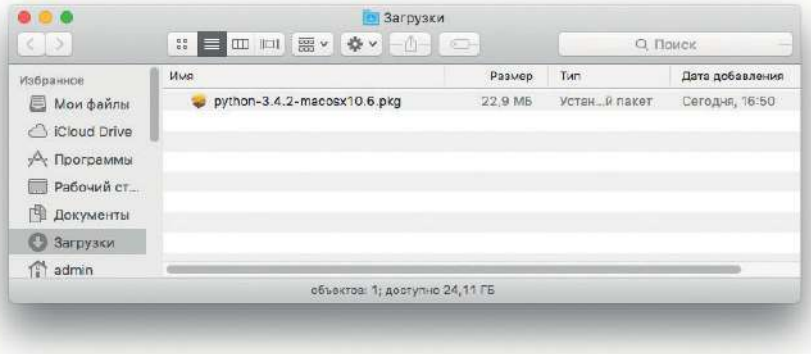


Рис. А.14. *Дважды щелкните мышью по файлу установщика в папке* ***Загрузки***

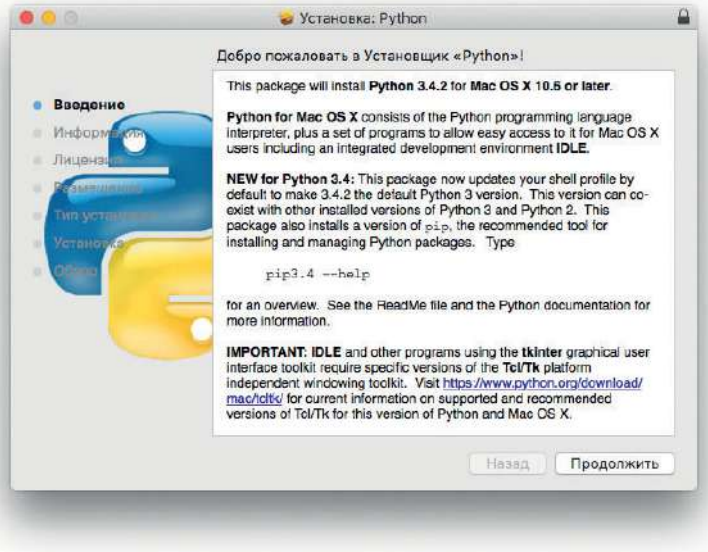


Рис. А.15. *В приветственном окне нажмите кнопку* ***Продолжить***

2. Двойной щелчок по файлу установщика откроет приветственное окно **Установка Python** (Install Python). Вы должны увидеть приветственное окно, наподобие того, что изображено на рис. А.15. Нажмите кнопку **Продолжить** (Continue).

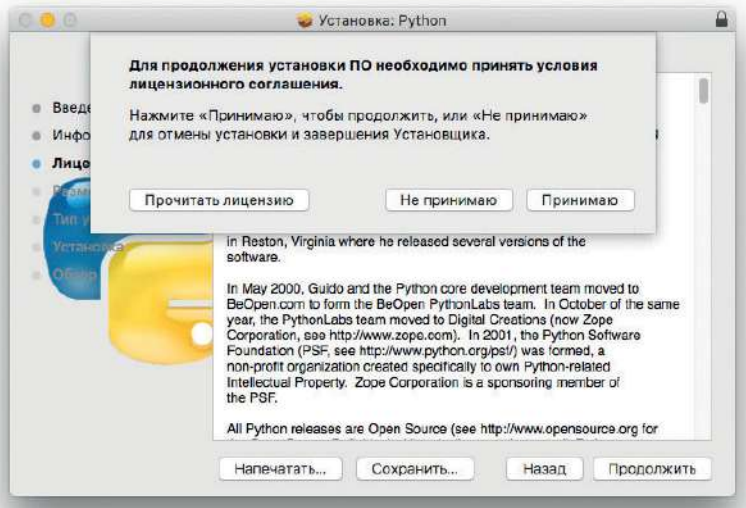


Рис. А.16. *Прочитайте и примите лицензионное соглашение, нажав кнопку* ***Принимаю*** *в диалоговом окне*

Рис. А.17. *Нажмите кнопку* ***Установить***

3. Прочитайте и примите лицензионное соглашение, нажав кнопку **Принимаю** (Agree) во всплывающем окне, как показано на рис. А.16.

4. В следующем окне нажмите кнопку **Установить** (Install), как показано на рис. А.17.

5. На экране должно появиться окно, подтверждающее завершение установки, наподобие того, что изображено на рис. А.18. Нажмите кнопку **Закрыть** (Close), чтобы выйти из установщика.

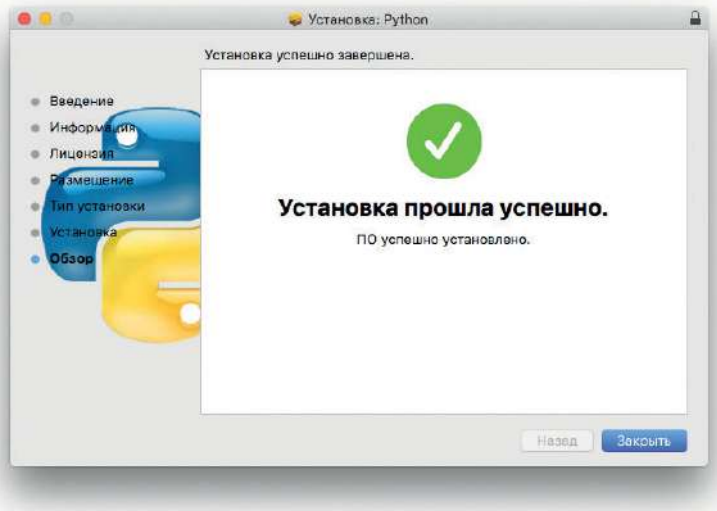


Рис. А.18. *Чтобы выйти из установщика, нажмите кнопку* ***Закрыть***

Вы установили Python! Теперь вы можете попробовать его, чтобы убедиться, что все работает.

Тестирование Python

1. Запустите приложение Launchpad и щелкните мышью по значку **IDLE** либо откройте в Finder папку **Приложения** (Applications), дважды щелкните мышью по папке **Python 3.4**, затем также дважды щелкните по значку **IDLE**, чтобы запустить оболочку Python, как показано на рис. А.19.

2. На экране должен появиться текстовый редактор оболочки Python. Все готово для того, чтобы вы попробовали программировать в оболочке. Введите команду `print("Здравствуй, Python!")` и нажмите клавишу **Return**. Оболочка Python должна ответить фразой

`Здравствуй, Python!`, как показано на рис. А.20. Также попробуйте ввести выражение сложения, например `2 + 3`. Нажмите клавишу **Enter** — и Python даст правильный ответ!

Рис. А.19. *Откройте IDLE из Launchpad (слева) или папки* ***Приложения*** *(справа)*

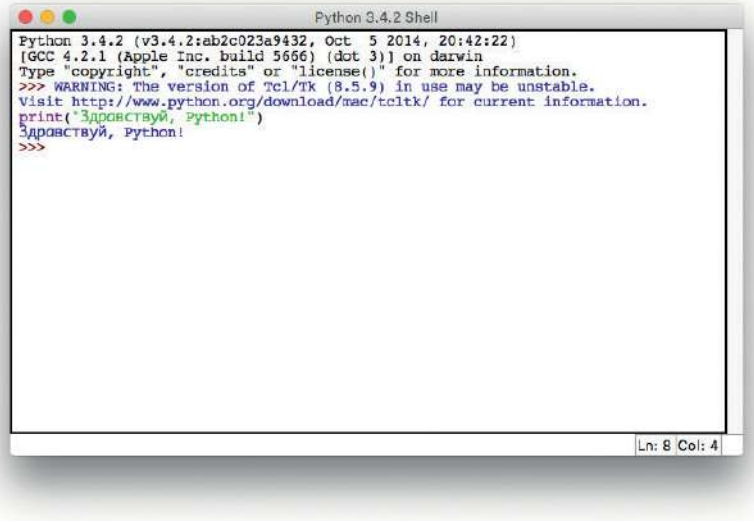


Рис. А.20. *Тестирование некоторых команд в оболочке Python*

3. Вы также можете изменить размер текста в окне IDLE, чтобы упростить чтение с экрана компьютера. Выполните команду **Options** ⇒ **Configure IDLE...** (Опции ⇒ Настроить IDLE...). На вкладке **Fonts/Tabs** (Шрифты/Отступы), изображенной на рис. А.21, измените параметр **Size** (Размер) на **20** или настройте размер шрифта так, чтобы вам было проще читать. Вы также можете установить флажок **Bold** (Жирный), чтобы сделать буквы толще. Выберите шрифт, который вам кажется более приятным для глаз.

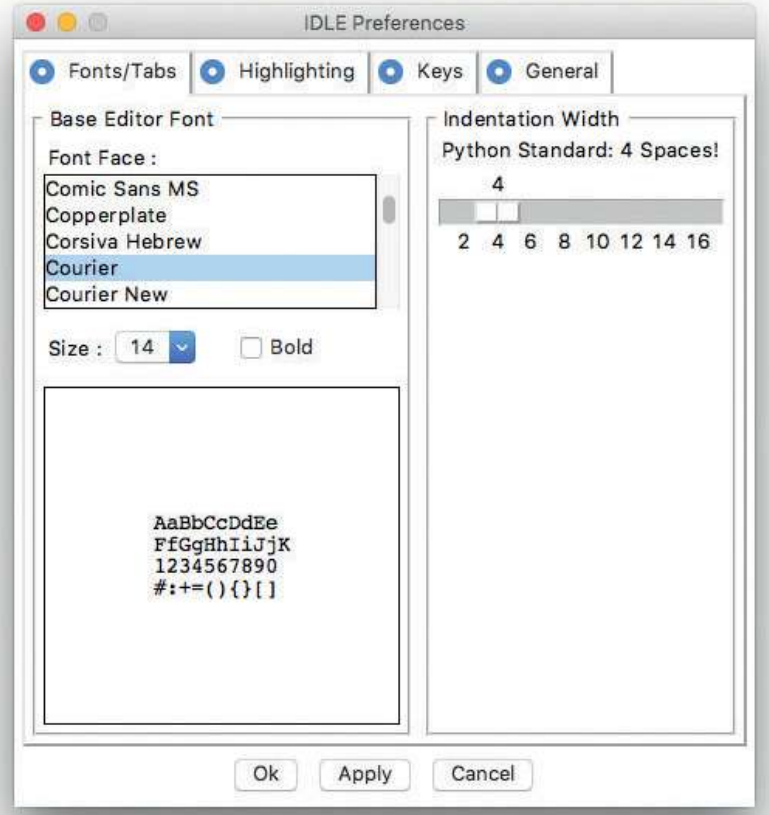


Рис. А.21. *Настройка предпочтений в программе IDLE*

4. После того как вы выбрали шрифт и установили размер, чтобы упростить чтение данных, вводимых в окно IDLE, нажмите кнопку **Apply** (Применить), а затем — кнопку **OK**, чтобы вернуться в окно оболочки Python IDLE. Теперь, выполняя ввод с клавиатуры, вы должны видеть на экране текст выбранного вами шрифта и размера.

