

PYTHON для ДЕТЕЙ

Эта книга – прекрасное руководство по программированию для детей на языке Python средней сложности. Читатели получают базовые знания о языке Python, узнают об объектно-ориентированном программировании, научатся работать с функциями, классами и модулями. Много внимания уделено работе с графикой, созданию анимации и разработке собственной игры.

Издание будет полезно школьникам средних и старших классов, увлекающимся программированием, а также может быть использовано как учебник на курсах дополнительного образования для детей.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
e-mail: books@aliants-kniga.ru

ДМК
издательство
www.dmk.pf

ISBN 978-5-97060-681-0



9 785970 606810 >

PYTHON для ДЕТЕЙ



Ханс-Георг Шуман

Python для детей

Hans-Georg Schumann



Python für Kids

Programmieren lernen ohne Vorkenntnisse



Ханс-Георг Шуман



Python для детей

Уроки программирования для чайников



Москва, 2019

УДК 004.438Python:004.6
ББК 32.973.22
Ш96

Шуман Х.-Г.
Ш96 Python для детей / пер. с нем. М. А. Райтман. – М.: ДМК Пресс, 2019. – 344 с.: ил.

ISBN 978-5-97060-681-0

Эта книга – прекрасное руководство по программированию для детей на языке Python средней сложности. Читатели получают базовые знания о языке Python, узнают об объектно-ориентированном программировании, научатся работать с функциями, классами и модулями. Много внимания уделено работе с графикой, созданию анимации и разработке собственной игры.

Издание будет полезно школьникам средних и старших классов, увлекающимся программированием, а также может быть использовано как учебник на курсах дополнительного образования для детей.

УДК 004.438Python:004.6
ББК 32.973.22

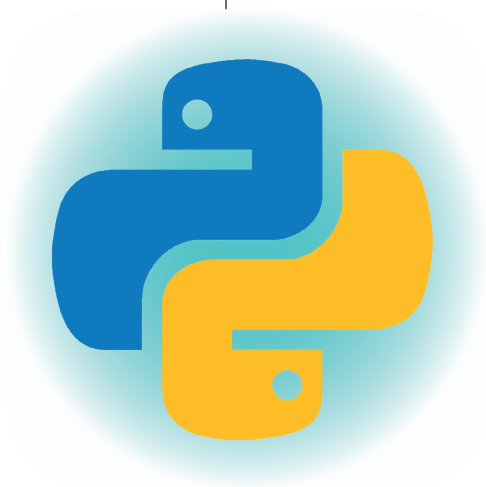
First published as Python für Kids by Hans-Georg Shuman. © 2018 by MITP Verlag GmbH & Co, KG. All rights reserved. Published with arrangements made by Maria Pinto-Peuckmann, Literary Agency-World Copyright Promotion, Kaufering, Germany.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-3-95845-320-3 (нем.)
ISBN 978-5-97060-681-0 (рус.)

Copyright © 2018 mitp Verlags GmbH & Co. KG, Frechen
© Оформление, издание, перевод, ДМК Пресс, 2019

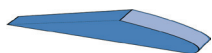
Посвящается Янне, Джулии, Катрин и Даниэль



Содержание

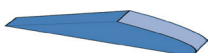
Введение	13
-----------------------	-----------

1



Твои первые шаги в программировании	19
Начало работы с Python.....	19
Числа и текст	23
Рабочая среда IDLE	27
Работа с PY-файлами	30
Эксперименты с исходным кодом.....	34
Выход из Python	37
Подведение итогов	37
Несколько контрольных вопросов.....	38
...а задач пока нет!.....	38

2



Условные конструкции	39
Конструкция if	39
Конструкция if else.....	44
Простые вычисления	48

Вкратце о числах..... 51

Работа с командами try и except 55

Подведение итогов 57

Несколько вопросов..... 58

...и задач 58

Сравнение и повторение 59

Оценки..... 59

Небольшая игра на угадывание 64

Компьютер считает попытки 69

Шлифуем игру 72

Подведение итогов 74

Несколько вопросов..... 75

...и задач 75

Азартная игра 76

Игра наудачу 76

Конструкция for 79

На пути к миллиону..... 82

Выиграть в лотерею? 87

Управление строками..... 90

Подведение итогов 92

Несколько вопросов ... 93

...и задач 93

Функции 94

Python учится 94

Локальные или глобальные переменные? 98

Параметры..... 100

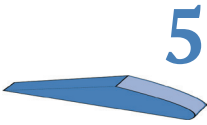
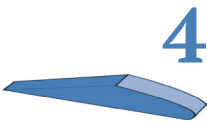
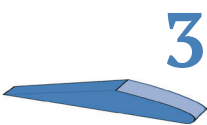
Обмен значений..... 104

Сортировка чисел..... 108

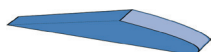
Подведение итогов 112

Несколько вопросов..... 112

...и задач 112

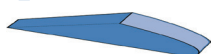


6



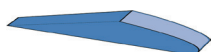
Классы и модули	113
Потомки	113
self и __init__	117
Наследование	120
Модули программы	125
Приватный или публичный?	130
Подведение итогов	134
Несколько вопросов	134
...а задач нет	134

7



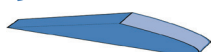
Введение в tkinter	135
Создаем окно	135
Что же происходит?	139
Разметка интерфейса программы	142
Диалоговые окна и заголовки	145
А теперь с классами	146
Подведение итогов	148
Несколько вопросов	148
...и одна задача	148

8



Библиотека компонентов	149
Череда кнопок	149
Кнопки и ответы	152
Списки выбора	155
О переключателях	157
...и флажках	160
Декорирование приложения	164
Подведение итогов	167
Несколько вопросов	168
...и задач	168

9



Домашний психолог	169
Пошаговая разработка программы-психолога	169
Приступим к диагностике?	173

Работа с файлами 177

Все вместе 181

Журнал диагностики 183

Подведение итогов 185

Несколько вопросов..... 185

...и задач 185

Меню и диалоговые окна 186

Меню для программы-психолога 186

Два диалоговых окна 190

Полный исходный код 192

Контекстные меню и всплывающие окна 194

Используем сочетания клавиш 198

Подведение итогов 200

Несколько вопросов..... 200

...и нет задач..... 200

Графика в Python 201

Точки и координаты 201

Первое изображение..... 204

Добавим цвета..... 208

Углы и круги 209

Эксперименты с текстом 212

Звездное небо..... 213

Сам себе художник..... 215

Черепашья графика 217

Подведение итогов 221

Несколько вопросов..... 221

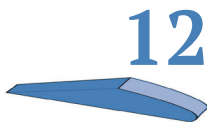
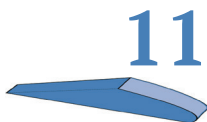
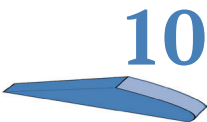
...и задач 222

Создание анимации 223

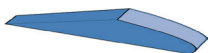
Начнем с круга 223

Загружаем на холст изображение 227

Коллекция изображений 230



13

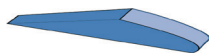


Класс Player	232
Все работает?	236
Повороты	238
Исчезновение и появление	240
Подведение итогов	243
Вопрос	243
...и несколько задач	244

Игра с насекомыми..... 245

Начало работы с Runame	245
Создаем в окне объект	248
Насекомое в качестве персонажа	251
Управление персонажем	256
Поворот персонажа	261
Отслеживание границ игрового поля	264
Подведение итогов	266
Несколько вопросов	267
...и задача	267

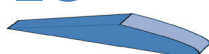
14



Как раздавить жука

Разбираемся с мышью	268
Никуда без математики	270
Собираем все вместе	275
Свободное ползание	277
Тапком по виртуальному жуку	281
Управление классами	283
Подведение итогов	286
Несколько вопросов	287
...и задача	287

15

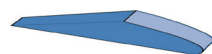


Уклониться или проиграть

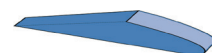
Новый персонаж	288
Стоять, приседать и подпрыгивать	292
Класс Thing	296

Учим персонажа уклоняться	299
Основная программа	303
Подведение итогов	304
Нет вопросов... ..	304
...и одна задача.....	305
Пора развлекаться.....	306
Игровой счет.....	306
Класс Game	310
Ошибка при сборке программы	313
Очки при попадании	316
Подведение итогов и заключение	320
Установка Python.....	321
Установка библиотеки Pygame, вариант 1.....	325
Установка библиотеки Pygame, вариант 2.....	329
Файлы примеров.....	331
Решение распространенных проблем	332
Ответы на вопросы и задачи.....	333
Предметный указатель.....	340

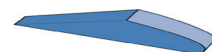
16



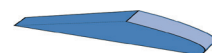
А



Б



В



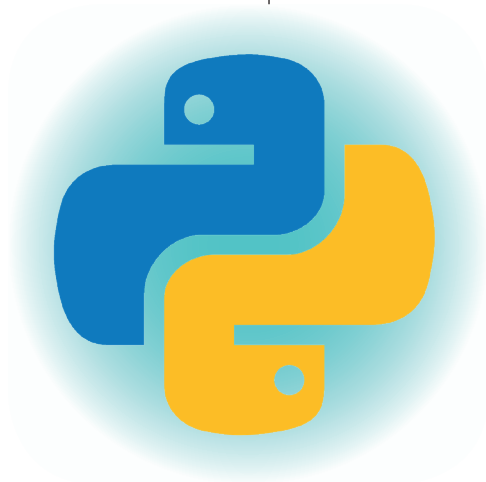


В своё время, для того чтобы изучить основы Python, я потратил очень много времени на поиск необходимой информации в интернете. Книг тогда было достаточно, но они были перегружены огромным количеством информации, которая только путала новичка.

Эта книга представляет собой руководство средней сложности для изучения языка программирования Python.

Она очень хорошо подойдёт для детей, изучающих Python с наставником, и для самостоятельного освоения данного языка, а также для составления методических пособий в организациях дополнительного образования.

Дмитрий Филиппов,
преподаватель курса по Python
в международной школе программирования
для детей «CODDY»



Введение

Python – это язык *программирования*, который был разработан в начале 1990-х годов. Название этого языка программирования произошло от имени английской комедийной труппы под названием «Монти Пайтон», которая в 70-е годы была довольно популярна и принимала участие в кинокомедиях, таких как «Жизнь Брайана».

Хотя Python во многом похож на другие языки программирования, он стал очень популярен, так как именно на этом языке довольно легко научиться программировать. Эта книга посвящена третьей версии языка программирования Python, на сегодня последней (и универсальной).

Что такое программирование?

Программирование – это когда ты записываешь то, что должен делать компьютер. Самое замечательное, что ты сам можешь решить, что конкретно ему нужно делать. Если ты запустишь свою программу, компьютер выполнит то, что ты только что придумал и записал.

Конечно, твой компьютер не сможет убрать твою комнату и не принесет тебе чашечку какао в постель. Но как только ты освоишь программирование, то сразу же сможешь позволить компьютеру, так сказать, танцевать под свою дудку.

Однако во время программирования часто бывает так, что компьютер не хочет выполнять то, что хочешь от него ты. Обычно такое поведение вызывает ошибка в программе. Проблема также может быть где-то еще в компьютере или

операционной системе. Проблема ошибок заключается в том, что им очень уж нравится прятаться, так что их поиск заставляет некоторых программистов попросту отчаиваться. Я очень надеюсь, что тебе все-таки захочется научиться программировать. Тогда все, что тебе нужно, – это подходящая среда разработки, и ты уже готов к работе!

Что такое среда разработки?

Чтобы создать программу, сначала нужно что-то куда-то ввести. Это как письмо или рассказ, который ты пишешь. Текстовая программа для этого может быть очень простой, поскольку не нужно выбирать разные шрифты или вставлять картинки. Такие программы принято называть *текстовыми редакторами*.

Если программа набрана в виде текста в редакторе, то компьютер не сможет просто прочитать этот документ и выполнить программу. Текстовая версия (*код*) программы должна быть переведена так, чтобы компьютер понял, что от него требуется в конечном итоге. Но поскольку он говорит на совершенно ином языке, чем ты или твои друзья, то придется использовать специальный переводчик.

Ты программируешь на понятном тебе языке, а специальная программа переводит то, что ты написал, в команды, понятные для компьютера. Такую программу называют *компилятором* или *интерпретатором*.

Для языка Python существуют интерпретаторы для нескольких операционных систем. Твой компьютер может работать под управлением операционной системы Windows, Linux или macOS. Независимо от этого одна и та же программа, написанная на языке программирования Python, будет работать (возможно, с небольшими изменениями) на любом компьютере.

Наконец, программы должны быть проверены, пересмотрены, усовершенствованы, доработаны и собраны. А еще есть вспомогательные инструменты. Все это становится целой системой – *средой разработки*.

Почему Python?

К сожалению, ты не сможешь программировать так, как захочется. Язык программирования должен быть структурирован таким образом, чтобы как можно больше людей могли понять его по всему миру.

В мире большинство людей может, по крайней мере, прочитывать несколько слов по-английски, и поэтому почти каждый язык программирования состоит из английских слов. Есть также проекты языков программирования на других языках, но в большинстве случаев команды звучат настолько непривычно, что ты, скорее всего, все же вернешься к английскому. На самом деле не имеет значения, какой конкретно язык программирования ты используешь. Конечно, лучше всего тот, которому легко научиться. Как ты уже понял, в этой книге ты научишься создавать программы с помощью языка программирования Python, который сейчас очень популярен. (Если ты захочешь познакомиться с другими языками, я рекомендую обратить внимание на языки программирования C++ и Java.)

Стать хорошим программистом может быть очень трудно. Нередко человек теряет желание, потому что у него просто не работают написанные им программы. Программа делает нечто совершенно другое, а он сам не может найти ошибку и спрашивает себя: зачем я учусь программировать, если в мире уже и так достаточно программ?

Хорошие программисты всегда легко находят работу, а потребность в качественном программном обеспечении будет только расти. Программисты на языке Python, безусловно, востребованы в трудовой сфере. И, честно сказать, хорошие программисты действительно получают хорошую заработную плату. Это не просто твоя первая попытка, ведь, может быть, тебе действительно станет интересно узнать, как программировать на Python.

Среда разработки

Тебе не нужно переживать о том, где взять среду разработки на языке Python, так как это достаточно просто. Ты получишь ее бесплатно в составе пакета Python (называемого IDLE (Integrated Development and Learning Environment) – *интегрированная среда разработки и обучения на языке Python*). Мы будем пользоваться ею в этой книге постоянно.

Скачать полный пакет ты сможешь на сайте www.python.org. Необязательно устанавливать самую последнюю версию. В этой книге мы будем работать с версией Python 3.1.

О чем эта книга?

В этой книге ты:

- получишь базовые знания о языке Python;

- узнаешь об объектно-ориентированном программировании;
- научишься работать с компонентами библиотеки `tkinter` (это строительные блоки, с помощью которых ты сэкономишь время в процессе программирования);
- узнаешь о возможностях Python при работе с графикой;
- обучишься основам работы с игровыми модулями `pygame`;
- узнаешь, как разработать собственную игру.

В приложениях содержится дополнительная информация и справочные сведения, которые помогут разобраться в настройках и способах устранения ошибок.

Как читать эту книгу

Эта книга состоит из большого количества текста с иллюстрациями. И я, конечно, старался написать ее так, чтобы она была понятной. Чтобы сделать информацию в ней еще более понятной для тебя, я добавил некоторые символы. Вот что они обозначают:

Практические шаги

- Если ты видишь такой символ, знай: тебе нужно выполнить задание в данном абзаце текста. Это приблизит тебя к новой цели в программировании.

В целом ты будешь быстрее усваивать материал, когда будешь вводить код программы или станешь изменять его самостоятельно. Но у тебя не всегда будет желание этим заниматься. Потому все примеры данной книги доступны для бесплатного скачивания по адресу dmkpress.com.

Скачав файлы примеров, ты сможешь найти проект по имени (например, \Rightarrow *project1.py*). Поэтому если ты не захочешь самостоятельно создавать проект, то всегда сможешь открыть соответствующий файл.

Задачи

В конце каждой главы ты найдешь несколько вопросов и задач. Эти упражнения не всегда просты, но они помогают освоить программирование. Решения задач можно найти в папке с файлами примеров. Ты сможешь просмотреть их в текстовом редакторе. Или распечатать на принтере и сложить стопкой рядом с компьютером.

Решение проблем

Случается, что ты не знаешь, как что-то сделать, или, возможно, просто что-то забыл. И тогда вся работа кажется невыполнимой и сложной. И ты спрашиваешь себя: что же мне делать? С помощью этого значка ты сможешь довольно легко найти решение или восстановить что-то в памяти. При необходимости также можешь заглянуть в последнюю часть этой книги, там расположено приложение Б, в котором приведены советы и другая справочная информация.



Важные примечания

Время от времени ты будешь встречать в книге такой восклицательный знак. Он указывает на текст с очень важной информацией.



Экспертное мнение

Если ты видишь такой значок, «Вау!», этот текст содержит дополнительную информацию по теме.



Что тебе нужно для этой книги

Среда разработки Python устанавливается в каталог на твой выбор, например в папку `C:\Python`. Там ты также сможешь разместить свои проекты Python, только чуть позже. Примеры программ в этой книге доступны для загрузки с главной страницы сайта издательства русского издания этой книги dmkpress.com.

Там же ты найдешь ответы на вопросы и решения задач (все они находятся в папке с примерами).

Операционная система

Большинство компьютеров сегодня работает под управлением операционной системы Windows. Для работы с примерами из этой книги тебе понадобится версия Windows 7 или 10. (Среда Python также доступна и для macOS и Linux.)

Носители информации

Тебе понадобится USB-накопитель (т. н. «флешка») или SD-карта, если ты захочешь сохранить свои программы на диск. На внешнем хранилище твоя работа всегда будет в безопас-

ности. При необходимости попроси своих родителей или преподавателя купить тебе такой носитель информации.

Примечания для преподавателей

Эта книга также может использоваться как учебный материал на уроках информатики в школе. Конечно, каждый учитель устанавливает свои собственные приоритеты в обучении детей программированию. Если вы уже используете другой учебник в своей работе, то сможете использовать это издание в качестве источника информации в дополнение к существующему учебнику. Эта книга начинается с «нуля», так сказать, является прямым вхождением в язык программирования Python, без необходимости наличия каких-либо навыков программирования.

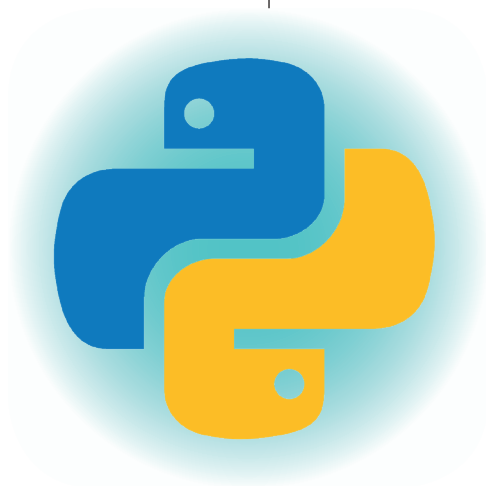
Важным направлением в этой книге является объектно-ориентированное программирование (ООП). Наиболее важные концепции (инкапсуляция, наследование и полиморфизм) обсуждаются в книге достаточно подробно. Еще одним направлением является игровое программирование. В проектах используются все основные элементы словаря Python, а также наиболее важные графические компоненты библиотеки tkinter. Вы встретите большое количество задач по программированию.

Хранение файлов проектов

На уроках информатики каждому ученику необходимо иметь собственный внешний носитель для хранения всех заданий и собственноручно написанных программ. Таким образом, жесткому диску школьного компьютера не придется накапливать горы лишнего «мусора». Кроме того, собственный носитель данных служит для их защиты: ученик не потеряет свои файлы и с удобством сможет ими управлять.

Постоянное сохранение прогресса

Отличная идея – сохранять файлы программы в процессе работы примерно каждые десять минут. Все знают, что компьютеры любят «выходить из строя» именно тогда, когда файл не был сохранен.



1

Твои первые шаги в программировании

В этой главе разговор пойдет о том, как после установки и запуска Python сделать свои первые шаги. Я расскажу тебе, как настроить рабочее окружение так, чтобы твоя первая, написанная на языке программирования Python программа появилась на свет.

В этой главе ты узнаешь:

- ⦿ как запустить Python;
- ⦿ как использовать инструкции для вывода и ввода данных;
- ⦿ что такое переменные;
- ⦿ что такое строковый тип данных;
- ⦿ как использовать среду разработки IDLE;
- ⦿ как создавать и сохранять программы;
- ⦿ как завершать работу с Python.

Начало работы с Python

Прежде чем ты начнешь программировать, давай настроим Python.

1

Настройка предполагает запуск установленной программы. Подробные сведения об установке ищи в приложении А. Тебе понадобится помощь взрослых, если ты не знаешь, как работать с такой программой самостоятельно. Один из способов запуска Python такой:

- Открой папку, в которую ты установил Python, например `C:\Program Files\Python` или `C:\Python` (рис. 1.1).

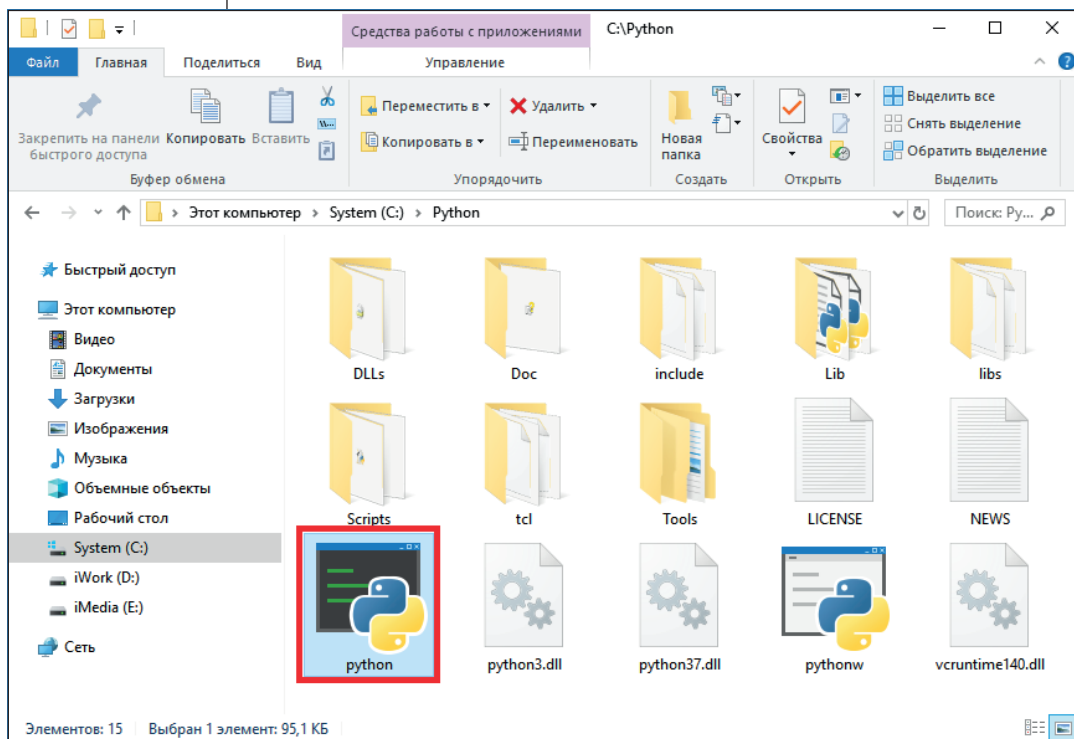


Рис. 1.1. Содержимое папки с установленным Python

- Найди среди значков файл с именем `python.exe`. Дважды щелкни мышью по значку с этим именем.

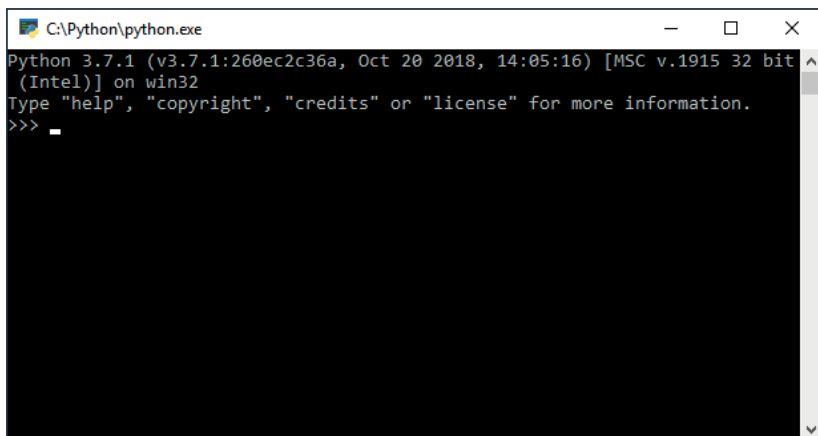
Для удобства ты можешь создать ярлык на рабочем столе:

- ❖ щелкни правой кнопкой мыши по значку `python.exe`;
- ❖ в контекстном меню выбери пункт **Отправить** ⇒ **Рабочий стол (Создать ярлык)** (Send To ⇒ Desktop (Create shortcut));
- ❖ не обязательно ставить стандартное имя для ярлыка на файл `python.exe`. Ты можешь использовать любое подходящее слово.

Теперь можешь дважды щелкнуть мышью по новому значку и запустить Python.



Что же произойдет после запуска? Появится окно оболочки командной строки, показанное на рис. 1.2.



```
C:\Python\python.exe
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Рис. 1.2. Запущенный Python

Первая строка сообщает о текущей версии Python, а ниже указано несколько справочных команд и три угловые скобки (>>>).

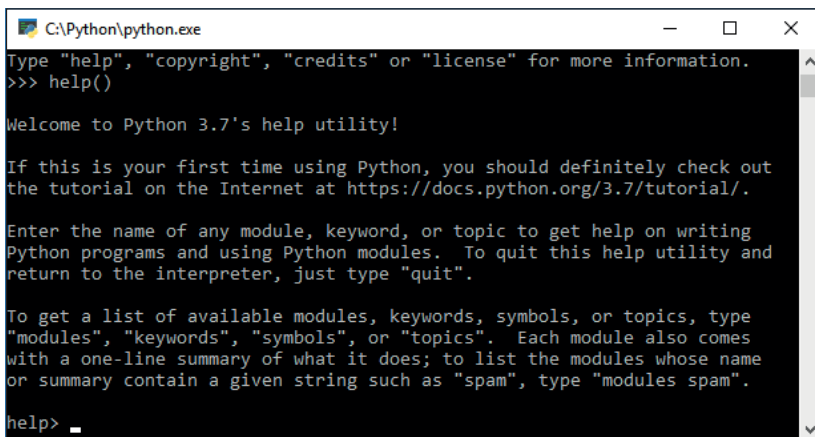
Эти три символа называются «приглашением». Это действительно своего рода приглашение, потому что ты сможешь ввести команду после них (и тебе придется это сделать, если ты хочешь продолжить работу).



- Давай попробуем поработать с Python с помощью команды `help`. Введи это слово. Обрати внимание, что помимо самого слова тебе нужно набрать две круглые скобки следом.
- Введи команду `help()` и нажми клавишу **Enter**.

На экране появится много текста (рис. 1.3).

1

A screenshot of a Windows command prompt window titled "C:\Python\python.exe". The window shows the output of the 'help()' command. The text displayed is: "Type 'help', 'copyright', 'credits' or 'license' for more information. >>> help() Welcome to Python 3.7's help utility! If this is your first time using Python, you should definitely check out the tutorial on the Internet at https://docs.python.org/3.7/tutorial/. Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type 'quit'. To get a list of available modules, keywords, symbols, or topics, type 'modules', 'keywords', 'symbols', or 'topics'. Each module also comes with a one-line summary of what it does; to list the modules whose name or summary contain a given string such as 'spam', type 'modules spam'. help> _".

```
C:\Python\python.exe
Type "help", "copyright", "credits" or "license" for more information.
>>> help()

Welcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

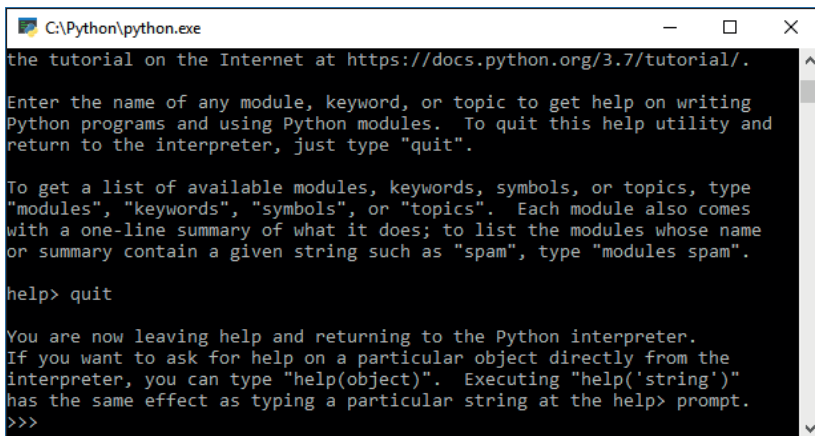
To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> _
```

Рис. 1.3. Результат выполнения команды help()

Теперь в оболочке командной строки указано слово help>, которое служит подсказкой. Ты можешь указать следом слово (на английском языке), и если оно есть в словаре Python, ты увидишь небольшое объяснение.

- Чтобы вернуться к приглашению, введи команду quit() (рис. 1.4).

A screenshot of a Windows command prompt window titled "C:\Python\python.exe". The window shows the output of the 'quit()' command. The text displayed is: "the tutorial on the Internet at https://docs.python.org/3.7/tutorial/. Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type 'quit'. To get a list of available modules, keywords, symbols, or topics, type 'modules', 'keywords', 'symbols', or 'topics'. Each module also comes with a one-line summary of what it does; to list the modules whose name or summary contain a given string such as 'spam', type 'modules spam'. help> quit You are now leaving help and returning to the Python interpreter. If you want to ask for help on a particular object directly from the interpreter, you can type 'help(object)'. Executing 'help('string')' has the same effect as typing a particular string at the help> prompt. >>>".

```
C:\Python\python.exe
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.

>>>
```

Рис. 1.4. Ввод команды quit()

И вот мы снова находимся в **интерпретаторе** языка Python.

Что же такое *интерпретатор* (другими словами, переводчик)? Во-первых, ты должен знать, что то, что ты вводишь в качестве команды, совершенно непонятно для компьютера. Как правило, он не может выполнить такую команду.

Интерпретатор *переводит* командную строку на *машинный* язык, который компьютер понимает, чтобы он мог выполнить данную тобой команду. Для программы, которая может состоять из многих-многих строк, интерпретатор переводит каждую строку по очереди и выполняет ее.

Кроме того, существуют *компиляторы*, которые переводят всю программу на машинный язык. Программа выполняется только после того, как скомпилирована и проверена, так что без ошибок может быть выполнена компьютером. В этой книге мы используем интерпретатор, но есть также и компиляторы для языка Python.



Числа и текст

Теперь попробуем что-нибудь ввести.

- Введи: `1+2+3`, а затем нажми клавишу **Enter** (рис. 1.5).

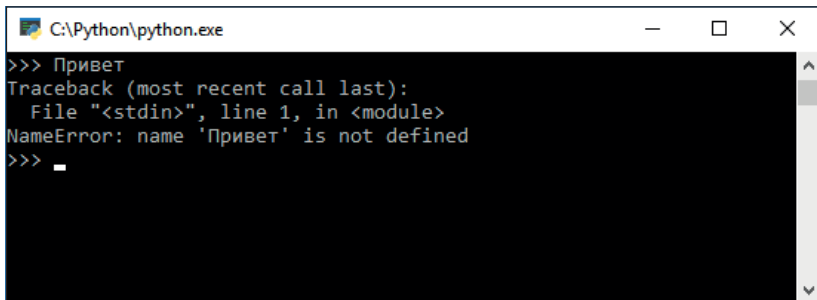
```
C:\Python\python.exe
>>> 1+2+3
6
>>> _
```

Рис. 1.5. Выполнение операции сложения

Результат решения этой маленькой математической задачи появился на экране.

- Выполни еще несколько задач, используя операции вычитания (`-`), умножения (`*`) и деления (`/`). Как калькулятор интерпретатор Python работает отлично, но мы же с тобой хотим гораздо большего, не так ли?
- Давай попробуем небольшое приветствие: напечатай Привет и нажми клавишу `\n` **Enter** (рис. 1.6).

1

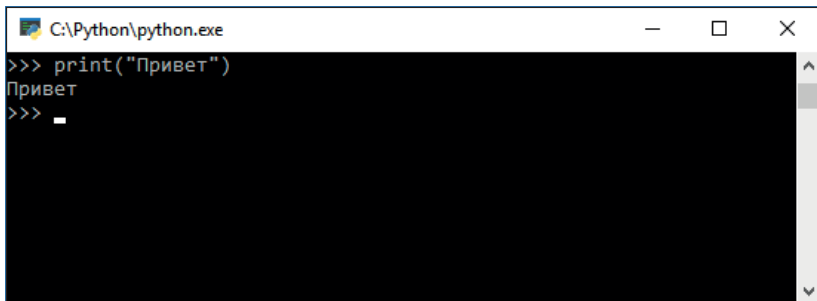
A screenshot of a Windows command prompt window titled "C:\Python\python.exe". The prompt shows the user typing "Привет" (Hello) and pressing Enter. The interpreter responds with a "NameError: name 'Привет' is not defined" message, indicating that "Привет" is not a recognized variable or function in Python. The prompt then returns to the user's input line, showing a cursor after a space character.

```
C:\Python\python.exe
>>> Привет
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Привет' is not defined
>>> _
```

Рис. 1.6. Ввод слова привел к ошибке

Что случилось? Тут явно что-то не работает. А я хотел, чтобы компьютер сказал мне (т. е. написал) дружеское «Привет!».

- Введи строку `print("Привет")` и нажми клавишу **Enter** (рис. 1.7):

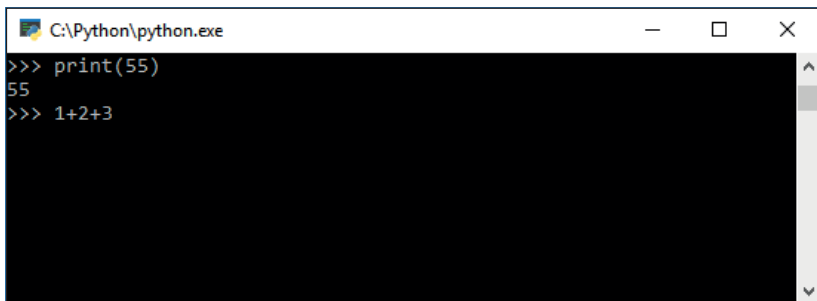
A screenshot of a Windows command prompt window titled "C:\Python\python.exe". The prompt shows the user typing "print('Привет')", pressing Enter, and then another Enter. The interpreter first prints "Привет" (Hello) to the screen, and then returns to the user's input line, showing a cursor after a space character.

```
C:\Python\python.exe
>>> print("Привет")
Привет
>>> _
```

Рис. 1.7. Отображение приветствия

Работает! Здесь имя команды `print()` означает отображение текста, вывод на экран. В круглых скобках после слова `print` ты вводишь именно то, что хочешь отобразить на экране. Это называется *параметром*.

Конечно, такое же возможно и с числами (рис. 1.8):

A screenshot of a Windows command prompt window titled "C:\Python\python.exe". The prompt shows the user typing "print(55)", pressing Enter, and then "1+2+3", pressing Enter. The interpreter first prints "55" to the screen, and then prints "6" (the result of 1+2+3). The prompt then returns to the user's input line, showing a cursor after a space character.

```
C:\Python\python.exe
>>> print(55)
55
>>> 1+2+3
6
>>> _
```

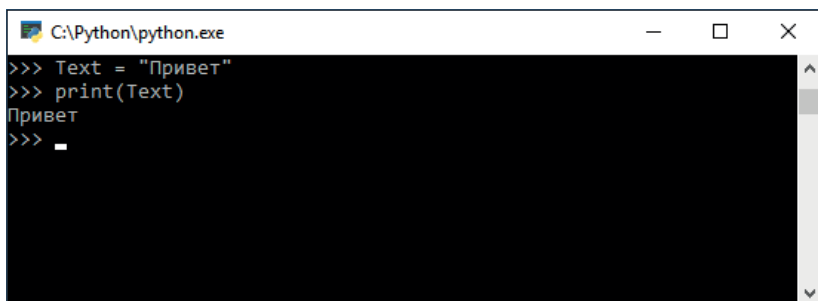
Рис. 1.8. Отображение чисел

- Поэкспериментируй с командой `print()`.

Теперь задание будет немного сложнее. До сих пор мы вводили только одну команду. Но все программы, конечно, состоят из нескольких строк. Попробуем ввести небольшую программу:

```
Text = "Привет"
print(Text)
```

- Введи эти две строки (рис. 1.9). Каков результат?