Теперь все готово для изучения глав 1–7. Для использования программ из глав 8–10 перейдите к Приложению Б и выполните шаги для установки модуля Pygame. Счастливого программирования!

Python для Linux

Большинство дистрибутивов Linux, в том числе Ubuntu и даже Linux OS, распространяемый на микрокомпьютерах Raspberry Pi, поставляются с предустановленными более ранними версиями языка Python. Однако большинство приложений из этой книги требуют языка Python 3. Чтобы установить Python 3 на Linux, выполните следующие шаги.

1. Из основного меню запустите программу **Менеджер приложений Ubuntu** (Ubuntu Software Centre) или аналогичное приложение, использующееся в вашей версии Linux. На рис. A.22 показано приложение **Менеджер приложений Ubuntu** (Ubuntu Software Centre), запущенное в операционной системе Ubuntu.

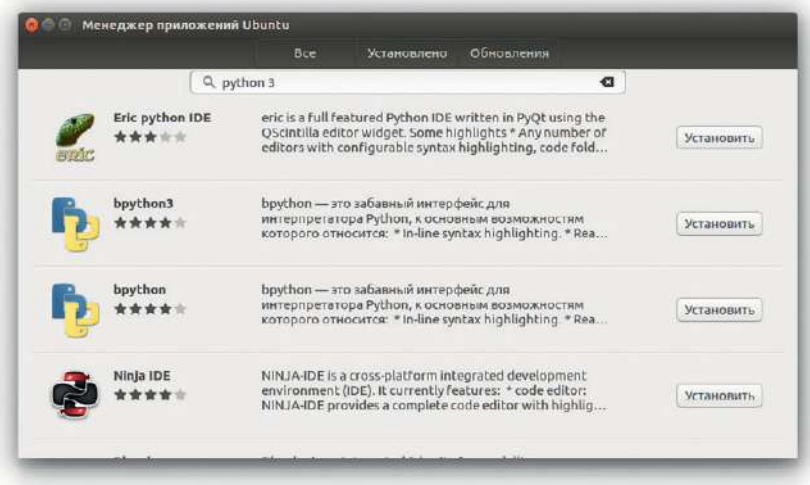


Рис. A.22. *Установка Python 3 на компьютер с операционной системой Ubuntu Linux*

2. Выполните поиск по запросу *python3* и найдите приложение *Eric Python IDE*. Щелкните мышью по кнопке **Установить** (Install).
3. По завершении установки на панели слева вы увидите значок **Eric Python IDE**, как показано на рис. A.23.

Рис. А.23. *Eric Python IDE — программа-оболочка Python*

4. Протестируйте программу Eric Python IDE, запустив ее, введя выражение `2 + 3` и нажав клавишу **Enter**. Введите команду `print("Здравствуй, Python!")` и нажмите клавишу **Enter**. Программа IDE должна вывести ответ, как показано на рис. А.24.

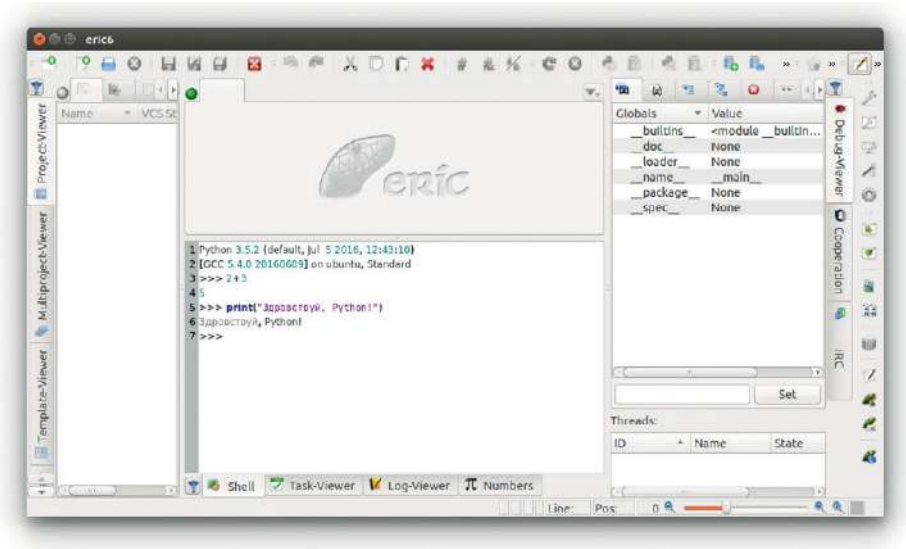


Рис. А.24. *Протестируйте Python, запустив Eric Python IDE. Вы готовы начать программировать!*

Вы готовы попробовать все программы из глав 1–7. Для использования программ из глав 8–10 см. Приложение Б, в котором описывается установка модуля Pygame для Linux. Счастливого программирования!

Приложение Б

УСТАНОВКА И НАСТРОЙКА PYGAME В СРЕДЕ WINDOWS, MACOS И LINUX

После установки Python (см. Приложение А) вам потребуется установить модуль Pygame, чтобы получить возможность запускать анимации и игры из глав 8–10. Это приложение поможет вам в установке и настройке. Если вы устанавливаете Pygame на компьютере в школе или на работе, возможно, для установки вам потребуется помощь или разрешение отдела IT. Если вы столкнулись с проблемами, попросите помощи в отделе IT.

Pygame для Windows

Для Windows мы будем использовать Pygame 1.9.2 для Python 3.2 (см. Приложение А для получения информации по установке Python 3.2.5).

1. Перейдите на сайт **pygame.org** и, как показано на рис. Б.1, в левой части сайта щелкните по ссылке **Downloads** (Загрузки).

2. В разделе Windows найдите ссылку на файл **pygame-1.9.2a0.win32-py3.2.msi** и щелкните мышью по ней для загрузки программы-установщика, как показано на рис. Б.2.

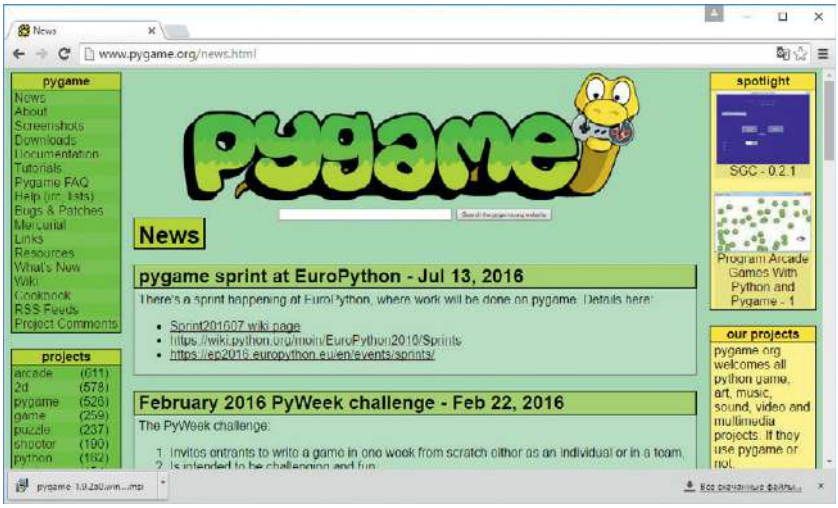


Рис. Б.1. *Щелкните по ссылке* ***Downloads***

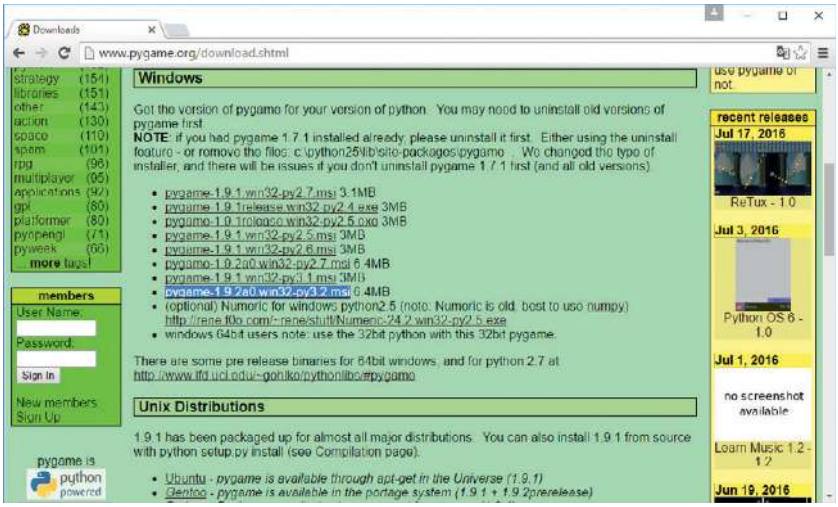


Рис. Б.2. *Загрузите программу-установщик для Windows*

3. По завершении загрузки перейдите в папку **Загрузки** (Downloads) и найдите файл установщика Windows **pygame-1.9.2a0.win32-py3.2**. как показано на рис. Б.3. Для начала установки дважды щелкните мышью по файлу. При появлении окна с предупреждением системы безопасности щелкните мышью по кнопке **Да** (Yes). Выводя это окно, Windows лишь пытается сообщить вам о том, какое программное обеспечение вы устанавливаете на свой компьютер.