```
C:\Python\python.exe
>>> Text = "Привет"
>>> print(Text)
Привет
>>> _
```

Рис. 1.9. Небольшая программа-приветствие

После первой строки ничего не выводится. Но, по-видимому, интерпретатор Python понимает, что означает `Text`. И он знает, какое значение в команде `print()` следует использовать в качестве параметра.

Чуть подробнее: `Text` – это так называемая *переменная*, которой *присваивается значение*, в нашем случае это слово "Привет". А знак равенства (=) называется *оператором присваивания*.

Переменная = Значение

В Python переменные создаются при первом присвоении им значений. При присвоении левая часть всегда является именем переменной, а правая – значением, поэтому знак равенства можно изобразить как стрелку:

Переменная ← Значение

Переменные полезны, потому что они сохраняют данные, чтобы компьютер мог запомнить значение. Для сложных программ важно, чтобы содержимое переменной можно было использовать несколько раз. Далее я покажу несколько примеров.



1



Раз уж компьютер с нами так хорошо поздоровался, давай усложним нашу программу. Введи:

```
Text = "Привет, кто ты?"
print(Text)
Name = input()
print(Name)
```

При программировании на языке Python текст, подобный приветствию, всегда должен быть указан в кавычках. В примере я использую двойные кавычки (""), но также допускаются и одинарные ('').

Поэтому следующие две строчки будут полностью эквивалентны:

```
Text = "Привет, кто ты?"
Text = 'Привет, кто ты?'
```

- Введи все эти строки по очереди. Обрати внимание, что в процессе тебе придется ввести свое имя.

Вот так программа выглядит в моем случае (рис. 1.10):

```
C:\Python\python.exe
>>> Text = "Привет, кто ты?"
>>> print(Text)
Привет, кто ты?
>>> Name = input()
Михаил
>>> print(Name)
Михаил
>>>
```

Рис. 1.10. Программа с запросом имени

Не так уж плохо! Заодно ты познакомился с новой командой `input()` – она означает ввод текста с клавиатуры. Отличная тренировка – напиши несколько вопросов `print()` и запроси ответы на них с помощью команды `input()` – и у тебя получится интересный чат с компьютером.

Но что-то здесь мне не очень нравится. Например, то, что приходится постоянно снова набирать строки кода про-

граммы. И не получается просто «прогуляться» по программе с помощью клавиш со стрелками. И также не получается щелкнуть в любом месте мышью и изменить там текст. Было бы лучше, если бы мы могли перемещаться в окне Python так же, как в текстовом редакторе.

Ужасно неудобно, если мы хотим создавать более крупные программные проекты! Для этого также должна быть возможность сохранить этот текст в виде файла. Одной этой программы, которая у нас есть, очевидно, недостаточно.

Рабочая среда IDLE

Поэтому нам нужен интерфейс, с помощью которого можно сохранить введенный текст и менять его при необходимости. Такая программа называется *редактором*. В дистрибутиве Python такой редактор присутствует, тебе просто нужно его найти. В меню **Пуск** (Start) операционной системы Windows ты найдешь ярлык **IDLE** (рис. 1.11).

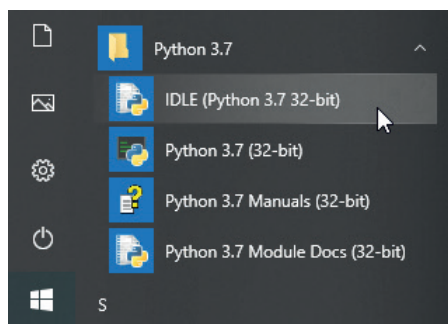


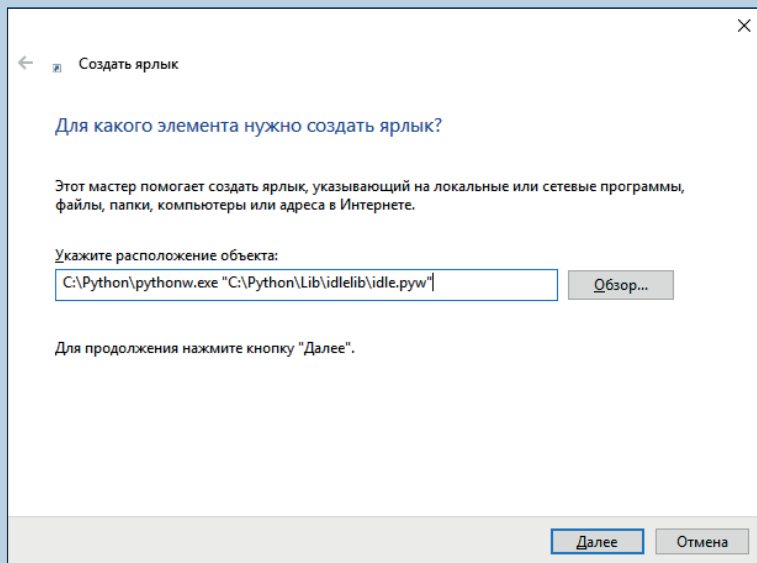
Рис. 1.11. Выбор пункта **IDLE** в меню **Пуск**

- Выбери значок с надписью **IDLE**, чтобы запустить эту программу.

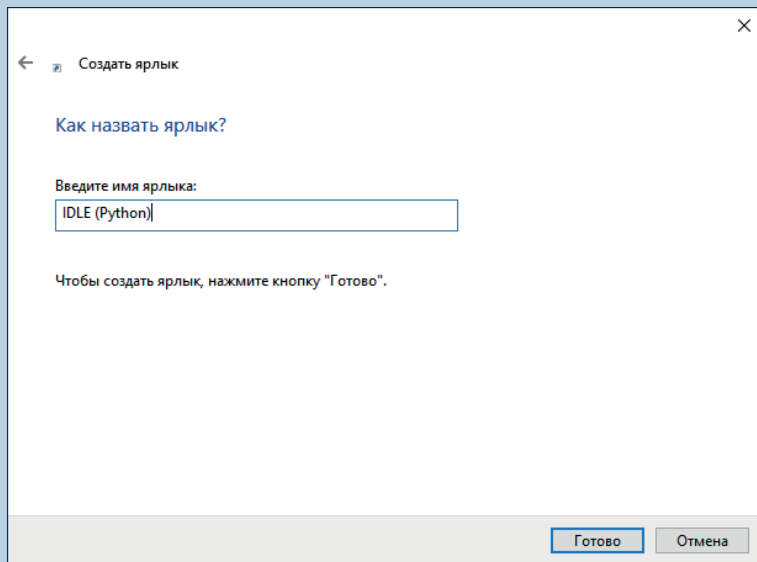
Если у тебя нет этого значка в меню **Пуск** (Start), тогда создай ярлык на рабочем столе самостоятельно.

- ❖ Щелкни правой кнопкой мыши на пустом месте рабочего стола и выбери пункт **Создать** ⇒ **Ярлык** (New ⇒ Shortcut).
- ❖ В появившемся диалоговом окне укажи строку **C:\Python\pythonw.exe** «C:\Python\Lib\idlelib\idle.pyw».





- ❖ Важно, чтобы вместо *C:\Python* ты указал ту папку, в которую установил Python. В противном случае ярлык не будет работать.
- ❖ Нажми кнопку **Далее** (Next).
- ❖ Присвой новому ярлыку на рабочем столе имя **IDLE (Python)**, как показано на рисунке ниже.



- ❖ Нажми кнопку **Готово** (Finish).

С этого момента ты сможешь запускать редактор Python прямо с рабочего стола, дважды щелкнув по ярлыку **IDLE (Python)**.



После запуска программы IDLE ты увидишь окно, показанное на рис. 1.12.

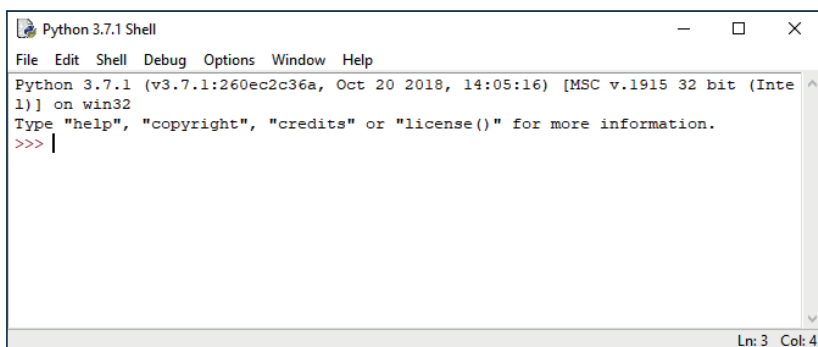


Рис. 1.12. Интерфейс IDLE

Сама программа напоминает и в то же время существенно отличается от предыдущего интерпретатора Python. Кроме того что вместо черного фона здесь мы будем работать в области белого цвета, тут также есть и строка меню. И теперь вся эта программа будет называться *оболочкой*.

IDLE – это аббревиатура названия Integrated Development and Learning Environment. То есть *интегрированная среда разработки и обучения*, которая необходима для программирования на языке Python.



Мы также можем вводить команды на языке Python в этой среде и получим те же результаты, что и в предыдущем окне (в «черном» интерфейсе Python) (рис. 1.13):

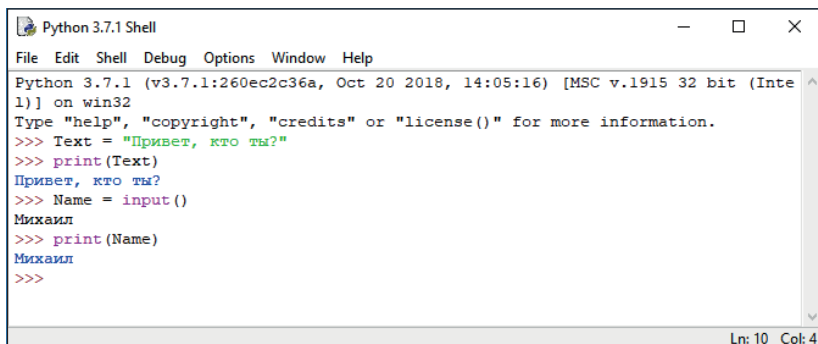


Рис. 1.13. Ввод команд в IDLE

Теперь среда разработки выглядит немного более красочной. Здесь команды наподобие `print()` и `input()`, а также вывод отображаются цветным шрифтом (это ведь удобно!).

- Попробуй сам, набрав строки примера в окне IDLE.

Работа с PY-файлами

Как же получается так, что всего лишь несколько строк становятся целой программой, которую можно загружать и выполнять снова и снова?

- Щелкни мышью по пункту **File** (Файл) в строке меню, а затем выбери пункт **New file** (Создать файл) (или нажми сочетание клавиш **Ctrl+N**) (рис. 1.14).

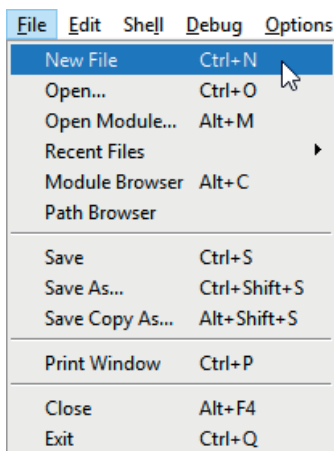


Рис. 1.14. Создание файла

Эта команда, только что выполненная тобой, создаст новый документ с именем *Untitled*. В этом окне тоже есть строка меню (рис. 1.15).

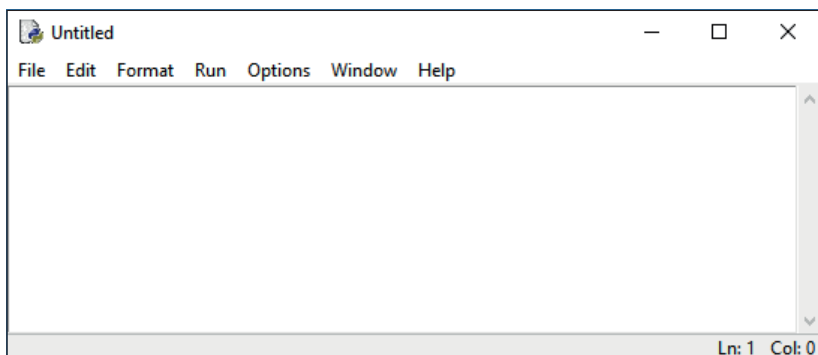


Рис. 1.15. Новый файл

Здесь мы сможем видеть нашу программу, не выполняя ни одной строки непосредственно в интерпретаторе Python.

- Введи следующие строки:

```
Text = "Привет! Кто ты?"
print(Text)
Name = input()
print(Name)
```

- Щелкни мышью по пункту **File** (Файл) в строке меню, а затем выбери пункт **Save as** (Сохранить как) (рис. 1.16).

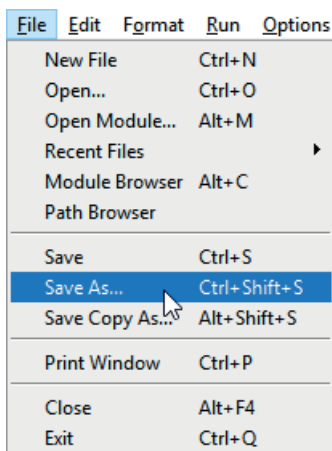


Рис. 1.16. Команда сохранения файла

1

- В диалоговом окне введи имя для этой программы, какое захочешь. Сама же программа автоматически добавит расширение *PY* как идентификатор программ для интерпретатора Python, если ты его сам не укажешь в имени (рис. 1.17). Нажми кнопку **Сохранить** (Save).

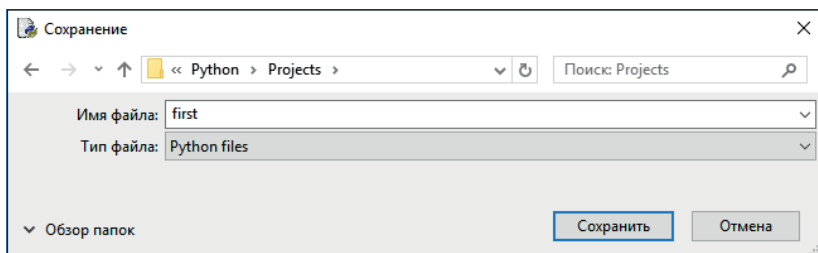


Рис. 1.17. Сохранение файла



Я создал подпапку под названием *Projects* в каталоге *Python* и сохранил в ней все свои проекты Python.

- Чтобы запустить программу, выбери команду меню **Run** ⇒ **Run Module** (Выполнение ⇒ Выполнить модуль). Или нажми клавишу **F5** (рис. 1.18).

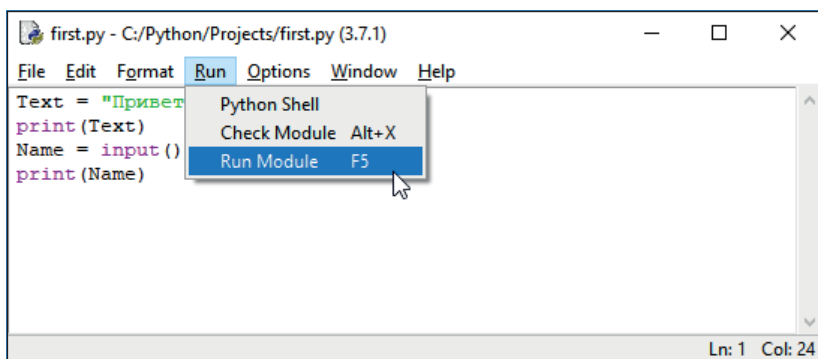


Рис. 1.18. Запуск программы

Ты вернешься к окну с приветствием «Привет! Кто ты?».

- Теперь введи свое имя и подтверди ввод с помощью клавиши **Enter**. Результат будет выглядеть так, как показано на рис. 1.19.

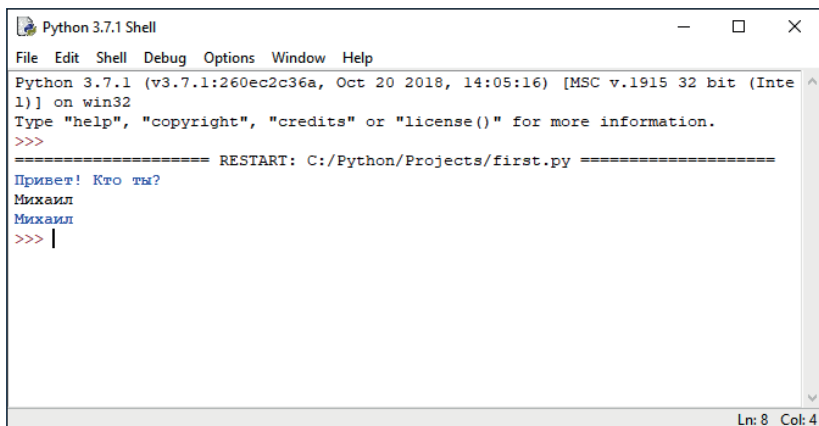


Рис. 1.19. Запущенная программа

Если сейчас ты закроешь это окно на своем компьютере, файл данной программы все равно будет сохранен. Тебе лишь нужно будет заново его открыть.

- Щелкни мышью по пункту **File** (Файл) в строке меню, а затем выбери пункт **Open** (Открыть) (или нажми сочетание клавиш **Ctrl+O**) (рис. 1.20).

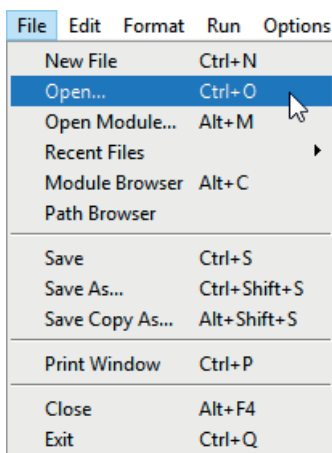


Рис. 1.20. Команда открытия файла

- В диалоговом окне выбери соответствующий файл (в папке *Python\Projects*) (рис. 1.21).

1

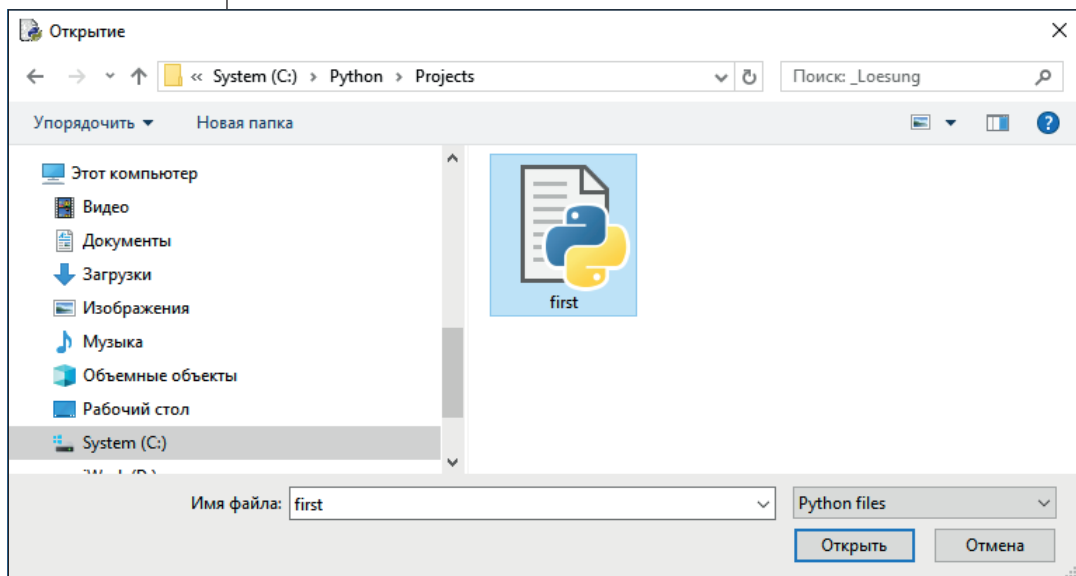


Рис. 1.21. Открытие файла

- Нажми кнопку **Открыть** (Open). Откроется окно редактора, и твоя первая программа будет готова к запуску.



В Python программу также называют *скриптом*.

Эксперименты с исходным кодом

Теперь давай более подробно рассмотрим *исходный код*, так как именно так называется последовательность этих строк:

```
Text = "Привет! Кто ты?"
print(Text)
Name = input()
print(Name)
```

Здесь есть две переменные, Text и Name. В каждой из них сохраняется последовательность символов, называемая *строкой*. Кроме того, здесь используются две функции:

`print()` Отвечает за вывод чисел и текста на экран

`input()` Отвечает за ввод чисел и текста с помощью клавиатуры

Как видишь, функции всегда имеют круглые скобки, в которых что-то может быть написано, но они также могут быть и пустыми, в зависимости от функции и ее применения. И если ты посмотришь немного внимательнее на них, то наверняка заметишь, что `print()` выглядит как инструкция, а `input()` – как присваивание. Таким образом, значение присваивается переменной `Name` с помощью этой функции.

Переменная = Формула

Я использую здесь термин «формула», потому что в правой части инструкции, помимо функций, можно использовать что-то вроде этого:

```
Amount = 1 + 2 * 3
Text = "Добрый вечер, " + Name
```

Обрати внимание, что символ плюс (+) играет двойную роль: позволяет складывать числа и конкатенировать (соединять) строки.



Теперь, когда ты знаешь, как написать несколько программных строк и сохранить их в файл, давай продолжим наш первый пример:

```
Text = "Привет! Кто ты?"
print(Text)
Name = input()
Text = "Так значит, ты - " + Name
print(Text)
print("А как у тебя дела?")
Answer = input()
print("Я рад, что у тебя дела " + Answer);
```

Как видишь, я не использую переменные на каждом шагу, на самом деле это необходимо, только если ты захочешь, чтобы компьютер что-то запомнил. В нашем примере это касается только данных, которые мы будем сами вводить. Соответственно, наша программа может выглядеть так:

```
print("Привет! Кто ты?")
Name = input()
print("Так значит, ты - " + Name)
print("А как у тебя дела?")
Answer = input()
print("Я рад, что у тебя дела " + Answer);
```

1

- Введи предыдущий исходный код и выбери команду меню **Run** ⇒ **Run Module** (Выполнение ⇒ Выполнить модуль) (или нажми клавишу **F5**). Затем измени код второй версией. Запусти программу снова.

Каждый раз, когда ты вносишь изменения, тебе будет предложено сохранить исходный код перед запуском программы (рис. 1.22):

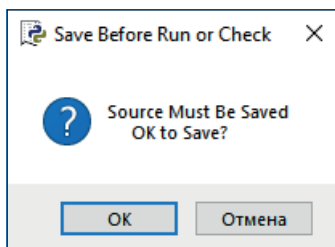


Рис. 1.22. Диалоговое окно с предупреждением

- Нажми кнопку **OK**.

Вот как будет выглядеть последняя версия нашей программы в окне интерпретатора Python (рис. 1.23):

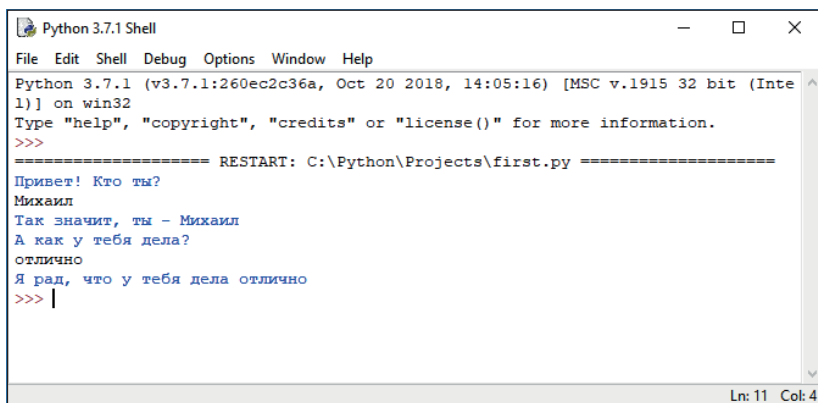
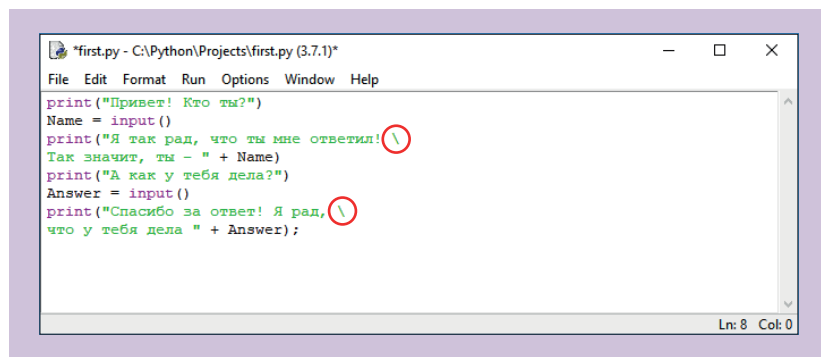


Рис. 1.23. Результат выполнения программы



Как видишь, интерпретатор Python читает код по строкам: в каждой строке есть оператор (даже знак равенства – это оператор присваивания). Поэтому ты не можешь просто перенести текст на две строки. Если это все же необходимо, в конце строки нужно указывать так называемый «обратный слеш» (\).



Выход из Python

Чтобы выйти из среды разработки Python, все открытые ранее окна должны быть, естественно, закрыты:

- Это можно сделать либо с помощью команды меню **File** ⇒ **Exit** (Файл ⇒ Выход), либо ты можешь щелкнуть по маленькому крестику (×) в правом верхнем углу каждого открытого окна. Или использовать сочетание клавиш **Ctrl+Q**.

Если ты ранее что-то изменил в коде и не сохранил файл, то увидишь окно, показанное на рис. 1.24.

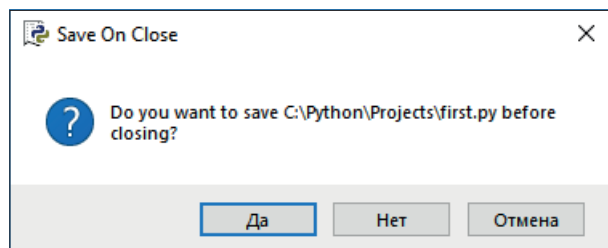


Рис. 1.24. Предупреждение о необходимости сохранения программы

- В зависимости от ситуации нажимай нужную кнопку. (Нажав кнопку **Отмена** (Cancel), ты вернешься обратно в окно редактора.)

Подведение итогов

С нашим первым проектом на языке программирования Python у тебя уже получилось кое-чего достичь. Давай-ка посмотрим, что ты узнал в этой главе. Прежде всего существует несколько команд, связанных с программой IDLE:

1

Запуск IDLE	Дважды щелкни мышью по ярлыку IDLE (Python)
Создать файл	Выбери команду меню File ⇒ New File (Файл ⇒ Создать файл)
Сохранить файл	Выбери команду меню File ⇒ Save (Файл ⇒ Сохранить)
Сохранить файл как...	Выбери команду меню File ⇒ Save as (Файл ⇒ Сохранить как)
Открыть файл	Выбери команду меню File ⇒ Open (Файл ⇒ Открыть)
Выполнить программу	Выбери команду меню Run ⇒ Run Module (Выполнение ⇒ Выполнить модуль)
Вызвать справочную систему	Выполни команду <code>help()</code> (используй команду <code>quit</code> для выхода)
Завершить работу Python	Выбери команду меню File ⇒ Exit (Файл ⇒ Выход)

А еще ты узнал кое-что из словаря языка Python:

<code>print()</code>	Функция, отвечающая за вывод чисел и текста на экран
<code>input()</code>	Функция, отвечающая за ввод чисел и текста с помощью клавиатуры
<code>=</code>	Оператор присваивания
<code>+</code>	Оператор сложения/конкатенации
<code>\</code>	Перенос строк (обратный слеш)

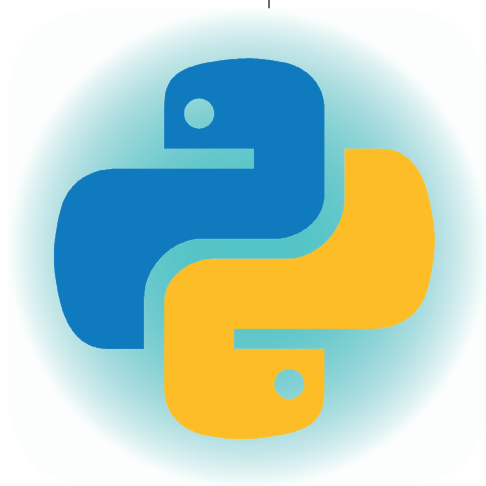
Несколько контрольных вопросов...

1. Какие функции отвечают за ввод и вывод информации?
2. Какая разница между компилятором и интерпретатором?
3. Инструкция присвоения – это то же самое, что и уравнение?

...а задач пока нет!

2

Условные конструкции



Только что тебе удалось немного «поговорить» со своим компьютером. Мне бы хотелось сказать это как-нибудь иначе, но выглядит это действительно глупо: твой компьютер делает только то, что говорит ему программа. Однако это не означает, что ты не сможешь довести компьютер до верного понимания и реакции.

В этой главе ты узнаешь:

- ⦿ что такое условные конструкции;
- ⦿ как сравнить значения переменных;
- ⦿ как использовать конструкции `if`, `else` и `elif`;
- ⦿ для чего преобразуются типы данных;
- ⦿ как использовать функции `int()` и `float()`;
- ⦿ как обрабатывать ошибки с помощью инструкций `try` и `except`.

Конструкция `if`

В предыдущей главе твоя небольшая программа спрашивала о твоих делах. Если ты запустишь эту программу несколько раз и введешь разные ответы, ты поймешь, насколько

нудным может быть такой разговор. Если же отвечать хорошо и плохо, твой оппонент будет выглядеть безразличным:

Я рад, что у тебя дела хорошо
Я рад, что у тебя дела плохо

Компьютер просто повторяет то, что ты ему говоришь. Если же ты ответишь вместо слова хорошо, например, грустно сегодня или просто наберешь несколько символов (например, +*!?фыва), тогда его реакция может выглядеть так (рис. 2.1):

Я рад, что у тебя дела грустно сегодня
Я рад, что у тебя дела +*!?фыва

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\first.py =====
Привет! Кто ты?
Михаил
Так значит, ты - Михаил
А как у тебя дела?
медленно двигаются вперед к светлому будущему
Я рад, что у тебя дела медленно двигаются вперед к светлому будущему
>>>
```

Рис. 2.1. Ответы компьютера

Итак, мы только что показали компьютеру, что надо правильно реагировать на такие ответы, как хорошо или плохо. Поэтому он не должен просто говорить: Я рад, что у тебя дела..., а наоборот – подбирать подходящую формулировку. Таким образом, его реакция может быть, например, такой:

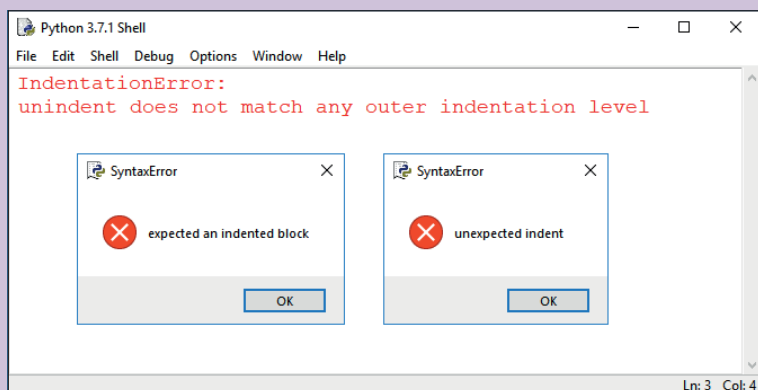
Твой ответ	Ответ компьютера
хорошо	Это радует!
плохо	Это огорчает!

Но как мы можем донести такое поведение до компьютера? В Python должно же быть что-то вроде понятия «если». Переведем это на английский (\Rightarrow *hello1*):

```
if Answer == "хорошо" :  
    print("Это радует!")  
if Answer == "плохо" :  
    print("Это огорчает!")
```

Ну как, стало интереснее? Содержание самого ответа сравнивается с текстом твоего ответа – хорошо или плохо. Если он совпадает, тут же выдается соответствующий ответ от компьютера.

Важная особенность программирования на языке Python, которую, к сожалению, можно легко упустить из виду: строка `print` ниже строки `if` имеет отступ. Для создания отступа нужно использовать пробелы (и никаких символов табуляции). Это необходимо для того, чтобы интерпретатор Python понимал, что строки `print` являются частью конструкции `if`. Если кто-то забывает о них или неправильно их вставляет (или слишком мало, или слишком много), появляется одно из таких сообщений об ошибке:



В зависимости от ситуации сообщение появляется либо непосредственно в окне оболочки, либо в виде диалогового окна.

Давай взглянем на оператор, который следует за именем переменной: в отличие от оператора присваивания (`=`), символ равенства указывается в этом случае дважды (`==`):

```
Answer == "хорошо"  
Answer == "плохо"
```

Это *оператор сравнения*, который проверяет, совпадают ли значения с обеих сторон. На самом деле очень легко можно



забыть указать символ равенства дважды и увидеть сообщение об ошибке, показанное на рис. 2.2.

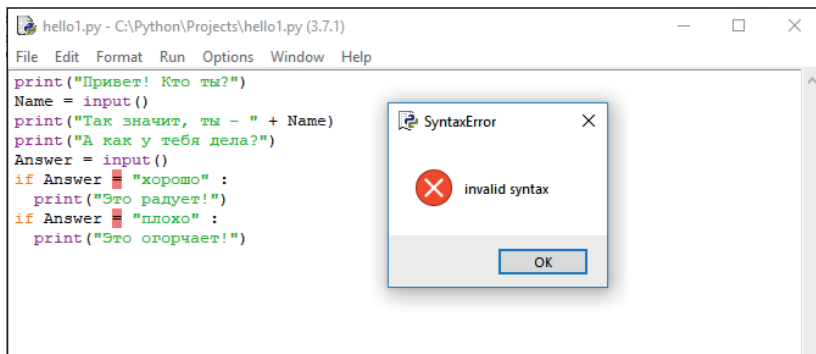


Рис. 2.2. Сообщение об ошибке – неправильный синтаксис

- Теперь открой файл примера из предыдущей главы и измени исходный код, как обсуждается на этом уроке. Не забудь про отступы! Сохрани код под новым именем (например, *hello1.py*).
- Затем запусти программу; введи ответ «хорошо» в первый раз, «плохо» во второй и посмотри, что произойдет. В соответствии с твоим ответом (как и должно быть) компьютер будет выдавать подходящий текст (рис. 2.3).

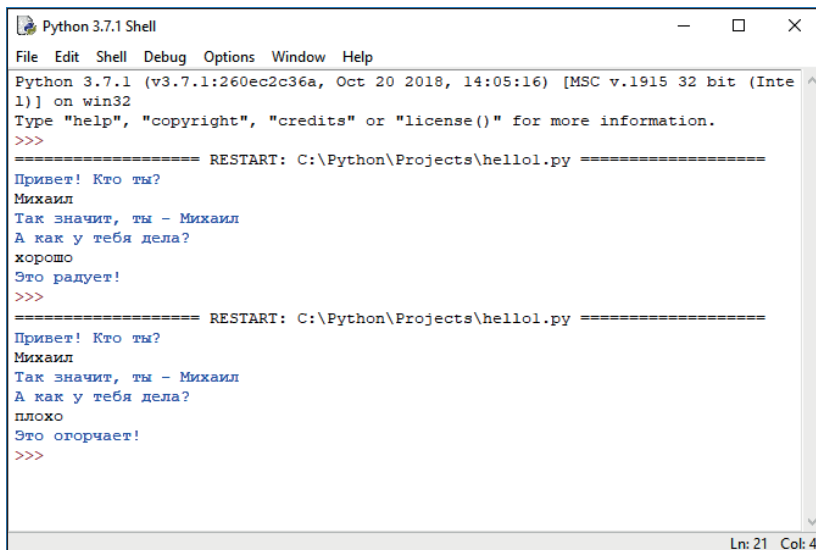


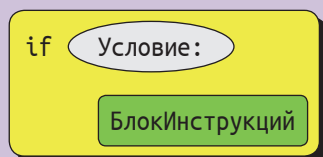
Рис. 2.3. Измененная версия программы из главы 1

Давай подробнее рассмотрим структуру, которая, кажется, дарит твоему компьютеру немного человеческого сострадания и сочувствия.

Ответ	Да	Нет
хорошо	Это радует!	(далее в программе)
плохо	Это огорчает!	(далее в программе)

В общем, это означает:

(Только) если выполнено конкретное условие, компьютер исполняет блок команд, если не выполнено – не исполняет.



Помни, что за условием всегда должно следовать двоеточие (:)! Кроме того, блок *инструкций* должен сопровождаться *отступом* (пробелами).



Условие здесь такое:

```
Answer == "хорошо" :
```

И следующее:

```
Answer == "плохо"
```

В конструкции имеются инструкции, в нашем примере такие:

```
print("Это радует!")
```

и

```
print("Это огорчает!")
```

Все это называется *конструкцией* или *условной конструкцией*, так как компьютер выбирает действие, исходя из со-

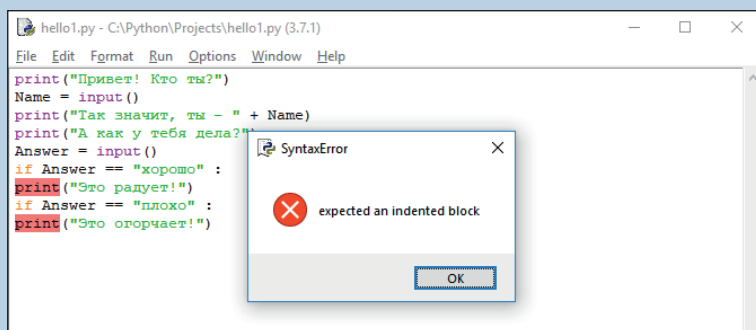
блюдения условия. Здесь его действие заключается в ответе тебе. Поэтому инструкции `print` являются частью конструкции `if`.

- Теперь попробуй запустить программу с кодом без пробелов.

Если исходный код выглядит следующим образом:

```
if Answer == "хорошо" :  
print("Это радует!")  
if Answer == "плохо" :  
print("Это огорчает!")
```

В этом случае Python не разберется, что здесь написано, и не поймет, что в коде есть конструкции `if`.



В начале строки обязательно должен быть, по крайней мере, один пробел. Когда ты вернешь пробелы в свой код, твоя программа снова заработает!

Конструкция `if else`

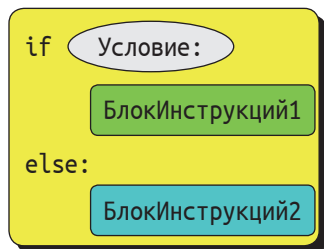
- Попробуй также изменить эту программу несколько раз с совсем другими словами в ответ на вопрос компьютера, кроме «хорошо» или «плохо». (Кстати, это также может быть и вариант «пучком», например.)

Поскольку компьютер не имеет инструкций насчет таких ответов, он и не сможет дать тебе ответ. Поэтому на экран ничего не выведется.

Для решения проблемы у нас есть такая команда, как `else`:

```
if Answer == "хорошо" :  
    print("Это радует!")  
else :  
    print("Я не понимаю ...")
```

Команда else означает «иначе». Это приводит нас к созданию новой условной конструкции с двумя альтернативными инструкциями.



Для нас с тобой это означает:

Если выполняется определенное условие, компьютер исполняет определенные инструкции (т.е. БлокИнструкций1). ЕСЛИ же не выполняется (т.е. else), компьютеру придется исполнять другой блок инструкций (БлокИнструкций2).



Условие:

```
Answer == "хорошо"
```

И в первом блоке инструкций находится эта строка:

```
print("Это радует!")
```

Компьютер выполняет эту инструкцию (выводит текст «Это радует!»), если в ответ на его вопрос ты ответил «хорошо», т.е. выполнил условие (==). Если ты отвечаешь что-то другое (=), выполняется второй блок инструкций – строка:

```
print("Я не понимаю ...")
```

Как и в случае простой инструкции if, мы говорим здесь об условной конструкции. Без команды else программа бы выглядела так:

```
If Answer == "хорошо" :  
    print("Это радует!")  
if Answer != "хорошо" :  
    print("Я не понимаю ...")
```

Второе условие в точности противоположно первому:

```
Answer != "хорошо"
```

Вместо двойного знака равенства (==) используется комбинация восклицательного знака и знака равно (!=).

- В приведенном выше коде замени строку `if Answer == "плохо"` конструкции строкой с ключевым словом `else`. Затем запусти свою программу.

Результат не совсем хорош: теперь компьютер знает только один подходящий ответ на слово `хорошо`! Я согласен, предыдущее решение было чуть лучше, так как тогда твой компьютер также отвечал и на слово `плохо`.

Как насчет такого решения (\Rightarrow *hello2*):

```
if Answer == "хорошо" :  
    print("Это радует!")  
if Answer == "плохо" :  
    print("Это огорчает!")  
else :  
    print("Я не понимаю ...")
```

Ну что, выглядит неплохо. Теперь оба условия снова оцениваются, и если они не подходят для ответа, выводится текст «Я не понимаю...». Нравится?

- Измени исходный код программы и запусти ее.

Если ввести слово `плохо`, программа работает прекрасно, но с ответом `хорошо` я получаю результат, показанный на рис. 2.4.

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\hello2.py =====
Привет! Кто ты?
Михаил
Так значит, ты - Михаил
А как у тебя дела?
хорошо
Это радует!
Я не понимаю ...
>>> |
```

Рис. 2.4. Проблема с отображением двух строк текста

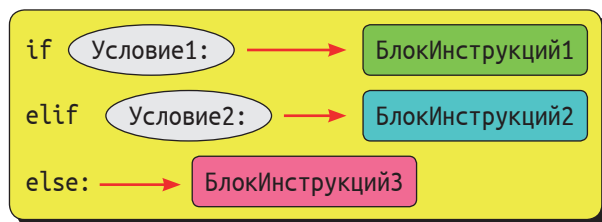
Оператор `else` срабатывает только вместе с последней конструкцией `if`. И как же нам добиться, чтобы этот оператор срабатывал и с первой конструкцией? Ты, наверное, уже подумал о чем-то вроде этого:

```
if Answer == "хорошо" :
    print("Это радует!")
else :
    print("Я не понимаю ...")
if Answer == "плохо" :
    print("Это огорчает!")
else :
    print("Я не понимаю ...")
```

Но если честно, мне эта идея не нравится. Тем более что я представляю, насколько сложным было бы, если бы я захотел оценить еще несколько ответов, кроме «хорошо» и «плохо». Но хорошее решение проблемы не так уж и сложно. Вот весь необходимый код (\Rightarrow *hello2*):

```
print("Привет! Кто ты?")
Name = input()
print("Так значит, ты - " + Name)
print("А как у тебя дела?")
Answer = input()
if Answer == "хорошо" :
    print("Это радует!")
elif Answer == "плохо" :
    print("Это огорчает!")
else :
    print("Я не понимаю ...")
```

Давай укажем слово `else` перед оператором `if` и объединим их вместе в инструкцию `elif`. Это т. н. вложенная инструкция.



- Измени исходный код, сохрани проект и запусти программу несколько раз. Попробуй ввести несколько ответов и, разумеется, наши допустимые варианты – хорошо и плохо.

Работает?

Простые вычисления

Мы возимся все с одной и той же программой. Пришло время чего-то новенького! Сначала тебе нужно закрыть окно редактора с загруженной программой приветствия. Так как эта программа больше не понадобится.

- Итак, щелкни мышью по маленькому значку «крестика» (*) в правом верхнем углу. Затем нам понадобится создать пустой файл.
- Щелкни мышью по пункту **File** (Файл) в строке меню, а затем выбери пункт **New file** (Создать файл) (или нажми сочетание клавиш **Ctrl+N**) (рис. 2.5).

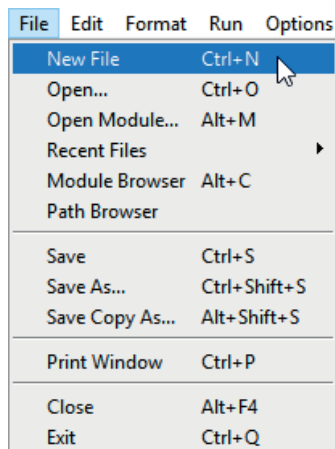


Рис. 2.5. Команда открытия файла

Откроется новый документ, в котором можно работать! Теперь твой компьютер должен доказать, что он умеет, по крайней мере, складывать числа. Для некоторых математика – веселое занятие, а вот для других она просто невыносима. Но поскольку компьютер должен уметь считать, все это пойдет нам только на пользу.

➤ Введи текст этой программы (\Rightarrow *mathe1*):

```
# Математика
print("Введи число: ")
Number1 = input()
print ("Введи еще одно число: ")
Number2 = input()
print ("А теперь выбери операцию (+,-,*,/): ")
Operator = input()
print ("Результат = ")
if Operator == "+" :
    print(Number1 + Number2)
if Operator == "-" :
    print(Number1 - Number2)
if Operator == "*" :
    print(Number1 * Number2)
if Operator == "/" :
    print(Number1 / Number2)
```

Полный исходный код ты сможешь найти в файлах примеров, которые можно скачать на странице dmkpress.com.



Прежде чем запускать эту программу, тебе нужно будет кое-что узнать. В верхней части кода программы присутствует строка с символом решетки (#). С помощью этого символа ты можешь записать комментарий, который игнорируется интерпретатором Python.

Таким образом также можешь отмечать версии программы с комментариями типа # Приветствие исходная версия или # Приветствие измененная версия. Или любые другие комментарии на твой вкус.



Вернемся к программе. Вначале запрашивается два любых числа и один оператор (математический). Переменные имеют соответственно имена Number1, Number2 и Operator. Затем следует текст, который применяется ко всем результатам: "Результат = ".



И наконец, конструкция `if`, которая выполняет определенную арифметическую операцию, в зависимости от того, какой оператор ты ввел.

В итоге тебе будет отображен результат этого вычисления.

Опять же, не забудь проверить написанное, так как некоторые инструкции должны иметь отступ (пробел) в начале строки.

Разумеется, в Python допустимы все четыре основные арифметические операции. Но нужно помнить, что некоторые из них отличаются от тех, которые ты знаешь из уроков математики или работы с калькулятором:

	Математика	Калькулятор	Компьютер
Сложение	+	+	+
Вычитание	-	-	-
Умножение	·	×	*
Деление	:	÷	/

Как ты видишь, различия только в оформлении операций умножения и деления.

- А теперь запусти программу и протестируй ее с несколькими разными числами и всеми типами арифметических операций.

Странно, но программа работает необычно. Особенно при сложении (+) программа хоть и работает, но выводит на экран странный результат, показанный на рис. 2.6.

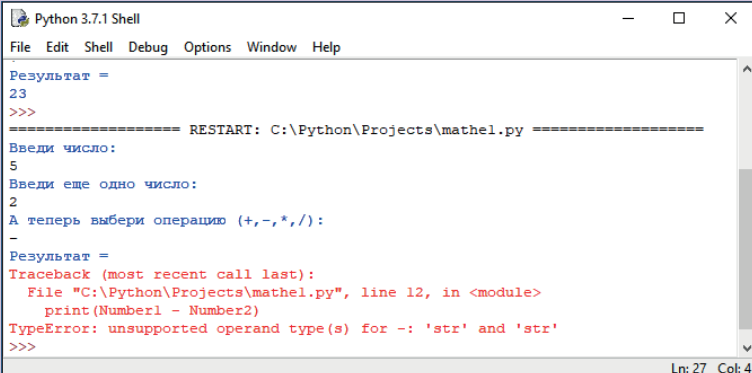
```

Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\mathel.py =====
Введи число:
2
Введи еще одно число:
3
А теперь выбери операцию (+, -, *, /):
+
Результат =
23
>>> |
  
```

Рис. 2.6. Непонятный результат сложения

Результат 2+3 равен 23? Это же неправда. Или правда? Проблема в том, что здесь все числа были введены функцией `input()`, которая применяется для символов или строк.

Все остальные операторы приводят лишь к сообщениям об ошибках:



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help

Результат =
23
>>>
===== RESTART: C:\Python\Projects\mathel.py =====
Введи число:
5
Введи еще одно число:
2
А теперь выбери операцию (+, -, *, /):
-
Результат =
Traceback (most recent call last):
  File "C:\Python\Projects\mathel.py", line 12, in <module>
    print(Number1 - Number2)
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>>
```

Потому что другие операторы строковых типов данных вообще не поддерживают.