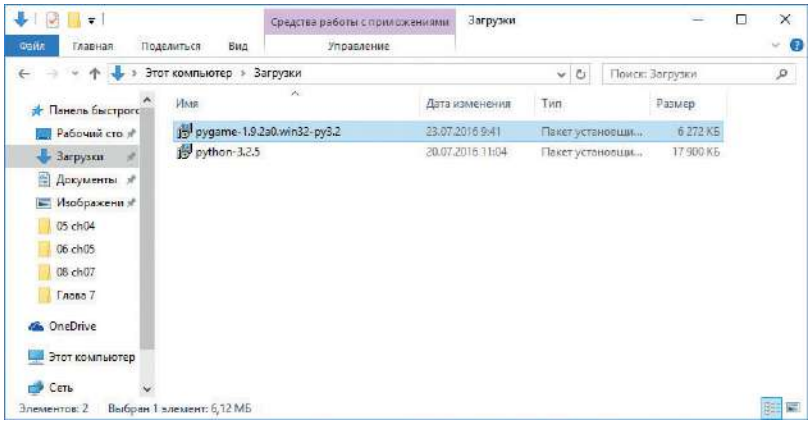


Рис. Б.3. *Дважды щелкните мышью по установщику в папке* ***Загрузки*** *(Downloads)*

4. Установщик может спросить у вас, желаете ли вы установить Pygame для всех пользователей или только для себя. Чаще всего лучше выбирать опцию **Install for all users** (Установить для всех пользователей), однако, если это запрещено у вас в школе, в офисе или у вас просто не получается выполнить корректную установку, попробуйте выбрать опцию **Install just for me** (Установить только для меня). Нажмите кнопку **Next** (Далее), как показано на рис. Б.4.

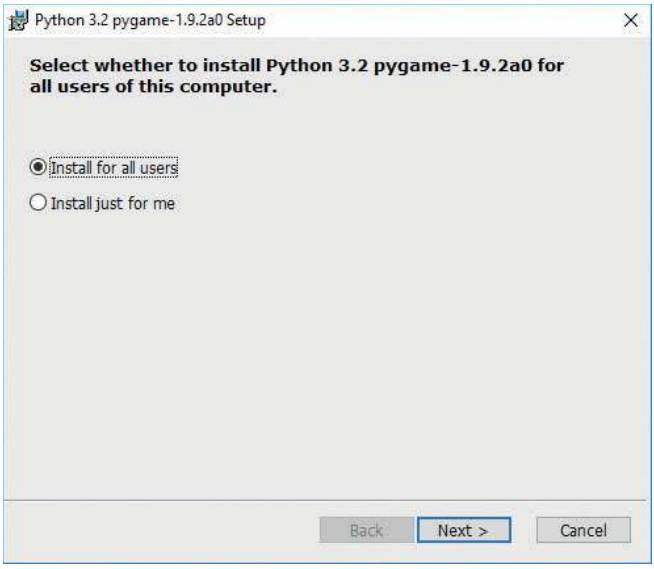


Рис. Б.4. *Установить для всех пользователей*

5. Программа должна автоматически найти установку Python 3.2.5, которую вы выполнили, прочитав Приложение А. Выберите опцию **Python 3.2.5 from registry** (Python 3.2.5 из реестра). Для продолжения установки нажмите кнопку **Next** (Далее), как показано на рис. Б.5. (Если вы выполняете установку в школе или на работе и столкнулись с трудностью, сотруднику вашего отдела IT, возможно, потребуется выбрать опцию **Python from another location** (Python из другого местоположения).)

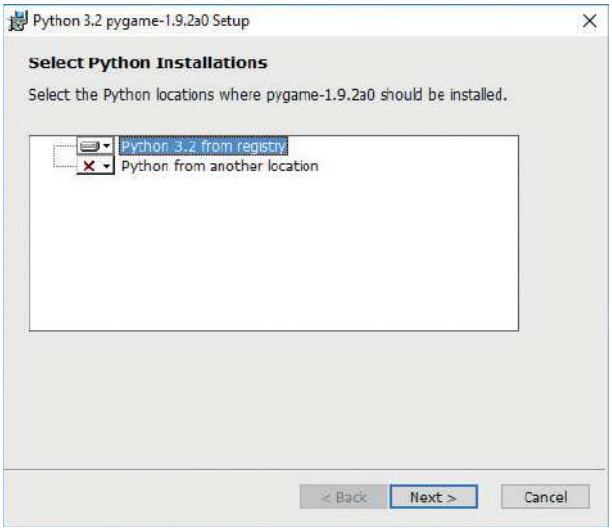


Рис. Б.5. *Выберите опцию* ***Python 3.2.5 from registry***

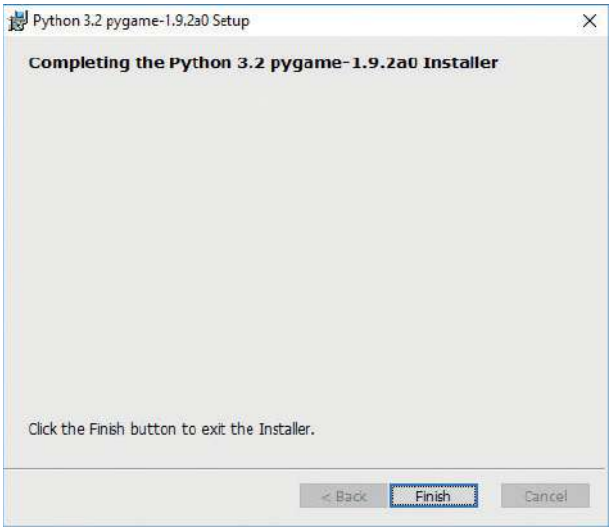


Рис. Б.6. *Нажмите кнопку* ***Finish*** *для выхода*

6. По завершении установки нажмите кнопку **Finish** (Закончить) для выхода, как показано на рис. Б.6.

7. Выполните команду **Start ⇒ Programs ⇒ Python 3.2 ⇒ IDLE (Python GUI)** (Пуск ⇒ Программы ⇒ Python 3.2 ⇒ IDLE (Python GUI)), как показано на рис. Б.7. (В Windows 8 или более поздних версиях вы можете нажать кнопку **Windows** или **Пуск** (Start), перейти к инструменту поиска и ввести с клавиатуры запрос `IDLE`.)

Рис. Б.7. *Запуск IDLE из меню* ***Пуск***

8. В текстовом редакторе оболочки Python введите команду `import pygame` и нажмите клавишу **Enter**. Оболочка Python должна будет

вывести ответ >>>, как показано на рис. Б.8. Если оболочка выводит именно такой ответ, значит, модуль Pygame установлен правильно и готов к использованию.

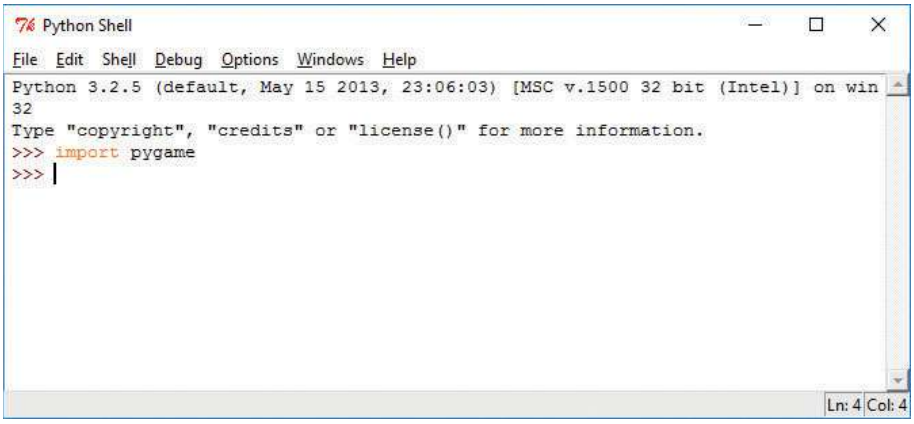


Рис. Б.8. *Импорт модуля Pygame в оболочку Python завершен успешно*

Теперь вы готовы к запуску программ из глав 8–10. Счастливого программирования!

Pygame для macOS

Установить модуль Pygame на Mac более сложно, чем на ПК. У вас есть три варианта.

1. Если у вас есть доступ к ПК с Windows, возможно, вам будет проще установить версии ПО Python и Pygame под Windows для запуска программ из глав 8–10. Если вы выберете эту опцию, выполните шаги, описанные в разделе «Python для Windows» на с. 298. Затем выполните шаги из раздела «Pygame для Windows» на с.316.
2. Вы можете установить в OS X более старую версию Python, например Python 2.7.9, вместе с модулем Pygame 1.9.2, чтобы запускать программы Pygame из глав 8–10. Установить Python 2.7.9 и модуль Pygame 1.9.2 значительно проще, чем заставить Pygame работать с Python 3.4.2. Однако между Python 2 и 3 есть существенные отличия, поэтому для работы с главами 1–7 я бы порекомендовал вам использовать Python 3.4.2, чтобы быть уверенными в том, что примеры из этих глав будут

работать. Затем для глав 8–10 вы можете использовать Python 2.7.9 с модулем Pygame 1.9.2 для запуска примеров с Pygame. Если вы решите выбрать эту опцию, выполните шаги из следующего раздела «Python 2.7 и Pygame 1.9.2».

3. Чтобы установить модуль Pygame для Python 3.4 на компьютер Mac, воспользуйтесь инструкциями на сайте **https://eksmo.ru/files/Python_deti.zip**. Если вы будете делать это в школе или на работе, то, скорее всего, вам потребуется помощь отдела IT. Передайте инструкции специалисту вашего отдела IT, чтобы он использовал их в качестве руководства.

Python 2.7 и Pygame 1.9.2

Более новые модели компьютеров Mac поставляются с предустановленной компанией Apple версией Python 2.7 в качестве части операционной системы OS X. Однако версия Python, предоставляемая Apple, может не работать с установщиком Pygame. Я рекомендую установить последнюю версию дистрибутива Python 2.7 с сайта **python.org**, прежде чем вы попробуете установить Pygame.

1. Чтобы установить Python 2.7 на свой компьютер Mac, вернитесь к Приложению А и выполните шаги из раздела «Python для Mac» на с. 308. Однако в этот раз вместо загрузки установщика 3.4.2 со страницы загрузок для Mac сайта **python.org** загрузите и запустите установщик версии 2.7 (2.7.12 на момент написания книги), как показано на рис. Б.9.
2. Процесс установки Python 2.7 не должен сильно отличаться от установки версии 3.4. Следуйте шагам из раздела «Python для Mac» до тех пор, пока не завершите установку.
3. Проверьте папку **Приложения** (Applications), в ней теперь должна содержаться папка **Python 2.7** вдобавок к уже имевшейся ранее **Python 3.4**, как показано на рис. Б.10.
4. Перейдите на страницу **pygame.org/download.shtml** и загрузите установщик модуля Pygame 1.9.2 для Python 2.7: **pygame-1.9.2pre-py2.7-macosx10.7.mpkg.zip**.
5. Запустите установщик Pygame, нажав и удерживая клавишу **Control**, щелкните мышью по файлу и из открывшегося меню выберите пункт

Открыть в программе ⇒ Установщик (Open with ⇒ Installer). Последующие шаги будут аналогичны установке Python: нажмите несколько раз кнопку **Продолжить** (Continue), примите лицензионное соглашение и выберите диск для установки. Нажмите кнопку **Закрыть** (Close), когда установщик завершит работу.