Вкратце о числах

Нам нужны числа для вычислений. Но как превратить их из строкового типа данных в числовой? Взгляни на небольшой пример обработки чисел:

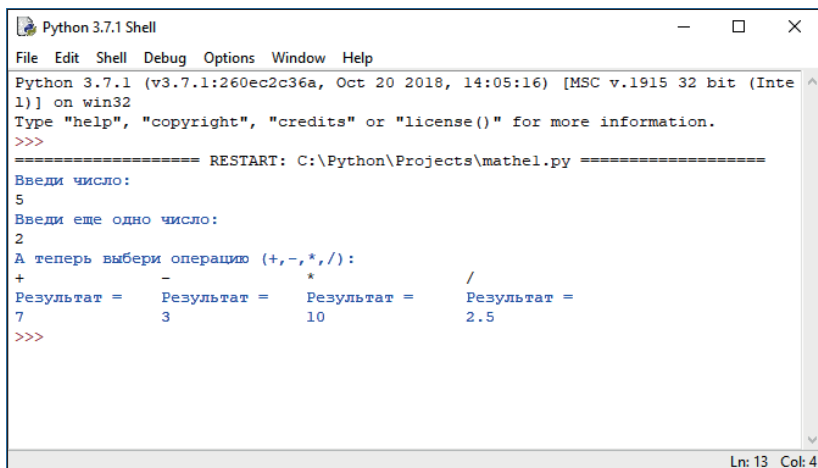
```
print("Введи число: ")
Number1 = int(input())
print("Введи еще одно число: ")
Number2 = int(input())
```

`int()` – функция целочисленного типа данных. Она превращает значение переменной (строку) в целое число (сокращение *int* происходит от слова *integer*, что в переводе с английского означает «целое число»).

Здесь вложены две функции, одна в другую: `input()` принимает ввод символов с клавиатуры, а `int()` превращает введенные (строковые) символы в целые числа. Другими словами, `int()` принимает значение другой функции, указанной в скобках.



- Измени исходный код, а затем сохрани файл под именем *mathe1.py*. Теперь запусти программу несколько раз и попробуй все математические операторы (рис. 2.7).



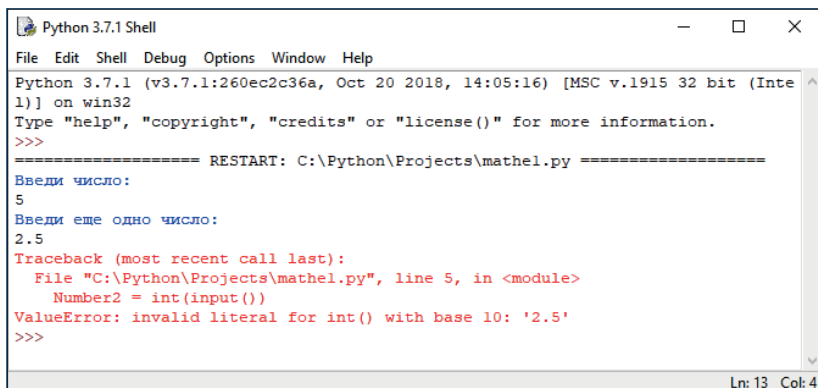
```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\mathe1.py =====
Введи число:
5
Введи еще одно число:
2
А теперь выбери операцию (+, -, *, /):
+ - * /
Результат =      Результат =      Результат =      Результат =
7          3          10          2.5
>>>
```

Рис. 2.7. Теперь программа работает корректно (на этой иллюстрации я немного схитрил и собрал все результаты вместе)

Как видишь, операция деления выводит правильный результат; даже если два целых числа фактически делятся на нецелое число. Но когда ты пытаешься ввести число с плавающей запятой, возникает проблема (рис. 2.8).



Важно, что, вводя «числа с плавающей запятой», всегда нужно указывать *точку* вместо запятой.



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\mathe1.py =====
Введи число:
5
Введи еще одно число:
2.5
Traceback (most recent call last):
  File "C:\Python\Projects\mathe1.py", line 5, in <module>
    Number2 = int(input())
ValueError: invalid literal for int() with base 10: '2.5'
>>>
```

Рис. 2.8. Ошибка при вводе чисел с плавающей запятой

В этом примере Python не способен ничего сделать с числом с плавающей запятой, таким как 2,5, потому что оно не является целым числом, вот и возникает ошибка. Чтобы решить проблему и избежать сообщения об ошибке, нужно использовать другой тип чисел, отличный от целых (`int`). Для этого используется тип `float`, означающий *число с плавающей запятой*. Изменения в файле выглядят примерно так (\Rightarrow *mathe2*):

```
print("Введи число: ")
Number1 = float(input())
print("Введи еще одно число: ")
Number2 = float(input())
```

- Измени соответствующие строки в своей программе, а затем запусти ее. И, как ты убедился, теперь она спокойно работает с числами с плавающей запятой (рис. 2.9).

The screenshot shows a Python 3.7.1 Shell window with the following content:

```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\mathe2.py =====
Введи число:
2.5
Введи еще одно число:
0.5
А теперь выбери операцию (+, -, *, /):
+
Результат =
3.0
>>> |
```

The status bar at the bottom right indicates "Ln: 13 Col: 4".

Рис. 2.9. Теперь числа с плавающей запятой поддерживаются

Осталась последняя проблема, которая возникает только в том случае, если ввести число 0, а затем выполнить операцию деления. Ты тут же получишь уведомление об ошибке, как показано на рис. 2.10.

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\mathe2.py =====
Введи число:
5
Введи еще одно число:
0
А теперь выбери операцию (+, -, *, /):
/
Результат =
Traceback (most recent call last):
  File "C:\Python\Projects\mathe2.py", line 16, in <module>
    print(Number1 / Number2)
ZeroDivisionError: float division by zero
>>>
```

Рис. 2.10. Ошибка при делении на 0

Проблему с делением на 0 проще решить, чем ты мог себе представить (\Rightarrow *mathe2*):

```
if Operator == "/" :
    if Number2 != 0:
        print(Number1 / Number2)
    else:
        print("На ноль делить нельзя!")
```

Все, что требуется сделать, – добавить еще одну конструкцию `if`. Теперь компьютер выполнит операцию деления только в том случае, если ты ввел число, отличное от 0. В противном случае будет выведено сообщение `На ноль делить нельзя!`.

- Дополни код программы указанными строками. Обрати внимание, что добавленная конструкция отбивается соответствующими отступами (пробелами) (рис. 2.11).

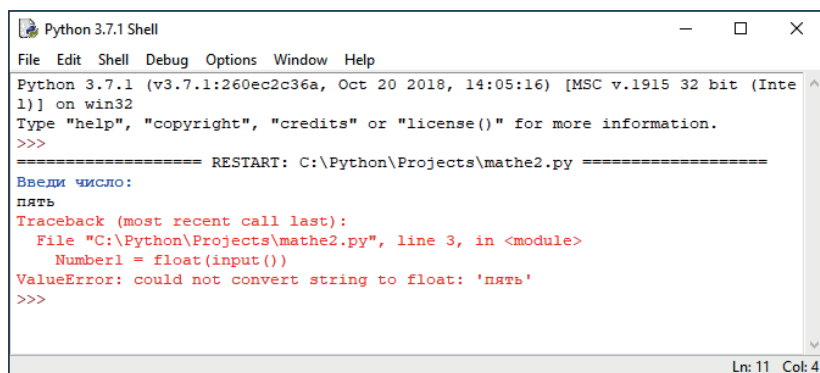
```
if Operator == "+" :
    print(Number1 + Number2)
if Operator == "-" :
    print(Number1 - Number2)
if Operator == "*" :
    print(Number1 * Number2)
if Operator == "/" :
    if Number2 != 0:
        print(Number1 / Number2)
    else:
        print("На ноль делить нельзя!")
```

Рис. 2.11. Обрати внимание на дополнительные отступы у вложенной конструкции

Теперь ты сможешь запустить программу и ввести любые числа, включая ноль.

Работа с командами try и except

Будучи программистом, ты не можешь быть стопроцентно уверен в том, что пользователь твоей программы случайно не сделает ошибки. Вполне возможно, что он случайно введет букву или какой-то иной символ вместо числа. Это однозначно приведет к прерыванию исполнения программы с выводом сообщения об ошибке.



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\mathe2.py =====
Введи число:
пять
Traceback (most recent call last):
  File "C:\Python\Projects\mathe2.py", line 3, in <module>
    Number1 = float(input())
ValueError: could not convert string to float: 'пять'
>>>
```

Рис. 2.12. Ошибка при вводе текста вместо числа

Я попытался ввести в этой программе «пять» в качестве числа. Ясно, что слово невозможно преобразовать в число. Но нельзя ли использовать в программе механизм, позволяющий реагировать на такие небольшие промахи?

Подобное поведение программируется, так сказать, исключительно в экспериментальных целях.

Если сработает – отлично. Если нет, тогда предпримем действие по обработке ошибок, которое ты выберешь сам. И такой механизм действительно существует в Python.

Такие предсказуемые ошибки называются *исключениями*. А механизм их обработки используется для того, чтобы *перехватывать* исключения до того, как произойдет *прерывание* программы.



Давай подробнее рассмотрим показанный ниже исходный код (\Rightarrow *mathe3*).

```
try :
    print("Введи число: ")
    Number1 = float(input())
    print ("Введи еще одно число: ")
    Number2 = float(input())
    print ("А теперь выбери операцию (+,-,*,/): ")
    Operator = input()
    print ("Результат = ")
    if Operator == "+" :
        print(Number1 + Number2)
    if Operator == "-" :
        print(Number1 - Number2)
    if Operator == "*" :
        print(Number1 * Number2)
    if Operator == "/" :
        if Number2 != 0:
            print(Number1 / Number2)
        else:
            print("невозможно вычислить")
except :
    print("Не число!")
```

Видишь два новых слова в этом тексте программы? Сначала указано ключевое слово `try`, а затем `except`.



Как тебе скоро станет понятно, это новые помощники в нашем деле программирования. Они влияют на каждую строку исходного кода предыдущей версии программы.

Давай рассмотрим эту «сладкую» парочку, `try` и `except`, немного поближе.

Опять же, их можно назвать условной конструкцией:

```
try :
    БлокИнструкций1
except :
    БлокИнструкций2
```

Для интерпретатора это означает примерно следующее:

Сначала попробовать выполнить первый блок инструкций, а если будет выброшена ошибка, тогда выполнить второй блок инструкций.



Блок инструкций `try` включает все инструкции предыдущей версии программы:

```
try:
    # здесь находятся все инструкции,
    # способные привести к исключениям
```

Блок инструкций `except` включает инструкции, которые следует исполнять в случае перехвата исключения:

```
except :
    # здесь находятся все инструкции,
    # исполняемые в случае
    # перехвата исключения
```

Я хотел бы еще раз напомнить, что символ решетки (#) в начале строки обозначает комментарий, который пропускает интерпретатор Python.



Теперь программа перехватывает исключения, как показано на рис. 2.13.

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\mathe3.py =====
Введи число:
пять
Не число!
>>> |
```

Рис. 2.13. Реакция программы на ввод слова вместо числа

Подведение итогов

Прежде чем ты сможешь насладиться заслуженным отдыхом, давай сначала посмотрим, что же получилось у нас с тобой в итоге. В любом случае, ты однозначно расширил свой словарный запас при работе в интерпретаторе Python. Давай еще раз освежим знания.

2

int()	Преобразование значения переменной (цифр) в целое число
float()	Преобразование значения переменной в число с плавающей запятой
if	Если выполняется условие, выполняется БлокИнструкций1
else	Исполняется БлокИнструкций2
elif	Выполняется проверка сложного условия, и выполняется БлокИнструкций, если условие истинно
try	Попытка исполнить БлокИнструкций1
except	Если возникает ошибка, исполняется БлокИнструкций2
=	Оператор присваивания (напоминаю!)
+ - * /	Базовые арифметические операторы
+	Оператор сложения
==	Оператор сравнения, проверка на равенство
!=	Оператор сравнения, проверка на неравенство
#	Комментарий



Слова if, else, elif, try, кроме того что относятся к основному словарю Python, также называются *ключевыми словами*.

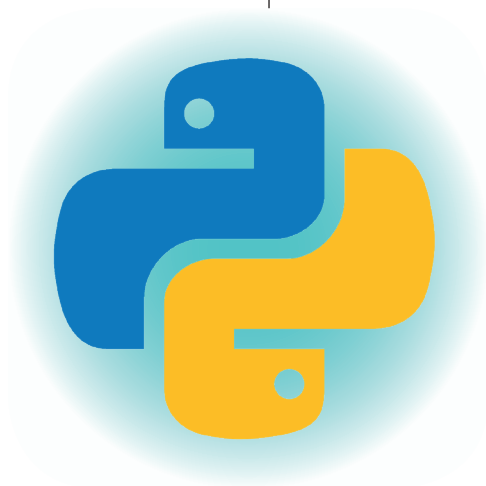
Несколько вопросов...

1. В чем разница между символами = и ==?
2. Как упростить показанный ниже фрагмент программы:

```
if Number == 0 :  
    print("Нет результата");  
if Number != 0 :  
    print(1/Number);
```

...и задач

1. Напиши программу, в которой после ввода числа вычисляется и отображается его обратное значение. (Выполни также эту программу, введя ноль.)
2. Напиши программу, запрашивающую пароль. Если пароль введен правильно, выдается сообщение «Доступ разрешен», в противном случае отображается текст «Доступ разрешен!» – или, если захочешь, в каждом случае напиши собственный вариант ответа.



3

Сравнение и повторение

Здесь речь будет идти все о том же, что и в предыдущих главах, только примеры станут сложнее и будут включать проверку не только на равенство. Кроме того, мы еще раз рассмотрим, как обрабатывать неправильный ввод и предотвращать ошибки программы, например путем предоставления нескольких попыток ввода.

Попытки ввода помогут не только в такой ситуации. Существует множество приложений и игр, где предоставление попыток пользователю – просто необходимая функция!

В этой главе ты узнаешь:

- ⦿ как связывать условия;
- ⦿ как интегрировать новые словари (модули);
- ⦿ о генерации случайных чисел;
- ⦿ как многократно исполнять фрагменты кода программы;
- ⦿ как использовать циклы `while`;
- ⦿ как выходить из цикла.

Оценки

Давай создадим еще одну математическую программу. Наш третий проект касается оценок. Логичный шаг после

3

того, как мы поработали с математической программой, которая уже привнесла в эту книгу немного напоминаний о школьных буднях.

- Итак, создай новый файл в среде разработки Python (IDLE), щелкнув мышью по пункту **File** (Файл) в строке меню, а затем выбрав пункт **New file** (Создать файл) (или нажав сочетание клавиш **Ctrl+N**).

Эта программа должна будет оценивать тебя по введенному целому числу. Для вычисления оценки я взял табл. 3.1.

Таблица 3.1. Оценка за введенное число

Диапазон введенных чисел	Оценка (текст)	Оценка (балл)
0–24	ужасно	6
25–44	плохо	5
45–64	сойдет	4
65–79	средне	3
80–89	хорошо	2
90–100	отлично	1

Конечно, ты можешь изменить эту таблицу по своему усмотрению, если хочешь.

Давай рассмотрим, как же ее задать компьютеру. Начнем с худшей оценки:

Если введенное число находится в диапазоне от 0 до 24, пусть программа выводит текст ужасно (6).

Это будет первая конструкция. Выглядеть она будет так:

```
if Score between 0 and 24 :  
    print("ужасно (6)")
```

Как жаль, что Python не понимает слово *between*! Здесь потребуется немного больше, чем одна лишь математика. Попробуем сформулировать это так: слово *between* означает, что число, с одной стороны, должно быть больше или равно 0: сформулируем это как условие:

```
Score >= 0
```

С другой стороны, число меньше 25. Это приводит к следующему условию:

```
Score < 25
```

(Учтем значения с плавающей запятой, скажем 24,5, так как числа не обязательно должны быть целыми.)

Теперь важно совместить два условия. За это в Python отвечает команда `and`:

```
(Score >= 0) and (Score < 25)
```

Круглые скобки в коде необязательны, так как Python распознает то, что относится к первому условию, и что ко второму. Скобки я добавил для наглядности, чтобы вы не запутались.



Вот это круто! А теперь создадим такие конструкции для остальных пяти оценок – как это будет выглядеть?

Полный код показан ниже (\Rightarrow *grade*):

```
# Отметка
print("Введи число: ", end="")
Score = int(input())
print("Это ", end = "")
if (Score >= 0) and (Score < 25) :
    print("ужасно (6)")
if (Score >= 25) and (Score < 45) :
    print("плохо (5)")
if (Score >= 45) and (Score < 65) :
    print("сойдет (4)")
if (Score >= 65) and (Score < 80) :
    print("средне (3)")
if (Score >= 80) and (Score < 90) :
    print("хорошо (2)")
if (Score >= 90) and (Score <= 100) :
    print("очень хорошо (1)")
if (Score > 100) or (Score < 0) :
    print("неправильно (0)")
```

- Введи этот исходный код и сохрани его в файл под именем *grade1.py*.

Ты можешь ускорить работу с кодом, если будешь применять в работе команды копирования, вырезания и вставки.

- ❖ Сначала необходимо выделить текст, удерживая клавишу **Shift** и нажимая клавиши со стрелками, либо мышью, удерживая ее левую кнопку.



3



- ❖ С помощью команды **Cut** (Вырезать) в меню **Edit** (Правка) или в контекстном меню ты можешь вырезать выделенный текст и поместить его в буфер обмена.
- ❖ С помощью команды **Copy** (Копировать) в меню **Edit** (Правка) или в контекстном меню ты можешь скопировать выделенный текст в буфер обмена.
- ❖ С помощью команды **Insert** (Вставить) в меню **Edit** (Правка) или в контекстном меню ты можешь вставить скопированный код из буфера обмена в файл.

Затем тебе нужно скорректировать скопированные строки кода.

- Запусти программу, выбрав команду меню **Run** ⇒ **Run Module** (Выполнение ⇒ Выполнить модуль). Также ты можешь нажать клавишу **F5**. И попробуй выполнить ее с несколькими значениями (попробуй вводить также несуществующие значения для проверки) (рис. 3.1).

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\gradel.py =====
Введи число: 88
Это хорошо (2)
>>>
===== RESTART: C:\Python\Projects\gradel.py =====
Введи число: 101
Это неправильно (0)
>>> |
```

Рис. 3.1. Проверка программы для оценки

Обрати внимание, что запрос и введенное тобой число размещаются в одной строке. Так же, как и финальный вывод программы со словом Это.

Мы обязаны этому благодаря дополнительному параметру в круглых скобках в инструкции `print`:

```
print("Введи число: ", end="")
print("Это ", end = "")
```

Все дело в том, что по умолчанию функция `print()` после вывода текста переводит текстовый курсор на одну строку ниже, поэтому следующий вывод находится в следующей строке. Вероятно, ты обратил внимание на такое поведение.

ние в наших предыдущих программах, хотя это и не ошибка программы в принципе.

Чтобы заставить нашу любимую функцию `print()` отказаться от разрыва строки, нужно использовать дополнительный параметр. С помощью параметра `end = ""` мы даем ей понять, что должно происходить после выполнения инструкции `print`.

Две двойные кавычки означают, что после выполнения инструкции `print` не следует никакая операция, в том числе разрыв строки. Текстовый курсор остается там, где остановился после выполнения инструкции, и с этого места начнется следующий ввод или вывод. (Чуть больше практики в шлифовке математических программ ты получишь немного позже.)

Так, все условия собраны вместе? Если ты внимательно изучишь код, то в предпоследней строке программы увидишь вместо знакомого оператора `and` слово `or`.

Эти два слова являются так называемыми *связывающими операторами*. Они представляют собой, так сказать, клей для соединения нескольких условий вместе. Вот значения, которые они имеют:

<code>and</code>	Должны выполняться все условия для исполнения последующего блока инструкций
<code>or</code>	Должно выполняться только одно условие для исполнения последующего блока инструкций

Понятно, что для всех диапазонов чисел каждый раз должны выполняться оба условия. Но есть еще одна инструкция `if`, последняя:

```
if (Score > 100) or (Score < 0) :
    print("неправильно (0)")
```

То есть в случае, если кто-то решит обмануть программу и ввести число, превосходящее 100 (а набрать столько баллов, скажем, в ЕГЭ, нереально), мы и определяем подобное условие:

```
Score > 100
```

Отрицательные числа также не допускаются в программе, так как человек не может набрать на экзамене меньше 0 баллов! Поэтому это условие выглядит так:

```
Score < 0
```

Одно из двух условий должно быть истинным. (Или ты знаешь число, одновременно отрицательное и больше 100?) Чтобы предотвратить эту ошибку, которую создаст оператор `and`, нам придется заменить его на оператор `or`.

В зависимости от условий в Python используются различные *операторы сравнения*. На данный момент ты их все уже изучил. Вот они:

Оператор	Описание	Оператор	Описание
<code>==</code>	Равно	<code>!=</code> ¹	Не равно
<code><</code>	Меньше	<code>>=</code>	Больше или равно
<code>></code>	Больше	<code><=</code>	Меньше или равно

Инструкции с операторами сравнения всегда указываются в одной строке, первое значение в которых противоположно второму. Операторы сравнения для равных и неравных значений ты уже знаешь.



Небольшая игра на угадывание

Почему бы нам не использовать наши математические познания в небольшой игре на угадывание? Тем более ты сможешь гордиться тем, что написал свою собственную небольшую игру.

Так о чем идет речь? Компьютер загадывает число в диапазоне от 1 до 1000. И ты должен угадать это случайное число за наименьшее число попыток.

Для этого сначала нужна функция, которая генерирует случайное число. К сожалению, мы не найдем такую в стандартном словаре языка программирования Python. Но во внешнем модуле, который также относится к Python, есть подходящая. Этот модуль называется `gandom` (от англ. – случайный).

Нам лишь нужно сообщить интерпретатору Python, что ему нужно использовать (т. е. *импортировать*) этот модуль. Начнем:

- закрой окно редактора с кодом предыдущей программы и создай новую. Введи следующий код в файле программы (\Rightarrow `guess.py`):

¹ С той же целью может использоваться оператор `<>`. – Прим. перев.

```
import random
Secret = "Я задумал число от 1 до 1000"
Case = random.randint(1,1000)
print(Case)
print(Secret)
print("Угадай число: ", end="")
Input = int(input())
if Input < Case :
    print("Слишком маленькое!")
if Input > Case :
    print("Слишком большое!")
if Input == Case :
    print("Правильно!")
```

В самом верху кода программы ты можешь увидеть, как мы добавляем (импортируем) нужный модуль:

```
import random
```

Обращение к внешнему модулю происходит с помощью ключевого слова `import`, за которым следует имя модуля, в нашем случае `random`, что означает «случайно». Из этого модуля нам нужна одна специальная функция с именем `randint()`. Обрати внимание, что ее обычным способом не вызвать:

```
Case = randint(1,1000)
```

Ничего не выйдет. Мы также должны указать имя импортируемого модуля, в котором содержится эта функция:

```
Case = random.randint(1,1000)
```

Значения в скобках – напомним, что они называются *параметрами*, – определяют начало и конец диапазона чисел, из которого компьютером будет выбираться случайное число. И как уже упоминалось, при использовании инструкции `randint` речь будет идти о целых числах.

Я добавил команду `print` для отображения числа, которое должно быть угадано. Так мы сможем проверить правильность работы программы. Позже, правда, придется удалить эту строку.



Ниже приводится правило (`Case`) игры, а затем появляется приглашение для ввода числа:

3

```
print(Case)
print(Secret)
print("Угадай число: ", end="")
Input = int(input())
```

Затем ты вводишь целое число, и компьютер проверяет, больше оно, меньше или даже совпадает с задуманным:

```
if Input < Case :
    print("Слишком маленькое!")
if Input > Case :
    print("Слишком большое!")
if Input == Case :
    print("Правильно!")
```

Таким образом, ты будешь понимать, в каком направлении думать, чтобы угадать загаданное компьютером число.

Ну как, не терпится запустить и опробовать программу? Давай сделаем это!

- Сохрани свою программу под именем *guess1.py*, а затем запусти ее.

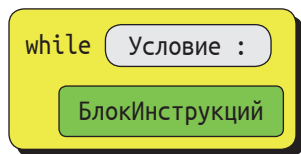
Программа работает, но, к сожалению, играть невозможно, так как допускается всего лишь единственная попытка угадать загаданное число. Этого недостаточно! Даже если ты будешь перезапускать программу снова и снова, это не поможет: компьютер каждый раз будет придумывать новое число при запуске. Итак, удовольствие от игры мы сможем получить тогда, когда у нас будет возможность попытаться угадать задуманное число несколько раз. Для этого нам необходимо повторять в программе некоторый код несколько раз.

Пока число не было угадано, необходимо продолжать запрашивать ввод и проверять числа, которые мы вводим. Для компьютера, естественно, это должно быть сформулировано на языке Python. Кроме того, «число не было угадано» – слишком размытое выражение. Оно должно быть математически точным:

```
Input != Case
```

И вот соответствующая конструкция для повторного исполнения кода:

```
while Input != Case
```



В общем, это значит:

Пока выполняется определенное условие, компьютер будет повторять исполнение блока инструкций.



Теперь мы должны добавить новую строку в нужную позицию кода. То есть применить ко всем строкам инструкций, которые необходимо повторить:

```
while Input != Case :
    print("Угадай число: ", end="")
    Input = int(input())
    if Input < Case :
        print("Слишком маленькое!")
    if Input > Case :
        print("Слишком большое!")
    if Input == Case :
        print("Правильно!")
```

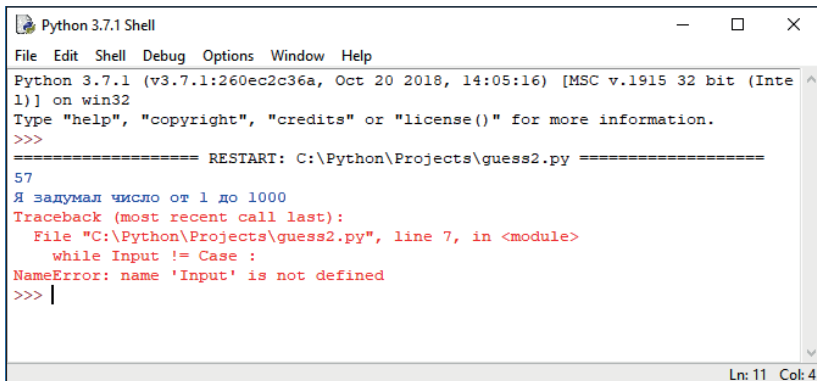
Здесь мы также используем двоеточие (как и в конструкциях if).

Все строки с инструкциями, относящиеся к условной конструкции while, обязательно должны иметь отступ (пробел слева).



- Измени исходный код своей программы, как показано выше, и запусти свою программу.

Не работает, как показано на рис. 3.2? Получается, что интерпретатор Python не понял, что ты добавил новую переменную?



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\guess2.py =====
57
Я задумал число от 1 до 1000
Traceback (most recent call last):
  File "C:\Python\Projects\guess2.py", line 7, in <module>
    while Input != Case :
NameError: name 'Input' is not defined
>>> |
```

Рис. 3.2. Ошибка интерпретатора

А ведь верно: во время обработки строки со сравнением `Input != Case` переменная `Input` еще не существует. Она будет создана только тогда, когда мы присвоим ей какое-то значение. А она создается позднее, в строке

```
Input = int(input())
```

Поэтому нам необходимо присвоить переменной значение перед конструкцией `while`, например так:

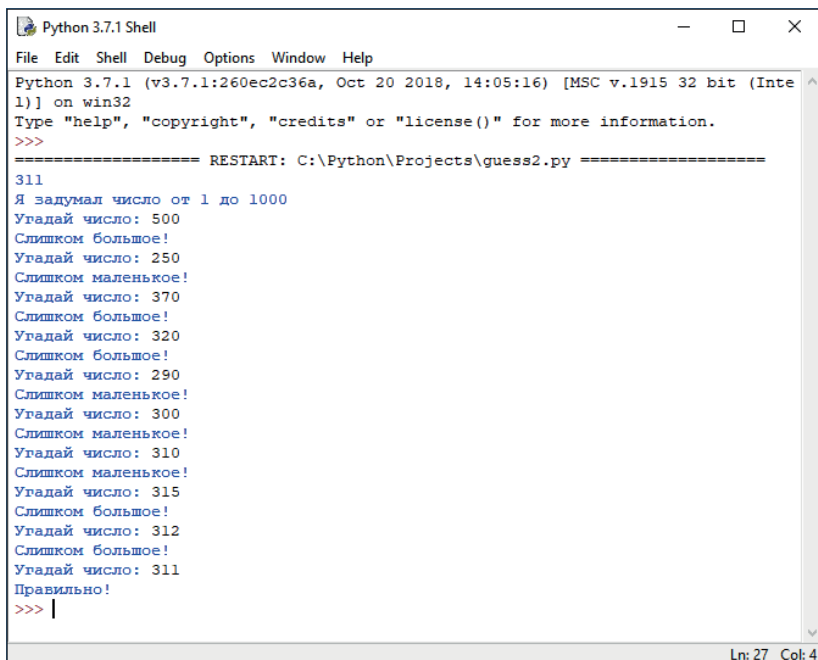
```
Input = 0
```

Это может быть любое число, но вне диапазона от 1 до 1000. В таком случае переменная будет существовать и может быть использована, в том числе и для сравнения.

Если мы соберем все вместе, финальная программа будет выглядеть так (\Rightarrow *guess2.py*):

```
import random
Secret = "Я задумал число от 1 до 1000"
Case = random.randint(1,1000)
print(Case)
print(Secret)
Input = 0
while Input != Case :
    print("Угадай число: ", end="")
    Input = int(input())
    if Input < Case :
        print("Слишком маленькое!")
    if Input > Case :
        print("Слишком большое!")
    if Input == Case :
        print("Правильно!")
```

- Измени код и начинай играть. Теперь ты можешь угадывать число так долго, сколько потребуется (рис. 3.3).



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\guess2.py =====
311
Я задумал число от 1 до 1000
Угадай число: 500
Слишком большое!
Угадай число: 250
Слишком маленькое!
Угадай число: 370
Слишком большое!
Угадай число: 320
Слишком большое!
Угадай число: 290
Слишком маленькое!
Угадай число: 300
Слишком маленькое!
Угадай число: 310
Слишком маленькое!
Угадай число: 315
Слишком большое!
Угадай число: 312
Слишком большое!
Угадай число: 311
Правильно!
>>> |
```

Рис. 3.3. Число успешно угадано!

Конструкция, в которой повторяется часть кода программы, называется *циклом*. Он выполняется повторно до тех пор, пока выполняется условие. Кстати, `while` – это тоже ключевое слово.

Компьютер считает попытки

Попрактиковавшись и применив смекалку, понадобится не более десяти попыток, чтобы угадать число. (Маленький намек: старайся всегда искать золотую середину!)

Если ты уже несколько раз играл в аналогичную игру, то, вероятно, не будешь возражать, если компьютер подсчитает количество твоих попыток угадать число. Для этого нам понадобится новая переменная, назовем ее `Attempt` (т. е. «Попытка»). Мы присвоим ей значение `0`, так как ни одной попытки еще не было совершено.

```
Attempt = 0
```

3

В цикле значение переменной `Attempt` должно увеличиваться на 1 с каждой новой попыткой угадывания. За это отвечает следующая инструкция:

```
Attempt = Attempt + 1
```

Твой компьютер берет текущее количество совершенных попыток и добавляет к нему 1. И вновь предоставляет возможность угадать число. В результате новое значение становится на 1 больше предыдущего.

Обрати внимание, это уравнение отличается от тех, которые ты привык видеть на уроках математики.

Компьютер всегда выполняет сначала то, что находится *справа* от оператора присваивания (`=`). В инструкции он вычисляет выражение с *правой* стороны.

`Attempt + 1`

Затем он переходит влево и присваивает результат в виде нового значения переменной. Возможно, стрелка поможет понять тебе этот процесс:

`Attempt ← Attempt + 1`

Предположим, что исходное значение равно 0. Процесс выглядит так:

Процесс	Схема	Значение переменной <code>Attempt</code>
Присвоение	<code>Attempt ← 0</code>	0
Вычисление результата	<code>Attempt + 1</code>	0 + 1
Присвоение	<code>Attempt ← Результат</code>	1

Таким образом, старое значение количества попыток угадывания заменяется новым значением. При следующей попытке оно становится 1, затем 2 и увеличивается каждый раз на единицу.

Кроме того, нам понадобится вот такая надпись, уведомляющая, с какой попытки ты угадал число, загаданное компьютером:

```
print("Ты попробовал " + Attempt + " раз.")
```



- Ну вот, снова пришло время набирать код программы. Полный код этой программы показан ниже (\Rightarrow *guess3*):

```
import random
Secret = "Я задумал число от 1 до 1000"
Case = random.randint(1,1000)
print(Secret)
Attempt = 0
Input = 0

# Процесс угадывания
while Input != Case :
    print("Угадай число: ", end="")
    Input = int(input())
    Attempt = Attempt + 1
    if Input < Case :
        print("Слишком маленькое!")
    if Input > Case :
        print("Слишком большое!")
    if Input == Case :
        print("Правильно!")
print("Ты попробовал " + Attempt + " раз.")
```

Обрати внимание, что последняя строка не имеет отступа, поэтому Python понимает, что она не относится к конструкции `while`.



К сожалению, у нас с тобой снова появляется небольшая проблема в самом конце игры. Интерпретатор Python не хочет принимать последнюю строку в нашей программе (рис. 3.4).

The screenshot shows a Python 3.7.1 Shell window with the following content:

```
File Edit Shell Debug Options Window Help
Слишком большое!
Угадай число: 700
Слишком большое!
Угадай число: 650
Слишком маленькое!
Угадай число: 675
Правильно!
Traceback (most recent call last):
  File "C:\Python\Projects\guess3.py", line 19, in <module>
    print("Ты попробовал " + Attempt + " раз.")
TypeError: can only concatenate str (not "int") to str
>>> |
```

Ln: 20 Col: 4

Рис. 3.4. В программу закралась какая-то ошибка

Он требует, чтобы значение переменной с количеством попыток было представлено исключительно в строковом фор-

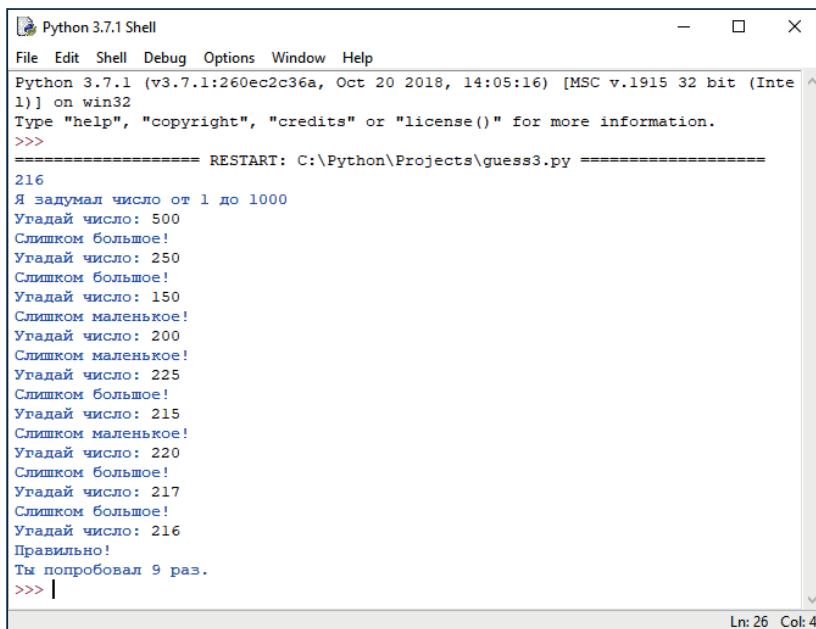
мате, иначе оно не поддерживается в сочетании с остальными словами в строке. С помощью оператора `+` ты можешь складывать числа или соединять (конкатенировать) строки, другими возможностями эта команда не обладает.

Таким образом, здесь вновь происходит преобразование типа данных, но на этот раз не в числовой формат, а в строковый. И это происходит следующим образом:

```
print("Ты попробовал " + str(Attempt) + " раз.")
```

Функция `str()` отвечает за преобразование в строковый формат.

- Измени исходный код программы и затем запусти игру еще раз. И в конце концов компьютер все-таки признается, сколько раз тебе пришлось угадывать это число (рис. 3.5).



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\guess3.py =====
216
Я задумал число от 1 до 1000
Угадай число: 500
Слишком большое!
Угадай число: 250
Слишком большое!
Угадай число: 150
Слишком маленькое!
Угадай число: 200
Слишком маленькое!
Угадай число: 225
Слишком большое!
Угадай число: 215
Слишком маленькое!
Угадай число: 220
Слишком большое!
Угадай число: 217
Слишком большое!
Угадай число: 216
Правильно!
Ты попробовал 9 раз.
>>> |
```

Рис. 3.5. Программа успешно считает количество попыток

Шлифуем игру

Поиграв, в какой-то момент тебе одозначно придет на ум такая мысль: «Игра хорошая. Но мне больше не хочется играть. Я хочу выйти из игры!»

Так, ты можешь выйти из игры, нажав кнопку в виде крестика (значок × в верхнем правом углу окна), используемую для закрытия окон в операционной системе Windows. Но нет ли более элегантного выхода?

Можно реализовать, например, введя ноль. Это можно сделать с помощью всего лишь одной строки:

```
if Input == 0 :  
    exit()
```

Функция `exit()` завершает программу. Несмотря на то что программа полностью прервется, она не совсем корректно завершит работу. Ты можешь это понять по окну сообщения, которое всегда появляется, когда программа прерывается (рис. 3.6).

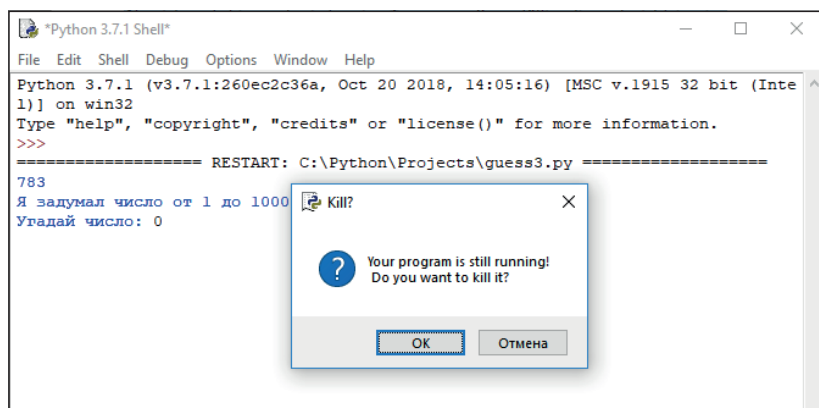


Рис. 3.6. Сообщение о принудительном завершении программы

➤ Нажми кнопку **ОК**, если ты решил завершить работу своей программы.

Это, конечно, не изящное решение, но как же иначе нам выйти из программы до момента, когда число будет угадано? Попробуем:

```
if Input == 0 :  
    break
```

С помощью команды `break` ты можешь выйти из своей программы. Если вы введешь ноль сразу или после нескольких неудачных попыток угадать загаданное число, то игра завершится.

3

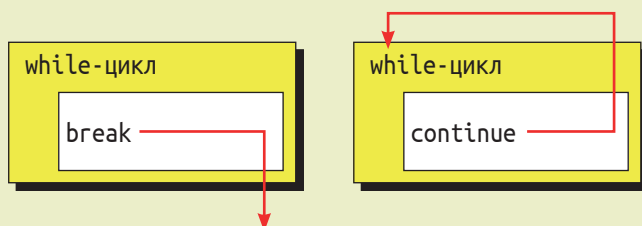
И будет гораздо лучше, если по завершении программы ты еще увидишь число, которое загадал компьютер (\Rightarrow *guess4*):

```
if Input == 0 :
    print("Правильное число: " + str(Case))
    break
```

В любом случае, в конечном итоге компьютер покажет тебе, сколько раз ты пытался угадать.

- Обнови код программы, а затем играй столько, сколько захочешь! (А может, даже захочешь увеличить диапазон загадываемого числа с 1000 до 500 000?)

Помимо ключевого слова `break`, в языке программирования Python есть команда `continue`. В чем же разница?



В обоих случаях цикл сначала прекращается. Если обнаружена команда `break`, компьютер переходит к выполнению инструкций *после* цикла. Если обнаружена команда `continue`, он переходит к *началу* цикла (и, следовательно, происходит следующая *итерация* (повтор) цикла).

Подведение итогов

Ну что ж, ты создал первую игру с моими небольшими рекомендациями, и для начала это не так уж плохо!

В этой главе ты встретил новые слова из языка программирования Python (и, конечно, использовал и ранее выученные!):

and	Связывающий оператор. Должны выполняться все условия для исполнения последующего блока инструкций
or	Связывающий оператор. Должно выполняться только одно условие для исполнения последующего блока инструкций

<, <=, >, >=	Операторы сравнения: больше, больше или равно, меньше, меньше или равно
while	Цикл. Если условие выполнено, выполняется блок инструкций внутри цикла
break	Выход из цикла и переход к последующим инструкциям
continue	Выход из цикла и переход к началу цикла для следующей его итерации
import	Импорт внешнего модуля (словарь)
random	Модуль random для генерации случайных чисел
randint()	Функция для генерации случайных чисел
str()	Преобразование типа данных в строку (String)
end=""	Параметр, который предотвращает переход на новую строку

Несколько вопросов...

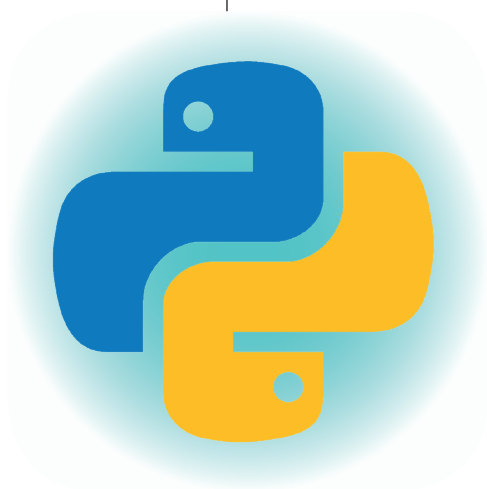
1. Объясни разницу между операторами < и <= или > и >=.
2. Что именно происходит в этих инструкциях:

```
Value = Value + 1
Value = Value - 1
Value = Value * 2
```

3. Как сгенерировать случайное число в диапазоне от 1 до 6 (например, для игры в кости)?

...и задач

1. Измени программу приветствия так, чтобы введенные значения указывались непосредственно после строки (а не ниже).
2. В школах, кроме ЕГЭ, есть еще один тип оценок. Это баллы от 0 до 10. Здесь «0» очков считается «недостаточно», остальные баллы рассчитываются в два шага (1 и 2 = 1, 3 и 4 = 2 и т. д.). Измени программу оценок так, чтобы ее можно было также использовать для 10-балльной системы.
3. В программе с указанием возраста ты можешь ввести свой возраст (или, например, друга). Аналогичным образом, как программа с оценками, компьютер должен давать определенному возрасту некий подходящий ответ.



4

Азартная игра

Ты когда-нибудь играл в лотерею? Несмотря на то что многие математики настоятельно советуют этого не делать, большинство людей считает, что в конечном итоге они все-таки станут миллионерами. Шансы существенно повышаются, если у тебя есть небольшой капитал и ты можешь куда-нибудь инвестировать деньги и «немного» подождать (пару лет или десятилетий) до получения своего первого миллиона. В данной главе компьютер может помочь тебе разобраться в этом вопросе.

Итак, в этой главе ты узнаешь:

- ⊙ как сделать перерыв;
- ⊙ что такое цикл `for`;
- ⊙ о списках и массивах;
- ⊙ о вложенных циклах.

Игра наудачу

Что ты думаешь насчет азартного варианта игры в числа? Если компьютер может генерировать случайные числа от 1 до 1000, то, конечно, сможет и сгенерировать выпадения игральной кости (т. е. от 1 до 6). В таком случае нам ничего не надо делать в программе, только ждать.

Бросок обеих игровых костей выполняется компьютером, и бросает он их совершенно случайно.

- Создай новый файл и введи показанный ниже код (⇒ *dice1*):

```
import random, time
print("Давай бросим кубики!")
Attempt = 0
YourNumber = 0
MyNumber = 0

# Игральные кубики
while Attempt < 5 :
    Attempt = Attempt + 1
    print(str(Attempt) + ". Раунд")
    print("Твой бросок: ", end="")
    Shoot1 = random.randint(1,6) # Твой бросок
    time.sleep(0.5) # Ожидание в полсекунды
    print(Shoot1)
    print("Мой бросок: ", end="")
    Shoot2 = random.randint(1,6) # Мой бросок
    time.sleep(0.5) # Ожидание в полсекунды
    print(Shoot2)
    if Shoot1 > Shoot2 :
        YourNumber = YourNumber + 1
    if Shoot1 < Shoot2 :
        MyNumber = MyNumber + 1
    print(str(YourNumber) + " и " + str(MyNumber))
    time.sleep(1) # Ожидание в секунду
    print()

# Вычисления
if YourNumber > MyNumber :
    print("Ты выиграл")
elif YourNumber < MyNumber :
    print("Я выиграл")
else :
    print("Ничья")
```

Программа имеет определенное сходство с версией из предыдущей главы: речь идет о случайных числах, которые здесь также будут сравниваться и подсчитываться.

Однако ты не имеешь в этой игре никакого влияния. Это просто игра, где ты можешь лишь наблюдать, выиграешь ты или проиграешь.

Если ты просмотришь весь код программы, то наверняка воскликнешь: ничего нового! Даже комментарии (строки, которые снабжены символом #), тебе уже знакомы.

4

Единственно, ты встретил новые ключевые слова, `time` и `sleep`. Кроме того, я интегрировал в программу два модуля с помощью команды `import`. Если первый модуль, `random`, тебе знаком по прошлой программе, то второй – это модуль `time`, он предоставляет функцию `sleep()`, которая может приостановить выполнение программы на указанное время. Параметром в скобках является время в секундах, допустимы как целочисленные, так и значения с плавающей запятой.

В нашем случае игра делает небольшие перерывы после «подбрасывания игровых костей» общей длительностью в полторы секунды. (Если это для тебя слишком мало или много, измени значения в коде.)

- Сохрани программу в файл с именем *dice1.py*, а затем запусти ее несколько раз для проверки (рис. 4.1).



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\dicel.py =====
Давай бросим кубики!
1. Раунд
Твой бросок: 4
Мой бросок: 4
0 и 0

2. Раунд
Твой бросок: 3
Мой бросок: 6
0 и 1

3. Раунд
Твой бросок: 2
Мой бросок: 3
0 и 2

4. Раунд
Твой бросок: 3
Мой бросок: 4
0 и 3

5. Раунд
Твой бросок: 4
Мой бросок: 2
1 и 3

Я выиграл
>>> |
```

Рис. 4.1. Запуск программы из файла *dice1.py*

Ты можешь заметить, что для функции `print()` ничего не указано в качестве параметра в круглых скобках. Это означает, что она ничего не будет выводить, но текстовый курсор тем не менее переходит на следующую строку: так мы создаем расстояние между строками. Это особенно полезно, когда текст должен быть визуально отделен – просто потому, что так программа выглядит лучше.



Конструкция *for*

В предыдущей версии игры мы не указывали явно количество попыток угадывания числа, а теперь зададим количество подбрасываний кубиков. В языке Python есть другая условная конструкция, отличная от `while` (\Rightarrow *dice2.py*):

```
# Игральные кубики
import random, time
print("Давай бросим кубики!")
YourNumber = 0
MyNumber = 0

for Nr in range(5) :
    print(str(Nr+1) + ". Раунд")
    print("Твой бросок: ", end="")
    Shoot1 = random.randint(1,6) # Твой бросок
    time.sleep(0.5) # Ожидание в полсекунды
    print(Shoot1)
    print("Мой бросок: ", end="")
    Shoot2 = random.randint(1,6) # Мой бросок
    time.sleep(0.5) # Ожидание в полсекунды
    print(Shoot2)
    if Shoot1 > Shoot2 :
        YourNumber = YourNumber + 1
    if Shoot1 < Shoot2 :
        MyNumber = MyNumber + 1
    print(str(YourNumber) + " и " + str(MyNumber))
    time.sleep(1) # Ожидание в секунду
    print()

if YourNumber > MyNumber :
    print("Ты выиграл")
elif YourNumber < MyNumber :
    print("Я выиграл")
else :
    print("Ничья")
```

4



- Измени код предыдущей версии программы в соответствии с листингом и запусти ее.

Прежде всего в глаза бросается, вероятно, такая строка:

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

Вот что она означает:

Для переменной `Nr` компьютер должен **ПОВТОРИТЬ** блок инструкций указанное количество раз.

```
for Nr in range(Значение) :
```

БлокИнструкций

Начнем разбираться: функция `range()` определяет список, который в этом случае содержит целые числа от 0 до 4. Для каждого элемента этого списка цикл повторяется.

Переменная `Attempt` не нужна, потому что подсчетом теперь занимается переменная `Nr`, но мы должны учитывать, что подсчет компьютер начинает с 0:

```
print(str(Nr+1) + ". Раунд")
```

Остальные же инструкции точно такие же, что и в цикле `while`. Получилась так называемая конструкция `for`. И поскольку она занимается подсчетом, ты можешь назвать ее считающим циклом.

А как быть в том случае, если тебе нужен список, который начинается не с 0, а, скажем, с 1? Это можно решить следующим образом:

```
for Nr in range(1,6) :
```

Упростить можно так:

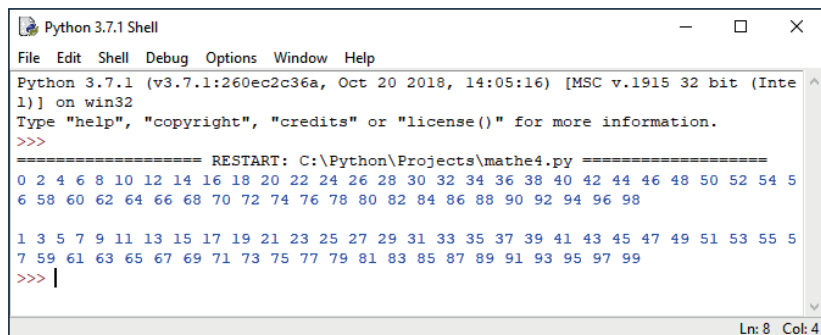
```
print(str(Nr) + ". Раунд")
```

Мы можем задать функции `range()` еще и третий параметр, который определяет ее математический шаг. Мы не можем

вывести разные числа в определенном диапазоне, но мож-
но показать, допустим, каждое второе (\Rightarrow *mathe4.py*):

```
# Четные числа
for Nr in range (0,100,2) :
    print(Nr, end=" ")
print()
print()
# Нечетные числа
for Nr in range (1,100,2) :
    print(Nr, end=" ")
```

- Если хочешь, создай новую программу и введи исход-
ный код. Затем запусти свою программу (рис. 4.2).



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\mathe4.py =====
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98

1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
>>> |
```

Рис. 4.2. Запуск программы *mathe4.py*

Как видишь, здесь можно промахнуться с функцией `print()`: я сделал так, чтобы с помощью параметра `end=" "` между каждым выводимым числом отображался пробел. Кроме того, в коде используются две инструкции с функцией `print()` без параметров, необходимые для создания промежутка между строками четных и нечетных чисел.



Вместо функции `range()` ты также можешь применить список, который затем использовать в цикле `for`. Вот небольшой пример:

```
Week = ["Понедельник", "Вторник", "Среда", "Четверг", "Пятница",
        "Суббота", "Воскресенье"]
for Day in Week :
    print(Day)
```

Программа выводит списоком названия дней недели.

4



Квадратные скобки ты можешь ввести, если переключишься на клавиатуре на английскую раскладку и нажмешь клавиши **X** и **Ъ**.

Важно, что дни недели перечисляются в одной строке программы (это не всегда получается показать в книге).

На пути к миллиону

Со знаниями, которые ты уже получил, ты легко можешь создать программу, способную чуть подправить твоё финансовое положение.

➤ Введи следующий код программы (\Rightarrow *million1.py*):

```
import random
Capital = random.randint(2,10)*10000
print("Ты выиграл в лотерею " + str(Capital) + " рублей!")
print("Ты можешь не забирать выигрыш сразу, ", end="")
print("а вложить деньги и заработать на этом!")
print("Процентная ставка: ", end="")
Percent = float(input())
Term = 0
while Capital < 1000000 :
    Fee = Capital * Percent / 100
    Capital = Capital + Fee
    Term += 1
print("Чтобы стать миллионером, тебе понадобится ", end="")
print(str(Term) + " лет.")
```

(Обрати внимание, что инструкции `print`, если даже в книге распределены на две строки, в программе должны быть однострочными!)



Если ты столкнулся с очень длинными строками, то в Python ты сможешь разделить одну строку на две части:

```
print("Ты выиграл в лотерею " + str(Capital) + \
" рублей!")
```

Так называемая «косая черта», или «слеш» (`\`), в конце строки сообщает Python, что следующая строка относится к той же инструкции.

Так что же происходит в этой программе? Тот факт, что она имеет какое-то отношение к математике, выражается не только в именах переменных, например: `Fee`, `Capital` и `Percent`.

Поскольку капитал (`Capital`) должен быть (случайной) прибылью, мы попытаемся использовать генератор чисел. Чтобы эта прибыль не оказалась слишком маленькой, мы начнем с 20 тыс. рублей:

```
Capital = random.randint(2,10)*10000
```

Теперь устанавливаем прибыль в диапазоне от 20 тыс. до 100 тыс. рублей, ведь не будем же мы ставить на выигрыш сразу миллион?! (И делать выигравшего сразу миллионером.)

Какой процент выбранный тобой банк (или другая финансовая организация) предлагает, ты вводишь сам:

```
print("Процентная ставка: ", end="")  
Percent = float(input())
```

Разумеется, здесь также можно использовать числа с плавающей запятой. Затем срок устанавливается равным нулю:

```
Term = 0
```

А теперь речь пойдет о вычислении процентов. Если ты еще не проходил (или уже забыл) эту тему на уроках математики, все не так уж плохо. Просто вводи данные и доверься мне – я знаю, как это делается.

Что происходит в цикле `while`? Во-первых, вычисляется доход за один год:

```
Fee = Capital * Percent / 100
```

Затем он будет засчитан в капитал:

```
Capital = Capital + Fee
```

И поскольку каждый год доход увеличивается, срок вклада будет увеличиваться на 1:

```
Term += 1
```

4



Но это выглядит немного странно. Фактически вот что делает Python в этой строке:

```
Term = Term + 1
```

Можно заменить следующим кодом:

```
Term += 1
```

Для строки

```
Term = Startzeit + 1-
```

прием, разумеется, не сработает.

И так будет продолжаться, пока не будет достигнут первый миллион:

```
Capital < 1000000
```

- Запусти программу несколько раз, чтобы узнать, каковы твои шансы стать миллионером в ближайшие несколько десятилетий, при условии, конечно, что у тебя будет достаточно денег.



Если ты введешь ноль в качестве процентной ставки (Percent), возникнет проблема: твоя программа зависнет, поскольку миллион никогда не будет достигнут и выполнение будет происходить бесконечно. Тебе придется закрыть окно программы.

Если хочешь, то также можешь создать дополнительный цикл, который будет повторять запрос ввода процентной ставки, пока не будет введено число больше нуля.

Может быть, лучше не полагаться на удачу в финансовом вопросе, а начать с капитала, который ты действительно можешь иметь?

Для этого мы немного изменим начало программы, чтобы у тебя была возможность указать свой начальный капитал:

```
print("Какую сумму ты хочешь инвестировать: ", end="")
Capital = float(input())
```

Затем код программы продолжается, как и раньше:

```
print("Процентная ставка: ", end="")
Percent = float(input())
```

```
Term = 0  
...
```

Ниже представлен полный код новой версии игры (\Rightarrow *million2.py*):

```
print("Какую сумму ты хочешь инвестировать: ", end="")  
Capital = float(input())  
print("Процентная ставка: ", end="")  
Percent = float(input())  
Term = 0  
while Capital < 1000000 :  
    Fee = Capital * Percent / 100  
    Capital = Capital + Fee  
    Term += 1  
if Term > 0 :  
    print("Чтобы тебе превратиться в миллионера, твои деньги  
в течение ", end="")  
    print(str(Term) + " лет должны быть в банке.")  
else :  
    print("Добро пожаловать в Клуб миллионеров!")
```

Добавим еще кое-что в код программы. Если ты начал игру с миллиона (или более), срок вклада (Term) станет бессмысленным (то есть равным 0), но ты все еще можешь кое-что сделать в игре.

➤ Измени свою программу и попробуй запустить новую версию программы несколько раз.

Ты увидишь, что в некоторых случаях должно пройти много лет, да и ты сам, скорее всего, выйдешь на пенсию, когда получишь возможность снять свой первый миллион.

Может быть, тебе будет интересно узнать, сколько сможет накапливаться денег за определенный срок? В этом случае ты должен сам определить срок вклада.

Таким образом, у нас получается три строки кода для ввода:

```
print("Какую сумму ты хочешь инвестировать: ")  
Capital = input()  
print("Процентная ставка: ")  
Percent = input()  
print("На какой срок ты вкладываешь деньги: ")  
Term = input()
```

Например, цикл для расчета процентов и общего капитала может изменяться по мере его выполнения:

4



```
while Term > 0 :  
    Fee = Capital * Percent / 100  
    Capital = Capital + Fee  
    Term -= 1
```

Опять же, я для краткости использовал символы `-=`, но на этот раз для обратного отсчета. Это важный момент, который очень легко упустить. Если вместо минуса будет плюс, тогда условие `> 0` всегда будет истинно, и программа завязнет (вновь) в бесконечном цикле.

Я рекомендую не использовать цикл из прошлой версии программы. Поскольку срок нам уже известен и понятно, как часто этот цикл нужно будет повторить, ты можешь просто позволить компьютеру провести вычисления самостоятельно. Выглядеть это будет так:

```
for Value in range(Term) :  
    Fee = Capital * Percent / 100  
    Capital = Capital + Fee
```

Совсем неплохо! И даже немного лаконичнее, чем писать целую конструкцию `while`. Ниже представлен полный исходный код (\Rightarrow *million3.py*):

```
print("Какую сумму ты хочешь инвестировать: ")  
Capital = float(input())  
print("Процентная ставка: ")  
Percent = float(input())  
print("На какой срок ты вкладываешь деньги: ")  
Term = int(input())  
for Value in range(Term) :  
    Fee = Capital * Percent / 100  
    Capital = Capital + Fee  
print("Так ты получишь " + str(Capital) + " рублей")  
if Capital < 1000000 :  
    print("Но так ты не станешь миллионером!")  
else :  
    print("Добро пожаловать в Клуб миллионеров!")
```

- Измени код в соответствии с листингом и запусти программу. Ты видишь, что за короткие сроки накапливают неплохие проценты.

Выиграть в лотерею?

Конечно, также можно выиграть деньги в настоящую азартную игру. Если ты действительно выиграешь, не так важно, что шансы проиграть в реальности были намного выше. В качестве примера возьмем одну лотерею, где 6 из 49 номеров (шаров) выбираются случайным образом. Набросаем такой код программы-лотереи (\Rightarrow *lotto1.py*)

```
import random
for Nr in range(6) :
    Case = random.randint(1,49)
    print("№ " + str(Nr+1) + " => " + str(Case))
```

И это все? И да, и нет. Потому что в случае проигрыша не только числа, выбранные компьютером, будут неправильные, но может выйти и так, что числа сгенерируются дважды (рис. 4.3).

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\lottol.py =====
№ 1 => 45
№ 2 => 7
№ 3 => 10
№ 4 => 3
№ 5 => 39
№ 6 => 10
>>>
```

Рис. 4.3. Компьютер сгенерировал одни и те же числа несколько раз

Чтобы узнать, генерировался ли уже один из номеров, компьютер должен запоминать числа, которые он сгенерировал. Затем каждая новая попытка должна быть по отдельности сопоставлена с прошлыми выпадениями чисел, и таким образом получится определить, есть ли это число в списке.

Но как компьютер может запомнить 49 чисел? Довольно просто: мы составим ему список, который изначально пуст:

```
Ball = []
```

4

Я выбрал имя `Ball`, так как в классической лотерее «Спортлото» происходит розыгрыш из контейнера с 49 пронумерованными шарами.

Итак, сначала у нас есть 0 элементов. В цикле мы теперь добавляем 49 элементов к пустому списку:

```
for Nr in range(1,50) :  
    Ball.append(0)
```

Это выполняется с помощью функции `append()`, для которой указывается значение конкретного элемента в круглых скобках. Итак, у нас, наконец-то, есть 49 элементов списка, который я назвал `Ball`. И каждый элемент несет значение 0, которое является для нас с тобой синонимом значения «еще не использовано».



В функции `range()` второе число является верхним пределом, который не включен, поэтому используются элементы от 1 до 49.

Во втором цикле, который ты уже знаешь из первой версии нашей лотереи, создается случайное значение:

```
Case = random.randint(1,49)
```

Но сначала ты должен проверить, не был ли этот шар уже выбран компьютером. Вот почему мы помещаем случайное число в конструкцию `while`:

```
while Ball[Case] == 1 :  
    Case = random.randint(1,49)
```

Если номер уже был выбран компьютером и показан нам, то будет создано новое случайное значение. (Может потребоваться немного времени на поиск компьютером «свободного» номера.)

Если это шар с номером `[Case] == 0`, то больше он не будет выпадать. Будет присвоено значение 1, говорящее, что шар «уже используется в игре».

```
Ball[Case] = 1
```

Это предотвратит вытаскивание второго шара с тем же числом. Ниже представлен полный код данной программы (\Rightarrow *lotto2.py*):

```
import random
Ball = []
# Все шары еще не "вытащены"
for Nr in range(1,50) :
    Ball.append(0)
Case = random.randint(1,49)
# "Вытягивается" шесть шаров
for Nr in range(6) :
    # поиск неиспользованных шаров
    while Ball[Case] == 1 :
        Case = random.randint(1,49)
    # Пометка использованного шара "вытасканным"
    Ball[Case] = 1
    print("№ " + str(Nr+1) + " => " + str(Case))
```

То, что переменная Nr используется дважды для разных целей, не имеет значения. Но ты можешь выбрать два разных имени для этой переменной.



Я хотел бы показать тебе еще один вариант, в котором используется новый тип переменной (\Rightarrow *lotto3.py*):

```
import random
Ball = []
# Все шары еще не "вытащены"
for Nr in range(1,50) :
    Ball.append(False)
Case = random.randint(1,49)
# "Вытягивается" шесть шаров
for Nr in range(6) :
    # поиск неиспользованных шаров
    while Ball[Case] :
        Case = random.randint(1,49)
    # Пометка использованного шара "вытасканным"
    Ball[Case] = True
    print("№ " + str(Nr+1) + " => " + str(Case))
```

Вместо 0 и 1 здесь у нас появляются значения False и True для списка Ball. Это так называемая *логическая переменная*. Она может принимать только одно из двух значений – истина (True) или ложь (False): одно противоположно другому, поэтому операция присвоения, такая как `ball = not ball`, непосредственно меняет значение переменной.

4

Попробуй запустить такой код:

```
Ball = True
print(Ball)
Ball = notBall
print(Ball)
```

Ты еще должен узнать про ключевое слово `not`, прежде чем логическая переменная изменит ее значение.

Если внимательно рассмотришь код программы-лотереи со значениями `True` и `False`, то наверняка заметишь еще одно изменение. Вместо

```
while Ball[Case] == True :
```

у нас используется

```
while Ball[Case] :
```

Это связано с тем, что условие в конструкции (например, `if` или `while`) может быть только в одном из двух состояний: либо оно выполняется (`= True`), либо нет (`= False`). Поэтому мы можем опустить `== True` и использовать логическую переменную непосредственно в качестве условия. Для полноты картины тебе также следует знать, что

```
while Ball[Case] == False :
```

можно заменить на

```
while not Ball[Case] :
```

Управление строками

Сейчас мы в основном занимаемся числами, но нам будет необходим также и текст. Ранее мы использовали его в виде строк. И хотя ты уже знаешь некоторые возможности использования текста в языке программирования Python, это еще не все.

Я хотел бы закончить главу чем-то, не имеющим ничего общего с деньгами. Мне удалось найти подходящие предложения: «А роза упала на лапу Азора» и «Аргентина манит негра» (оба предложения – *палиндромы*, или *перевертыши*, одинаково читающиеся в обоих направлениях).

Давай проверим эти фразы в нашей программе (\Rightarrow *palindrom.py*):

```
print("Напиши короткий текст (небольшой, без пробелов): ")
Text1 = input()
Text2 = ""
Chain = len(Text1)
for Nr in range(0, Chain) :
    Text2 += Text1[Chain-Nr-1]
print(Text2)
if Text1 == Text2 :
    print ("Палиндром")
```

Сначала компьютер попросит нас написать что-нибудь. Это необязательно должны быть только строчные буквы, да и пробелы, конечно, допускаются. Но если ты хочешь проверить свой текст, лучше все-таки написать его последовательностью обычных строчных (маленьких) букв.

С помощью функции `input()` ты вводишь свое предложение или слово. Попробуй ввести в этой программе одно из двух предложений, которые я упомянул выше. Кроме того, мы также определим вторую пустую строку. Тогда нам еще понадобится длина набранной строки:

```
Chain = len(Text1)
```

Теперь мы используем цикл `for`, который «соответствует» каждой букве первого текста:

```
for Nr in range(0, Chain) :
```

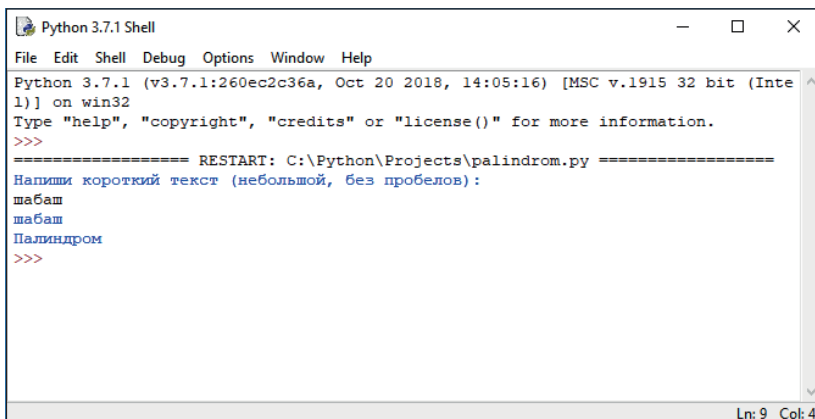
Несмотря на то что отсчет начинается с самого начала, номеру 1 соответствует последняя буква, затем идет предпоследняя и т. д. добавляется к пустой второй строке:

```
Text2 += Text1[Chain-Nr-1]
```

Опять же, используются комбинации `+` и других знаков.

Наконец, ты можешь увидеть обращенный текст. И так как фраза является палиндромом, компьютер сообщит об этом (рис. 4.4).

```
print(Text2)
if Text1 == Text2 :
    print ("Палиндром")
```



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\palindrom.py =====
Напишите короткий текст (небольшой, без пробелов):
шабаш
шабаш
Палиндром
>>>
```

Рис. 4.4. Пример выполнения программы

И еще одна особенность в работе с текстами: здесь также можно использовать символ `*` для умножения (дублирования) слов, и записать это можно было бы так:

```
Text1 = input()
Text2 = ""
for Nr in range(0,3) :
    Text2 += Text1
```

...но ты можешь укоротить все это вот так:

```
Text1 = input()
Text2 = 3 * Text1
```

Подведение итогов

Посмотрим, что у нас с тобой получилось после этой главы. Речь шла о другой управляющей конструкции, которую мы использовали в двух небольших игровых программах. Как и в предыдущей главе, мы использовали эффект случайности. Кроме того, нам везло, и иногда мы получали деньги и выигрывали.

В целом же вот что мы узнали нового из языка программирования Python:

time	Модуль времени
sleep()	Ожидание некоторого времени (указывается в секундах)
for	Для определенного количества элементов списка следует выполнить блок инструкций
in range()	Управление элементом в диапазоне или списке
append()	Добавление элемента в список
True, False	Значения логической переменной или условия
not	Обратное значение Истина/Ложь для логической переменной или условия
[]	Диапазон списка или номер элемента списка
\	Распределение длинного кода на две строки
len()	Длина строки
+, *	Складывает строки Умножает строки

Несколько вопросов...

1. Как сделать так, чтобы компьютер приостановил работу программы?
2. Что происходит в программе для миллионеров, если ты введешь 0 в качестве значения начального капитала или процентной ставки?
3. Пробел, новая строка, пустая строка: в чем разница?

...и задач

1. Напиши математическую программу, в которой два числа генерируются случайным образом. Другая случайность определяет, нужно ли добавлять, вычитать, умножать или делить эти два числа. Компьютер выполнит подсчеты, затем ты сможешь ввести результат. В конце компьютер проверяет, правильно ли введено твое число.
2. Измени игру в угадывания числа так, чтобы ты задумывал число, а компьютер пытался угадывать его. Указывая «меньше» или «больше», ты сможешь вести компьютер к правильному результату.
3. И еще одна вариация игры в «угадайку»: компьютер играет сам с собой, угадывая число и сравнивая его со своим случайным значением. (Он должен играть «с системой», поэтому никаких случайных догадок!)



5

Функции

Наши предыдущие программы были небольшими, даже достаточно маленькими, чтобы мы могли увидеть весь исходный код целиком. Но с действительно «большим» проектом вроде профессионального приложения или, к примеру, более сложной игры количество исходного кода будет серьезно увеличиваться. Это означает возможность просто упустить смысл в большом количестве строк кода.

Однако если мы разделим большую программу на небольшие фрагменты, это поможет нам сохранить, так сказать, «здоровомыслие».

Итак, в этой главе ты узнаешь:

- ⊙ как создавать свои собственные функции с помощью ключевого слова `def`;
- ⊙ как применять параметры;
- ⊙ о локальных и глобальных переменных;
- ⊙ о ключевом слове `return`;
- ⊙ как менять местами и сортировать числа.

Python учится

У языка программирования Python есть базовый словарь, называемый «ключевыми словами». К ним относятся такие слова, как `if`, `for`, `while`, но не `print()`, `input()` и `int()`. Они ин-

тегрируются из базовых модулей, которые импортируются отдельно.

Хотя это и происходит автоматически, некоторые модули, такие как `random` и `time`, должны быть импортированы в программу с помощью инструкции `import` (кстати, также ключевое слово). Если же импортированных модулей тебе будет недостаточно, ты всегда можешь создать свои собственные функции.

Это означает, что ты сможешь научить язык программирования Python новым словам и, таким образом, постоянно расширять его словарный запас. Это также означает, что Python способен запоминать эти новые изученные слова, и ты сможешь, немного позже, использовать слова из своего расширенного словаря в будущих программах.

Давай проверим это прямо сейчас. Я просто возьму функцию, имя которой я сам придумал:

```
tunix()
```

Конечно, Python ничего не сможет с ней в этот момент сделать (рис. 5.1).

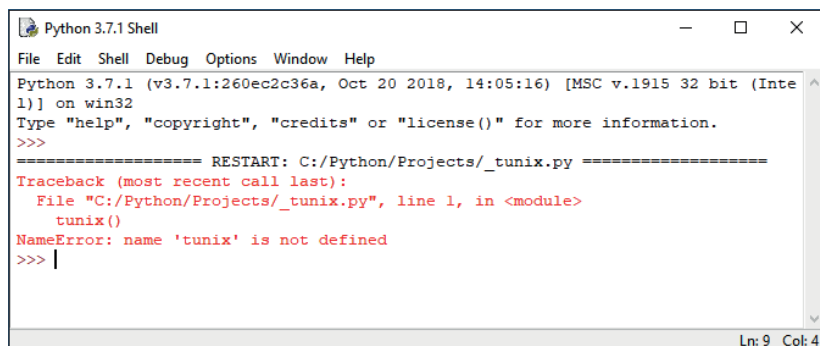


Рис. 5.1. Ошибка: функция не определена

Вот почему мы должны определить код `tunix()` как функцию, которая ничего особенного здесь не делает, например так:

```
def tunix() :
    nix = 0
    tunix()
```

Ниже идет вызов функции, и, как видишь, ничего в реальности не происходит, что нам с тобой было бы видно в результате.

Теперь давай я приведу пример, где будет виден результат работы программы:

```
# Определение функции
def sayHello() :
    print("Привет!")
# Вызов функции
sayHello()
```

Если ты наберешь эту программу и запустишь ее, то получишь короткое приветствие «Привет!».



Прежде чем мы продолжим, ты должен решить, как будешь придумывать имена функциям. Я решил использовать английские слова. С одной стороны, они часто более лаконичны, чем другие, и все уже существующие функции также имеют английские имена. Многие профессиональные программисты предпочитают их, часто и для переменных. Но ты всегда можешь использовать свои (например, русские (транслитом)) слова для имен функций, если тебе так больше нравится.

Теперь вернемся к нашей новой функции. Она состоит из имени и тела.

```
def ИмяФункции () :  
    БлокИнструкций
```

Немного напоминает условные конструкции, которые ты видел до этого. Начало функции состоит из одной строки. Сначала указывается вводное ключевое слово `def`, затем имя, всегда сопровождаемое круглыми скобками.

```
def sayHello() :
```

Тело же функции состоит из блока инструкций, в данном случае только из одной инструкции:

```
print("Привет!")
```

Разумеется, эта инструкция используется после строки `def`.



Как видишь, в коде определение новой функции указывается выше ее вызова. Это необходимо, чтобы интерпретатор Python узнал о функции прежде, чем попытался ее вызвать.

И теперь мы создадим новый проект, в котором функции будут определены и, конечно же, немедленно использованы. Как насчет проекта-загадки? Давай посмотрим, что происходит со встроенными функциями.

Начнем же мы с функции для начальных значений:

```
def initGame() :  
    Secret = "Я задумал число от 1 до 1000"  
    print(Secret)  
    Attempt = 0  
    Input = 0
```

В следующей функции дело касается игры:

```
def playGame() :  
    Case = random.randint(1,1000)  
    while Input != Case :  
        print("Угадай число: ", end="")  
        Input = int(input())  
        Attempt + 1  
        if Input < Case :  
            print("Слишком маленькое!")  
        if Input > Case :  
            print("Слишком большое!")  
        if Input == Case :  
            print("Правильно!")
```

В основе функции `playGame()` лежит цикл, который гарантирует, что игра продолжается до тех пор, пока ты не угадаешь число. Как видишь, я использовал здесь упрощенную версию, где ты не сможешь просто взять и завершить игру в любой момент. И поэтому для последней функции остается только одна строка:

```
def endGame() :  
    print("Ты попробовал " + str(Attempt) + " раз.")
```

Наконец, все три функции должны сработать, это и называется *основной программой*, которая здесь довольно небольшая:

```
# Основная программа
initGame()
playGame()
endGame()
```

- Итак, теперь твоя очередь. Возможно, тебе будет удобнее и проще создать новый проект. Необходимо убедиться, что функции определены. Как считаешь, у тебя получится? Если нет, позже я приведу код целиком.

Локальные или глобальные переменные?

Если твоя программа внезапно завершается с сообщением об ошибке, это происходит не из-за тебя, а из-за того, что с переменными что-то не так.

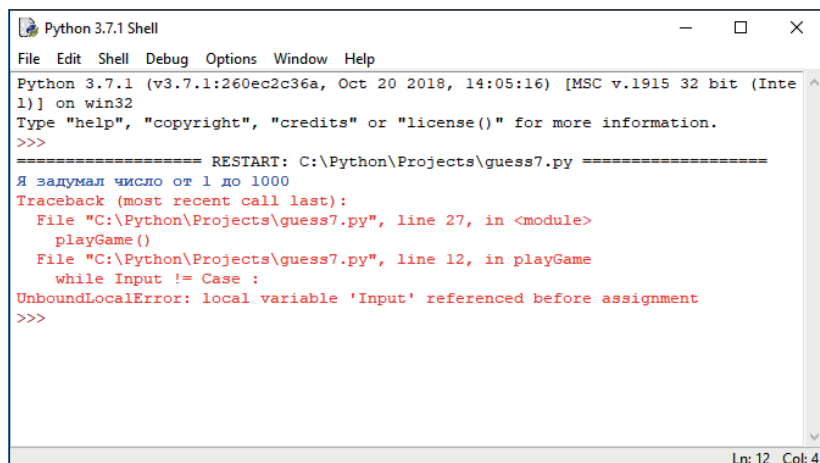


Рис. 5.2. Ошибка из-за недоступности переменных

По-видимому, функция `playGame()` не может ничего сделать с переменной `Input`. Хотя она использовалась ранее в функции `initGame()`, она должен быть знакома твоему компьютеру. Что же происходит не так?

Дело в том, что в первую очередь все переменные будут действительны только там, где они были объявлены. В последних версиях игры переменные были действительны для всей основной программы. А основной программой был исходный код целиком.

Теперь же код выглядит так: сначала указывается несколько определений, а далее приводится код основной программы. И как я только что объяснил, переменные действуют только в той функции, в которой используются. Это также означает, что могут существовать две разные переменные с одним и тем же именем и одна из них работает в функции `initGame()`, а другая – только в `playGame()`.

Все переменные, которые были определены в пределах какой-либо функции, действуют только в ней – с точки зрения общей программы, которая может состоять из множества функций, это *локальные переменные*.

После выхода из функции локальные переменные станут недоступны.

Но поскольку нам нужны переменные, которые будут доступны повсюду, поэтому во всех наших функциях у нас есть возможность сделать их глобальными.



И как мы достигнем подобной «глобализации»? Это проще объяснить, если ты посмотришь на исходный код целиком (\Rightarrow *guess7*):

```
import random

# Функции
def initGame() :
    global Attempt, Input
    Secret = "Я задумал число от 1 до 1000"
    print(Secret)
    Attempt = 0
    Input = 0
def playGame() :
    global Attempt, Input
    Case = random.randint(1,1000)
    # print(Case)
    while Input != Case :
        print("Угадай число: ", end="")
        Input = int(input())
        Attempt += 1
        if Input < Case :
            print("Слишком маленькое!")
        if Input > Case :
            print("Слишком большое!")
        if Input == Case :
            print("Правильно!")
def endGame() :
```

```
global Attempt
print("Ты попробовал " + str(Attempt) + " раз.")

# Основная программа
initGame()
playGame()
endGame()
```

Вероятно, ты заметил эти дополнительные строки. Так:

```
global Attempt, Input
```

мы также сообщаем соответствующей функции о том, что одна или несколько переменных должны быть доступны глобально, то есть никакая новая переменная с этим именем не должна создаваться внутри функции. Таким образом, в обеих функциях могут использоваться те же переменные `Attempt` и `Input`.



Переменная должна объявляться на глобальном уровне, если ее значение должно быть изменено. Функция `endGame()` не меняет значение переменной, а лишь считывает ее. Таким образом, ты также можешь опустить для нее инструкцию `global`.

- Теперь проверь, что твой исходный код выглядит так, как листинг выше, сохрани все изменения и запусти игру. Попробуй запустить ее несколько раз.