Рис. Б.9. *Установка Python 2.7*

Рис. Б.10. *У вас должны быть обе версии:* ***Python 2.7*** *и* ***Python 3.4***

6. Чтобы протестировать корректность установки Pygame, перейдите в папку **Приложения** (Applications), откройте каталог **Python 2.7** и запустите IDLE. В IDLE для Python 2.7 введите команду `import pygame`. Оболочка IDLE должна будет вывести ответ >>>, как показано на рис. Б.11.

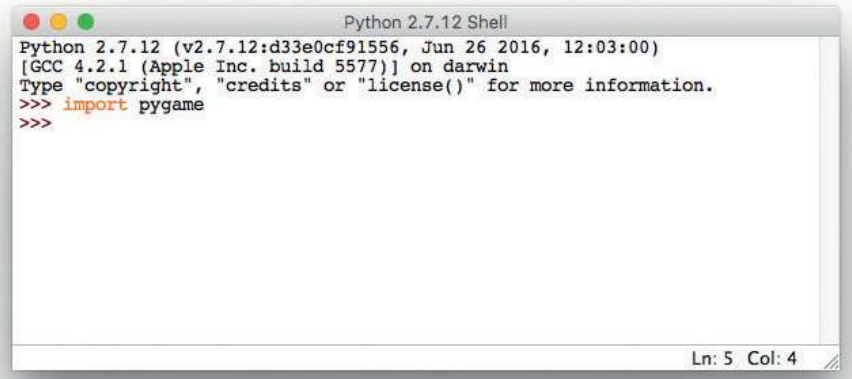


Рис. Б.11. *Импорт модуля Pygame в оболочку Python*

7. На экране может появиться всплывающее предупреждение, говорящее, что вы должны установить X11 — систему окон, используемую модулем Pygame. Нажмите кнопку **Продолжить** (Continue), чтобы перейти на сайт XQuartz, **xquartz.macosforge.org**. Загрузите и откройте файл *XQuarz.dmg*, запустите установщик и следуйте указаниям мастера установки.

8. Для запуска программ из глав 8–10 используйте IDLE Python 2.7 вместо IDLE Python 3.4.

На более новых моделях компьютеров Mac с дисплеями Retina модуль Pygame с Python 2.7 будет выглядеть несколько иначе, чем на других компьютерах, так как в дисплеях Retina используется более высокое разрешение. Ваши программы должны работать как нужно, просто они будут занимать меньшую площадь на экране.

Pygame для Linux

Аналогично с установкой Pygame на Mac при установке Pygame на Linux у вас есть две опции.

1. Вы можете установить Pygame для Python 2, версии языка Python, которая, скорее всего, будет предустановлена в вашей версии Linux. Для работы с главами 1–7 вам потребуется использовать Python 3, поэтому выполните инструкции из Приложения А и используйте эту версию IDE для создания приложений из первых семи глав. Затем для глав 8–10 вы можете воспользоваться модулем Pygame для Python 2, чтобы запускать приложения с примерами Pygame из этих глав. Если вы выберете эту опцию, выполните шаги из следующего раздела «Pygame для Python 2».

2. Чтобы установить Pygame для Python 3.4 под Linux, воспользуйтесь инструкциями на сайте **https://eksmo.ru/files/Python_deti.zip**. Если вы делаете это в школе или на работе, то вам, скорее всего, потребуется помощь отдела IT. Передайте инструкции специалисту вашего отдела IT, чтобы он использовал их в качестве руководства.

Pygame для Python 2

Большинство операционных систем Linux, поставляются с предустановленной средой Python, как правило, Python 2. Игровые и графические приложения из глав 8–10 отлично работают с этой, более старой, версией языка Python.

1. В основном меню перейдите в раздел **System Tools** (Системные) и запустите утилиту Менеджер пакетов Synaptic (Synaptic Package Manager) или аналогичное приложение, использующееся в вашей версии Linux. На рис. Б.12 показано окно менеджера пакетов дистрибутива Ubuntu.

2. Выполните поиск по запросу *python-pygame*. Установите флажок рядом с опцией *python-pygame* в результатах поиска и щелкните мышью по кнопке **Apply** (Применить), чтобы завершить установку

3. Запустите программу Терминал (Terminal) (XTerm или аналогичное приложение, использующееся в вашей версии Linux). Вы можете запустить Python 2, введя команду `python2` в окне терминала. Затем протестируйте корректность установки Pygame, введя в строке

с приглашением >>> команду `import pygame`, как показано на рис. Б.13. Python должен вывести ответ >>>, это позволит вам знать, что модуль Pygame был успешно импортирован.

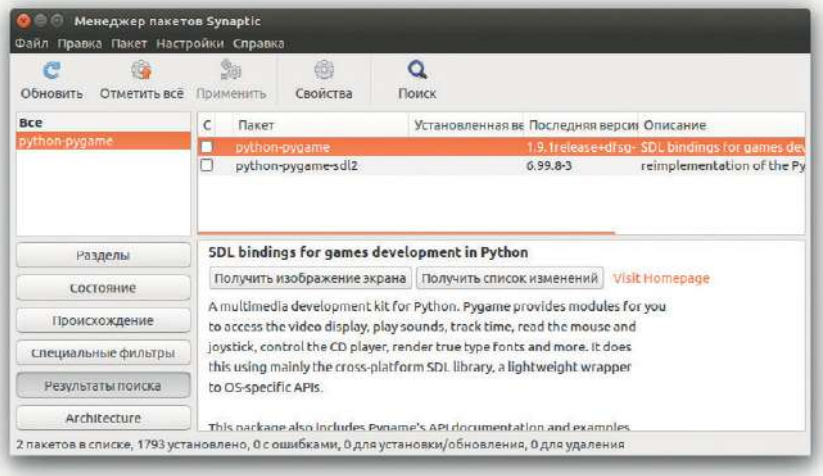


Рис. Б.12. *Установка Pygame для Python 2 под Linux*

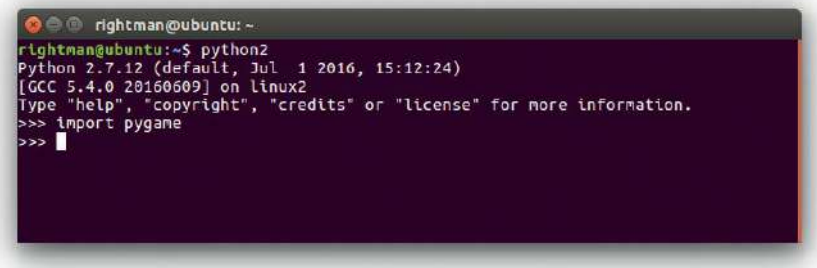


Рис. Б.13. *Вы можете протестировать корректность установки Pygame для Python 2 с помощью терминала командной строки Linux*

4. Для поиска и установки IDE для Python 2 вы можете воспользоваться как центром приложений (как показано в разделе «Python для Linux» на с. 314), так и менеджером пакетов Synaptic, изображенным на рис. Б.13. Используйте эту версию IDE для запуска приложений Pygame из глав 8–10.

Приложение B

СОЗДАНИЕ ВАШИХ СОБСТВЕННЫХ МОДУЛЕЙ

На протяжении всей книги вы импортировали различные модули, например `turtle`, `random` и `pygame`, в свои программы, чтобы добавить функции рисования, генерации случайных чисел, анимированной графики без необходимости создавать все эти функции с нуля. Однако знаете ли вы, что можете создавать собственные модули и затем импортировать их в свои программы? Python упрощает создание модулей для сохранения и последующего использования полезного кода во многих программах.

Для создания модуля, пригодного для повторного использования, мы должны написать такой модуль в окне редактора файлов IDLE, как мы создавали все прочие программы. Мы также должны сохранить модуль как файл *.py* с именем, совпадающим с названием модуля. Например, *colorspiral.py* могло бы быть именем модуля, рисующего на экране цветные спирали. В модуле мы определяем функции и переменные. Затем для повторного использования этих функций и переменных мы импортируем модуль в программу, введя команду `import` и имя модуля (например, команда `import colorspiral` позволила бы программе использовать код модуля *colorspiral.py* для отрисовки разноцветных спиралей).

Для того чтобы попрактиковаться в написании собственных модулей, давайте создадим настоящий модуль `colorspiral` и посмотрим, каким образом этот модуль помогает нам сэкономить время на повторном написании кода.

Создание модуля colorspiral

Давайте создадим модуль colorspiral, который помог бы нам легко и быстро рисовать спирали в собственных программах просто путем вызова import colorspiral. Введите следующий код в новое окно IDLE и сохраните как *colorspiral.py*.

colorspiral.py

```
❶ """Модуль для рисования цветных спиралей до 6 граней"""
  import turtle
❷ def cspiral(sides=6, size=360, x=0, y=0):
❸     """Рисует цветную спираль на черном фоне. ↵

      Аргументы: ↵
      sides — количество граней спирали (по умолчанию 6) ↵
      size — длина последней грани (по умолчанию 360) ↵
      x, y — местоположение спирали от центра экрана ↵
      """
      t=turtle.Pen()
      t.speed(0)
      t.penup()
      t.setpos(x, y)
      t.pendown()
      turtle.bgcolor("black")
      colors=["red", "yellow", "blue", "orange", "green", ↵
      "purple"]
      for n in range(size):
            t.pencolor(colors[n%sides])
            t.forward(n * 3/sides + n)
            t.left(360/sides + 1)
            t.width(n*sides/100)
```

В данном модуле выполняется импорт модуля turtle, а также определяется функция с именем cspiral() для рисования в разных участках экрана цветных спиралей различных форм и размеров. Давайте взглянем на отличия между этим модулем и теми программами, что мы создавали раньше. Во-первых, в строке ❶ мы поместили специальный комментарий, называемый также *строкой документации*. Строка документации — это один из способов добавления пояснений в файл, который мы хотели бы использовать повторно или которым мы желаем поделиться с другими людьми. Модули Python должны содержать строки документации, помогающие другим пользователям понять, какие

функции выполняет данный модуль. Строка документации всегда будет первым выражением в модуле или функции, каждая строка документации начинается и заканчивается *тремя двойными кавычками* (`"""`, три двойные кавычки подряд без пробелов). После строки документации мы импортируем модуль `turtle` — да, мы можем импортировать модули в наши модули!