На самом деле все должно работать как в последней версии этой игры-загадки. Новое здесь то, что переменные были объявлены в глобальной области видимости.

Параметры

Если бы ты объявил все переменные, которые встречаются в программе, глобально, то у тебя всегда был бы доступ к ним. Но имеет ли это смысл? Представь, что ты работаешь над колоссальным проектом, большим приложением или серьезной игрой. В таком проекте огромное количество переменных: десятки, сотни и даже свыше тысячи.

Зачем же нам доступность этих переменных во всей программе, если большинство из них необходимо только в определенных ее частях? Если бы все переменные были глобальными, произошло бы наверняка и такое, что вне-

запно их значения изменились бы, чего на самом деле быть не должно. А вот найти эту ошибку, а тем более исправить ее, в большом количестве исходного кода быстро не получится.

Не было бы лучшим решением этой проблемы всегда использовать переменную именно там, где она будет нужна? Давай взглянем на наш текущий проект:

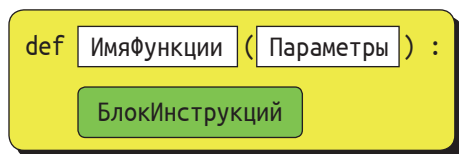
Переменная	Используется в:
Secret	initGame()
Case	playGame()
Input	initGame(), playGame()
Attempt	initGame(), playGame(), endGame()

Для всех функций требуется только одна из переменных. Еще две, Secret и Case, используются локально.

Но локальным переменным все же присущи проблемы, не так ли? Они усложняют программу, особенно когда проект становится все больше и больше.

А как теперь вызвать функции с применением переменных Attempt и Input? Для этого ведь нужны круглые скобки? Значит, они не должны оставаться пустыми.

Представь себе, что скобки – это руки, которыми функция «обнимает» параметры. Термин *параметр* мы уже рассматривали с тобой в предыдущей главе.



Когда функция получает имена переменных в качестве параметров, она может сделать с ней некое действие. Например, вот так:

```
def initGame(Attempt, Input) :  
def playGame(Attempt, Input) :  
def endGame(Attempt) :
```

В основной программе первая из трех функций может быть вызвана с числовыми значениями в качестве параметров, например так:

```
initGame(0,0)
```

Таким образом, внутри функции переменные `Attempt` и `Input` будут иметь значение 0. И тогда эти присваивания больше не нужны:

```
Attempt = 0  
Input = 0
```

Теперь ты можешь забыть про них. Да, и еще кое-что: здесь нам не нужны какие-либо параметры, поэтому функция может быть сокращена следующим образом:

```
def initGame() :  
    Secret = "Я задумал число от 1 до 1000"  
    print(Secret)
```

Но как мы теперь присвоим переменным `Attempt` и `Input` значение 0? Передав два нуля в качестве параметров при вызове следующей функции:

```
playGame(0,0)
```

Внутри себя функция теперь может продолжать использовать эти две переменные. И поэтому код нашей функции `playGame()` может выглядеть так:

```
def playGame(Attempt, Input) :  
    Case = random.randint(1,1000)  
    while Input != Case :  
        print("Угадай число: ", end="")  
        Input = int(input())  
        Attempt += 1  
        if Input < Case :  
            print("Слишком маленькое!")  
        if Input > Case :  
            print("Слишком большое!")  
        if Input == Case :  
            print("Правильно!")
```

Ввод известен перед переходом к строке конструкции `while`. И значение переменной `Attempt` будут рассчитываться соответствующим образом, исходя из начального 0.

Как видишь, в обоих определениях функций инструкции `global` не используются.



Теперь нам нужен параметр для последней функции, поскольку мы не можем просто передать ей фиксированное значение:

```
endGame(10)
```

Текущее значение переменной `Attempt` должно быть передано после того, как вы поиграли и число увеличилось. Но это работает, только если это значение каким-то образом покидает функцию `playGame()`. В языке Python есть решение этой проблемы. Инструкция `return` используется для возврата из функции, т. е. для прекращения ее работы и выхода из нее.

```
return Attempt
```

Инструкция `return` должна находиться в *последней строке* определения функции `playGame()`. Эта функция может быть вызвана двумя способами:

```
playGame(0,0)  
Game = playGame(0,0)
```

Во втором варианте сама функция назначается (новой) переменной (которую я назвал `Game`). Так мы получаем значение, которое возвращает функция с помощью инструкции `return`.

Поэтому функции могут использоваться двумя различными способами: как простая инструкция или как присвоение:

Функция ()

Функция (Параметр)

Переменная = Функция ()

Переменная = Функция (Параметр)

Переменная, которая приняла значение функции, теперь может служить параметром для следующей функции:

```
endGame(Game)
```

Вот полный листинг исходного кода программы (\Rightarrow *guess8.py*):

```
import random

# Функция
def initGame() :
    Secret = "Я задумал число от 1 до 1000"
    print(Secret)
def playGame(Attempt, Input) :
    Case = random.randint(1,1000)
    # print(Case)
    while Input != Case :
        print("Угадай число: ", end="")
        Input = int(input())
        Attempt += 1
        if Input < Case :
            print("Слишком маленькое!")
        if Input > Case :
            print("Слишком большое!")
        if Input == Case :
            print("Правильно!")
    return Attempt
def endGame(Attempt) :
    print("Ты попробовал " + str(Attempt) + " раз.")

# Основная программа
initGame()
Game = playGame(0,0)
endGame(Game)
```

- Введи исходный код и запусти свою программу. Получилось ли угадать число, и сколько в итоге вышло попыток?

Если игра работает, то ты сделал все правильно!

Обмен значений

Продолжая работу с функциями, давай рассмотрим новый пример их использования. Он относится к числам, но не очень тесно связан с математикой. Попробуем поменять местами два значения – например, когда необходима сортировка данных.

Как же произойдет обмен? Представь, что у тебя обе руки заняты, например, бутылками газировки. Теперь поменяй бутылки в руках!

Скорее всего, у тебя ничего не получится. Хотя можно попробовать подбросить бутылки в воздух и заменить их таким образом. Или разбить. Для программирования это в любом случае было бы слишком рискованно.

Поэтому для обмена значений двух переменных необходима еще одна вспомогательная переменная, чтобы ничего не пошло наперекосяк.

Назовем переменные, значения которых необходимо обменивать, `Number1` и `Number2` и присвоим вспомогательной переменной имя `Swap`. Процесс обмена будет выглядеть так:

```
Swap = x1
x1 = x2
x2 = Swap
```

Поместим код в функцию и сразу же ее вызовем.

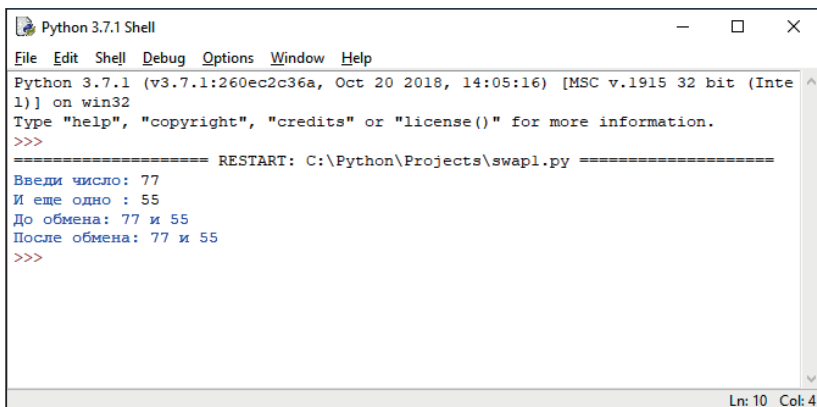
Поскольку я буду использовать переменные с именами `Number1` и `Number2` в основной программе (а также как параметры), я сокращаю их имена: `x1` и `x2`.

Теперь давай посмотрим, как все это может выглядеть в полном коде программы (\Rightarrow `swap1.py`):

```
def exchange(x1, x2) :
    Swap = x1
    x1 = x2
    x2 = Swap

# Основная программа
print("Введи число: ", end="")
Number1 = int(input())
print("И еще одно : ", end="")
Number2 = int(input())
print ("До обмена: " + str(Number1) + " и " + str(Number2))
exchange(Number1, Number2)
print("После обмена: " + str(Number1) + " и " + str(Number2))
```

В основной программе вводятся первые два числа, и они отображаются в правильном порядке. Затем они меняются местами, это делается в строке `exchange(Number1, Number2)`, и, наконец, числа в итоге будут отображаться в обратном порядке (рис. 5.3).



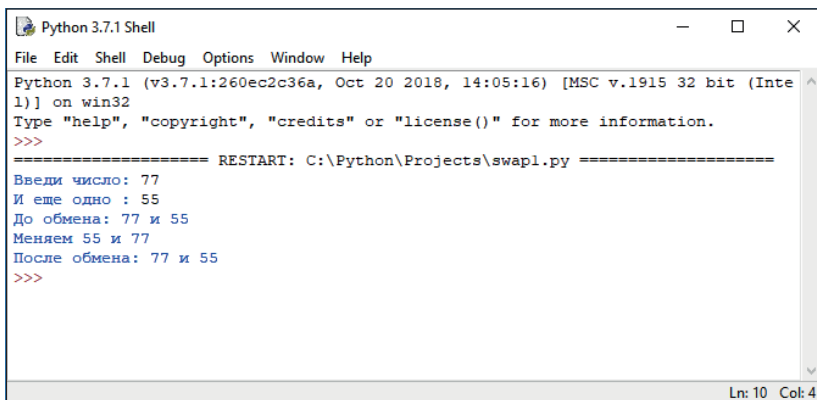
```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\swap1.py =====
Введи число: 77
И еще одно : 55
До обмена: 77 и 55
После обмена: 77 и 55
>>>
```

Рис. 5.3. Что-то пошло не так

Подожди-ка, здесь что-то не так. Ничего не поменялось. Почему? Давай для проверки временно вставим инструкцию `print` в функцию `exchange()`:

```
def exchange(x1, x2) :
    Swap = x1
    x1 = x2
    x2 = Swap
    print ("Меняем " + str(x1) + " и " + str(x2))
```

Программа, запускаемая с теми же числами, что и ранее, выдаст следующий результат (рис. 5.4):



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\swap1.py =====
Введи число: 77
И еще одно : 55
До обмена: 77 и 55
Меняем 55 и 77
После обмена: 77 и 55
>>>
```

Рис. 5.4. Результат работы программы

Два числа поменялись местами, но результат неочевиден.

Все это имеет простую причину: функция не получает в качестве параметра само значение переменной, а только *копию* ее значения. Исходное значение переменной не изменяется. Внутри функции изменяется только значение копии. И оно удаляется после выхода из функции.



Как насчет инструкции `return`? Будет ли она работать с двумя значениями? Давай попробуем:

```
def exchange(x1, x2) :
    Swap = x1
    x1 = x2
    x2 = Swap
    return x1, x2
```

Если мы так изменим исходный код, у нас не будет сообщения об ошибке. Фактически инструкция `return` может вернуть два (или более) значения.

Это особенность языка программирования Python. Другие известные языки программирования, такие как C++, C# или Java, могут возвращать только одно значение с помощью инструкции `return`. Для этого есть возможность сослаться на переменную как на параметр, а не копию, чтобы изменить оригинальное значение.

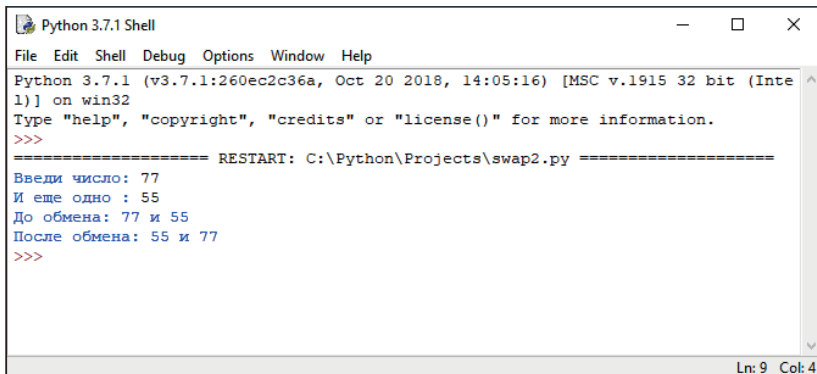


И как же нам использовать это в основной программе? Давай посмотрим (\Rightarrow `swap2.py`):

```
# Основная программа
print("Введи число: ", end="")
Number1 = int(input())
print("И еще одно : ", end="")
Number2 = int(input())
print("До обмена: " + str(Number1) + " и " + str(Number2))
Number1, Number2 = exchange(Number1, Number2)
print("После обмена: " + str(Number1) + " и " + str(Number2))
```

Здесь уже нет смысла повторно использовать переменные `Number1` и `Number2`.

- Введи исходный код и попробуй запустить программу с парой чисел (рис. 5.5).



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\swap2.py =====
Введи число: 77
И еще одно : 55
До обмена: 77 и 55
После обмена: 55 и 77
>>>
```

Рис. 5.5. Исправленная версия программы



Как видишь, в Python ты можешь выполнять несколько заданий *квазипараллельно*. Поэтому, если ты захочешь присвоить значение нескольким переменным, это делается так:

```
Number = 5
Text = "Привет"
```

Или так:

```
Number, Text = 5, "Привет"
```

Это работает, даже если используются переменные разных типов. Инструкция `print` может использоваться для проверки того, что они являются переменными одного типа:

```
print(str(Number), Text)
```

Сортировка чисел

Я сейчас вернусь немного назад – в нашу игру-лотерею. Там у нас была серия случайных чисел, сгенерированных в надежде, что некоторые из них в итоге будут «правильными». При выводе чисел мне все-таки не нравится то, что это происходит вразнобой. Аккуратная строка отсортированных чисел выглядела бы в этой игре намного лучше, как считаешь? Для этого нам нужно собрать все числа из списка, а затем передать их функции сортировки. Простой процесс сортировки ты можешь увидеть в следующем примере:

```
def bubblesort(x, Index) :
    for i in range(Index) :
        for j in range(Index-i-1) :
            if x[j] > x[j+1] :
                x[j], x[j+1] = exchange(x[j], x[j+1]);
```

В качестве параметра функция принимает список отсортированных номеров и количество элементов для сортировки.

bubblesort() – это метод сортировки, в котором отдельные значения как бы поднимаются, как пузыри (от англ. *bubble* – пузырик), до тех пор, пока не будут отсортированы по размеру. Это происходит в двух циклах for:

- во внутреннем цикле два числа сравниваются и меняются местами, если первое число больше второго. Это повторяется, пока обмен необходим и возможен. Для обмена мы можем использовать функцию `exchange()`, которую обсуждали ранее;
- после каждой итерации внешнего цикла, подсчитывающего элементы, внутренний цикл укорачивается (чем меньше значение `Index-i-1`, тем больше значение `i`).

Во время процесса сортировки меньшие числа как бы «скользят» все дальше и дальше вперед, как пузырьки в воде движутся на поверхность.

- Чтобы проверить эту сортировку, введи следующий код или измени свою старую версию игры (\Rightarrow *lotto4.py*):

```
import random

# Функция
def exchange(x1, x2) :
    Swap = x1
    x1 = x2
    x2 = Swap
    return x1, x2
def bubblesort(x, Index) :
    for i in range(Index) :
        for j in range(Index-i-1) :
            if x[j] > x[j+1] :
                x[j], x[j+1] = exchange(x[j], x[j+1]);

# Основная программа
Ball = []
Lotto = [0,0,0,0,0,0]
# Все шары еще не "вытащены"
for Nr in range(1,50) :
```

```
Ball.append(False)
Case = random.randint(1,49)
# "Вытягивается" шесть шаров
for Nr in range(6) :
    # поиск неиспользованных шаров
    while Ball[Case] :
        Case = random.randint(1,49)
    # Пометка использованного шара "вытащенным"
    Ball[Case] = True
    Lotto[Nr] = Case
# Сортировка и отображение
bubblesort(Lotto,6)
print(Lotto)
```

Сначала мы определим две функции, которые хотим использовать, и одну из них в основной программе. Там мы создадим два списка:

```
Ball = []
Lotto = [0,0,0,0,0,0]
```

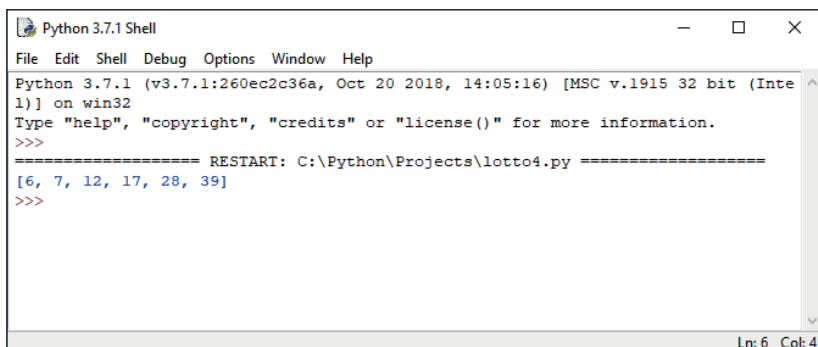
Один из них остается пустым, а другой заполним шестью нулями. Далее будут добавлены лотерейные номера. Это выполняется следующим образом:

```
Ball[Case] = True
Lotto[Nr] = Case
```

Ball – это число в лотерее, помечаемое как выпавшее, и новое случайное число добавляется в список Lotto. Как видишь, функция append() имеет смысл только тогда, когда в список добавляется новое число.

В конце эти числа сортируются (шесть номеров лотереи), а в качестве второго параметра указываются 6 выбранных чисел, формируя список выпавших чисел:

```
bubblesort(Lotto,6)
print(Lotto)
```



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\lotto4.py =====
[6, 7, 12, 17, 28, 39]
>>>
```

Рис. 5.6. Числа отсортированы и выведены в виде строки

В конце есть кое-что удивительное. С помощью функции `print()` выводится содержимое списка в виде строки. Но если ты хочешь вывести их так же, как мы делали ранее в игре-лотерее, тебе нужно будет заменить последнюю строку на следующий дополнительный цикл:

```
for Nr in range(6) :
    print("№ " + str(Nr+1) + " => " + str(Lotto[Nr]))
```

Ничего не заметил? Кажется, что список, подобный `Lotto`, также является *глобальным* в функции сортировки, по крайней мере мы не использовали инструкцию `return`, а в конце все равно все отсортировалось.

Это происходит из-за списков, которые передаются в качестве параметров, а не копии, поскольку они иногда могут быть очень большими. Вместо этого вы работаете с так называемой ссылкой: функция обращается непосредственно к содержимому списка (а не к его копии).

Поэтому операция, применяемая к списку в функции, например сортировка, также работает и вне функции.



Наконец, существует метод «сортировки» для списков – `sort`. С его помощью мы можем упростить код и поместить следующую строку перед последней инструкцией `print`:

```
Lotto.sort()
```

Даже в этом случае список, о котором идет речь, сортируется. Тогда мне не пришлось бы здесь использовать ссылки, потому что, как ты уже знаешь, списки в качестве парамет-

ров отличаются от обычных переменных. Таким образом, теперь ты умеешь сортировать элементы.

Подведение итогов

Прежде чем отдохнуть, я хочу понять, что ты усвоил. В этой главе мы рассмотрели не так много нового, но все это достаточно важно, потому что ты теперь сможешь определить любую (почти любую) функцию:

<code>def</code>	Определение функции
<code>return</code>	Выход из функции
<code>global</code>	Объявление переменной как глобальной
<code>sort()</code>	Сортировка элементов в списке

Несколько вопросов...

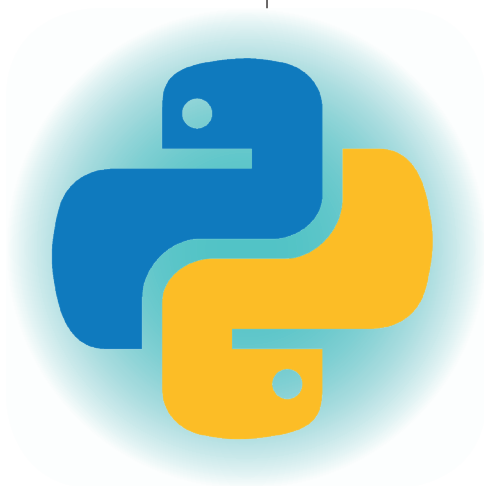
1. В чем основное различие между глобальными и локальными переменными?
2. Могут ли параметры быть переменными?

...и задач

1. Измени программу обмена так, чтобы менять местами две строки вместо двух чисел.
2. Необходимо определить сумму целых чисел от 1 до предела. Напиши программу.
3. Вычисли среднее из целых чисел от 1 до предела. Напиши программу.
4. Измени предыдущие проекты: в каких из них можно использовать функции?

6

Классы и модули



В предыдущих главах ты написал довольно много кода. Если ты продолжишь в том же духе, исходный код станет все более сложным. Конечно, некоторые фрагменты можно объединить в функции, но если программы содержат не тридцать строк, а более ста или даже тысячи, возникает вопрос, нельзя ли скомбинировать функциональность в методы, или разделить их, чтобы сохранить общую картину.

Конечно, Python предлагает соответствующие конструкции, о которых ты узнаешь в этой книге. Речь пойдет о так называемом объектно-ориентированном программировании (сокращенно ООП).

Итак, в этой главе ты узнаешь:

- ⊙ как написать свой собственный класс;
- ⊙ что такое инкапсуляция и наследование;
- ⊙ как разделить программу на несколько модулей;
- ⊙ кое-что о публичности и приватности.

Потомки

Начнем с малого, чтобы лучше понять, что такое объектно-ориентированное программирование.

Как видно из названия, речь идет об объектах. Это «объекты», которые всегда окружают нас. Так, например, дома, де-

6

ревя, автомобили, люди. Ты – это тоже объект. Аналогично и в Python есть объекты, но, разумеется, они виртуальные. Концепция заключается в обобщении данных и функций. Вместо данных можно также сказать атрибуты и свойства. А функции здесь можно назвать методами. Они используются для обработки данных или для обработки атрибутов. Все характеристики (т. е. атрибуты, методы) аналогичных объектов суммируются в так называемые классы. В класс собран весь функционал, на который способен объект. Это называется *инкапсуляцией* (рис. 6.1).

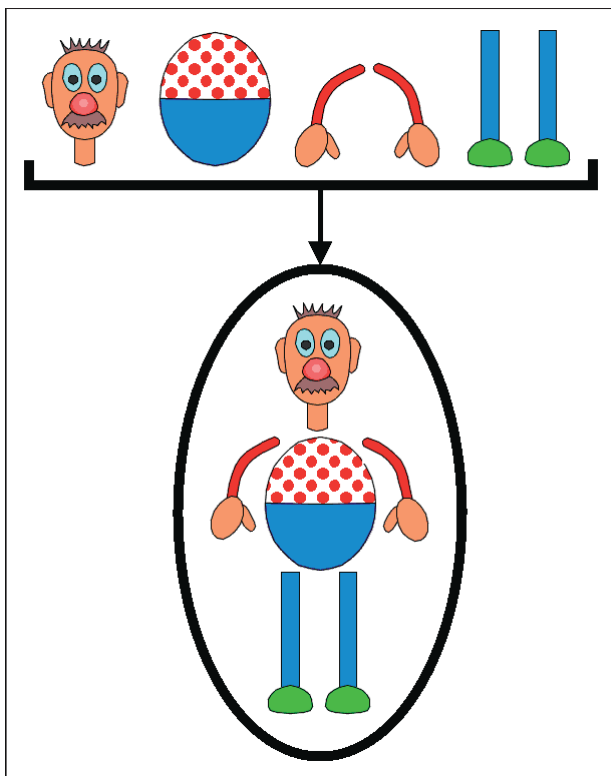


Рис. 6.1. Инкапсуляция на примере задорного мужичка

Ты уже использовал такой класс несколько раз. Это класс списка. Вот так, например, будет выглядеть объект списка:

```
List = []
```

Он также называется *экземпляром* класса списка. В нашем примере он пустой. Чтобы добавить элемент, необходим метод `append()`:

```
List.append(Element)
```

Класс списка допускает использование дополнительных методов, в том числе:

```
# Вставить элемент в определенной позиции
List.insert(Nr, Element)
# Удалить определенный элемент
List.remove(Element)
# Значение элемента списка
Value = len(List)
```

Теперь мы можем сами написать собственный класс. Он может быть связан с математикой, но не обязательно. Давай вернемся на пару сотен лет назад и посетим некоего доктора Франкенштейна. Он разработал аналогичный класс:

```
class Monster :
    pass
```

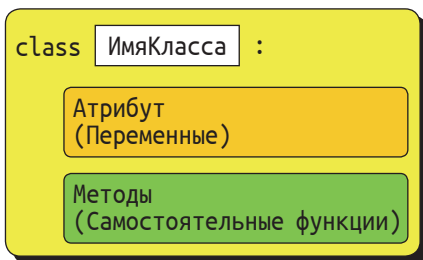
Для чего нужно слово `pass`? Это просто заполнитель. Потому что в Python конструкция с двоеточием (:) в конце никогда не может быть пустой. Поскольку в нашем мини-классе все еще нет элементов, вставим слово `pass`. Мы заменим его, как только класс получит свои атрибуты и методы.

Это очень простой монстр, у которого есть *имя* и *сущность* в качестве атрибута. Существует также подходящий *метод*. У доктора Франкенштейна не было такого удобного способа, чтобы проверить свое творение, как у нас. Рассмотрим нашего милого монстра. Давай соберем все вместе:

```
class Monster :
    # Атрибут
    Name = "Фрэнки"
    Character = "необычный"
    # Метод
    def show(self) :
        print("Имя: " + Name)
        print("Особенность: " + Character)
```

В отображаемом методе `show` два атрибута получают одинаковое значение.

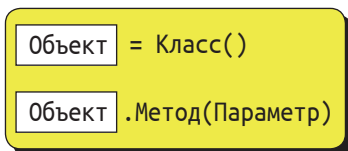
6



В чем разница между функцией и методом? Прежде всего они определяются по-разному. Хотя функция допустима и используется повсюду, метод привязан к объекту. Вот почему они совершенно разные. Давай посмотрим, как выглядит основная часть программы:

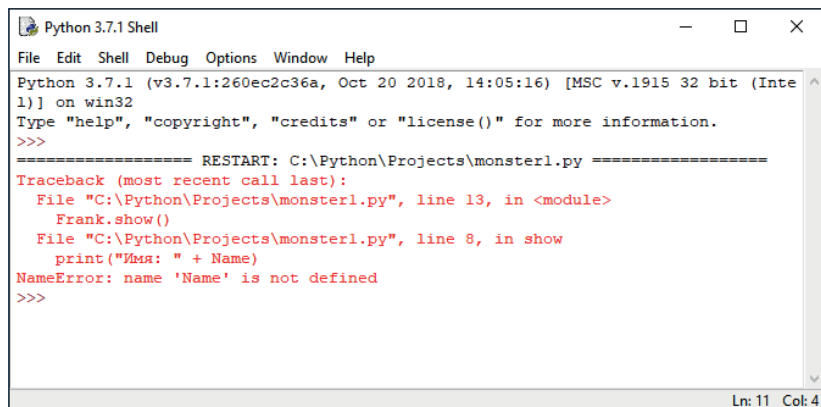
```
Frank = Monster()  
Frank.show()
```

Во-первых, создается объект, то есть экземпляр класса `Monster`. Скобки после имени класса важны. Для доступа к методу `show()` необходимо указать имя объекта (в нашем примере – `Frank`).



Выше показано, что методы могут также иметь параметры. Точка (`.`), которая соединяет объект и связанный с ним метод, тоже важна.

Если ты запустишь нашу небольшую программу, то, к сожалению, будешь разочарован сообщением об ошибке (рис. 6.2).



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\monster1.py =====
Traceback (most recent call last):
  File "C:\Python\Projects\monster1.py", line 13, in <module>
    Frank.show()
  File "C:\Python\Projects\monster1.py", line 8, in show
    print("Имя: " + Name)
NameError: name 'Name' is not defined
>>>
```

Рис. 6.2. Сообщение об ошибке: имя *Name* не определено

Почему имя не определяется? Объект не знает свои собственные атрибуты? То же самое произойдет с атрибутом. Как решить эту проблему?

self и __init__

Прежде всего нужно знать, что при определении класса методы доступны однократно. Если, например, ты создаешь пять объектов типа *Monster*, атрибуты *Name* и *Entity* используются пять раз, а метод *show()* – только один раз. Хотя атрибуты теперь «находятся» в объектах (и могут иметь разные значения для каждого объекта), методы остаются в пределах класса.

Чтобы метод получил доступ к атрибуту объекта, он должен быть связан с атрибутами с помощью первого параметра – *self* и только в определении класса. Посмотри, как меняется наш код (\Rightarrow *monster1.py*):

```
class Monster :
    # Атрибут
    Name = "Фрэнки"
    Character = "необычный"
    # Метод
    def show(self) :
        print("Имя: " + self.Name)
        print("Особенность: " + self.Character)

# Основная программа
Frank = Monster()
Frank.show()
```

6

Методу `show` передается первый (только здесь) параметр, `self`. Кроме того, отдельные атрибуты связаны через `self` и точку. Код основной программы не меняется, метод `show()` по-прежнему вызывается без параметров.

Если метод связан с несколькими параметрами, первым всегда должен указываться `self`.

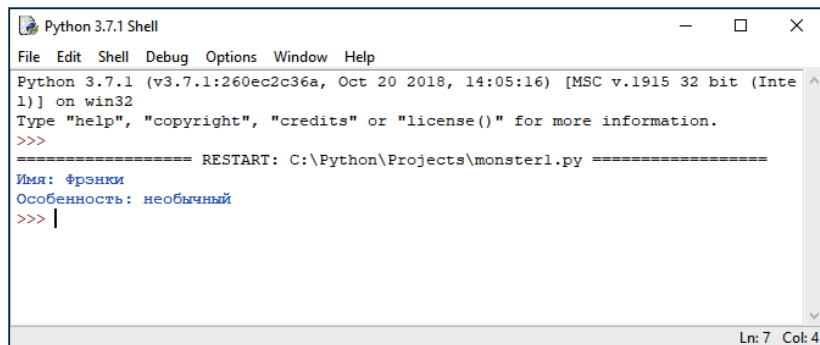
```
def Имя (self, параметр) :  
    БлокИнструкций
```

Кроме того, все атрибуты доступны только через `self`, например:

```
self.Атрибут = Результат|Формула
```

Все это работает только внутри класса, но не вне.

- Создай программу с показанным выше исходным кодом. Сохрани программу под именем *monster1.py*, а затем запусти ее (рис. 6.3).



```
Python 3.7.1 Shell  
File Edit Shell Debug Options Window Help  
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
===== RESTART: C:\Python\Projects\monster1.py =====  
Имя: Фрэнки  
Особенность: необычный  
>>> |
```

Рис. 6.3. Теперь программа работает

Теперь монстр Frank откликается на свое имя и имеет собственный характер.

Далее мы установим значения двух атрибутов. Попробуем сделать метод более гибким, если эти значения использовать только при создании объекта. Для этого нам нужно расширить наше определение класса (\Rightarrow *monster2.py*):

```
class Monster :  
    # Инициализация атрибута  
    def __init__(self, name, character) :  
        self.Name = name  
        self.Character = character  
    # Метод  
    def show(self) :  
        print("Имя: " + self.Name)  
        print("Особенность: " + self.Character)  
  
# Основная программа  
Frank = Monster("Фрэнки", "необычный")  
Frank.show()
```

- Измени исходный код своей программы и запусти ее. Как ты видишь, мы получили тот же результат, что и раньше.

Что изменилось? Первое, что ты заметишь, – это добавленный новый метод. Метод `__init__` иногда называют конструктором, потому что он вызывается автоматически при создании (конструировании) объекта.

А зачем нужно так много «черточек»? Имя метода `__init__` содержит два символа подчеркивания (`_`) до и после своего слова `init`, которые вводятся сочетанием клавиш **Shift+–** (клавиша после 0 (нуля)). Они необходимы, а для чего – я расскажу позже.



В рамках метода `__init__` двум атрибутам присваивается значение:

```
def __init__(self, name, character) :  
    self.Name = name  
    self.Character = character
```

По этой причине при создании объекта в основной программе требуется два параметра:

```
Frank = Monster("Фрэнки", "необычный")
```

Конечно, можно было бы одновременно сгенерировать сразу два объекта, например так:

```
Albert = GMonster("Альберт", "задумчивый")
```



Ты можешь попытаться избавиться от «черточек» и просто запустить метод `init()`. Когда ты запустишь программу, то получишь ошибку, показанную на рис. 6.4.

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\monster2.py =====
Traceback (most recent call last):
  File "C:\Python\Projects\monster2.py", line 13, in <module>
    Frank = Monster("Фрэнки", "необычный")
TypeError: Monster() takes no arguments
>>> |
```

Рис. 6.4. Ошибка передачи аргументов

Во время создания не получается использовать какие-либо параметры. Но это можно решить следующим образом:

```
Frank = Monster()
Frank.init("Фрэнки", "необычный")
Frank.show()
```

В этом случае метод `init()` должен быть вызван отдельно.

Наследование

Доктор Франкенштейн планирует создать еще двух монстров. Конечно, мы хотим на них взглянуть. Сначала можно подумать, что нужно создать один и тот же класс три раза, каждый раз с другим именем. Но есть более короткий способ (\Rightarrow *monster3.py*):

```
class Monster :
    # Инициализация атрибута
    def __init__(self, name, character) :
        self.Name = name
        self.Character = character
    # Метод
    def show(self) :
        print("Имя: " + self.Name)
        print("Особенность: " + self.Character)

class GMonster (Monster):
    pass
```

```
class SMonster (Monster):
    pass
```

И больше ничего? Нет. Достаточно указать имя уже определенного класса в первой строке, например в виде параметра в круглых скобках. Будет создан *производный* класс. Другими словами, это *дочерний* класс, и он *наследует* все элементы родительского класса (рис. 6.5).

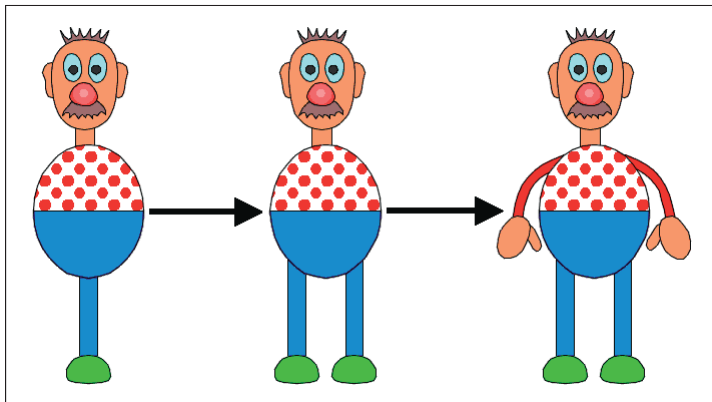


Рис. 6.5. Наследование на примере задорного мужичка

Кстати, ты мог создать класс `Monster` следующим образом:

```
class Monster(object):
```

В более ранних версиях Python это было обычным явлением, что новый класс автоматически производился из объекта «родительского класса».



В основной программе теперь может быть создано три (разных) монстра:

```
Frank = Monster("Фрэнки", "необычный")
Frank.show()
Albert = GMonster("Альберт", "задумчивый")
Albert.show()
Sigmund = SMonster("Зигмунд", "веселый")
Sigmund.show()
```

- Напечатав код, а затем запустив программу, ты получишь результат, показанный на рис. 6.6.

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\--\monster4.py =====
Имя: Фрэнки
Особенность: необычный
Тип: Монстр
Имя: Альберт
Особенность: задумчивый
Тип: Дух монстра
Имя: Зигмунд
Особенность: веселый
Тип: Душа монстра
>>> |
```

Рис. 6.6. Результат работы программы: три монстра!

Теперь я хотел бы расширить все три класса с помощью метода, который имеет возвращаемое значение. Для родительского класса `Monster` код выглядит так:

```
def Type(self) :
    return "Монстр"
```

Метод `show` тоже должен быть расширен. Я сделал новую версию программы, код которой выглядит так (\Rightarrow `monster4.py`):

```
class Monster :
    # Инициализация атрибута
    def __init__(self, name, character) :
        self.Name = name
        self.Character = character
    # Метод
    def Type(self) :
        return "Монстр"
    def show(self) :
        print("Имя: " + self.Name)
        print("Особенность: " + self.Character)
        print("Тип: " + self.Type())

class GMonster (Monster):
    # Метод
    def Type(self) :
        return "Дух монстра"
```

```
class SMonster (Monster):
    # Метод
    def Type(self) :
        return "Душа монстра"

# Основная программа
Frank = Monster("Фрэнки", "необычный")
Frank.show()
Albert = GMonster("Альберт", "задумчивый")
Albert.show()
Sigmund = SMonster("Зигмунд", "веселый")
Sigmund.show()
```

Расширенный метод show() теперь также отображает тип соответствующего объекта:

```
print("Тип: " + self.Type())
```

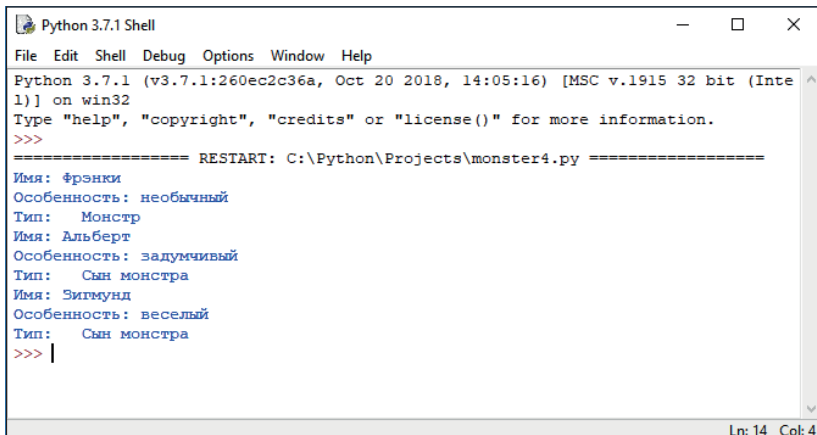
Помимо прочего, важно, чтобы слово self было помещено перед методом. Это касается всех элементов.



Теперь у дочернего класса есть все элементы родительского класса и, конечно, его собственные новые атрибуты и методы – в зависимости от того, что было переопределено.



- Введи код из листинга выше, а затем запусти программу. Должно получиться примерно так, как показано на рис. 6.7.



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (tags/v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\monster4.py =====
Имя: Фрэнки
Особенность: необычный
Тип: Монстр
Имя: Альберт
Особенность: задумчивый
Тип: Сын монстра
Имя: Зигмунд
Особенность: веселый
Тип: Сын монстра
>>> |
```

Рис. 6.7. Результат работы программы: отображается тип монстра

Обычно объект Frank вызывает собственный метод `Type()` в собственном методе `show`. Интересно, что два других объекта сначала вызывают унаследованный метод (`= show()`) и не получают доступ к дочернему (унаследованному) методу, а обращаются к собственному методу.

Как узнать, какой метод `Type()` использовать? Поскольку у них фактически есть два метода с одним и тем же именем `Type()`, они могут выбрать и использовать неправильный метод «по ошибке». В Python существует механизм, благодаря которому автоматически используется дочерний метод.

На самом деле так происходит и в других языках программирования, таких как, например, C++ или Java: программа, подобная приведенной выше, выводит тип Монстр три раза. Только если определить методы `Type()` по-разному, код будет работать автоматически, как в Python. Почему так происходит?

Существует два способа вызова метода. Обычно в той позиции, где вызывается метод, задается начальный адрес. Этот параметр определяет, какой метод используется. Или ты помечаешь позицию вызова с помощью заполнителя. Только во время запуска программы будет использоваться адрес соответствующего метода.



Другие языки программирования:

```
def show() :
```

фиксированный адрес

нормальный
метод

Python:

```
def show() :
```

заполнитель

виртуальный
метод

Это виртуальные методы. В то время как в других языках программирования такие методы имеют метку `virtual`, все методы в Python по умолчанию виртуальны.



Модули программы

Исходный код будет только увеличиваться в объеме со временем. Эта программа не слишком велика, но представь, что ты захотел расширить код на нужное приложение или реализовать дополнительный функционал. Тогда код можно разделить на несколько файлов – *модулей*.

moster.py

Определение класса
с помощью атрибутов
и методов

Ctrl+X

Основная программа:
создание и использование
объекта

mosterlab.py

Определение класса
с помощью атрибутов
и методов

Ctrl+V

Давай создадим новый файл, в котором мы сможем сохранить определения классов. В предыдущем файле остается только основная программа.

- Убедись, что последняя версия программы открыта. Создай новый файл в среде разработки Python (IDLE), щелкнув мышью по пункту **File** (Файл) в строке меню, а затем выбрав пункт **New file** (Создать файл) (или нажав сочетание клавиш **Ctrl+N**).

- Перейди в файл с кодом программы, выдели целиком определения классов монстров и вырежи их (нажав сочетание клавиш **Ctrl+X**) (рис. 6.8). Затем перейди в окно нового файла и нажми сочетание клавиш **Ctrl+V**, чтобы вставить скопированный код.

Рис. 6.8. Определения классов монстров выделены

- Сохрани оба файла, первый (старый) под новым именем (у меня это *monster5.py*). Новый файл назовем *monsterlab.py*.

Скопированный код теперь находится в файле модуля:

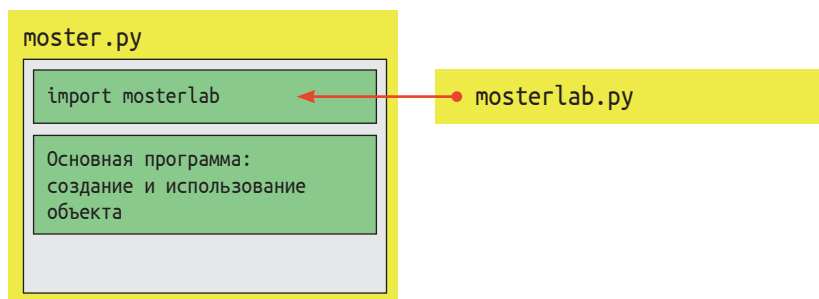
```
class Monster :
    # Инициализация атрибута
    def __init__(self, name, character) :
        self.Name = name
        self.Character = character
```

```
# Метод
def Type(self) :
    return "Монстр"
def show(self) :
    print("Имя: " + self.Name)
    print("Особенность: " + self.Character)
    print("Тип: " + self.Type())

class GMonster (Monster):
    # Метод
    def Type(self) :
        return "Дух монстра"

class SMonster (Monster):
    # Метод
    def Type(self) :
        return "Душа монстра"
```

Разумеется, старая программа без определения классов больше не работает. Но как основная программа узнает, где теперь находятся определения классов?

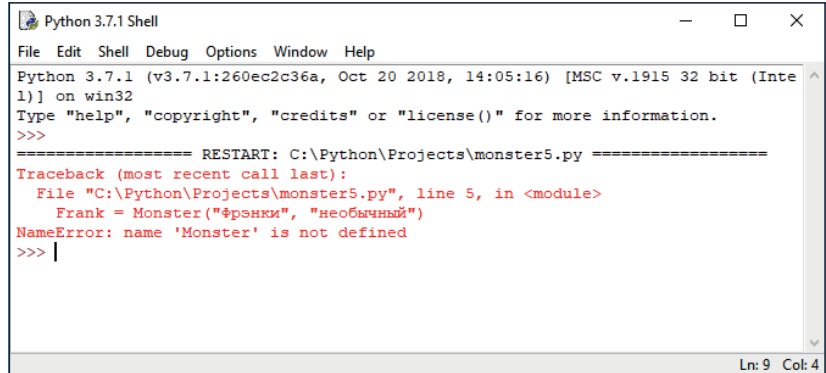


Классы, с которыми мы работали раньше, определены в одном модуле, который импортируется в программу. Это выглядит так:

```
import monsterlab
```

Заверши код программы, вставив эту строку кода вверху. Затем запусти программу.

Тебя удивило сообщение об ошибке? Так или иначе компьютер не узнаёт класс `Monster` (и, разумеется, два других дочерних класса) (рис. 6.9).



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\monster5.py =====
Traceback (most recent call last):
  File "C:\Python\Projects\monster5.py", line 5, in <module>
    Frank = Monster("Фрэнки", "необычный")
NameError: name 'Monster' is not defined
>>> |
```

Рис. 6.9. Ошибка определения имени *Monster*

Как сообщить компьютеру, что мы уже импортировали соответствующий модуль? Недостаточно просто импортировать модуль, также нужно указать его имя, если ты хочешь использовать некий код из этого модуля (например, функцию). Давай сделаем это здесь (\Rightarrow *monster5.py*):

```
import monsterlab

# Основная программа
Frank = monsterlab.Monster("Фрэнки", "необычный")
Frank.show()
Albert = monsterlab.GMonster("Альберт", "задумчивый")
Albert.show()
Sigmund = monsterlab.SMonster("Зигмунд", "веселый")
Sigmund.show()
```

Поместив перед именем каждого класса имя модуля, из которого происходит класс, компьютер не будет делать ошибок.

- Исправь исходный код основного модуля, затем запусти программу.

Теперь все работает (рис. 6.10).

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\monster5.py =====
Имя: Фрэнки
Особенность: необычный
Тип: Монстр
Имя: Альберт
Особенность: задумчивый
Тип: Дух монстра
Имя: Зигмунд
Особенность: веселый
Тип: Душа монстра
>>> |
```

Рис. 6.10. Ошибка исправлена

Наша семейка монстров готова, но не все еще идеально. Во-первых, существует множество атрибутов и методов, которые могут быть согласованы в обоих классах – в зависимости от твоего воображения. Займись этим.

А тем временем я хотел бы внести несколько потенциальных улучшений, чтобы основная программа стала чуть более элегантной. Если мы немного изменим строку кода, отвечающую за импорт модуля, мы сможем сократить код ниже (\Rightarrow *monster6.py*):

```
from monsterlab import *

# Основная программа
Frank = Monster("Фрэнки", "необычный")
Frank.show()
Albert = GMonster("Альберт", "задумчивый")
Albert.show()
Sigmund = SMonster("Зигмунд", "веселый")
Sigmund.show()
```

Вместо импорта всего модуля мы лишь извлекаем отдельные классы, обозначенные звездочкой (*) после слова `import`. Также может использоваться и такая строка:

```
from monsterlab import Monster
```

Но тогда программа будет работать только для объекта класса, потому что ссылка на вызов `monsterlab` указана до вызова класса.

```
import Модуль
from Модуль import Элемент
```

Таким образом, ты можешь интегрировать полный модуль или его части. И если ты хочешь извлечь все элементы, просто укажи звездочку:

```
from Modul import *
```

На первый взгляд кажется, что это сложнее и лучше – просто импортировать модуль. Но разница в том, что тебе больше не нужно перечислять имена классов в модуле. Также ты можешь сократить объявление объекта – вот так:

```
Frank = Monster("Фрэнки", "необычный")
Albert = GMonster("Альберт", "задумчивый")
Sigmund = SMonster("Зигмунд", "веселый")
```

Приватный или публичный?

Остается недостаток, который ты даже не заметил вначале и, возможно, никогда не заметил бы. После создания объекта, такого как `Frank`, можно напрямую получить доступ к атрибутам внутри объекта (\Rightarrow `monster7.py`):

```
Frank = Monster("Фрэнки", "необычный")
print(Frank.Name);
print(Frank.Character)
print(Frank.Type())
```

Что не так? А вот что. В нашем примере ничего страшного нет, но представь себе определение более сложных классов, которые имеют много атрибутов. Часто нежелателен прямой доступ к атрибутам извне. Потому что в зависимости от уровня доступа можно изменить значение атрибута. В случае с многочисленными объектами может случиться так, что что-то изменилось, хотя не должно было. И найти такую ошибку часто бывает нелегко.

Вернемся к термину, который я упомянул в начале: инкапсуляция. Ее также можно назвать замкнутостью и защитой от внешнего доступа. В нашем случае достаточно получить доступ к методу `show()`, и ты получишь доступ ко всему

остальному коду, т. к. он является внутренним. А нужно ли его менять с наружным доступом?

Давай посмотрим, как Python защищает атрибуты и методы от внешнего доступа. Для этого нам нужно войти в модуль `monsterlab`.

➤ Открой файл `monsterlab.py` и сохрани его под именем `monsterlabx.py`.

Затем тебе нужно добавить два символа подчеркивания (`_`) для каждого из атрибутов `Name` и `Character`, а также метода `Type()`:

```
class Monster :
    # Инициализация атрибута
    def __init__(self, name, character) :
        self.__Name = name
        self.__Character = character
    # Метод
    def _Type(self) :
        return "Монстр"
    def show(self) :
        print("Имя: " + self.__Name)
        print("Особенность: " + self.__Character)
        print("Тип: " + self._Type())

class GMonster (Monster):
    # Метод
    def __Type(self) :
        return "Дух монстра"

class SMonster (Monster):
    # Метод
    def __Type(self) :
        return "Душа монстра"
```

Импортируем новый модуль в основную программу и соответствующим образом настроим имена (\Rightarrow `monster8.py`):

```
from monsterlabX import *

# Основная программа
Frank = Monster("Фрэнки", "необычный")
print(Frank.__Name);
print(Frank.__Character)
print(Frank.__Type())
```

Запуск программы выдаст сообщение об ошибке (рис. 6.11):

```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\monster8.py =====
Traceback (most recent call last):
  File "C:\Python\Projects\monster8.py", line 6, in <module>
    print(Frank.__Name__);
AttributeError: 'Monster' object has no attribute '__Name'
>>> |
```

Рис. 6.11. Ошибка: не обнаружены атрибуты объекта

Хотя `_Name` (а также `_Character` и `_Type`) четко определены и модуль импортирован, основная программа не может получить доступ к этим элементам. Потому что рассматриваемые атрибуты и метод `Type` теперь считаются закрытыми. И именно этого мы и хотели достичь.

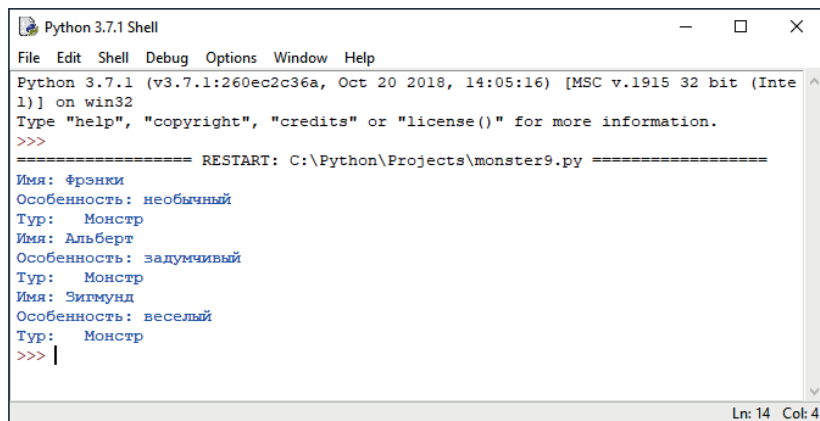


В Python применяется следующее правило: если ты добавляешь к имени класса два символа подчеркивания, элемент становится закрытым. В противном случае каждый элемент считается публичным. Доступ к публичным элементам можно получить из любой позиции, но в пределах объекта. (Вспомни о доступности глобальных и локальных переменных.)

- Чтобы попробовать новый модуль, тебе нужно изменить только первую строку в исходном коде `monster6.py` (если ты назвал свой модуль иначе, разумеется, указывай свое имя):

```
from monsterlabX import *
```

Запуск программы заканчивается горьким разочарованием (рис. 6.12). Что происходит?



```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\monster9.py =====
Имя: Фрэнки
Особенность: необычный
Тур: Монстр
Имя: Альберт
Особенность: задумчивый
Тур: Монстр
Имя: Зигмунд
Особенность: веселый
Тур: Монстр
>>> |
```

Рис. 6.12. Все монстры стали одного типа

Теперь дочерние классы `Monster` забыли про свой тип и используют родительский. Поэтому, очевидно, нельзя сказать, что программа стала лучше. Итак, все элементы должны быть определены заново?

Улучшение было своего рода «промежуточным». Взгляни на решение, которое предоставляет Python (\Rightarrow `monsterlabx.py`):

```
class Monster :
    # Инициализация атрибута
    def __init__(self, name, character) :
        self.__Name = name
        self.__Character = character
    # Метод
    def _Type(self) :
        return "Монстр"
    def show(self) :
        print("Имя: " + self.__Name)
        print("Особенность: " + self.__Character)
        print("Тип: " + self._Type())

class GMonster (Monster):
    # Метод
    def _Type(self) :
        return "Дух монстра"

class SMonster (Monster):
    # Метод
    def _Type(self) :
        return "Душа монстра"
```

Смотри внимательно, чтобы увидеть разницу: теперь метод `Type()` имеет только одно подчеркивание (`_`). И вот разница между соответствующими параметрами доступа (табл. 6.1).

6

Таблица 6.1. Типы доступа

Тип	Имя	Доступ
Public (публичный)	Без подчеркивания	В любом месте
Protected (защищенный)	С одним подчеркиванием	В дочерних классах
Private (приватный)	С двумя подчеркиваниями	Только внутри объекта

Внеси изменения, а затем запусти программу еще раз. (Ты можешь скачать все файлы примеров с сайта dmkpress.com.) Теперь, как раньше, все монстры знают свой тип. Снаружи ты не можешь получить доступ к методу, если только он не касается класса `Monster`. (Таким образом доступ к методу остается «защищенным».)

Подведение итогов

Вот и закончилась очередная глава. Ты написал собственные классы и создал собственные объекты. Однако ты еще не гуру программирования (все самое интересное еще впереди). Посмотрим, что ты узнал о Python.

<code>class</code>	Определяет класс
<code>self</code>	Ссылка на объект или класс
<code>_init</code>	Метод инициализации
<code>from</code>	Импортирует отдельные элементы модуля
<code>*</code> (звездочка)	Доступ ко всем элементам модуля
<code>pass</code>	Пустая строка, отображающая пустую структуру
<code>__</code>	Определяет приватный элемент
<code>_</code>	Определяет защищенный элемент
<code>.</code>	Связывает объект с атрибутом или методом

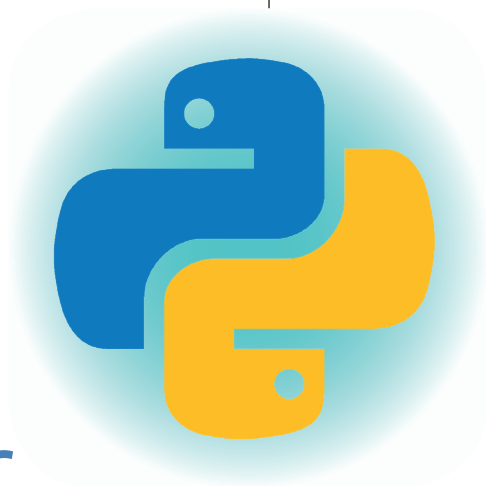
Несколько вопросов...

1. Что такое методы и атрибуты класса?
2. Что такое инкапсуляция?
3. В чем разница между приватными и публичными элементами?

...а задач нет

7

Введение в tkinter



До сих пор мы ограничивались вводом кода вручную, но в операционной системе Windows мы привыкли к чему-то большему. Мы используем графический интерфейс, открываем меню, щелкаем по окнам с кнопками, изображениям и полям ввода. Но наши проекты на Python до сих пор были невзрачными, без красивого интерфейса. Теперь это нужно изменить. Ты можешь сделать это с помощью окон, кнопок и т. д.

Итак, в этой главе ты узнаешь:

- ⦿ что такое tkinter;
- ⦿ об элементах Label и Button;
- ⦿ как работать с событиями;
- ⦿ что такое компоновка окон;
- ⦿ о диалоговых окнах.

Создаем окно

Работая с интерфейсом IDLE, мы не использовали многие компоненты, которые содержит операционная система Windows. Было бы круто вывести хотя бы одно диалоговое окно на экран. И если нам это удастся, мы заполним это окно разными компонентами.

7



Что такое *компоненты*? Это объекты, которые обычно используются для управления программой. Например, кнопки (Button), меню и метки (Label). *Виджеты* тоже относятся к компонентам tkinter.

Разумеется, нам нужен модуль, который мы должны сначала импортировать:

```
import tkinter
```

Модуль tkinter содержит все компоненты, необходимые для создания графического пользовательского интерфейса. Особенностью модуля является то, что он работает независимо от того, что делает операционная система.



Кстати, слово tkinter является аббревиатурой для *Toolkit interface* – инструменты для разработки интерфейсов. Модуль содержит инструменты для разработки так называемого GUI – *Graphical User Interface* – графического пользовательского интерфейса.

Наша первая графическая программа очень короткая:

```
import tkinter
Window = tkinter.Tk()
```

Если ты запустишь этот код, то увидишь окно, показанное на рис. 7.1.

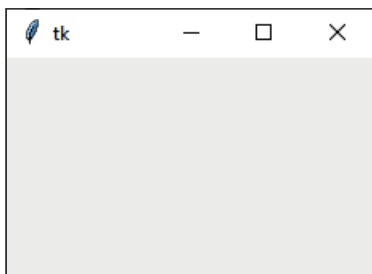


Рис. 7.1. Созданное окно

Сначала создается объект окна, который запускается нажатием функцией Tk(). Это основной класс модуля tkinter. Важно, что имя Tk написано с прописной буквы «Т», а слово tkinter со строчной «т».

- Создай новый файл, напиши исходный код и запусти программу.

Как насчет приятного приветствия, как мы это делали в главе 1? Воспользуемся компонентом, называемым `Label`.

- Обнови исходный код, а затем перезапусти программу (\Rightarrow `window1.py`):

```
import tkinter
Window = tkinter.Tk()
Display = tkinter.Label(Window, text="Привет!")
Display.pack()
```

Функция `Display` создает объект `Label`, первый параметр указывает на окно, в котором должно отображаться содержимое `Label`, а второй представляет собой текст, который надо отобразить. Все это «обернуто», так сказать, в метод `pack`. Без этой инструкции ты ничего не увидишь в окне (рис. 7.2).

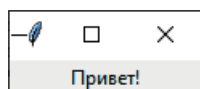


Рис. 7.2. Приветствие в окне

Окно стало еще более маленьким, чем предыдущее. Это связано с тем, что размер окна адаптируется под размер компонентов. Позже мы рассмотрим, как можем влиять на размер окна.

Теперь я хотел бы усложнить программу, чтобы за «приветствием» последовал вопрос «как дела?», а также добавить две кнопки `Button`, чтобы окно выглядело следующим образом:

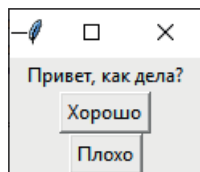


Рис. 7.3. Усложненная программа

Выглядит (по-прежнему) не очень красиво, но ключевое слово `pack` в `tkinter` гарантирует, что окно занимает столько места, сколько необходимо. Кроме того, все компоненты расположены один за другим. Мы изменим это позже.

7

Ниже представлен исходный код этой версии программы (⇒ *window2.py*):

```
from tkinter import *
Window = Tk()
Display = Label(Window, text="Привет, как дела?")
Display.pack()
Button1 = Button(Window, text="Хорошо")
Button2 = Button(Window, text="Плохо")
Button1.pack()
Button2.pack()
Window.mainloop()
```

- Введи код и запусти программу. Получилось ли у тебя окно, которое ты видел на рисунке выше?

Здесь добавлены два экземпляра нового для тебя компонента *Button*, созданных и вставленных так же, как объект *Label*.



Примечательно, что при использовании *tkinter* параметры имеют одну «цель», которая характеризует их значение.

Обычная функция или метод в Python выглядит, к примеру, так:

```
print("Привет, как дела?")
```

Параметр в *tkinter* передается функции *print* следующим образом:

```
print(text="Привет, как дела?")
```

Порядок перечисления параметров соблюдать необязательно.

В исходном коде есть еще пара изменений: в верхней его части я изменил инструкцию импорта. Строка

```
from tkinter import *
```

позволяет избежать постоянного указывания модуля *tkinter* при создании объектов.

А внизу я добавил еще кое-что: *mainloop()* – это бесконечный цикл, который предоставляет модуль *tkinter*.



Почему именно такой цикл? Операционная система, такая как Windows, ориентирована на *события*. Это означает, что все, что происходит (будь то действия мыши, нажатия клавиш, открытие, закрытие или перемещение окон), считается событием (на англ. языке – **Event**).

Цикл `mainloop()` аккумулирует все события, что важно для работы окна `tkinter` и его компонентов. Цикл бесконечен, поскольку не является условным, и он не останавливается, пока программа не завершит работу.

Что же происходит?

После запуска программы ты быстро разочаруешься, когда поймешь, что нажатие кнопки не приводит к какому-либо результату. Было бы неплохо, если бы, как в наших предыдущих программах, появилось сообщение типа «Это радует!» или «Это огорчает!» – в зависимости от того, какую кнопку ты нажал.

Но как это сделать с двумя компонентами? Как ты видишь, они реагируют на щелчки мыши при нажатии кнопок. Нам нужны две функции или два метода, которые выполняются при таком щелчке мыши.

В первую очередь на ум приходит конструкция `if`:

```
if Button1.pressed :  
    print("Это радует!")  
if Button2.pressed :  
    print("Это огорчает!")
```

К сожалению, нет такого события, как `pressed`, но есть возможность связать кнопку с функцией в самом начале (разумеется, в одной строке):

```
Button1 = Button(Window, text="Хорошо",  
command = buttonClick1())  
Button2 = Button(Window, text="Плохо",  
command = buttonClick2())
```

Параметр `command` связывает кнопку с функцией, выполняющейся, как только с кнопкой «что-то происходит», в частности ее нажатие. Поэтому нам нужно запрограммировать две функции:

```
def button1Click() :  
    print("Это радует!")  
def button2Click() :  
    print("Это огорчает!")
```

Как только событие инициируется щелчком мыши по одной из кнопок, выполняется функция, связанная с помощью ключевого слова `command` (рис. 7.4).

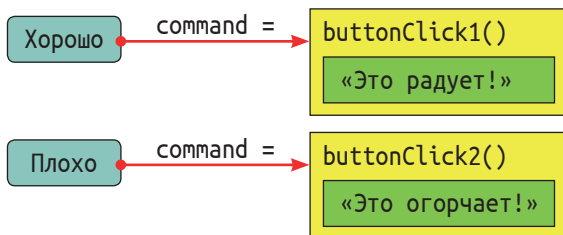


Рис. 7.4. Схема работы кода программы

Давай соберем все это вместе. Ниже представлен полный код программы (\Rightarrow `window3.py`).

```

# Окно с событиями
from tkinter import *

# Функция для события
def button1Click() :
    print("Это радует!")
def button2Click() :
    print("Это огорчает!")

# Основная программа
Window = Tk()
Display = Label(Window, text="Привет, как дела?")
Display.pack()
Button1 = Button(Window, text="Хорошо", command=button1Click)
Button2 = Button(Window, text="Плохо", command=button2Click)
Button1.pack()
Button2.pack()
Window.mainloop()

```

- Измени исходный код, помня, что оба события должны быть в одной строке со словом `command`.
- Запусти программу и нажми по очереди каждую кнопку (если хочешь, нажми несколько раз) (рис. 7.5).

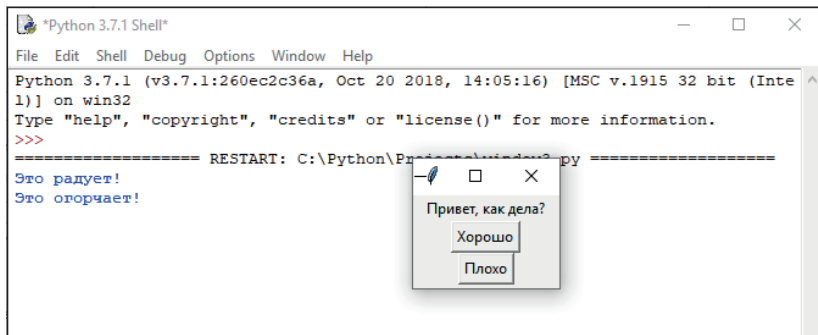


Рис. 7.5. Результат работы программы

Результат отобразится в окне среды Python. А я бы предпочел видеть все в нашем окне tkinter. Давай посмотрим, как это сделать. Например, мы могли бы взять компонент Label с текстом «Привет, как дела?». Как только нажимается кнопка, содержимое метки Label можно заменить на другой подходящий текст. Так мы получим следующую версию программы (\Rightarrow *window4.py*):

```
# Окно с событиями
from tkinter import *

# Функция для события
def button1Click() :
    Display.config(text="Это радует!")
def button2Click() :
    Display.config(text="Это огорчает!")

# Основная программа
Window = Tk()
Display = Label(Window, text="Привет, как дела?")
Display.pack()
Button1 = Button(Window, text="Хорошо", command=button1Click)
Button2 = Button(Window, text="Плохо", command=button2Click)
Button1.pack()
Button2.pack()
Window.mainloop()
```

Метод `config()` может использоваться для изменения свойств компонента, например текста, отображаемого меткой Label:

```
Display.config(text="Это радует!")
Display.config(text="Это огорчает!")
```

- Обнови код программы и запусти ее. Несколько раз нажми каждую кнопку (рис. 7.6).

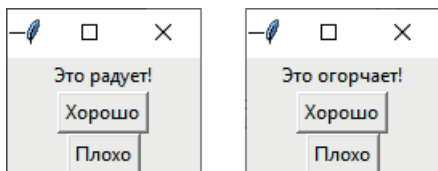


Рис. 7.6. Доработанная версия программы

Разметка интерфейса программы

Все неплохо, но и не хорошо. Мне не нравится расположение компонентов, они слишком сильно сжаты в окне. Мы должны сейчас это проработать. Причина в том, что разметка интерфейса контролируется методом `pack()`. Модуль `tkinter` предлагает три разные разметки интерфейса:

<code>pack()</code>	Компоненты автоматически размещаются как можно более компактно
<code>place()</code>	Компоненты могут быть расположены точно, их размер можно настроить
<code>grid()</code>	Компоненты распределяются по невидимой сетке в окне

Вариант `pack()` – наиболее удобное решение, но, к сожалению, не самое красивое. Ты также можешь передать параметры этому методу, чтобы изменить разметку. К примеру, так можно разместить кнопки далеко друг от друга, а не рядом:

```
Button1.pack(side="left")
Button2.pack(side="right")
```

Теперь окно выглядит так, как показано на рис. 7.7.

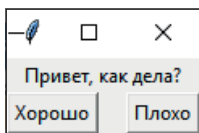


Рис. 7.7. Кнопки выровнены по левому и правому краям

Неплохое улучшение, но не слишком значимое. Посмотрим, как работает метод `place()`. В этом случае мы должны указывать определенные числовые значения. Попробуй прямо сейчас с новой версией программы (\Rightarrow `window5.py`).

```
from tkinter import *

def button1Click() :
    Display.config(text="Это радует!")
def button2Click() :
    Display.config(text="Это огорчает!")

Window = Tk()
```

```
Window.config(width=260, height=120)
Display = Label(Window, text="Привет, как дела?")
Display.place(x=50, y=20, width=160, height=30)
Button1 = Button(Window, text="Хорошо", command=button1Click)
Button2 = Button(Window, text="Плохо", command=button2Click)
Button1.place(x=20, y=70, width=100, height=30)
Button2.place(x=140, y=70, width=100, height=30)
Window.mainloop()
```

- Если ты введешь это и запустишь программу, результат будет выглядеть так, как показано на рис. 7.8.

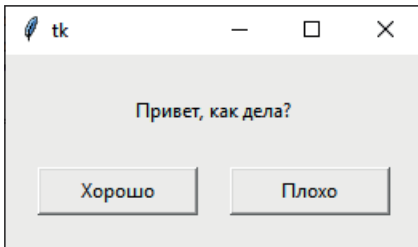


Рис. 7.8. Дизайн программы стал намного лучше

Совершенно точно лучше, чем предыдущие варианты. Давайте подробнее рассмотрим изменения. Во-первых, я установил размер окна с помощью инструкции config:

```
Window.config(width=260, height=120)
```

С помощью модуля tkinter размеры окна можно определить двумя способами:

```
Window.config(width=400, height=300)
```

или

```
Window.config(height=300, width=400)
```

Такой подход недоступен в классической версии Python.



После создания компонентов и кнопок в интерфейсе я использую три инструкции place для определения местоположения и размера элементов:

```
Display.place(x=50, y=20, width=160, height=30)
Button1.place(x=20, y=70, width=100, height=30)
Button2.place(x=140, y=70, width=100, height=30)
```

7



С помощью значений переменных *x* и *y* я определяю позицию верхнего левого угла каждого компонента, а остальные параметры настраивают их ширину и высоту.

Отличный пример, как в tkinter используются параметры. Выглядит необычно, так как порядок произволен, а в классическом коде Python должен соблюдаться определенный порядок. Таким образом, ты можешь оформлять функцию `place()` как угодно, например так:

```
Button1.place(height=30, width=100, x=20, y=70)
```

Честно говоря, аналогичного результата с помощью метода `grid` достичь сложнее. Взгляни на исходный код (\Rightarrow `window6.py`):

```
Window = Tk()
Display = Label(Window, text="Привет, как дела?")
Display.grid(row=0, column=1)
Button1 = Button(Window, text="Хорошо", command=button1Click)
Button2 = Button(Window, text="Плохо", command=button2Click)
Button1.grid(row=2, column=0, padx=10, pady=10)
Button2.grid(row=2, column=2, padx=10, pady=10)
Window.mainloop()
```

На область окна накладывается невидимая сетка, ячейки которой необязательно должны быть одного размера (рис. 7.9).

row=0		Привет, как дела?	
row=1	Хорошо		Плохо
	column=0	column=1	column=2

Рис. 7.9. Компоновка на основе сетки

С помощью параметра `row` ты определяешь строки, а с помощью `column` – столбцы (нумерация начинается с 0). С помощью параметров `padx` и `pady` определяется, какое расстояние будет слева направо (*x*) и сверху вниз (*y*).



Существует еще параметр `sticky`: в случае его использования компонент может быть выровнен по указанному краю ячейки: например, `sticky=E` выравнивает кнопку в крайнем правом углу

ячейки, `sticky=W` – в крайнем левом углу. Используются первые буквы четырех направлений света (на английском языке, т. е. W (запад), N (север), E (восток) и S (юг)).

Кроме того, с помощью кода

```
Display.grid(row=0, column=0, columnspan=2)
```

один компонент может занимать две ячейки.



- Для закрепления материала поэкспериментируй со значениями, чтобы понять тонкости разметки интерфейса.

Диалоговые окна и заголовки

Я также хотел бы рассмотреть в этой главе еще один способ отображения ответов программы. Для этого строки `import` недостаточно, нам нужно отдельно импортировать класс, который мы будем использовать:

```
from tkinter import *  
from tkinter import messagebox
```

Под *диалоговым окном* понимается небольшое окно, в котором отображаются сообщения. Также это может быть диалоговое окно предупреждения или ошибки и, конечно же, содержащее сообщение об успешном результате.

В определении двух функций для событий я теперь заменяю код для отображения текста в виде сообщения, а не метки (\Rightarrow *window7.py*):

```
def button1Click() :  
    messagebox.showinfo("Ответ", "Это радует!")  
def button2Click() :  
    messagebox.showinfo("Ответ", "Это огорчает!")
```

Метод `showinfo()` позволяет вывести на экран указанный текст. Первый параметр определяет строку заголовка, а второй – текст в диалоговом окне.

- Если ты поместишь этот код в свою программу и запустишь ее, то увидишь сообщение в дополнительном окне, которое закрывается нажатием кнопки **ОК** (рис. 7.10).

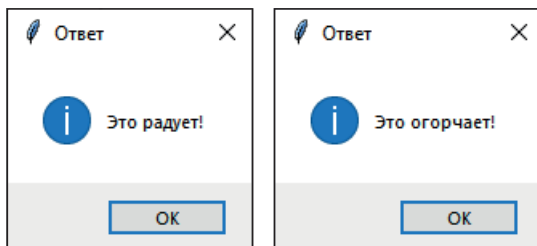


Рис. 7.10. Дополнительные диалоговые окна

В этом случае следует также рассказать, как задать главному окну заголовок (который до сих пор содержал буквы «tk»). Все, что тебе нужно сделать, – это добавить следующую строку кода:

```
Window.title("Привет")
Window.config(width=260, height=120)
```

Я также указал здесь строку, в которой определены размеры окна. Если ты не определишь их, окно может быть слишком маленькое, и ты увидишь лишь часть заголовка. Для длинного заголовка окно должно быть шире (рис. 7.11).

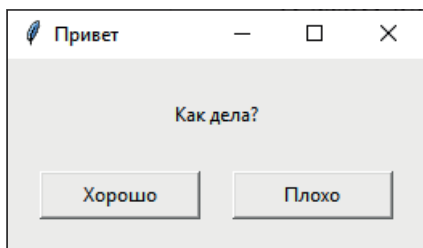


Рис. 7.11. Окно с заголовком

Поскольку слово «Привет» теперь находится в строке заголовка, высоту окна можно уменьшить.

А теперь с классами

Освоив классы в предыдущей главе, ты, возможно, задался вопросом: а где классы в этой главе? Возможно, ты даже захочешь узнать, как код нашей программы-приветствия будет выглядеть в случае применения классов. Ниже представлен полный исходный код такого варианта (\Rightarrow `window8.py`):

```
# Разметка окна
from tkinter import *

class Dialog() :

    # Инициализация
    def __init__(self, Title) :
        self.Window = Tk()
        self.Window.title(Title)
        self.Window.config(width=260, height=120)
        self.Display = Label(self.Window, text="Как дела?")
        self.Display.place(x=50, y=20, width=160, height=30)
        self.Button1 = Button(self.Window, text="Хорошо", \
            command=self.button1Click)
        self.Button2 = Button(self.Window, text="Плохо", \
            command=self.button2Click)
        self.Button1.place(x=20, y=70, width=100, height=30)
        self.Button2.place(x=140, y=70, width=100, height=30)
        self.Window.mainloop()

    # Метод
    def button1Click(self) :
        self.Display.config(text="Это радует!")
    def button2Click(self) :
        self.Display.config(text="Это огорчает!")

# Основная программа
Window = Dialog("Привет")
```

- Напиши весь этот код и протестируй программу, происходит ли все то же самое, что и при запуске предыдущих версий приложения (вновь обрати внимание, что однострочный код в этой книге записан в две строки).

Я снова переключился на вывод текста в виде метки. В этом коде также много раз используется ключевое слово `self`. В противном случае все, что мы запрограммировали ранее, фактически будет относиться к классу `Dialog`.

Метод `__init__()` определяет окно, ниже которого указываются методы событий. Основная программа получилась очень короткой, поэтому мы можем не разбивать ее код на два файла. Размеется, метод `init` можно использовать так, чтобы применять больше параметров, а не только заголовок окна:

```
def __init__(self, Title, Text, But1, But2) :
```

Что привело бы к следующему вызову в основной программе:

```
Window = Dialog("Привет", "Как дела?", "Хорошо", "Плохо")
```

Подведение итогов

Сейчас мы снова сделаем перерыв, прежде чем продолжить работу с модулем tkinter. На данный момент ты многое узнал. Подведем итоги этой главы:

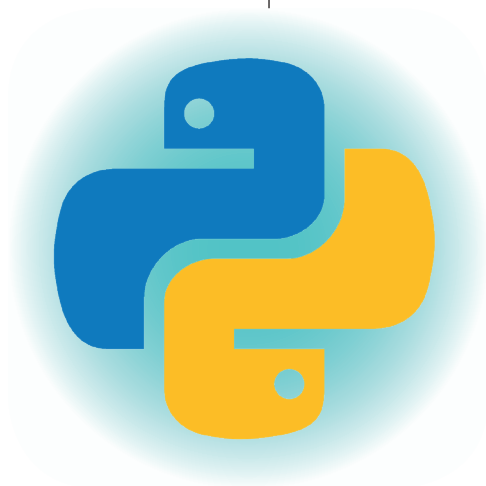
tkinter	Модуль для графики, окон и компонентов
Tk	Класс главного окна
mainloop()	(Бесконечный) цикл событий
Label	Метка (текстовая)
Button	Кнопка
MessageBox	Окно сообщения
showinfo()	Отображение окна сообщений
title()	Строка заголовка главного окна
config()	Настройка компонента
pack()	Упаковка компонента
Place()	Точное размещение, определение размера компонента
grid()	Расположение компонентов на сетке
Command=	Ссылка на функцию/метод события
height=	Высота компонента
width=	Ширина компонента
column=, row=	Количество столбцов и строк сетки
padx=, pady=	Расстояние компонентов от краев сетки
side=	Расположение компонента в окне (слева или справа)
sticky=	Расположение компонента на краю (к основным направлениям)

Несколько вопросов...

1. Какие компоненты модуля tkinter ты знаешь?
2. Как связаны компоненты и события?

...и одна задача

1. Создай программу-гороскоп, в которой есть кнопка для каждого знака зодиака и которая выводит короткое сообщение (например, месяц знака зодиака) после щелчка мышью.



8

Библиотека компонентов

Теперь ты уже знаком с некоторыми важными компонентами, но `tkinter` предлагает гораздо больше. На самом деле ты мог бы использовать еще несколько элементов в окне. Программа-приветствие из прошлой главы предполагает еще некоторые возможности, которые ты скоро увидишь.

Итак, в этой главе ты узнаешь:

- ⦿ как использовать списки;
- ⦿ о разнице между переключателями и флажками;
- ⦿ что такое лямбда-выражения;
- ⦿ еще о разметке интерфейса;
- ⦿ о преимуществах использования рамок.

Череда кнопок

Давай еще раз взглянем на программу из предыдущей главы. Возможны два ответа на вопрос: «Как дела?» Не так ли? Теперь мы должны мужественно расширить наш проект до шести кнопок. В табл. 8.1 ты найдешь мои варианты надписей на кнопках и соответствующие варианты ответа:

Таблица 8.1. Варианты кнопок и ответов для программы

Кнопка	Результат
Супер	Это здорово!
Хорошо	Это радует!
Так себе	Все возможно.
Плохо	Это огорчает!
Ужасно	Это плохо!
Не скажу	Раз ты так думаешь...

Таким образом программа сможет отреагировать на большинство ответов пользователя. Порадоваться за него или посочувствовать.

Придется довольно много набрать кода – фактически код программы меняется лишь в своем объеме.



Поскольку в исходном коде, за исключением экранного текста, много повторений, для ускорения работы ты можешь использовать команды **Copy** (Копировать) и **Insert** (Вставить) в меню **Edit** (Правка). Или применять в Windows горячие сочетания клавиш: **Ctrl+C** и **Ctrl+V**.

Понадобится изменить только несколько чисел и текст.

- Расширь свой (уже имеющийся) проект путем ввода следующего длинного кода. Убедись, что метка, кнопки и текст сообщения совпадают (\Rightarrow *hello3.py*):

```
from tkinter import *

# Функция для события
def button1Click() :
    Display.config(text="Это здорово!")
def button2Click() :
    Display.config(text="Это радует!")
def button3Click() :
    Display.config(text="Все возможно.")
def button4Click() :
    Display.config(text="Это огорчает!")
def button5Click() :
    Display.config(text="Это плохо!")
def button6Click() :
    Display.config(text="Раз ты так думаешь ...")

# Основная программа
Window = Tk()
```

```

Window.title("Привет!")
Window.config(width=300, height=190)
Display = Label(Window, text="Как это сделать?")
Display.place(x=80, y=10, width=160, height=30)
# Текст на кнопках
Button1 = Button(Window, text="Супер", command=button1Click)
Button2 = Button(Window, text="Хорошо", command=button2Click)
Button3 = Button(Window, text="Так себе", command=button3Click)
Button4 = Button(Window, text="Плохо", command=button4Click)
Button5 = Button(Window, text="Ужасно", command=button5Click)
Button6 = Button(Window, text="Не скажу", command=button6Click)
# Позиционирование кнопок
Button1.place(x=20, y=60, width=120, height=30)
Button2.place(x=160, y=60, width=120, height=30)
Button3.place(x=20, y=100, width=120, height=30)
Button4.place(x=160, y=100, width=120, height=30)
Button5.place(x=20, y=140, width=120, height=30)
Button6.place(x=160, y=140, width=120, height=30)
# Цикл событий
Window.mainloop()

```

В первую очередь обрати внимание, что определены шесть функций, обрабатывающих события, затем (в основной программе) в окне генерируется в общей сложности семь компонентов (1 метка, 6 кнопок). Кроме того, кнопки расположены парами.

➤ Теперь запусти программу и протестируй каждую кнопку (рис. 8.1).

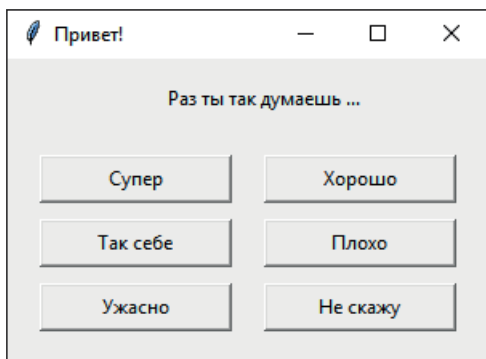


Рис. 8.1. Программа с шестью вариантами ответов

Выглядит довольно громоздко. Но как быть, если понадобится значительно больше шести кнопок: явно возникнет хаос в исходном коде. Разве нет способа упростить это?

8

Кнопки и ответы

Ранее у нас было что-то подобное в игре-лотерее, где мы все включили в список. Сработает ли способ с компонентами? Попробуем, но давай сначала объявим пустой список.