В строке ❷ мы определяем функцию `cspiral()`, которая принимает до четырех аргументов: `sides`, `size`, `x`, `y`, определяющие соответственно количество сторон спирали, ее размер и местоположение (x, y) относительно центра экрана `turtle`. Строка документации функции `cspiral()` начинается в строке ❸. Такое многострочное документирование дает более детальную информацию о функции. Строка документации начинается с трех двойных кавычек. Первая строка описывает функцию в целом. Далее следует пустая строка, после которой приводится перечень принимаемых функцией аргументов. Такое документирование позволит будущему пользователю легко прочитать и узнать, какие аргументы принимаются функцией и что означает каждый из них. Оставшаяся часть функции — это код для рисования разноцветной спирали, наподобие кода из глав 2, 4 и 7.

Использование модуля `colorspiral`

После завершения написания и сохранения файла *colorspiral.py* мы можем использовать его в качестве модуля и импортировать в другие программы. В IDLE создайте новый файл и сохраните его в файл с именем *MultiSpiral.py* в той же папке, что и файл *colorspiral.py*.

MultiSpiral.py

```
import colorspiral
colorspiral.cspiral(5,50)
colorspiral.cspiral(4,50,100,100)
```

Эта трехстрочная программа импортирует модуль `colorspiral`, который мы создали, и использует функцию `cspiral` из этого модуля для отрисовки двух спиралей на экране, как показано на рис. B.1.

Благодаря модулю `colorspiral` каждый раз, когда программист захочет создать разноцветные спирали, все, что ему нужно сделать, — импортировать данный модуль и вызывать функцию `colorspiral.cspiral()`!

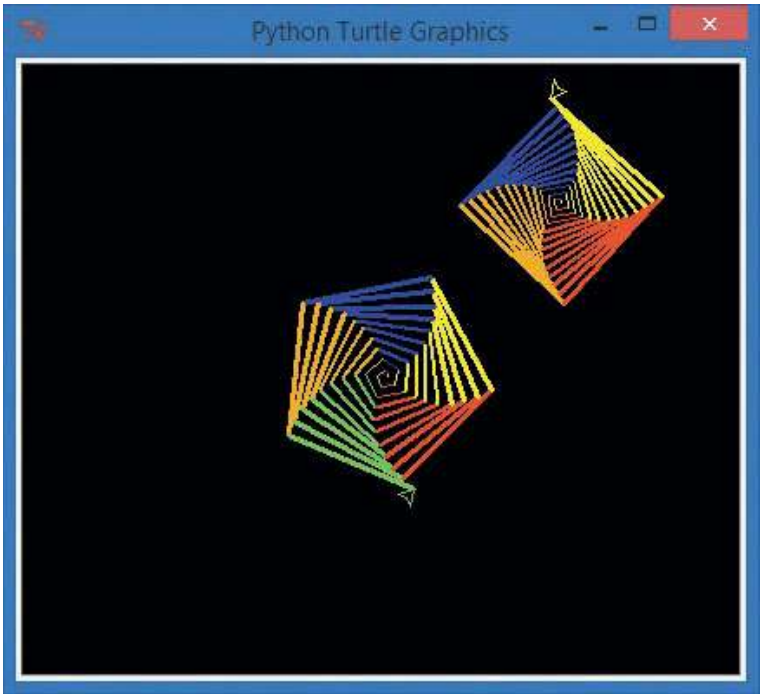


Рис. B.1. *Две разноцветные спирали, созданные с помощью трехстрочной программы и благодаря модулю colorspiral.py*

Повторное использование модуля `colorspiral`

Давайте повторно используем модуль `colorspiral` для отрисовки 30 случайных разноцветных спиралей. Чтобы сделать это, мы импортируем еще один модуль, который мы также уже использовали ранее, `random`. Введите следующие восемь строк кода в окно нового файла IDLE и сохраните его в файл с именем *SuperSpiral.py*.

SuperSpiral.py

```
import colorspiral
import random
for n in range(30):
    sides = random.randint(3,6)
    size = random.randint(25,75)
    x = random.randint(-300,300)
    y = random.randint(-300,300)
    colorspiral.cspiral(sides, size, x, y)
```

Эта программа начинается с двух выражений `import`: одна для созданного нами модуля `colorspiral`, а вторая — для модуля `random`, который мы использовали на протяжении почти всей книги. Цикл `for` будет выполнен 30 раз. В цикле генерируются четыре случайных числа, задающих количество сторон (от 3 до 6), размер спирали (от 25 до 75), а также координаты *x* и y в диапазоне от (–300, –300) до (300, 300) для рисования спирали на экране. (Помните, что точка начала координат модуля Turtle (0,0) находится в центре экрана для рисования.) Наконец, при каждом проходе по циклу выполняется вызов функции `colorspiral.cspiral()` из созданного нами модуля, что приводит к отрисовке цветных спиралей со случайным образом сгенерированными атрибутами из цикла.

Несмотря на то что эта программа состоит всего лишь из восьми строк, она производит потрясающие графические узоры, как тот, что изображен на рис. B.2.

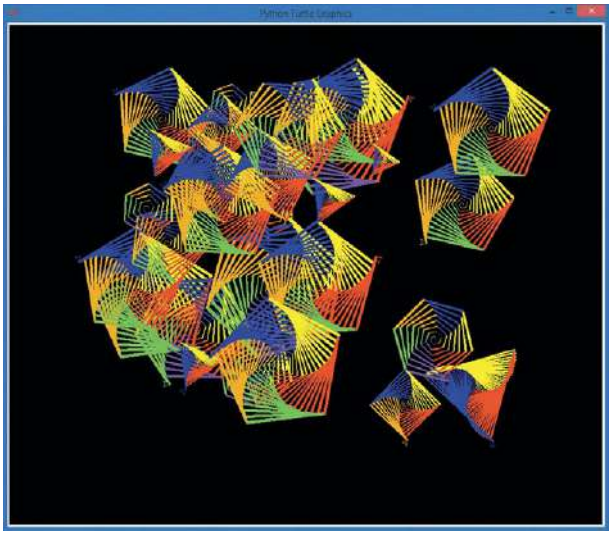


Рис. B.2. *Модуль `colorspiral` позволяет программе SuperSpiral.py создавать красивый коллаж из нескольких спиралей с помощью лишь восьми строк кода*

Возможность создавать модули для повторного использования означает, что вы можете посвятить больше времени решению новых задач, а не повторному написанию программного кода старых решений. Каждый раз при создании полезной функции или набора функций, которые вы хотите

использовать вновь и вновь, вы можете создать модуль, использовать его сами или поделиться с другими программистами.

Дополнительные ресурсы

Официальная документация по Python на сайте **docs.python.org/3/** содержит еще больше информации о модулях и языке Python в целом. **Python Tutorial** (Руководство по Python) также содержит раздел, посвященный модулям: **docs.python.org/3/tutorial/modules.html**. По мере осваивания новых навыков программирования на языке Python используйте эти ресурсы для обогащения вашего набора инструментов программирования.

Приложение Г

УСТАНОВКА PYGAME ДЛЯ PYTHON 3 В СРЕДЕ MACOS И LINUX

В этом приложении вы найдете инструкции по установке Pygame для Python 3 на Mac (с. 333) и Linux (с.342). Эти процессы установки могут быть сложными, поэтому, если вам нужна помощь, обратитесь к вашему специалисту по IT или другому человеку, обладающему необходимыми знаниями. Более простые инструкции по использованию Pygame для Python 2 вы можете найти в Приложении Б.

Pygame для Python 3.4 в среде macOS

Следующие шаги могут оказаться очень сложными, поэтому не пытайтесь выполнить их самостоятельно, лучше передайте их программисту Python или помогающему вам специалисту по IT.

Подготовка к командам терминала

1. **Установка Xcode.** Запустите программу App Store. Найдите приложение *Xcode* и щелкните мышью по кнопке **Загрузить** (Download) для установки инструментов разработчика Xcode, как показано на рис. Г.1. Вам понадобятся эти инструменты для выполнения некоторых команд установки.

Рис. Г.1. *Установка Xcode из App Store*

2. **Установка XQuartz.** Перейдите на сайт по адресу **xquartz.macosforge.org/** и загрузите самую новую версию приложения XQuartz (на момент написания — версия 2.7.9), как показано на рис. Г.2. Откройте папку **Загрузки** (Downloads), дважды щелкните мышью по файлу **XQuartz-2.7.9.dmg**, затем дважды щелкните по файлу пакета **XQuartz.pkg** — и проследуйте инструкциям для завершения установки.

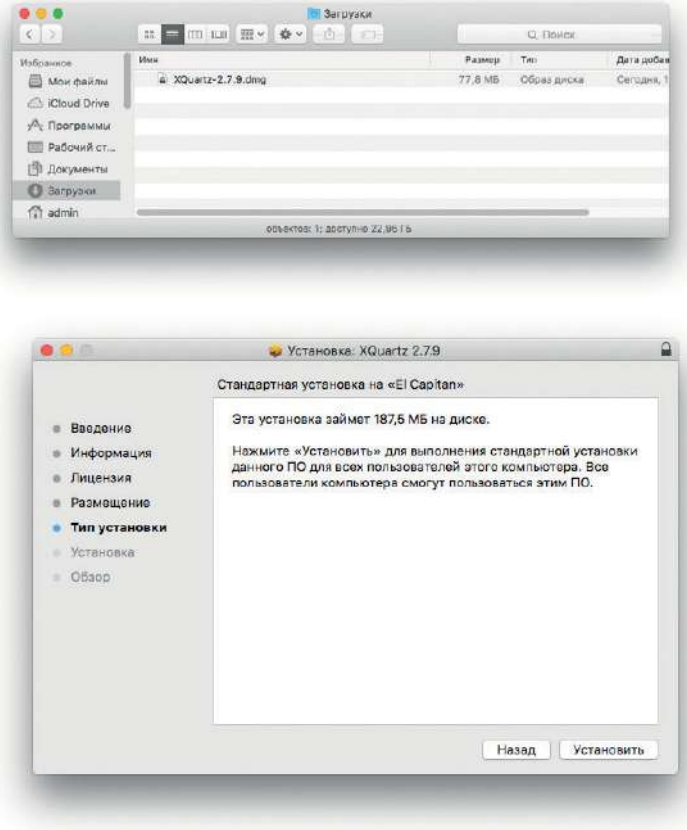


Рис. Г.2. *Установка XQuartz*

1. Откройте окно терминала (командная строка). Чтобы сделать это, перейдите в папку **Приложения ⇒ Утилиты** (Applications ⇒ Utilities) и дважды щелкните мышью по значку приложения **Terminal** (Терминал), как показано на рис. Г.3. Откроется окно терминала.