```
Кнопка = []
```

Я не использую в коде слово `Button`, потому что это имя класса. В цикле списка ты можешь теперь генерировать шесть таких «кнопок»:

```
for Nr in range(0,6) :  
    Кнопка.append(Button(Window, text=Answer[Nr]))
```

После создания каждая кнопка должна быть добавлена в список с помощью функции `append()`. Я просто добавил еще один список, который нам нужно сгенерировать в начале программы:

```
Answer = ["Супер", "Хорошо", "Так себе", "Плохо", "Ужасно", \n          "Не скажу"]
```

В другом цикле `for` мы организуем шесть кнопок (обе инструкции должны быть однострочными):

```
for pos in range(0,3) :  
    Кнопка[pos].place(x=20, y=60+pos*40, width=120, height=30)  
    Кнопка[pos+3].place(x=160, y=60+pos*40, width=120, height=30)
```

Это три строки, в каждой из которых есть пара кнопок. Значения по оси `x` фиксированы, значения по оси `y` изменяются в каждой строке. Поэтому счетчик `pos` учитывается, когда речь идет о вычислении позиции по оси `y`.

На данный момент я не включил функции события, потому что это не так просто. Ты можешь оставить все как есть, назначить функцию

```
Кнопка[0].config(command=buttonClick)
```

и продублировать ее шесть раз (с параметром от 0 до 5). Я поместил параметр `command` в метод `config`, что вполне себе работает. Но здесь я бы предпочел цикл.

Однако тогда нам сначала нужно сделать одну функцию для многих кнопок. Для этого нам нужен еще один список:

```
Diagnose = ["Это здорово!", "Это радует!", "Все возможно.", \
            "Это огорчает!", "Это плохо!", \
            "Раз ты так думаешь..."]
```

Здесь и в других похожих ситуациях ты можешь использовать символ `\`, чтобы разбить длинную инструкцию на две и более строк.



Функция события выглядит следующим образом:

```
def buttonClick(Nr) :
    Display.config(text=Diagnose[Nr])
```

И вот уже у нас возникла проблема. Сообщение об ошибке не появится, если мы расширим блок инструкций первого цикла `for` следующим образом:

```
Кнопка[Nr].config(command=buttonClick(Nr))
```

Однако начало программы вызывает разочарование: независимо от того, какую кнопку я нажимаю, не появляется ответ. На самом деле мы не должны передавать какие-либо параметры функции `buttonClick`, но компьютер не ругается, а лишь игнорирует параметр.

Как же использовать функцию `buttonClick()`, которая настолько красива, с короткими параметрами? Ключевое слово `lambda` поможет нам в этом. Используется оно так:

```
Кнопка[Nr].config(command=lambda: buttonClick(Nr))
```

Если ты применишь его в первом цикле `for`, исходный код будет выглядеть так:

```
for Nr in range(0,6) :
    Кнопка.append(Button(Window, text=Answer[Nr]))
    Кнопка[Nr].config(command=lambda: buttonClick(Nr))
```

Но даже это не приносит желаемого эффекта. Ты увидишь текст, когда нажмешь кнопку, но только последний вариант ответа: «Раз ты так думаешь...».

Каждому параметру `command` нужен уникальный номер, поэтому у нас нет выбора, кроме как обойтись без цикла и назначить определенное число каждой из шести кнопок:

8

```
Кнопка[0].config(command=lambda: buttonClick(0))
Кнопка[1].config(command=lambda: buttonClick(1))
Кнопка[2].config(command=lambda: buttonClick(2))
Кнопка[3].config(command=lambda: buttonClick(3))
Кнопка[4].config(command=lambda: buttonClick(4))
Кнопка[5].config(command=lambda: buttonClick(5))
```

В конце концов, мы смогли избежать проблем. Теперь я приведу исходный код целиком (\Rightarrow *hello4.py*):

```
# Приветствие с кнопками
from tkinter import *

# Константы текста
Answer = ["Супер", "Хорошо", "Так себе", "Плохо", "Ужасно", \
          "Не скажу"]
Diagnose = ["Это здорово!", "Это радует!", "Все возможно.", \
            "Это огорчает!", "Это плохо!", \
            "Раз ты так думаешь..."]

# Функция события
def buttonClick(Nr) :
    Display.config(text=Diagnose[Nr])

# Основная программа
Window = Tk()
Window.title("Привет!")
Window.config(width=300, height=190)
Display = Label(Window, text="Как это сделать?")
Display.place(x=80, y=10, width=160, height=30)

# Кнопки
Кнопка = []
for Nr in range(0,6) :
    Кнопка.append(Button(Window, text=Answer[Nr]))
for pos in range(0,3) :
    Кнопка[pos].place(x=20, y=60+pos*40, width=120, height=30)
    Кнопка[pos+3].place(x=160, y=60+pos*40, width=120, height=30)

# Индивидуальная настройка событий
Кнопка[0].config(command=lambda: buttonClick(0))
Кнопка[1].config(command=lambda: buttonClick(1))
Кнопка[2].config(command=lambda: buttonClick(2))
Кнопка[3].config(command=lambda: buttonClick(3))
Кнопка[4].config(command=lambda: buttonClick(4))
Кнопка[5].config(command=lambda: buttonClick(5))

# Цикл событий
Window.mainloop()
```

- Лучше создай новый файл. Или сохрани изменения в файл под новым именем. Введи указанный исходный код и протестируй программу. Если хочешь, можешь попробовать реализовать такой функционал с помощью метода `config` в цикле.

Списки выбора

В поисках альтернативы кнопкам мы теперь немного покопаемся в коллекции компонентов модуля `tkinter`. Без кнопок наша программа может работать, если мы создадим список со всеми ответами, которые раньше были на кнопках. И тогда нам больше не понадобятся кнопки. Поэтому тебе нужно их удалить, либо ты можешь создать совершенно новый файл и скопировать туда необходимые строки исходного кода.

Теперь мы имеем дело только с одним компонентом, который нам нужно создать:

```
Box = Listbox(Window)
```

Разумеется, сейчас этот список пуст, поэтому мы заполним его текстом:

```
for Nr in range(0,6) :  
    Box.insert(Nr, Answer[Nr])
```

Метод `insert()` принимает в качестве параметра сначала номер записи, затем текст ответа. Наконец, мы помещаем блок компонента и указываем его размеры:

```
Box.place(x=30,y=50, width=200, height=150)
```

(Тебе может понадобиться настроить высоту и ширину, особенно если ты используешь другие, более длинные строки ответов.)

Как связать список с функцией события? Здесь речь идет не о нажатии на кнопку, а о том, какой элемент был выбран. Поэтому мы используем совершенно другой метод:

```
Box.bind("<<ListboxSelect>>", listboxSelect)
```

Инструкция `bind()` привязывает событие к функции (или методу). Первый параметр определяет событие, которое не

8

только заключается в кавычки, но и оборачивается в двойные угловые скобки.

Я переименовал функцию события: `listboxSelect()` (имя `button-Click()` больше не подходит). И вот как выглядит эта функция сейчас:

```
def listBoxSelect(event) :  
    Select = Box.curselection()  
    Nr = Select[0]  
    Display.config(text=Diagnose[Nr])
```

Здесь необходим параметр, который принимает событие (английское слово `event`). Сначала функция определяет, какая запись была выбрана (отмечена или нажата):

```
Select = Box.curselection()
```

Результат – это не число, а выражение, из которого число сначала должно быть «выделено». Мы делаем это следующим образом:

```
Nr = Select[0]
```

Теперь можно отобразить соответствующий текст ответа. Давай рассмотрим исходный код целиком (\Rightarrow *hello5.py*):

```
from tkinter import *  
  
# Константы текста  
Answer = ["Супер", "Хорошо", "Так себе", "Плохо", "Ужасно", \  
          "Не скажу"]  
Diagnose = ["Это здорово!", "Это радует!", "Все возможно.", \  
            "Это огорчает!", "Это плохо!", \  
            "Раз ты так думаешь..."]  
  
# Функция события  
def listBoxSelect(event) :  
    Select = Box.curselection()  
    Nr = Select[0]  
    Display.config(text=Diagnose[Nr])  
  
# Основная программа  
Window = Tk()  
Window.title("Привет!")  
Window.config(width=260, height=230)  
Display = Label(Window, text="Как это сделать?")
```

```
Display.place(x=20, y=10, width=160, height=30)
```

```
# Список
Box = Listbox(Window)
for Nr in range(0,6) :
    Box.insert(Nr, Answer[Nr])
Box.bind("<<ListboxSelect>>", listboxSelect)
Box.place(x=30,y=50, width=200, height=150)

# Цикл событий
Window.mainloop()
```

- Введи этот исходный код и запусти программу (рис. 8.2).
Выбери каждую запись один раз.

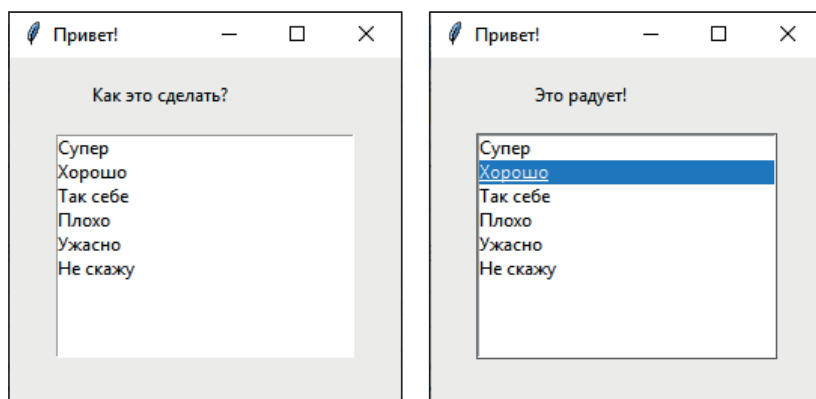


Рис. 8.2. Результат работы программы со списком

О переключателях...

Существует еще один способ визуализировать твоё настроение. Для этого ты можешь взять свой старый проект, работающий с помощью кнопок. В принципе, компоненты могут быть изменены непосредственно там.

Изучаемый компонент носит имя `Radiobutton` и называется *переключателем*. Такой объект содержит текст, отображаемый справа от кружочка. Если в кружочке установлена точка – переключатель считается установленным.

Как и в случае с кнопками, нам нужны шесть компонентов, которые мы создаем в виде пустого списка:

```
Option = []
```

8

Затем мы создаем шесть элементов, список параметров которых будет немного отличаться от кнопок:

```
for Nr in range(0,6) :  
    Option.append(Radiobutton(Window, variable=Number, value=Nr, \  
        text=Answer[Nr]))
```

Здесь создается позиция переключателя с определенным текстом. Пока это похоже на работу с кнопками. Но есть два новых параметра:

- `variable` – вставляет дополнительную переменную, которая делит переключатель с другими позициями. Они образуют группу, в которой можно активировать только один параметр. У нас это номер выбранного элемента;
- `value` получает последовательный номер соответствующей позиции в группе.

Теперь число не является обычной численной переменной так, как это было ранее в классическом Python. Модуль `tkinter` использует свой собственный тип, который описывается следующим образом:

```
Number = IntVar()  
Number.set(-1)
```

Этой переменной присваивается значение с помощью метода `set()`. Это не может быть число из списка с номерами позиций (от 0 до 5). В противном случае соответствующий компонент будет считаться активированным и переключатель будет установлен в соответствующую позицию при запуске программы. Тем не менее должен отмечаться только тот вариант, который мы выбираем.



Функция `IntVar()` так же, как, например, `StringVar()`, – это собственное творение модуля `tkinter` для работы с целыми числами и строками. Это методы, которые создают «обычную» переменную, нужную для работы программы. Важно, чтобы такая переменная не создавалась, пока не появится окно как экземпляр `Tk()`!

Функция события не нуждается в параметре, потому что значение переменной `Number` глобально. Однако ты получишь ее значение только с помощью метода `get()`:

```
def buttonClick() :  
    Display.config(text=Diagnose[Number.get()])
```

Используя дополнительные переменные, многое можно упростить, как показывает следующий полный листинг (\Rightarrow *hello6.py*):

```
from tkinter import *  
  
# Константы текста  
Answer = ["Супер", "Хорошо", "Так себе", "Плохо", "Ужасно", \  
          "Не скажу"]  
Diagnose = ["Это здорово!", "Это радует!", "Все возможно.", \  
            "Это огорчает!", "Это плохо!", \  
            "Раз ты так думаешь..."]  
  
# Функция события  
def buttonClick() :  
    Display.config(text=Diagnose[Number.get()])  
  
# Основная программа  
Window = Tk()  
Window.title("Привет!")  
Window.minsize(width=260, height=260)  
Display = Label(Window, text="Как это сделать?")  
Display.pack()  
  
# Вспомогательная переменная  
Number = IntVar()  
Number.set(-1)  
  
# Опции  
Option = []  
for Nr in range(0,6) :  
    Option.append(Radiobutton(Window, variable=Number, value=Nr, \  
                              text=Answer[Nr]))  
    Option[Nr].config(command=buttonClick)  
    Option[Nr].pack(anchor="w")  
  
# Цикл событий  
Window.mainloop()
```

На этот раз я выбрал вариант `pack()`, который показался мне проще. Тем не менее пришлось помешать окну стать слишком маленьким. Вот почему я определил минимально допустимые размеры и использовал метод `minsized()` вместо `config()`:

```
Window.minsize(width=260, height=260)
```

Кроме того, нам нужен параметр `anchor="w"`, позволяющий убедиться, что кружочки позиций переключателя выровнены должным образом. (Вариант `anchor="e"` тоже подойдет.)

- Введи показанный код и запусти программу. Щелкни по каждому положению переключателя один раз.

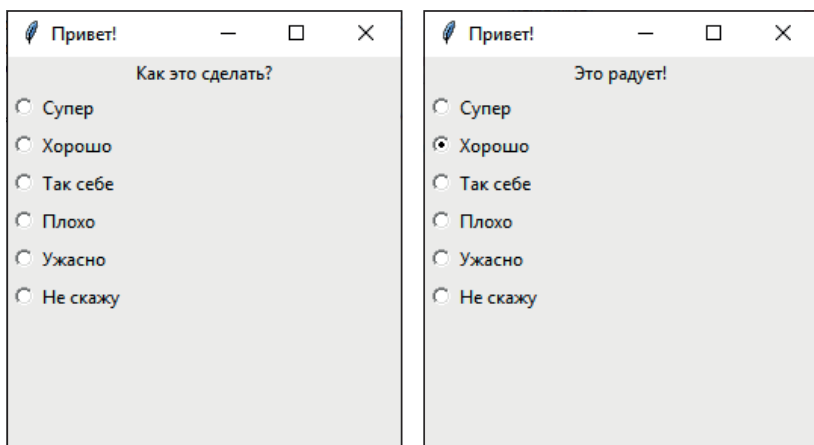


Рис. 8.3. Результат работы версии программы с переключателем

...и флажках

Современная медицина уже признала, что помимо физического есть еще моральные и духовные аспекты состояния человека. Поэтому давай применим новейшие продвижения в науке в нашем текущем проекте.

Для этого мы используем новый объект типа `Checkbox`, также называемый *флажком*. Этот компонент отображает квадратик слева от текста. Если в нем установлен флажок (галочка), запись считается отмеченной (рис. 8.4).

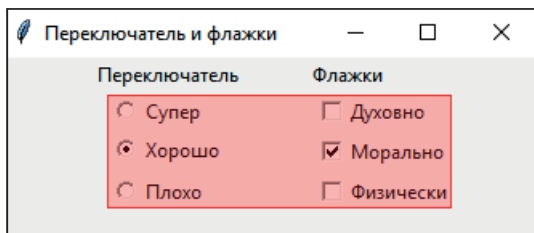


Рис. 8.4. Сгруппированные переключатель и флажки

Чтобы охватить все аспекты состояния человека, мы дополним нашу программу тремя флажками:

```
Choice = []  
for Nr in range(0,3) :  
    Choice.append(Checkbutton(Window, variable=Number, \  
        text=State[Nr]))
```

Нам также нужен еще один список строк:

```
State = ["Духовно", "Морально", "Физически"]
```

Похоже на переключатели, но здесь нам нужен лишь параметр `variable`, и мы можем избавиться от переменной `value`. Однако мы не можем использовать одну и ту же переменную здесь, если она уже используется в коде переключателей. По этой причине я ее переименовал и кое-что дополнил:

```
ChkNum = [0,0,0]  
OptNum = IntVar()  
OptNum.set(-1)
```

В то время как элемент теперь назван `OptNum`, у меня есть переменная `ChkNum`, которая сразу становится списком из трех элементов. Почему? В то время как переключатель может быть установлен только в одну позицию, флажков можно установить любое количество (даже все). Поэтому каждый элемент `Checkbutton` нуждается в собственной переменной. Затем мы создаем их в цикле `for`:

```
for Nr in range(0,3) :  
    ChkNum[Nr] = IntVar()  
    Choice.append(Checkbutton(Window, variable=ChkNum[Nr], \  
        text= State [Nr]))
```

Теперь что-то должно произойти, если установить флажок. Например, соответствующая «строка ответа» появится в строке заголовка окна.

Для этого нам нужен другой метод событий, который я назову `checkClick()`:

```
def checkClick() :  
    Text = "Привет! "  
    for Nr in range (0,3) :  
        if ChkNum[Nr].get() == 1 :  
            Text = Text + State[Nr] + " ";  
    Window.title(Text)
```

8

Сначала мы создаем строку с содержимым «Привет!», которое изначально присутствовало в заголовке окна. Затем эта строка проверяется в цикле, в котором установлен флажок. Так происходит, если связанный элемент `ChkNum` имеет значение 1:

```
if ChkNum[Nr].get() == 1 :
```

Если это так, то добавляется соответствующий текст («Духовно», «Морально», «Физически») (с пробелами между ними):

```
Text = Text + State[Nr] + " " ;
```

Наконец, у нас есть полноценный заголовок окна:

```
Window.title(Text)
```

➤ Теперь у нас есть все необходимое, чтобы собрать исходный код программы воедино (\Rightarrow *hello7.py*):

```
from tkinter import *

# Константы текста
Answer = ["Супер", "Хорошо", "Так себе", "Плохо", "Ужасно", \
          "Не скажу"]
State = ["Духовно", "Морально", "Физически"]
Diagnose = ["Это здорово!", "Это радует!", "Все возможно.", \
            "Это огорчает!", "Это плохо!", \
            "Раз ты так думаешь..."]

# Функция для события
def buttonClick() :
    Display.config(text=Diagnose[OptNum.get()])
def checkClick() :
    Text = "Привет! "
    for Nr in range (0,3) :
        if ChkNum[Nr].get() == 1 :
            Text = Text + State[Nr] + " " ;
    Window.title(Text)

# Основная программа
Window = Tk()
Window.title("Привет!")
Window.minsize(width=420, height=260)
Display = Label(Window, text="Как дела?")
Display.grid(row=0, column=1)
```

```
# Вспомогательные переменные
ChkNum = [0,0,0]
OptNum = IntVar()
OptNum.set(-1)

# Опции
Option = []
for Nr in range(0,6) :
    Option.append(Radiobutton(Window, variable=OptNum, value=Nr, \
        text=Answer[Nr]))
    Option[Nr].config(command=buttonClick)
    Option[Nr].grid(row=Nr+1, column=0, sticky="w")

# Управление
Choice = []
for Nr in range(0,3) :
    ChkNum[Nr] = IntVar()
    Choice.append(Checkbutton(Window, variable=ChkNum[Nr], \
        text= State [Nr]))
    Choice[Nr].config(command=checkClick)
    Choice[Nr].grid(row=Nr+1, column=2, sticky="w")

# Цикл событий
Window.mainloop()
```

Опять же, я немного экспериментировал с дизайном, а затем решил использовать для разметки сетку. Сначала поле отображается в первой строке, но во втором столбце:

```
Display.grid(row=0, column=1)
```

Я расположил позиции переключателя в первом столбце, строка меняется в соответствии с нумерацией отдельных элементов:

```
Option[Nr].grid(row=Nr+1, column=0, sticky="w")
```

Флажки попадают в столбец 3:

```
Choice[Nr].grid(row=Nr+1, column=2, sticky="w")
```

Ключевое слово `sticky` гарантирует, что все пункты выравниваются по левому краю.

- Запусти программу и протестируй все переключатели и галочки (рис. 8.5).

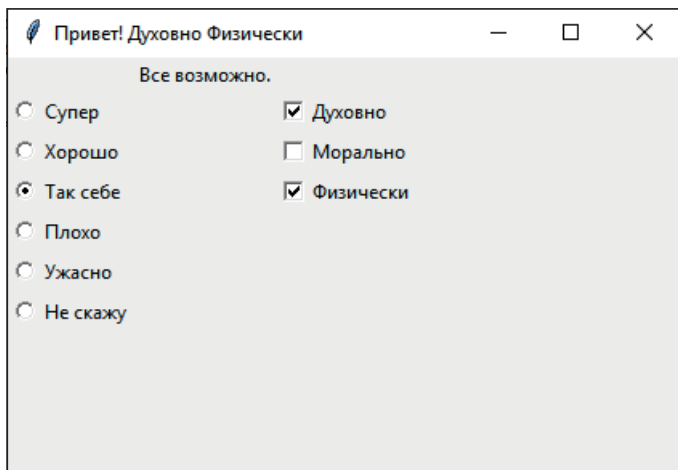


Рис. 8.5. Версия программы с флажками и переключателем

Декорирование приложения

Все эти пункты с возможностью выбора с их точками и флажками кажутся какими-то блеклыми. Посмотрим, удастся ли нам немного доработать внешний вид программы. Для этого я использую два других компонента того же типа, с которыми я хотел бы визуально совместить две группы элементов управления – переключателя и флажков:

```
Links = Frame(Window, borderwidth=2, relief="sunken")
Links.place(x=20, y=50, width=180, height=220)
Rechts = Frame(Window, borderwidth=2, relief="raised")
Rechts.place(x=220, y=50, width=180, height=110)
```

Frame – это тип компонента, который прежде всего предназначен для генерации красивой рамки. С помощью параметра `relief` я могу влиять на то, как выглядит внутренняя поверхность этого компонента, например утопленная или выступающая. А параметр `borderwidth` определяет толщину границы. Кроме того, с помощью функции `place()` можно тщательно позиционировать элементы управления.

Окно будет состоять лишь из двух новых компонентов, показанных на рис. 8.6.

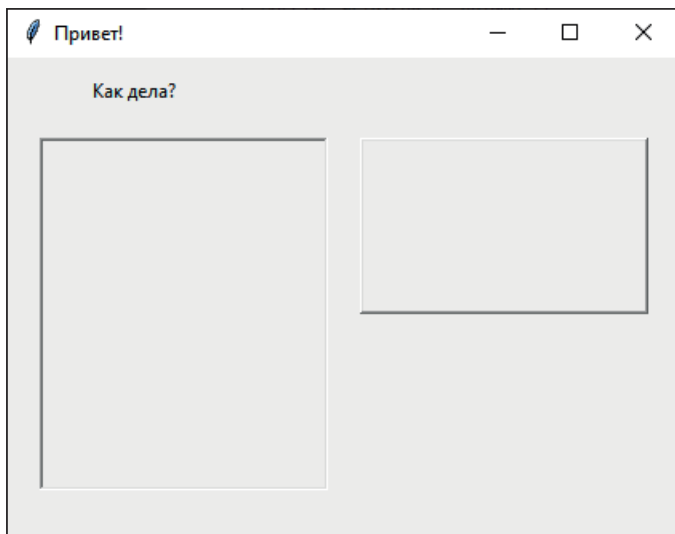


Рис. 8.6. Примеры рамок – утопленной и выступающей

Теперь мы должны убедиться, что позиции переключателя и флажки попадают в нужные рамки. Поэтому мы сделаем Radiobuttons и Checkbuttons элементами рамки:

```
Option.append(Radiobutton(Links, variable=OptNum, value=Nr, \
                          text=Answer[Nr]))
...
Choice.append(Checkbutton(Rechts, variable=ChkNum[Nr], \
                          text= State [Nr]))
```

И с этим мы приходим к полному исходному коду (\Rightarrow *hel-lo8.py*):

```
from tkinter import *

# Константы текста
Answer = ["Супер", "Хорошо", "Так себе", "Плохо", "Ужасно", \
          "Не скажу"]
State = ["Духовно", "Морально", "Физически"]
Diagnose = ["Это здорово!", "Это радует!", "Все возможно.", \
            "Это огорчает!", "Это плохо!", \
            "Раз ты так думаешь..."]

# Функция для события
def buttonClick() :
    Display.config(text=Diagnose[OptNum.get()])
def checkClick() :
    Text = "Привет! "
```

8

```

for Nr in range(0,3) :
    if ChkNum[Nr].get() == 1 :
        Text = Text + State[Nr] + " ";
Window.title(Text)

# Основная программа
Window = Tk()
Window.title("Привет!")
Window.minsize(width=420, height=300)
Display = Label(Window, text="Как дела?")
Display.place(x=50, y=10)
Links = Frame(Window, borderwidth=2, relief="sunken")
Links.place(x=20, y=50, width=180, height=220)
Rechts = Frame(Window, borderwidth=2, relief="raised")
Rechts.place(x=220, y=50, width=180, height=110)

# Вспомогательные переменные
ChkNum = [0,0,0]
OptNum = IntVar()
OptNum.set(-1)

# Опции
Option = []
for Nr in range(0,6) :
    Option.append(Radiobutton(Links, variable=OptNum, value=Nr, \
        text=Answer[Nr]))
    Option[Nr].config(command=buttonClick)
    Option[Nr].grid(row=Nr+1, column=0, sticky="w")

# Управление
Choice = []
for Nr in range(0,3) :
    ChkNum[Nr] = IntVar()
    Choice.append(Checkbutton(Rechts, variable=ChkNum[Nr], \
        text= State [Nr]))
    Choice[Nr].config(command=checkClick)
    Choice[Nr].grid(row=Nr+1, column=2, sticky="w")

# Цикл событий
Window.mainloop()

```

- Измени исходный код, сохрани файл и запусти программу. Результат должен выглядеть примерно так, как показано на рис. 8.7.

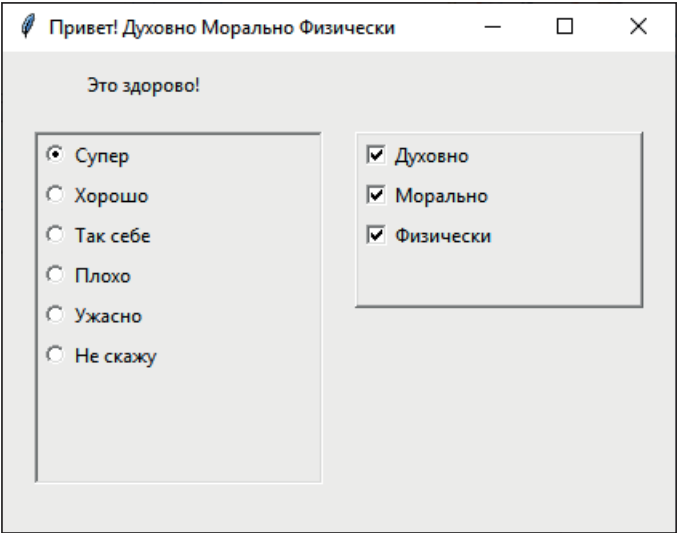


Рис. 8.7. Финальная версия программы с рамками

Подведение итогов

Данный раздел завершает эту главу. Потребовалось немало усилий, чтобы выполнить примеры из нее, в том числе и потому, что каждый компонент имеет свои особенности, да и визуальное приукрашивание интерфейса занимает время. Вот что ты узнал:

Frame	Рамка (может вмещать другие компоненты)
Listbox	Список (многострочное поле)
Radiobutton	Переключатель (можно установить только в одну позицию)
Checkbox	Флажок (можно установить сколько угодно, хоть все или ни одного)
minsize()	Определяет минимальный размер окна
append()	Добавляет элемент в список
insert()	Вставляет элемент в список
IntVar()	Создает переменную типа tkinter (целочисленную)
set()	Присваивает значение переменной tkinter
get()	Извлекает значение переменной tkinter
bind()	Связывает событие с функцией
lambda=	Вызов функции события с параметром
curselection()	Текущая выбранная запись в списке
<<ListboxSelect>>	Событие выбора в списке
variable=	Переменная для номера или состояния в переключателе или флажке

8

value=	Номер пункта в группе
anchor=	Отвечает за «привязку» компонента слева или справа (pack)
sticky()	Отвечает за «привязку» компонента слева или справа (grid)
relief=	Создает рамку утопленную или выступающую

Несколько вопросов...

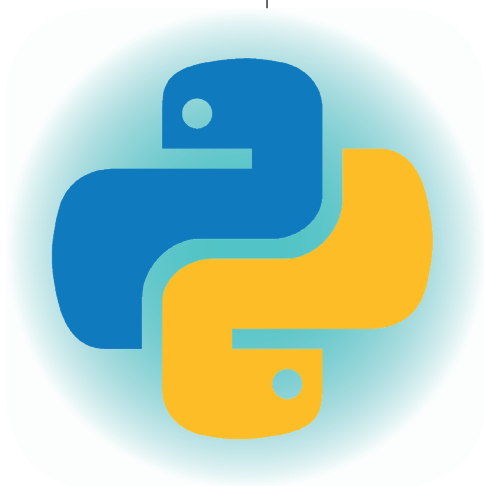
1. Как определить в списке, какой пункт выбран?
2. Как проверить, установлен ли флажок или переключатель?

...и задач

1. В проекте программы-гороскопа из главы 7 (*horoscope1.py*) замени отдельные кнопки списком кнопок.
2. В этом же проекте используй компонент `Listbox`.
3. Примени переключатель в том же проекте вместо кнопок.

9

Домашний психолог



Действительно ужасно: все эти стрессы в школе и на работе. Вся эта суета. Только за компьютером ты находишь пару минут отдыха – пока программа или игра не запустится.

Как насчет терапии, которая вернет тебе равновесие и покой? Но зачем тратить уйму денег на психолога? Для чего ты на самом деле учишься программировать? Почему бы тебе не создать своего собственного терапевта? Он даже сможет записывать ваши разговоры в качестве бонуса.

Итак, в этой главе ты узнаешь:

- ⊙ какие бывают поля ввода;
- ⊙ подробнее о списках;
- ⊙ про ползунковые регуляторы;
- ⊙ как загрузить и сохранить текстовые данные.

Пошаговая разработка программы-психолога

Предыдущая программа с приветствием вдохновила меня написать новое приложение, которое я хотел бы скромно назвать «домашним психологом». Первые варианты программы-приветствия были не более чем приятным корот-

ким разговором. Это то, что ты получаешь бесплатно в разговоре с соседями или продавщицей в магазине за углом. Наш новый проект должен предложить гораздо больше диагностических средств. Для этого он должен сначала занять «лицо» (рис. 9.1).

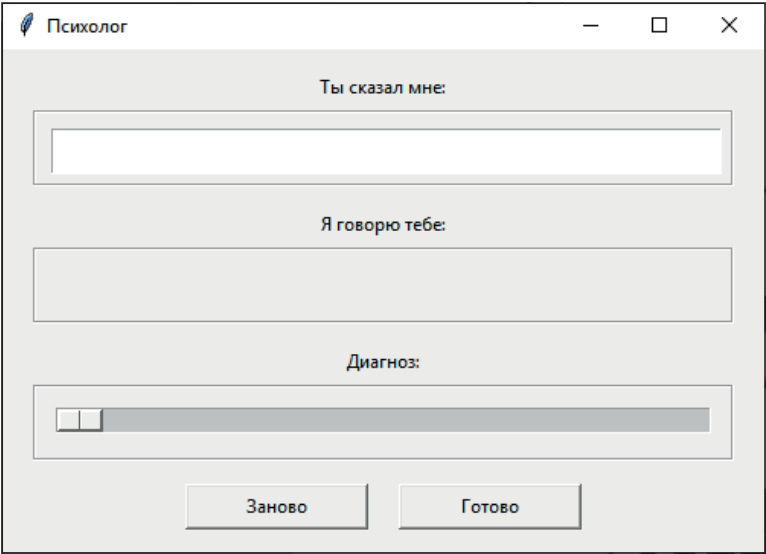


Рис. 9.1. Интерфейс программы-психолога

Потребуется много компонентов, и необходимо увеличить окно. Давай посмотрим, что нам нужно в качестве строительных материалов (табл. 9.1).

Таблица 9.1. Компоненты для программы-психолога

Компоненты	Текст
Вверху (Frame, Label, Entry)	Ты сказал мне:
В центре (Frame, Label, Label)	Я говорю тебе:
Внизу (Frame, Label, Scale)	Диагноз:
Button1	Заново
Button2	Готово
Заголовок	Психолог

В таблице есть несколько незнакомых тебе элементов, которые мы добавим позже. Я начну с трех компонентов, которые мы создадим и разместим следующим образом:

```
Border = []
for pos in range(0,3) :
```

```
Border.append(Frame(Window, borderwidth=2, relief="groove"))
Border[pos].place(x=20, y=40+pos*90, width=460, height=50)
```

Здесь я выбрал новое значение параметра `relief`: `groove` выделяет рамку рельефной линией. Это лучше, чем значение `sunken` или `raised`.

Для каждого элемента управления используется метка, с помощью которой указывается, для чего он нужен. Мы также упакуем их в конструкцию `for`, как показано ниже:

```
Display = []
for pos in range(0,3) :
    Display.append(Label(Window, text=State[pos]))
    Display[pos].place(x=20, y=10+pos*90, width=460, height=30)
```

Разумеется, список с соответствующими строками текста тоже должен быть заранее согласован:

```
State = ["Ты сказал мне:", "Я говорю тебе:", "Диагноз:"]
```

У нас уже есть структура приложения (рис. 9.2).

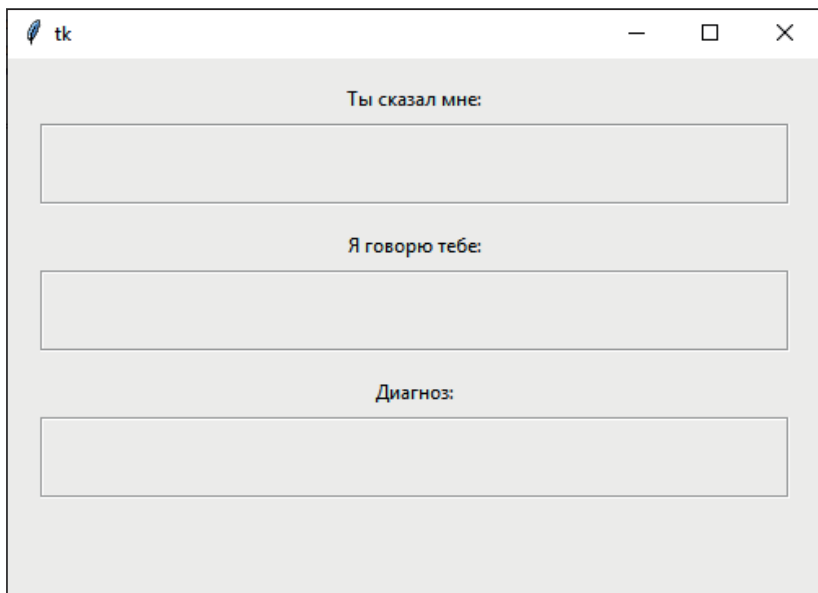


Рис. 9.2. Структура программы готова

Как все это будет работать? В верхней части окна ты можешь ввести любой текст. Для этого нам нужно поле вво-

9

да (компонент, который ты еще не знаешь). В центре, под строкой «Я говорю тебе:», должен выводиться ответ на твой вопрос. Все, что тебе нужно, – это еще одно поле вывода.

Внизу есть возможность управлять ответом. «Диагноз:» – не что иное, как ползунковый регулятор, который меняет положение каждый раз, когда в центре появляется другой ответ. Мы доберемся до него позже.

Сначала мы возьмем строки с ответами, которые ты уже использовал в предыдущей программе-приветствии. Позже разберемся, как сгенерировать больше вариантов для ответа. Теперь давай создадим остальные компоненты.

В нижней части окна программы находятся кнопки, которые также связаны с функцией события:

```
Button1 = Button(Window, text="Заново", command=button1Click)
Button1.place(x=120, y=285, width=120, height=30)
Button2 = Button(Window, text="Готово", command=button2Click)
Button2.place(x=260, y=285, width=120, height=30)
```

Ты нажимаешь кнопку «Готово», если хочешь завершить беседу, тогда ответ должен появиться в нижнем поле. Нажатие кнопки «Заново» гарантирует, что все поля будут очищены и доступны для нового ввода/вывода.

Перейдем к последним трем компонентам, два из которых тебе незнакомы. Начнем с блока «Ты сказал мне:»:

```
Input = Entry(Border[0])
Input.place(x=10, y=10, width=440, height=30)
```

Entry – это поле, в которое можно ввести что-то, текст или число. Это поле ввода относится не к главному окну, а к верхней рамке (фрейму). Соответственно, позиционирование выполняется относительно области рамки (а не всего окна программы).

Далее размещается метка, ее ты уже знаешь. Она относится к центральной рамке (фрейму):

```
Answer = Label(Border[1])
Answer.place(x=10, y=10, width=440, height=30)
```

Еще один незнакомый тебе тип – Scale – используется для изменения значения, он также называется *ползунковым регулятором*.

Его можно расположить по горизонтали или вертикали. За расположение отвечает параметр `orient`:

```
Slider = Scale(Border[2], orient="horizontal")
Slider.config(length=430, showvalue=0)
Slider.pack(pady=10)
```

Также мы настраиваем ползунковый регулятор, определяя его длину в соответствии с окружающей рамкой. (Примечание: не работает с параметром `place`.) Параметр `showvalue=0` предотвращает отображение значения при перемещении ползункового регулятора.

В конце мы используем `place()`, следя за тем, чтобы ползунковый регулятор был хорошо центрирован в пределах рамки.

Приступим к диагностике?

Давай посмотрим весь код программы целиком (\Rightarrow *psych1.py*):

```
# Психолог
from tkinter import *

# Константы текста
State = ["Ты сказал мне:", "Я говорю тебе:", "Диагноз:"]
Diagnose = ["Это здорово!", "Это радует!", "Все возможно.", \
            "Это огорчает!", "Это плохо!", \
            "Раз ты так думаешь..."]

# Функция для события
def button1Click() :
    pass
def button2Click() :
    pass

# Основная программа
Window = Tk()
Window.title("Психолог")
Window.minsize(width=500, height=330)

# Определение состояния
Display = []
Border = []
for pos in range(0,3) :
    Display.append(Label(Window, text=State[pos]))
    Display[pos].place(x=20, y=10+pos*90, width=460, height=30)
    Border.append(Frame(Window, borderwidth=2, relief="groove"))
    Border[pos].place(x=20, y=40+pos*90, width=460, height=50)

# Ввод, ответ и ползунковый регулятор
Input = Entry(Border[0])
Input.place(x=10, y=10, width=440, height=30)
```

```

Answer = Label(Border[1])
Answer.place(x=10, y=10, width=440, height=30)
Slider = Scale(Border[2], orient="horizontal")
Slider.config(length=430, showvalue=0)
Slider.pack(pady=10)

# Кнопки "Готово" и "Заново"
Button1 = Button(Window, text="Заново", command=button1Click)
Button1.place(x=120, y=285, width=120, height=30)
Button2 = Button(Window, text="Готово", command=button2Click)
Button2.place(x=260, y=285, width=120, height=30)

# Цикл событий
Window.mainloop()

```

- Создай новый файл и введи исходный код полностью. Или внеси в предыдущую программу приветствия все необходимые изменения. Затем запусти программу.

Ты можешь ввести любой текст в поле ввода, перетаскивать ползунковый регулятор и нажимать кнопки. Ничего не произойдет: психолог, похоже, не консультирует.

Неудивительно, потому что функция событий ни на что не способна – пока есть только один вариант передаваемого значения. Поэтому мы должны заняться решением проблемы. Начнем с функции `button2Click()`, потому что именно там происходит фактическая «консультация».

Что именно должно произойти? После того как я введу некий текст и нажму кнопку «Готово», должен появиться текст ответа. Лучше всего, если случайный. Для этого нам нужен модуль, который мы уже давно не использовали:

```
import random
```

Если ты вставил эту строку в верхней части исходного кода, функция события может быть задействована (\Rightarrow *psych2.py*):

```

def button2Click() :
    Nr = random.randint(0,9)
    Antwort.config(text=Diagnose[Nr])

```

Поскольку у меня есть десять записей в списке проблем, я использую код `randint(0,9)`; если ты используешь старые записи из программы-приветствия, ты должен изменить его на `randint(0,5)`. Также ты можешь изменить текст диагностики, например так:

```

Diagnose = ["Хм...", "Это важно!", \
            "Взгляни на это под другим углом!", \

```

```
"Ну что я могу сказать?", "На самом деле?", \
"Так, все понятно.", "У тебя явно проблемы.", \
"Как я тебя понимаю.", "У меня нет слов.", \
"Хорошо..."]
```

Если ты сейчас запустишь программу с этими изменениями, ты получишь ответ (который не всегда соответствует вопросу) на каждое предложение, сказанное тобой психологу. Теперь давай позаботимся о кнопке «Заново», позволяющей удалять введенный и показанный текст (\Rightarrow *psych2.py*):

```
def button1Click() :
    Input.delete(0,"end")
    Answer.config(text="")
```

Для компонента Answer достаточно указать параметр text со значением в кавычках "" (пустым), потому что пока не отображается какой-либо ответ. Для поля ввода нам нужен метод delete(), поскольку в качестве параметров указывается начало и конец строки символов, которые нужно удалить. ("end" задается для последнего символа в строке.)

- Добавь исходный код в свою программу, не забыв про определение двух методов событий (и, если хочешь, собственные диагнозы). Затем запусти программу, введи текст и нажми кнопку «Готово» (рис. 9.3). Перед следующим сеансом работы сначала нажми кнопку «Заново».

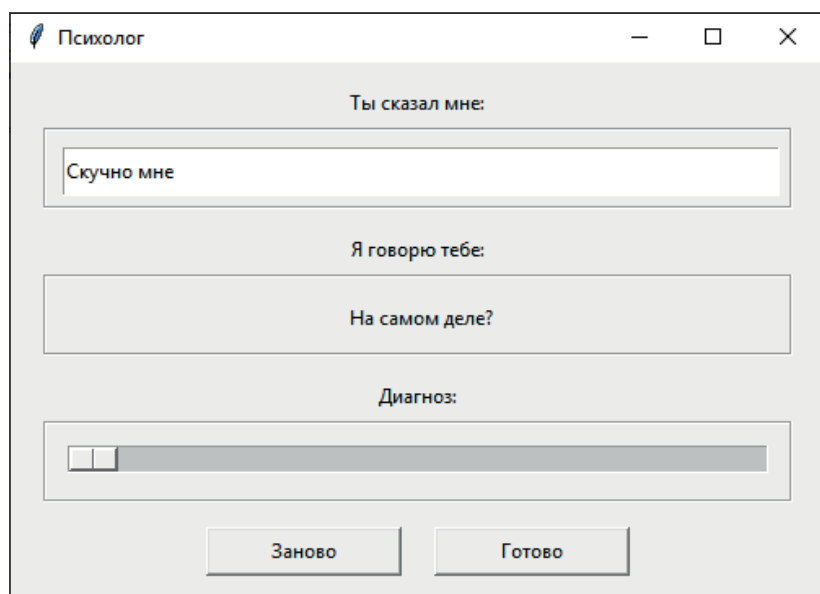


Рис. 9.3. Программа-психолог в работе

9

Теперь ты, вероятно, хочешь знать, что такое ползунковый регулятор. Чем на самом деле являются элементы управления «Диагноз»?

В зависимости от положения маленького ползункового регулятора в окне программы должен появиться соответствующий диагноз. Поскольку диагноз зависит от движения ползункового регулятора, здесь можно говорить о генерации. И поэтому ползунковый регулятор становится генератором диагнозов.

Как привести ползунковый регулятор в действие? Путем определения функции события, которая реагирует на его движение. Для этого нам нужно дополнить первые две строки кода компонента ползункового регулятора (\Rightarrow *psych3.py*).