Рис. Г.3. *Откройте окно терминала*

Далее вы введете в терминал несколько команд для установки программ, помогающих установить модуль Pygame.

Ввод команд терминала

В следующих нескольких шагах вы введете в терминал в общей сложности 10 команд. Вводите все команды по одной, строго так, как показано в тексте, по окончании ввода нажимайте клавишу **Return**. Выполнение некоторых команд может занять несколько минут, а на экран может быть выведено большое количество текста.

1. **Установка Homebrew.** Homebrew – это бесплатная программа, помогающая установить на Mac Python, Pygame и другие программы. В строке приглашения терминала введите одной строкой следующую команду и нажмите клавишу **Return**:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/master/install) "
```

(Возможно, вам понадобится расширить окно терминала, чтобы эта команда влезла в одну строку, однако, если произойдет ее автоматический перенос по границам окна, в этом нет ничего страшного.)

2. **Подготовка Homebrew к использованию.** В строке приглашения терминала введите все три следующие команды точно так, как показано в тексте. Выполнение двух последних команд может потребовать некоторого времени, а в окне будет отображено несколько экранов информации.

```
echo export PATH='/usr/local/bin:$PATH' >> ~/.bash_profile
brew update
brew doctor
```

3. **Установка Python 3 для Pygame.** В строке приглашения терминала введите:

```
brew install python 3
```

4. **Установка Mercurial.** Mercurial — это бесплатная система управления и контроля ресурсов, которая требуется для установки Pygame на Mac. В строке приглашения терминала введите:

```
brew install mercurial
```

5. **Установка зависимостей Pygame.** Для корректной работы Pygame требуется наличие нескольких вспомогательных программ, называемых *зависимостями*, позволяющих Pygame отображать анимацию, воспроизводить звуки и создавать графику. В строке приглашения терминала введите все три следующие команды, не забывая нажимать клавишу **Return** после каждой команды:

```
brew install sdl sdl_image sdl_mixer sdl_ttf portmidi
brew tap homebrew/headonly
brew install —HEAD smpeg
```

На выполнение каждой из этих команд потребуется время, и в окне каждый раз будет показано несколько экранов с информацией. Продолжайте, вы почти закончили.

6. **Установка Pygame.** Введите следующую команду в строке приглашения терминала и нажмите клавишу **Return**:

```
sudo pip3 install hg+bitbucket.org/pygame/pygame
```

7. Возможно, вам потребуется ввести пароль администратора (используйте свой пароль или спросите у системного администратора). Последующий процесс установки может занять несколько минут.

Создание ярлыка на рабочем столе

Только что установленные Pygame и новый Python 3 создают отдельный редактор IDLE, который вы будете использовать специально для создания приложений Python. (Вы можете использовать эту новую версию IDLE для любой программы из книги, однако для написания и запуска приложений Pygame вы *должны* использовать именно эту версию.)

1. В программе Finder выберите команду меню **Переход** ⇒ **Переход к папке** (Go ⇒ Go to Folder), как показано на рис. Г.4.

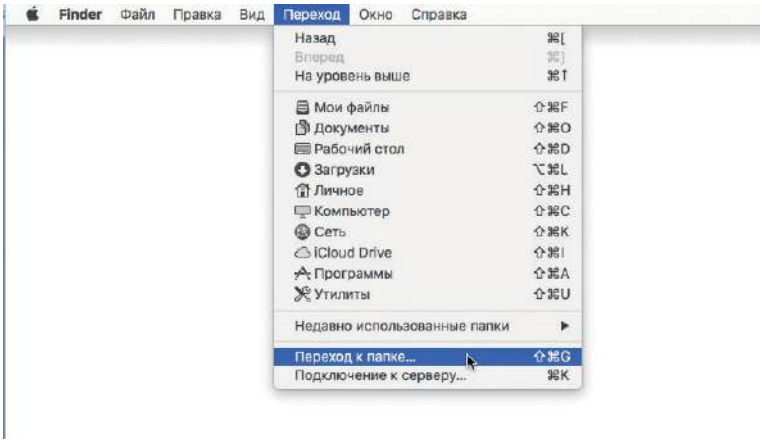


Рис. Г.4. *Выберите команду* ***Переход к папке*** *в меню* ***Переход***

2. В текстовое поле диалогового окна **Переход к папке** (Go to Folder) введите путь **usr/local/Cellar/python3** и нажмите кнопку **Перейти** (Go), как показано на рис. Г.5.

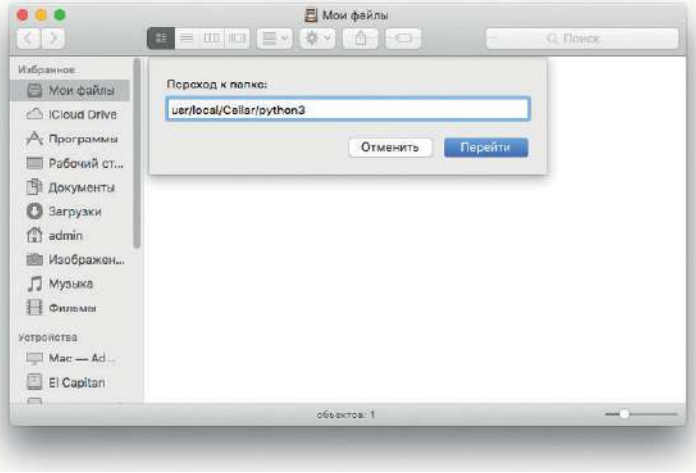


Рис. Г.5. *Введите путь* ***usr/local/Cellar/python3*** *и нажмите кнопку* ***Перейти***

3. Дважды щелкните мышью по папке **python3**, чтобы отобразить содержащуюся внутри папку, как показано на рис. Г.6.

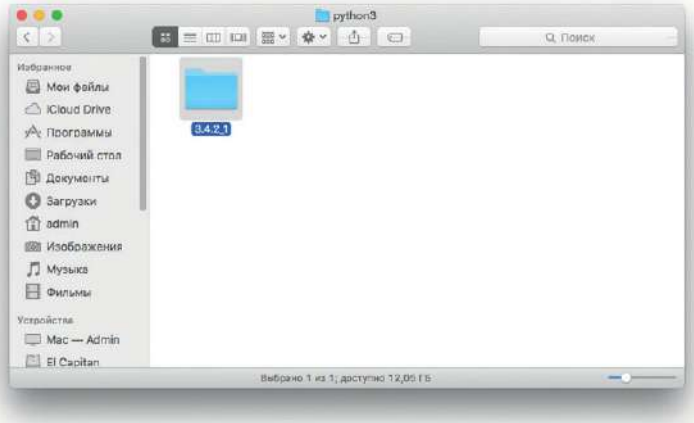


Рис. Г.6. *Дважды щелкните мышью по папке* ***python3****,*
чтобы отобразить содержащуюся внутри папку

4. В этой папке вы найдете приложение IDLE3. Нажав и удерживая кнопку **Control**, щелкните мышью по значку приложение IDLE3. Из открывшегося контекстного меню выберите команду **Создать псевдоним** (Make Alias), как показано на рис. Г.7.

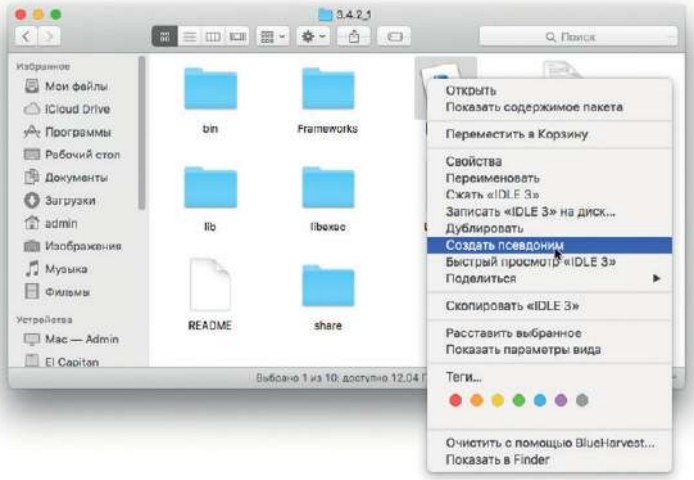


Рис. Г.7. *Выберите команду* ***Создать псевдоним*** *из открывшегося контекстного меню*

5. В папке появится новый *псевдоним*, или значок ярлыка, с именем, наподобие показанного на рис. Г.8.

Рис. Г.8. *Создан новый псевдоним*

6. Щелкните по имени файла, чтобы отредактировать его. Измените имя псевдонима на *pygame IDLE* или нечто подобное, как показано на рис. Г.9, помогающее запомнить, что для этой версии IDLE установлен модуль Pygame.

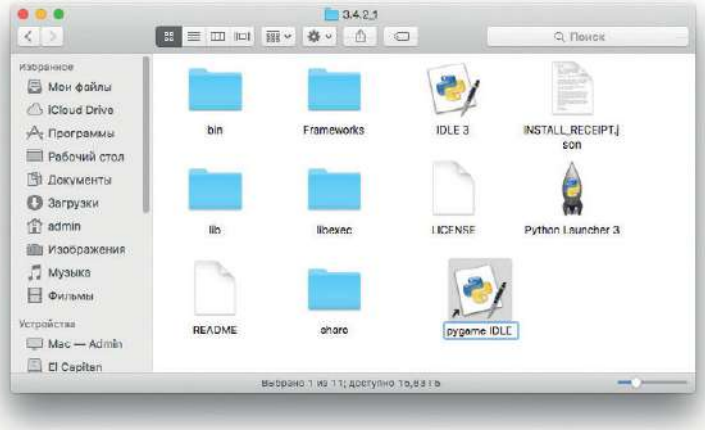


Рис. Г.9. *Назовите новый псевдоним pygame IDLE*

7. Перетащите значок ярлыка **Pygame IDLE** на рабочий стол, как показано на рис. Г.10. Это позволит вам получать доступ к нужной версии IDLE для Pygame прямо с рабочего стола.

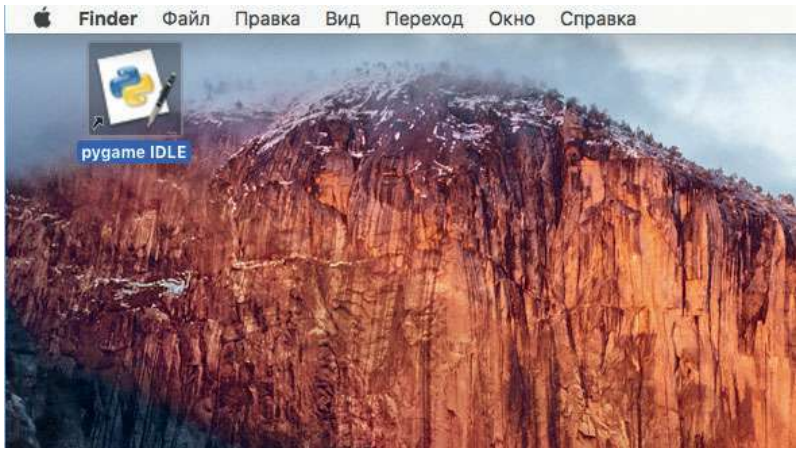


Рис. Г.10. *Перетащите* ***Pygame IDLE*** *на рабочий стол*

8. Дважды щелкните мышью по значку **Pygame IDLE**. Откроется окно редактора IDLE с поддержкой Pygame. Введите `import pygame` и нажмите клавишу **Return**, как показано на рис. Г.11. Редактор IDLE должен будет ответить приглашением >>> и не сообщить об ошибках.

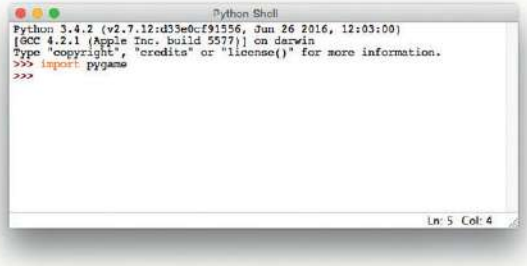


Рис. Г.11. *Импортируйте Pygame. Редактор IDLE должен будет ответить приглашением >>> и не сообщить об ошибках*

Теперь вы готовы программировать на вашем Mac приложения Pygame из глав 8–10. Приятного программирования!

Pygame для Python 3 в среде Linux

Следующие шаги могут оказаться очень сложными, поэтому не пытайтесь выполнить их самостоятельно, лучше передайте их программисту Python или помогающему вам специалисту по IT.

В первую очередь выполните указания из раздела Python для Linux (Приложение А, с. 298), чтобы установить IDE для Python 3.5 или более новой версии. Затем проследуйте нижеприведенным инструкциям для установки Pygame для Python 3 на компьютер с системой Linux.

1. Запустите программу Терминал (Terminal) (XTerm или аналогичное приложение, использующееся в вашей версии Linux).
2. **Установите зависимости Pygame.** В окно терминала введите следующие четыре строки кода точно так, как показано в тексте, и нажмите **Enter**, как показано на рис. Г.12.

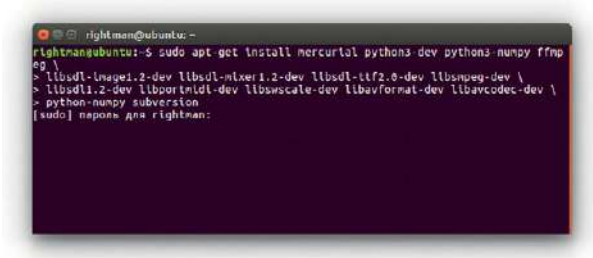


Рис. Г.12. *Установка Pygame для Python 3 для Linux требует ввода нескольких строк команд и наличия некоторых навыков работы с терминалом*

```
sudo apt-get install mercurial python3-dev python3-numpy ↵
ffmpeg libsdl-image1.2-dev libsdl-mixer1.2-dev ↵
libsdl-ttf2.0-dev libsmpeg-dev libsdl1.2-dev ↵
libportmidi-dev libswscale-dev libavformat-dev ↵
libavcodec-dev python-numpy subversion
```

3. **Скачайте и установите Pygame.** По окончании установки зависимостей введите следующие пять строк кода, поочередно введите следующие пять команд в строку приглашения терминала (обратите внимание, что версия языка Python в первой строке на вашем компьютере может отличаться и быть более поздней (в примере 3.5)!):

```
cd /usr/local/lib/python3.5/dist-packages/
sudo svn co svn://seul.org/svn/pygame/trunk pygame
cd pygame
sudo python3 setup.py build
sudo python3 setup.py install
```

4. По окончании установки Pygame откройте оболочку Python IDE.
5. Протестируйте установку Pygame для Python 3 на Linux, введя команду `import pygame` в строке приглашения >>> и нажмите клавишу **Enter**, как показано на рис. Г.13. Python должен ответить строкой >>>, что даст вам понять об успешном импорте модуля Pygame.

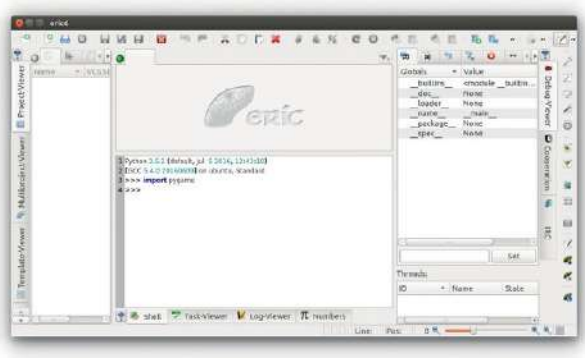


Рис. Г.13. *Тестирование Pygame для Python 3 с использованием IDE. Теперь вы готовы программировать приложения Pygame на Linux*

Теперь вы готовы использовать программы Pygame из глав 8–10 на вашей машине Linux. Счастливого программирования!

ГЛОССАРИЙ

Большинство терминов, с которыми вы встретились при изучении программирования, — это слова из повседневной жизни, которые вы и так уже понимаете. Однако некоторые из этих терминов — это новые слова или слова, имеющие особое значение в сфере компьютерного программирования. В этом глоссарии определены некоторые из новых слов, с которыми вы встретитесь в этой книге, а также знакомые слова, имеющие отдельное значение в мире программных кодов.

Алгоритм — набор шагов, необходимых для выполнения задания, например рецепт.

Анимация — иллюзия движения, создаваемая, когда похожие изображения по очереди быстро сменяют друг друга, как в мультфильме.

Аргумент — значение, передаваемое функции. В выражении `range(10)` `10` — это аргумент.

Блок — группа программных выражений.

Булевы значения — значение или выражение, которое может быть либо Истинным (True), либо Ложным (False).

Ввод — любые данные или информация, введенные в компьютер. Ввод может осуществляться с помощью компьютера, мыши, микрофона, цифровой камеры или иного устройства ввода.

Вложенный цикл — цикл внутри другого цикла.

Выражение — любой допустимый набор значений, переменных, операторов и функций, выполнение которого приводит к какому-то результату.

Диапазон — упорядоченный набор значений от известного начального до известного конечного значения. В языке Python функция `range` возвращает последовательность значений, например от `0` до `10`.

Импорт — перенос в программу кода, пригодного для повторного использования, из другой программы или модуля.

Индекс — позиция элемента в списке или массиве.

Инициализация — присвоение переменной или объекту их первого, или *начального*, значения.

Итеративное создание версий — пошаговое внесение небольших изменений в программу и сохранение изменений в качестве новых версий, например **Игра1**, **Игра2** и так далее.

Кадр — единичное изображение в подвижной последовательности для анимации, видео или компьютерной графики.

Кадров в секунду (fps) — скорость отрисовки изображений на экране для создания анимации в игре или фильме.

Класс — заготовка, определяющая функции и значения, которые будут храниться в любом объекте этого типа.

Ключевое слово — специальное, или зарезервированное, слово, имеющее некое значение на данном конкретном языке программирования.

Код — выражения или инструкции, написанные программистом на языке, понимаемом компьютером.

Конкатенация — объединение двух строк текста в одну.

Константа — поименованная переменная в памяти компьютера, значение которой остается неизменной, например `math.pi` (3.1415…).

Массив — упорядоченный список значений или объектов, как правило, одного типа, доступ к которым осуществляется по *индексу*, или положению в списке.

Модуль — файл или набор файлов со связанными переменными, функциями или классами, которые могут быть повторно использованы в других программах.

Обнаружение столкновений — проверка, не соприкасаются ли или не *сталкиваются* ли два виртуальных объекта на экране, как шарик и ракетка в игре Pong.

Оболочка — текстовая командная строка, читающая команды пользователя и выполняющая их. IDLE — это оболочка Python.

Объект — переменная, содержащая информацию об одной сущности класса, например одного спрайта из класса `Sprite`.

Объявление — выражение или группа выражений, сообщающие компьютеру, что означает имя переменной или функции.

Параметр — входящая переменная функции, указываемая во время определения функции.

Переменная — в компьютерном программировании — это поименованное значение, которое может быть изменено.

Пиксель — сокращение от английского словосочетания *picture element* — «элемент картинки», маленькие цветные точки, составляющие изображения на экране компьютера.

Подстановка — добавление чего-то в конец, например добавление букв к концу строки или добавление элементов в конец списка или массива.

Приложение — компьютерная программа, делающая что-то полезное (или веселое!).

Присвоение — установка значения переменной, как, например, в выражении `x = 5`, в котором переменной `x` присваивается значение `5`.

Программа — набор инструкций, написанных на понятном для компьютера языке.

Прозрачность — в графике — возможность видеть сквозь участки изображения.

Псевдослучайность — одно значение из последовательности значений, кажущееся случайным и непредсказуемым, такое значение обладает достаточной степенью случайности для симуляции броска игральной кости или монетки.

Синтаксис — грамматические и орфографические правила языка программирования.

Случайные числа — непредсказуемая последовательность чисел, равно распределенных по ограниченному диапазону.

Событие — активность, которую может обнаружить компьютер, например щелчок мышью, изменение значения, нажатие на клавишу, отсчет времени таймером и так далее. Выражения или функции, отвечающие на события, называются *обработчиками событий* или *слушателями событий*.

Строка — последовательность символов, в том числе букв, чисел, символов, знаков пунктуации и пробелов.

Условное выражение — выражение, позволяющее компьютеру проверить некое значение и выполнить разные действия в зависимости от результатов проверки.

Файл — набор данных или информации, сохраненный компьютером на некоем запоминающем устройстве, например жестком диске, DVD или USB-флеш-карте.

Функция — поименованный набор программных выражений, пригодных для выполнения конкретной задачи.

Цвет RGB — сокращение от *Цвет red-green-blue* (красный-зеленый-голубой), означает способ представления цветов: задается количество красного, зеленого и голубого компонентов, которые могут перемешиваться для воссоздания нужного цвета.

Цикл `for` — программное выражение, позволяющее повторять блок кода установленное количество раз.

Цикл `while` — программное выражение, позволяющее повторно выполнять блок кода до тех пор, пока верно определенное условие.

Цикл — набор инструкций, повторяемых до тех пор, пока не будет выполнено определенное условие.

Элемент — единица списка или массива.

ОБ АВТОРЕ

Доктор Брайсон Пэйн — заслуженный профессор компьютерной науки Университета Северной Джорджии, где он обучает целеустремленных программистов на протяжении более 15 лет. Его студенты построили успешную карьеру в таких компаниях, как Blizzard Entertainment, Riot Games, Equifax, CareerBuilder и др. Он был первым заведующим кафедрой компьютерной науки в Университете Северной Джорджии, кроме того, является обладателем степени доктора наук (PhD) от Государственного университета штата Джорджия. Доктор Пэйн также интенсивно сотрудничает со школами К-12 в области продвижения технического образования.

Доктор Пэйн обладает более чем 30-летним опытом программирования. Первую свою программу он продал за 10 долларов журналу *RUN* (Commodore 64), опубликованную в колонке «Магия» в 1985 году.

Доктор Пэйн живет к северу от Атланты, штат Джорджия, с женой Бев и двумя сыновьями, Алексом и Максом.

ОБ ИЛЛЮСТРАТОРЕ

Миран Липовача — автор книги «*Изучай Haskell во имя добра!*». Он увлекается боксом, игрой на бас-гитаре и, конечно, рисованием. Его привлекают танцующие скелеты и число 71, а также, проходя через автоматические двери, он представляет, что открывает их усилием собственной мысли.

БЛАГОДАРНОСТИ

Написать эту книгу было бы невозможно без недюжинной помощи и поддержки команды издательства No Starch Press. Выражаю особую благодарность Биллу Поллоку за то, что он верил в этот проект, Тайлеру Ортману за поддержку и редактуру, а также Лесли Шень, Райли Хоффману, Ли Аксельроду, Макензи Долгиноу, Серене Янг и Лаурель Чунь за неустанное редактирование, вычитывание, маркетинг и производственные навыки, а также за бессчетное количество советов, которые помогли мне улучшить эту книгу в сравнении с изначальной рукописью. Также спасибо Рейчел Монаган и Пауле Флеминг за помощь в исправлении формата и стиля и выверку текста.

Выражаю благодарность Мишель Френд и Эри Ласенски за внимательную и дотошную техническую редактуру, а также Конору Сенг за то, что он был первым, кто прочел эту книгу и протестировал предложенные в ней программы — в свои 10 лет.

Спасибо Мирану Липовача за восхитительные иллюстрации, они делают текст таким живым, каким я даже не мечтал его увидеть.

Спасибо моему свекру Норманну Пети, компьютерщику на пенсии, который начал самостоятельно изучать язык Python, используя ранние черновики этой книги.

Отдельная благодарность моей жене и лучшему другу Бев за постоянную поддержку, а также моим восхитительным сыновьям Алексу и Максу за помощь в тестировании каждой программы и советы по их улучшению. Эта книга и вся моя жизнь стали бесконечно лучше благодаря вам троим.

Наконец, хочу поблагодарить мою маму, Эсту, благодаря которой я полюбил изучать и решать головоломки.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

__init__()
функция *254*

B

blit()
метод *215*

C

circle()
функция *74*
Clock
класс *221*
collide_circle()
функция *260*
collide_rect()
функция *260*

D

def
ключевое слово *172*

E

elif
выражение *114*
else
выражение *107*

F

font.render()
функция *279*
FPS
количество кадров в секунду *218*

G

Group
класс *252*
GUI *206*

H

hideturtle()
функция *182*

I

if
выражение *100*
сложные условия *116*
if-elif-else
выражение *114*
if-else
выражение *108*
import
команда *31*
index()
функция *149*
islower()
функция *121*
isupper()
функция *121*

L

list()
функция *75*
lower()
функция *121*

M

MOUSEBUTTONDOWN
событие *244*
MOUSEBUTTONUP
событие *245*

O

onkeypress()
обработчик событий, функция *192*
onscreenclick()
обработчик событий, функция *188*

P

pygame
модуль *208*
Pygame
библиотека *207*
обработка событий *212*
структура программы *213*
pygame.draw.circle()
функция *211*
pygame.event.get()
функция *212*
pygame.font
модуль *278*
функция *278*
pygame.mixer
модуль *290*
pygame.quit()
функция *211*
Pygame, Turlte
отличия *211*
Python
установка на компьютер *23*

R

randint()
функция *131*
random
модуль *131*
random.choice()
функция *135*
randrange()
функция *140*
range()
функция *75*
rect.collidepoint()
функция *260*
remove()
функция *260*
return
ключевое слово *184*
RGB *210*

S

scale()
функция *257*
set_caption()
функция *242*
setheading()
функция *93*
setpos()

функция *137*
setx()
функция *93*
sety()
функция *93*
Smiley
класс *254*
sort()
функция *155*
speed()
функция *164*
Sprite
класс *251*

Т

text.get_rect()
функция *279*
turtle
скорость отрисовки *29*
Turtle
скорость отрисовки *164*
Turtle, Pygame
отличия *211*
turtle.window_height()
функция *138*
turtle.window_width()
функция *138*

U

update()
функция *211*
upper()
функция *120*

А

Алгоритм *38*
Анимация
кадр *217*
Арифметические операции
%, взятие по модулю *41*

В

Ввод строк с клавиатуры *51*
Выражения ветвления *100*, *102*

Г

Графика turtle *29*
Графический интерфейс пользователя
определение *206*

Д

Дизайн игр *267*

З

Заголовок окна *242*
Запуск программы *25*

И

Игровой цикл *214*
Импорт кода *31*
Интерактивные приложения *188*

К

Картезианские координаты *136*
Класс
конструктор *254*
методы и атрибуты *251*
определение *221*
расширение *253*
создание *253*
функция инициализации *254*
Класс-контейнер *252*
Комментарии *24*
Константа
определение *210*
Конструктор класса *254*
Координаты на экране *136*

Л

Логические операторы *117*

М

Массив
индекс *148*
определение *146*
поиск элемента *148*
сортировка элементов *155*
Математические операторы *53*

О

Оболочка Python
математические вычисления *54*
переменные *56*
Обработка событий
события клавиатуры *192*
события
с параметрами *194*
щелчок мышью *188*
Обработка событий Pygame *212*
Обработка щелчков мышью *243*
Объектно-ориентированное программирование
определение *252*
Окно
размер *138*
Операторы *53*
логические *117*
сравнения *103*
Операторы сравнения *103*
Отрисовка текста на экране *277*

П

Переменная
определение *49*
правила именования *50*
присвоение значения *50*
типы значений *49*
Переменная-флаг *151*
Переопределение функции *256*
Присваивание *50*

Р

Размер окна *138*

С

Синтаксис
определение *55*
Синтаксические ошибки *55*
Сложные условия *116*
Случайные числа
заданный диапазон *140*
Событийно управляемые приложения *188*

События мыши
раздельная обработка *245*
Создание игр
вычитание жизни *272*
геймплей (сюжет) *267*
звук *290*
конец игры *282*
отрисовка текста *277*
столкновения игровых фигурок *273*
ускорение игры *284*
учет очков *271*
элементы дизайна *267*
Создание новой программы *24*
список
поиск элемента *148*
Спрайт
графический элемент *250*
столкновение с точкой *260*
трансформация *256*
Спрайтовая графика *250*
Строки *61*
функция ввода *63*
Структура программы Pygame *213*
Счетчик *221*

У

Установка Python на компьютер *23*

Ф

Флаг *151*
функцией *66*
Функция
аргументы *175*
возврат значений *184*
объявление *172*
определение *172*
параметры *175*
переопределение *256*
Функция инициализации *254*
Функция обратного вызова *189*

Ц

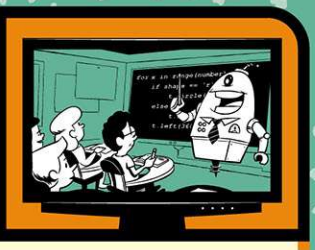
Цвета Python *38*
Цикл *31*
for *32*
Циклы
for *75*
while *82*
вложенный цикл *89*

Ч

Четные или нечетные числа *111*
Числа
виды чисел в Python *52*
функция ввода *66*

PYTHON ДЛЯ ДЕТЕЙ И РОДИТЕЛЕЙ

Программирование – это просто, даже родители справятся!



Python – это сверхпопулярный язык, на котором разрабатывается абсолютно все, от мобильных приложений и игр до научных расчетов и моделирования. Именно на нем ведутся разработки в таких компаниях, как Google и IBM.

Благодаря пошаговым инструкциям и иллюстрированным примерам, а также игровой составляющей этой книги ваш ребенок без труда освоит азы программирования на Python.

Чему вы научитесь вместе с ребенком по этой книге:

- **Создавать веселые семейные игры на компьютере**
- **Рисовать необыкновенную и красочную графику**
- **Писать программы для шифрования секретных посланий**
- **Разрабатывать собственные интерактивные компьютерные приложения со звуком и анимацией**

Об авторе

Доктор Брайсон Пэйн – профессор Университета Северной Джорджии, преподаватель компьютерных наук с более чем 15-летним стажем. Среди его бывших студентов – ведущие сотрудники компаний, подаривших миру такие игры, как World of Warcraft и League of Legends. Отец двоих сыновей, доктор Пэйн прекрасно знает, как сделать обучение увлекательным и веселым.

Когда я был маленьким, приходилось собирать информацию по крупицам и засыпать над слишком техническими томами по программированию. Насколько было бы проще, если бы у моих родителей была такая книга! Возьму на вооружение для своего первенца.

Евгений Пивень,
IPONWEB

no starch press
www.nostarch.com

ISBN 978-5-699-95169-7
9 785699 951697 >