```
Slider = Scale(Border[2], orient="horizontal", command=scaleValue)
Slider.config(from_=0, to=9, length=430, showvalue=0)
```

Первая строка реализует ссылку на функцию события *scaleValue()*, которую, конечно же, нам нужно определить. Во второй строке мы определяем диапазон перемещения, в моем случае, поскольку у меня десять строк текста, от 0 до 9.



Здесь имя *from_* не содержит опечатку: ключевое слово *from* уже есть в Python (см. строку *from Tkinter import **), поэтому в этом случае его можно использовать с символом подчеркивания.

Перейдем к функции *scaleValue()* (\Rightarrow *psych3.py*):

```
def scaleValue(event) :
    Nr = Slider.get()
    Answer.config(text=Diagnose[Nr])
```

Обрати внимание, что вновь в определении должен быть указан параметр *event*, но его не нужно использовать при назначении *command*. Во-первых, значение положения ползункового регулятора определяется с помощью метода *get*. Оно используется как номер показанного диагноза.

Измени исходный код своей программы, добавив приведенные выше строки кода ползункового регулятора и связанного с ним параметра *event*. Затем тщательно протестируй генератор диагнозов.

Работа с файлами

Если тебе больше не подходят тексты диагнозов, их можно легко изменить в любое время. Конечно, ты также можешь увеличить количество строк текстов.

Но со временем тебе понадобится более удобный способ добавления текста в программу-психолог.

Было бы круто написать диагнозы в текстовом редакторе и сохранить их в виде обычного текстового файла? Программа соберет все строки из этого файла и поместит их в свою диагностическую базу.

Для начала создай текстовый файл, например, с помощью программы Блокнот в операционной системе Windows. Если ты используешь мощный текстовый редактор, такой как Microsoft Word, ты должен сохранить файл как обычный текстовый файл (*diagnose.txt*). Заполни файл подходящими вариантами ответов, по одному в строку.

Ты можешь взять текстовый файл *diagnose.txt* из папки с примерами к этой книге и использовать его. Также ты можешь отредактировать его по своему усмотрению.



Как правило, мы не знаем, сколько строк находится в таком файле, поэтому подобный список удобен в качестве гибкого «контейнера данных». Давай для начала определим наш список диагнозов:

```
Diagnose = []
```

Теперь нам нужен файл, содержащий наши диагностические записи. Чтобы открыть и загрузить данные, нам нужен некий механизм их ввода по аналогии с информацией, вводимой с клавиатуры.

Тебе уже знаком обычный способ «ввода/вывода данных» с помощью клавиатуры и монитора. Для этого мы использовали функцию `input()` для ввода с клавиатуры и функцию `print()` для вывода на экран.

Теперь же мы имеем дело с «потокком данных»: хотя речь также идет о вводе и выводе, но уже с жесткого диска и других носителей.



9

Я думаю, что имеет смысл разместить весь код, касающийся работы с файлами, в отдельной функции, которую я определил следующим образом:

```
def loadDiagnose() :  
    global Max  
    Nr = 0  
    try :  
        File = open("diagnose.txt", "r")  
        for String in File :  
            Diagnose.append(String.rstrip())  
            Nr += 1  
        Max = Nr - 1  
        File.close()
```

Для успешной работы я определил глобальную переменную в верхней части исходного кода программы и присвоил ей значение 0:

```
Max = 0
```

Это приводит к изменению в двух местах кода программы, где раньше у меня было число 9 (ты увидишь изменения позже в целом листинге).

После того как мы определились с переменными Max и Nr в функции loadDiagnose, создадим объект файла:

```
File = open("diagnose.txt", "r")
```

Это делается с помощью функции open(), в которой параметры – это имя файла и режим. В данном случае используется режим «r» – это аббревиатура от английского слова «read», означающая, что файл должен быть прочитан.

Разумеется, указанный файл должен находиться в той же папке, что и программа. Либо ты можешь указать полный путь к файлу, например так:

```
File = open("d:\python\projects\diagnose.txt", "r")
```

или так:

```
File = open("d:/python/projects/diagnose.txt", "r")
```

Конечно, ты также можешь использовать другие имена папок и файлов. Важно правильно указать путь. Обрати внимание, что Python поддерживает как обратную косую черту (\), так и прямую (/). Только если файл находится в той же папке, что и проект, полное имя пути указывать не требуется.



Далее, текст из файла построчно считывается и вставляется в виде элементов в список диагнозов:

```
for String in File :  
    Diagnose.append(String.rstrip())  
    Nr += 1
```

На самом деле этого утверждения было бы достаточно:

```
Diagnose.append(String)
```

Дополнительный метод `rstrip()` гарантирует удаление лишних символов в конце сохраненной строки. Здесь это (невидимый) символ конца строки (если параметр не был передан в `rstrip()`). Если этого не сделать, каждый диагноз будет выводиться в две строки.

Чтобы сохранить текст с диагнозами в отдельных строках, редактор, в котором ты вводил текст, добавляет символ, который в Python можно симитировать с помощью последовательности `\n`.

Например, инструкция `print("Привет!\n")` возвращает еще одну (пустую) строку после слова «Привет!». Ты можешь не указывать метод `rstrip()`, но не удивляйся потом пустому пространству под текстом «Я говорю тебе».



Затем переменная `max` начинает «заполняться». Значение переменной на единицу меньше, так как отсчет начинается с 0:

```
Max = Nr - 1
```

И наконец, файл закрывается, как дверка шкафа, из которого ты что-то достал.

```
File.close()
```

9

Функция `close()` является противоположностью `open()`.

Есть еще кое-что, что мы не рассматривали здесь: как быть, если файл не существует? Или его невозможно прочитать? Нельзя ли установить механизм защиты, чтобы программа не выходила из-за этого из строя?

Для решения проблемы мы отступим от структуры, которую использовали долгое время. Взгляни на следующий код исключения (\Rightarrow *psych4.py*):

```
def loadDiagnose() :  
    global Max  
    Nr = 0  
    try :  
        File = open("diagnose.txt", "r")  
        for String in File :  
            Diagnose.append(String.rstrip())  
            Nr += 1  
        Max = Nr - 1  
        File.close()  
    except :  
        Diagnose.append("Что-то не так с файлом диагнозов")
```

Сначала функция пытается открыть файл и прочитать данные (`try`); если это не удастся (`except`), то она сообщает о проблеме с файлом.

```
except :  
    Diagnose.append("Что-то не так с файлом диагнозов")
```

Чтобы проверить работу программы в такой ситуации, ты должен на время переименовать файл (рис. 9.4).

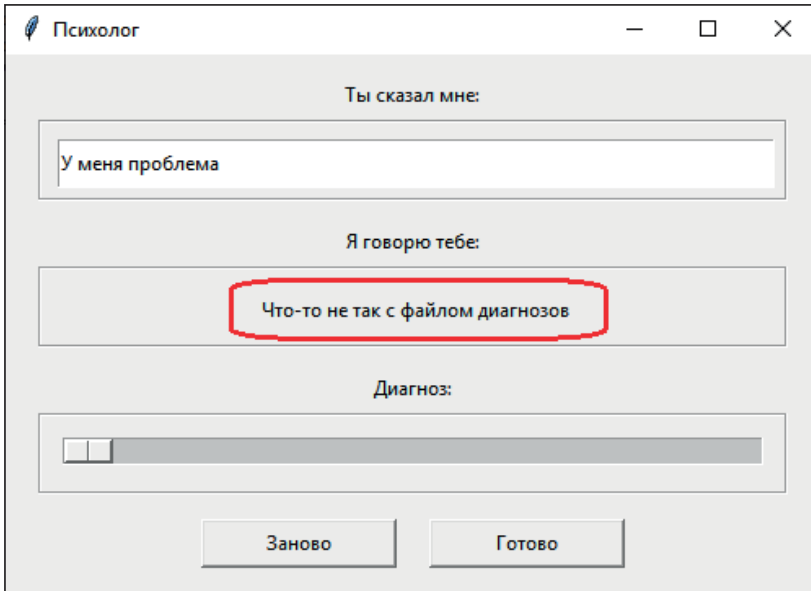


Рис. 9.4. Программа вызвала исключение

Все вместе

Давай перейдем к общему листингу, который немного увеличился (\Rightarrow *psych4.py*):

```
# Психолог
from tkinter import *
import random

# Константы текста
State = ["Ты сказал мне:", "Я говорю тебе:", "Диагноз:"]
Diagnose = []
Max = 0

# Файловая функция
def loadDiagnose() :
    global Max
    Nr = 0
    try :
        File = open("diagnose.txt", "r")
        for String in File :
            Diagnose.append(String.rstrip())
            Nr += 1
        Max = Nr - 1
        File.close()
    except :
```

```

        Diagnose.append("Что-то не так с файлом диагнозов")

# Функция для события
def button1Click() :
    Input.delete(0,"end")
    Answer.config(text="")
def button2Click() :
    Nr = random.randint(0,Max)
    Answer.config(text=Diagnose[Nr])
def scaleValue(event) :
    Nr = Slider.get()
    Answer.config(text=Diagnose[Nr])

# Основная программа
Window = Tk()
Window.title("Психолог")
Window.minsize(width=500, height=330)
loadDiagnose()

# Определение состояния
Display = []
Border = []
for pos in range(0,3) :
    Display.append(Label(Window, text=State[pos]))
    Display[pos].place(x=20, y=10+pos*90, width=460, height=30)
    Border.append(Frame(Window, borderwidth=2, relief="groove"))
    Border[pos].place(x=20, y=40+pos*90, width=460, height=50)

# Ввод, ответ и ползунковый регулятор
Input = Entry(Border[0])
Input.place(x=10, y=10, width=440, height=30)
Answer = Label(Border[1])
Answer.place(x=10, y=10, width=440, height=30)
Slider = Scale(Border[2], orient="horizontal", command=scaleValue)
Slider.config(from_=0, to=Max, length=430, showvalue=0)
Slider.pack(pady=10)

# Кнопки "Готово" и "Заново"
Button1 = Button(Window, text="Заново", command=button1Click)
Button1.place(x=120, y=285, width=120, height=30)
Button2 = Button(Window, text="Готово", command=button2Click)
Button2.place(x=260, y=285, width=120, height=30)

# Цикл событий
Window.mainloop()

```

Как видишь, существенные изменения претерпела только верхняя часть исходного кода.

- Измени код своей предыдущей версии программы и убедись, что у тебя есть текстовый файл с корректным

именем. Разумеется, ты можешь воспользоваться файлом с диагнозами из папки с примерами. Да начнется терапия!

Ты действительно не понял, как работает программа?

- ❖ В поле ввода ты можешь ввести все, что хочешь. Тебе не нужно подбирать слова, потому что здесь нет цензуры.
- ❖ Затем нажми кнопку «Готово» и получи ответную фразу в объявлении под строкой «Я говорю тебе», что, конечно же, не всегда соответствует введенному тобой тексту. Но может быть, ответ подкинет тебе какую-то идею?
- ❖ Нажми кнопку «Заново», чтобы удалить свой текст из поля ввода, после чего ты можешь ввести что-то новое. Это может быть комментарий к диагнозу или что-то еще.
- ❖ Если ответы программы кажутся тебе неуместными, перетаскивай ползунковый регулятор, чтобы найти более подходящий ответ.
- ❖ Если программа-психолог действует тебе на нервы, то ты знаешь, как завершить ее работу.



Журнал диагностики

Теперь ты узнаешь, как сохранить текст из программы в файл: мы немного расширим функционал нашего виртуального психолога. Для этого нам нужен еще один список:

```
Psychosis = []
```

Этот список должен собирать все, что ты скажешь психологу посредством ввода текста, а в конце сеанса терапии сохранить собранные данные в файл *psychox.txt*. Вместо символа *x* ты также можешь использовать число, если хочешь сохранить текст в нескольких файлах. (Если ты придумал лучшее имя своему файлу, используй его!)

Лучше всего собирать ответы вместе с вводимым тобой текстом. Но сначала мы должны создать и открыть соответствующий файл:

```
File = open("psychoX.txt", "w")
```

Здесь используется режим "w", который является аббревиатурой слова «write», запись. Файл создается в той же папке, где запускается текущая программа.

Затем используется цикл, в котором считывается каждая строка:

```
for String in Psychosis :
    File.write(String+"\n")
```

К каждой строке в файле добавляется последовательность `\n`, чтобы каждое предложение размещалось в отдельной строке. Если ты откроешь этот файл с помощью текстового редактора, то сможешь посмотреть его (и изменить, если нужно).



Подобно функции `write()`, существует также метод чтения строк. Нам они не нужны в функции `loadDiagnose()`, где строки берутся непосредственно из файла.

Можно также «обернуть объекты» функцией `read()`. Это громоздкий метод, но ты наверняка встретишь такой способ, и не раз, поэтому стоит о нем упомянуть. Выглядит он так:

```
String = file.read()
```

Затем содержимое переменной `String` должно стать элементом ответа.

Теперь функция `saveDialog()` выглядит следующим образом (\Rightarrow *psych5.py*):

```
def saveDiagnose() :
    try :
        File = open("psychoX.txt", "w")
        for String in Psychosis :
            File.write(String+"\n")
        File.close()
```

- Дополни свою программу этой функцией. Ее лучше всего вставить непосредственно под определение `loadDiagnose()`. Не забудь о диагнозах в виде списка.

Перейдем к функции события `button2Click()`. Вводимый текст и строки ответов должны вставляться в список диагнозов, а затем сохраняться. Так мы получаем дополненную версию программы (\Rightarrow *psych5.py*):

```
def button2Click() :
    Nr = random.randint(0,Max)
    Answer.config(text=Diagnose[Nr])
    Psychosis.append(Input.get())
    Psychosis.append(Diagnose[Nr])
    saveDiagnose()
```

Измени свою программу. Теперь ты можешь запустить ее, а затем посмотреть, что находится в файле *psychox.txt*.

Ты найдешь все файлы для программы-психолога в папке с примерами по адресу dmkpress.com.



Подведение итогов

Мы совершили большой прорыв! Из-за обилия компонентов ты можешь не разглядеть важную часть исходного кода. Давай соберем в таблице то, что упоминается в главе:

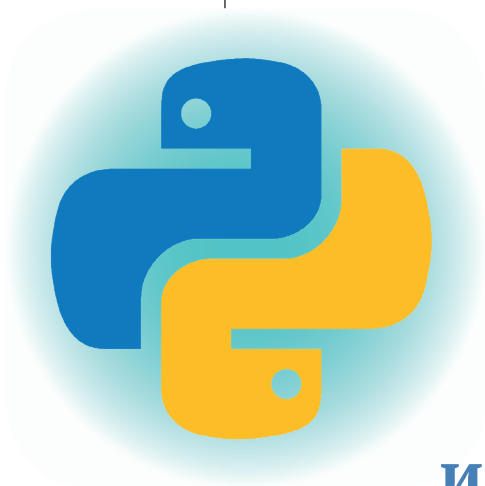
<code>open()</code>	Открывает файл для записи ("w") или чтения ("r")
<code>close()</code>	Закрывает файл
<code>read()</code>	Считывает строковые данные
<code>write()</code>	Записывает строковые данные
<code>rstrip()</code>	Обрезает строку по указанным символам (по умолчанию <code>\n</code> – символ конца строки)
<code>Entry</code>	Поле ввода (для текста и цифр)
<code>delete()</code>	Удаляет ввод
<code>Scale</code>	Ползунковый регулятор для числовых значений
<code>orient=</code>	Отвечает за расположение ползункового регулятора по горизонтали или вертикали
<code>showvalue=</code>	Отвечает за отображение значения ползункового регулятора: да (1) или нет (0)
<code>relief=</code>	Оформление компонента рамкой: например, плоской, утопленной, выступающей, рельефной

Несколько вопросов...

1. В чем разница между компонентами `Label` и `Entry`?
2. Как открыть и сохранить текстовые файлы с помощью программы?

...и задач

1. Расширь функцию `saveDiagnose()` с помощью конструкции `try-except`.
2. Создай программу, которая отображает значение от 0 до 100 %, изменяемое с помощью ползункового регулятора.



10

Меню и диалоговые окна

Возможность программы-психолога загружать и сохранять текст – это прекрасно. Но ты можешь открыть только конкретный файл и сохранить данные в определенный файл. Этого недостаточно.

В этой главе мы расширим программу, реализовав то, что есть у многих других приложений: меню. Помимо всего, ты сможешь выбирать, какой файл ты хочешь открыть или под каким именем должен быть сохранен полученный диагноз.

Итак, в этой главе ты узнаешь:

- ⊙ как добавить в программу меню;
- ⊙ как использовать диалоговые окна открытия и сохранения файлов;
- ⊙ кое-что о контекстных меню и всплывающих окнах;
- ⊙ подробнее о событиях;
- ⊙ как использовать сочетания клавиш;
- ⊙ как с удобством завершать работу программы.

Меню для программы-психолога

В первую очередь нам нужно меню, с помощью которого мы сможем открыть или сохранить файл.

Однако мы не сможем обойтись одним компонентом, так как меню включает в себя не только команды меню, но так-

же саму строку меню. Необходимым компонентом, как минимум, является Menu:

```
Menuitem = Menu(Window)
Window.config(menu=Menuitem)
Filemenu = Menu(Menuitem)
```

Но так ты ничего не увидишь, в строке меню должен появиться текст. Ты добавляешь его с помощью метода `add_cascade()`:

```
Menuitem.add_cascade(label="Файл", menu=Filemenu)
```

Параметр `menu` связывает пункт непосредственно с первым меню (с именем «Файл»).

Выполнив эти инструкции, ты уже можешь увидеть некое подобие меню, как показано на рис. 10.1.

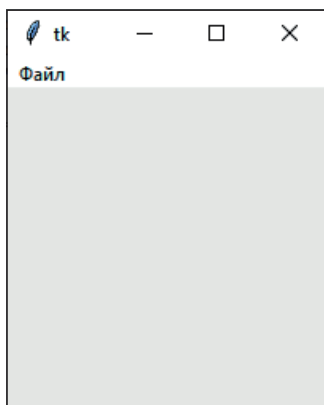


Рис. 10.1. Первое меню для программы

Прежде чем мы внесем все это в нашу программу, я хотел бы предложить тебе сделать небольшую дополнительную программу, в которой ты сможешь открыть меню и щелкнуть по нему:

```
from tkinter import *

Window = Tk()
Menuitem = Menu(Window)
Window.config(menu=Menuitem)

Filemenu = Menu(Menuitem)
Menuitem.add_cascade(label="Файл", menu=Filemenu)
```

```
Filemenu.add_command(label="Открыть")
Filemenu.add_command(label="Сохранить")
Filemenu.add_command(label="Выход")

Helpmenu = Menu(Menuitem)
Menuitem.add_cascade(label="Справка", menu=Helpmenu)
Helpmenu.add_command(label="О программе")

Window.mainloop()
```

Функция `add_cascade()` отвечает за пункты в строке меню, а `add_command()` компилирует команды в меню.

Первый запуск программы демонстрирует результат, показанный на рис. 10.2 (я открыл меню **Файл**).

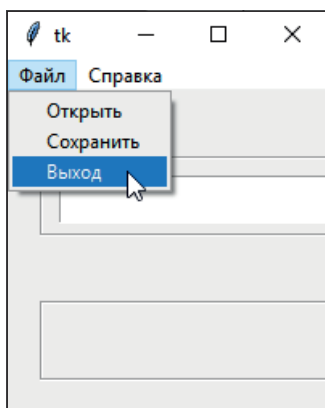


Рис. 10.2. Рабочее меню **Файл**

Теперь ты можешь, по желанию, открыть или сохранить любой файл. А выбрав команду **Выход**, можешь закрыть программу. Здесь также есть меню **Справка**. Конечно, ты можешь выбрать его, но не увидишь никакого эффекта. Это тот результат, которого мы хотели достичь: у нас теперь есть система меню, встроенная в программу.

Теперь, когда основная программа уже довольно сложная, лучше всего создать систему меню в виде отдельной функции. Вот моя первая попытка:

```
def initMenu() :
    Menuitem = Menu(Window)
    Window.config(menu=Menuitem)
    Filemenu = Menu(Menuitem, tearoff = 0)
    Menuitem.add_cascade(label="Файл", menu=Filemenu)
```

```
Filemenu.add_command(label="Открыть", command=openFile)
Filemenu.add_command(label="Сохранить", command=saveFile)
Filemenu.add_command(label="Выход", command=closeAll)
Helpmenu = Menu(Menuitem, tearoff = 0)
Menuitem.add_cascade(label="Справка", menu=Helpmenu)
Helpmenu.add_command(label="О программе")
```

Обрати внимание, два пункта строки меню имеют параметр `tearoff = 0`. Он удаляет довольно уродливую, с моей точки зрения, пунктирную линию в меню. Также я подготовил несколько методов `command` для команд в меню **Файл**, но пока они не работают.

Основной код программы требует внесения дополнительной строки (\Rightarrow *psych6.py*):

```
initMenu()
```

- Дополни код последней версии программы-психолога новой функцией и вызови ее, а затем запусти программу. Выбирай пункты меню и проверь, все ли команды работают (рис. 10.3).

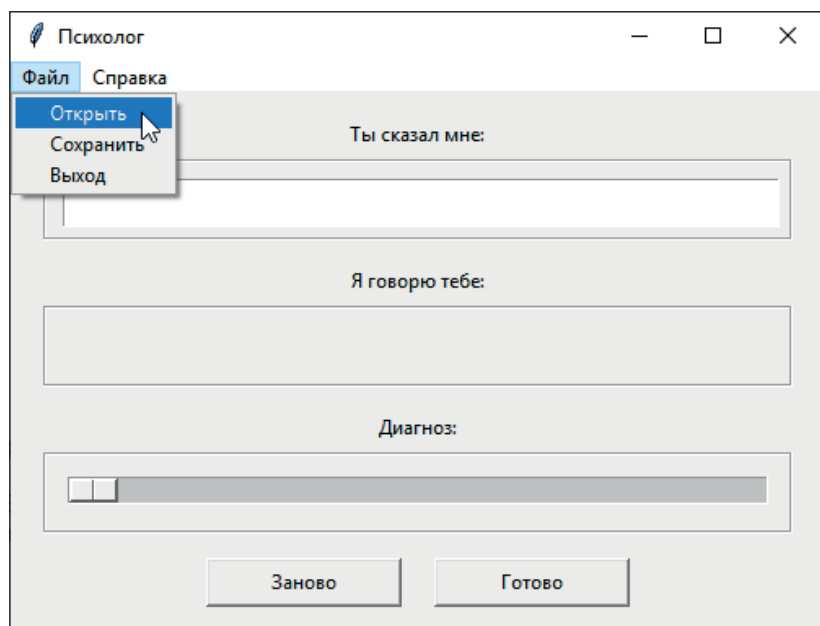


Рис. 10.3. Версия программы с меню

Два диалоговых окна

Теперь нам понадобятся дополнительные функции событий. Давай проработаем команды **Открыть** и **Сохранить**. Чего бы нам хотелось?

- Чтобы в открывшемся диалоговом окне мы могли выбрать текстовый файл. Затем этот файл можно было бы загрузить.
- Чтобы при сохранении можно было ввести имя в диалоговом окне. Затем файл должен быть сохранен.

Звучит как что-то сложное, но не в случае с библиотекой `tkinter`. У нее есть что предложить – модуль `filedialog`, который может выполнить часть работы за нас. `askopenfilename()` и `asksaveasfilename()` – еще два хороших вспомогательных метода.

Для этого сначала нужно добавить еще одну строку `import` в верхней части исходного кода:

```
from tkinter import filedialog
```

Даже если указать символ звездочки, с помощью которой импортируются все классы модуля, этого недостаточно, `filedialog` нужны «особенные привилегии».

Давай посмотрим, что происходит, когда ты выбираешь один из пунктов меню – **Открыть** или **Сохранить**. Ниже представлены две функции события (\Rightarrow *psych6.py*):

```
def openFile() :
    Name = filedialog.askopenfilename(filetypes= \
        (("Текстовые файлы", "*.txt"), ("Все файлы", "*.*")))
    if Name != "":
        Diagnose.clear()
        loadDiagnose(Name)
        Slider.config(to=Max)
def saveFile() :
    Name = filedialog.asksaveasfilename()
    if Name != "":
        saveDiagnose(Name)
```

Сначала функция `openFile()` открывает диалоговое окно, показанное на рис. 10.4, в котором ты можешь выбрать имя файла или ввести его. Это достигается с помощью метода `askopenfilename()`. В качестве параметров ты можешь указать определенные параметры по умолчанию.

С помощью параметра `filetypes` ты определяешь, файлы с какими расширениями будут отображаться. Я указал текстовые файлы (`txt`), но есть также вариант **Все файлы**.

Если имя выбрано, текущий диагноз удаляется, а затем загружается содержимое нового файла. Но если выбор отменен, `name!=""` и все прочее остается без изменений. Важно, чтобы после успешного завершения загрузки ползунковый регулятор подгонялся под количество строк текста в новом файле.

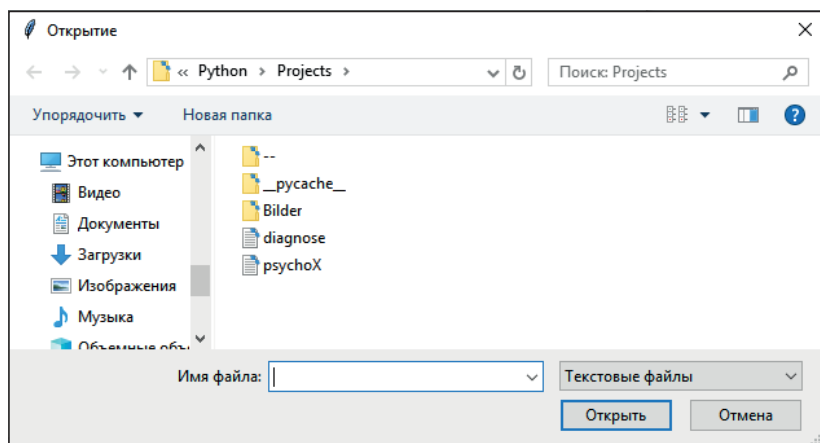


Рис. 10.4. Диалоговое окно открытия файла

При сохранении открывается диалоговое окно, в котором ты можешь указать имя, под которым хочешь сохранить файл. Это делается с помощью метода `requestsaveasfilename()`, тогда файл можно сохранить, только если имя указано.

Поскольку параметры файлов с диагнозами больше не определяются однозначно командами **Открыть** и **Сохранить**, две файловые функции теперь имеют соответствующий параметр (\Rightarrow `psych6.py`):

```
def loadDiagnose(Name) :
def saveDiagnose(Name) :
```

Внутри двух функций вызов метода `open()` должен быть настроен так, чтобы он принимал общее имя файла:

```
File = open(Name, "r")
File = open(Name, "w")
```

По желанию ты можешь оставить методы `loadDiagnose()` и `saveDiagnos()` в программе, но теперь добавь по одному параметру в каждую функцию:

```
loadDiagnose("diagnose.txt")
saveDiagnose("psychoX.txt")
```

Этот код загрузит стандартный файл в начале, затем нажми кнопку **Готово**, чтобы сохранить сеанс в файл с именем по умолчанию. Все как раньше. Но если ты воспользуешься меню, то сможешь обойти дефолтное поведение.

Последняя команда в меню **Файл** должна также иметь функцию события:

```
def closeAll() :
    Window.destroy()
```

Как только ты выбрал команду **Выход**, окно и его содержимое удаляются с помощью метода `trouthe()`, таким образом завершая работу tkinter-программы.

Полный исходный код

Теперь соберем полный исходный код новой версии программы-психолога. Внеси изменения в свой файл, а затем запусти программу (\Rightarrow *psych6.py*):

```
# Психолог
from tkinter import *
from tkinter import filedialog
import random

# Константы текста
State = ["Ты сказал мне:", "Я говорю тебе:", "Диагноз:"]
Diagnose = []
Psychosis = []
Max = 0

# Функции меню
def initMenu() :
    Menuitem = Menu(Window)
    Window.config(menu=Menuitem)
    Filemenu = Menu(Menuitem, tearoff = 0)
    Menuitem.add_cascade(label="Файл", menu=Filemenu)
    Filemenu.add_command(label="Открыть", command=openFile)
    Filemenu.add_command(label="Сохранить", command=saveFile)
```

```

Filemenu.add_command(label="Выход", command=closeAll)
Helpmenu = Menu(Menuitem, tearoff = 0)
Menuitem.add_cascade(label="Справка", menu=Helpmenu)
Helpmenu.add_command(label="О программе")

# Файловые функции
def loadDiagnose(Name) :
    global Max
    Nr = 0
    try :
        File = open(Name, "r")
        for String in File :
            Diagnose.append(String.rstrip())
            Nr += 1
        Max = Nr - 1
        File.close()
    except :
        Diagnose.append("Что-то не так с файлом диагнозов")
def saveDiagnose(Name) :
    try :
        File = open(Name, "w")
        for String in Psychosis :
            File.write(String+"\n")
        File.close()
    except :
        print("Не получается сохранить файл")

# Функция для события
def button1Click() :
    Input.delete(0,"end")
    Answer.config(text="")
def button2Click() :
    Nr = random.randint(0,Max)
    Answer.config(text=Diagnose[Nr])
    Psychosis.append(Input.get())
    Psychosis.append(Diagnose[Nr])
    saveDiagnose("psychoX.txt")
def scaleValue(event) :
    Nr = Slider.get()
    Answer.config(text=Diagnose[Nr])

# Команды меню
def openFile() :
    Name = filedialog.askopenfilename(filetypes= \
        (("Текстовые файлы", "*.txt"),("Все файлы", "*.*")))
    if Name != "":
        Diagnose.clear()
        loadDiagnose(Name)
        Slider.config(to=Max)
def saveFile() :
    Name = filedialog.asksaveasfilename()

```

```
if Name != "":
    saveDiagnose(Name)
def closeAll() :
    Window.destroy()

# Основная программа
Window = Tk()
Window.title("Психолог")
Window.minsize(width=500, height=330)
initMenu()
loadDiagnose("diagnose.txt")

# Определение состояния
Display = []
Border = []
for pos in range(0,3) :
    Display.append(Label(Window, text=State[pos]))
    Display[pos].place(x=20, y=10+pos*90, width=460, height=30)
    Border.append(Frame(Window, borderwidth=2, relief="groove"))
    Border[pos].place(x=20, y=40+pos*90, width=460, height=50)

# Ввод, ответ и ползунковый регулятор
Input = Entry(Border[0])
Input.place(x=10, y=10, width=440, height=30)
Answer = Label(Border[1])
Answer.place(x=10, y=10, width=440, height=30)
Slider = Scale(Border[2], orient="horizontal", command=scaleValue)
Slider.config(from_=0, to=Max, length=430, showvalue=0)
Slider.pack(pady=10)

# Кнопки "Готово" и "Заново"
Button1 = Button(Window, text="Заново", command=button1Click)
Button1.place(x=120, y=285, width=120, height=30)
Button2 = Button(Window, text="Готово", command=button2Click)
Button2.place(x=260, y=285, width=120, height=30)

# Цикл событий
Window.mainloop()
```

Теперь ты можешь присваивать файлам любые имена, а также создать целую коллекцию сеансов.

Контекстные меню и всплывающие окна

Всегда что-то можно улучшить. Ты уже мог заметить, что многие разработчики программного обеспечения выпускают обновления для своих продуктов через определенные

промежутки времени. Иногда, к сожалению, ухудшая таким образом продукт.

Мы скромно добавим новую версию меню в наш проект. Ты знаешь из операционной системы Windows, что если щелкнуть правой кнопкой мыши, всплывет контекстное меню, которое не показывается в строке меню. Никто не возражает, если мы сделаем в новой версии программы-психолога такую же «штучку»?

Прежде всего подобное меню похоже на обычное меню (\Rightarrow *psych7.py*):

```
def initPopup() :
    global ContextMenu
    ContextMenu = Menu(Window, tearoff = 0)
    ContextMenu.add_command(label="Открыть", command=openFile)
    ContextMenu.add_command(label="Сохранить", command=saveFile)
    ContextMenu.add_command(label="Выход", command=closeAll)
```

Только в программе оно не отображается в виде строки меню и не зафиксировано в конкретной позиции окна. Почему же ContextMenu является глобальным? Потому что оно нам необходимо в другой функции.

Всплывающие меню называются контекстными, потому что их содержимое обычно связано с текущей средой (состоянием) программы.

Всплывающее меню, выпадающее меню? Оно появляется где-то посреди интерфейса, поэтому всплывающее. Пункт меню свернут, и команды выпадают из него, поэтому это выпадающее меню. Фактически любые меню просто полностью появляются или полностью исчезают.



После того как меню создано, ты должен проверить, что оно открывается щелчком правой кнопки мыши. За это отвечает функция, показанная ниже (\Rightarrow *psych7.py*):

```
def openMenu(event):
    global ContextMenu
    ContextMenu.post(event.x_root, event.y_root)
```

Метод `post()` инициирует открытие меню в той позиции, где была нажата кнопка мыши. Ты должен использовать параметры `x_root` и `y_root` вместо `x` и `y` для правильной интерпретации положения мыши.

Соответствующая привязка создает указанные две строки непосредственно под функцией `initMenu()`:

```
initPopup()  
Window.bind("<Button-3>", openMenu)
```

Сначала создается контекстное меню, затем оно связывается с помощью метода `bind()` с событием, которое происходит при щелчке правой кнопкой мыши в любой позиции (окна программы).

Параметр `<Button-1>` определяет левую кнопку мыши, параметр `<Button-3>` – правую (`<Button-2>` – среднюю, если она есть).

Этого достаточно, так как контекстное меню использует уже существующие функции стандартного меню.

Дополни версию своей программы-психолога указанными выше строками кода, а затем проверь все команды в обоих меню. (Или используй файлы примеров, скачанные с сайта dmkpress.com.)

Команда в меню **Справка** по-прежнему остается нерабочей, но это легко изменить. Для начала нам нужен параметр `command` для этой команды меню:

```
Helpmenu.add_command(label="О программе", command=showInfo)
```

Затем, естественно, необходимо определить соответствующую функцию события (\Rightarrow *psych8.py*):

```
def showInfo() :  
    messagebox.showinfo("О программе", "Персональный психолог")
```

Конечно, ты можешь написать все, что угодно. Но не забудь, что нужно импортировать модуль `messagebox`:

```
from tkinter import messagebox
```

Ты познакомился с новым интересным способом вывода сообщения (в отдельном окне).

Было бы хорошо, если бы при завершении работы программа спрашивала, нужно ли сохранить данные сеанса. Так обычно и происходит в правильном приложении.

Все, что нам нужно сделать, – это дополнить кодом метод `closeAll()` (\Rightarrow *psych8.py*):

```
def closeAll() :
    if messagebox.askyesno("Выход", "Сохранить данные?") :
        saveDiagnose("backup.txt")
    window.destroy()
```

Вновь появляется окно сообщения. Но если функция `showinfo()` просто отображает текст и кнопку **ОК** для закрытия окна, функция `askyesno()` открывает окно с двумя кнопками на выбор. Тут ты можешь ответить на запрос с помощью кнопки **Да** или **Нет** (рис. 10.5).

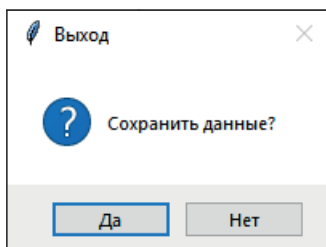


Рис. 10.5. Диалоговое окно с запросом сохранения данных

Не следует забывать, что есть, конечно, и другие возможности для использования такого диалогового окна. К примеру, метод `askokcancel()` отображает кнопки **ОК** и **Отмена**.



Условная конструкция `if` выполняется при нажатии кнопки **Да**. Затем данные сохраняются в файл под новым именем в качестве резервной копии. Разумеется, ты можешь вызвать функцию `saveFile()`.

Теперь наша программа стала довольно удобной, но все еще не хватает одной мелочи: возможности выходить из программы не только с помощью команды **Выход** меню, но и щелчком мыши по кнопке закрытия программы (т. е. по маленькому значку `x` в строке заголовка). Для этого надо написать новую функцию.

Нам нужна связь между новым событием и функцией `closeAll()`. Событие называется `WM_DELETE_WINDOW` и используется в самом конце исходного кода (перед объявлением `mainloop()`):

```
Window.protocol("WM_DELETE_WINDOW", closeAll)
```

Метод `bind` не работает, поскольку он вмешивается в процесс событий Windows: эта операционная система обычно

берет под контроль изменения: открытие или закрытие окна. Здесь же хотим вмешаться мы и взять на себя «протокол» управления – т. н. метод `protocol()`. Он направляет событие закрытия согласованному методу, которым мы управляем. (Если ты не укажешь инструкцию `Window.protocol()`, то не сможешь закрыть окно щелчком мыши по значку × в строке заголовка.)

Используем сочетания клавиш

Почти каждое меню имеет собственные сочетания клавиш, которые ты можешь использовать вместо выбора команд меню мышью. В нашем случае это может быть сочетание клавиш **Ctrl+O** для открытия файла и **Ctrl+S** для сохранения. Кроме того, ты можешь добавить сочетание клавиш **Ctrl+X**, чтобы закрывать программу (или **Ctrl+Q**).

Такие события клавиатуры могли бы выглядеть так:

```
Window.bind('<Control-o>', openFile)
Window.bind('<Control-s>', saveFile)
Window.bind('<Control-x>', closeAll)
```

Здесь задействованы три функции, которые мы используем для соответствующих пунктов меню. К сожалению, все не так просто, потому что когда вызывается метод `bind()`, функция события нуждается в параметре события. Вспомни функцию, которую мы подключили к событию `<Button-3>` для контекстного меню:

```
def openMenu(event) :
```

Поэтому нам нужно будет связать все три функции (`openFile`, `saveFile` и `closeAll`) с одним параметром, но тогда они больше не подойдут для меню. По этой причине мы используем отдельную функцию здесь только для событий клавиатуры (\Rightarrow *psych9.py*):

```
def getShotcut(event) :
    if event.keysym == "o" :
        openFile()
    if event.keysym == "s" :
        saveFile()
    if event.keysym == "x" :
        closeAll()
```

event принимает событие, `keysum` определяет, какая клавиша (кроме **Ctrl**) была нажата. И в зависимости от того, является ли эта клавиша буквой **O**, **S** или **X**, задействуется функция события, которая также соответствует меню. Конечно, только если ты изменишь связи следующим образом (\Rightarrow *psych9.py*):

```
Window.bind('<Control-o>', getShotcut)
Window.bind('<Control-s>', getShotcut)
Window.bind('<Control-x>', getShotcut)
```

Чтобы увидеть сочетания клавиш рядом с пунктами меню, ты должен дополнить методы `add_command` (\Rightarrow *psych9.py*):

```
Filemenu.add_command(label="Открыть", command=openFile, \
                      accelerator="Ctrl+O")
Filemenu.add_command(label="Сохранить", command=saveFile, \
                      accelerator="Ctrl+S")
Filemenu.add_command(label="Выход", command=closeAll, \
                      accelerator="Ctrl+X")
```

Для каждой команды параметр `accelerator` отображает соответствующее сочетание клавиш справа от имени команды (рис. 10.6).

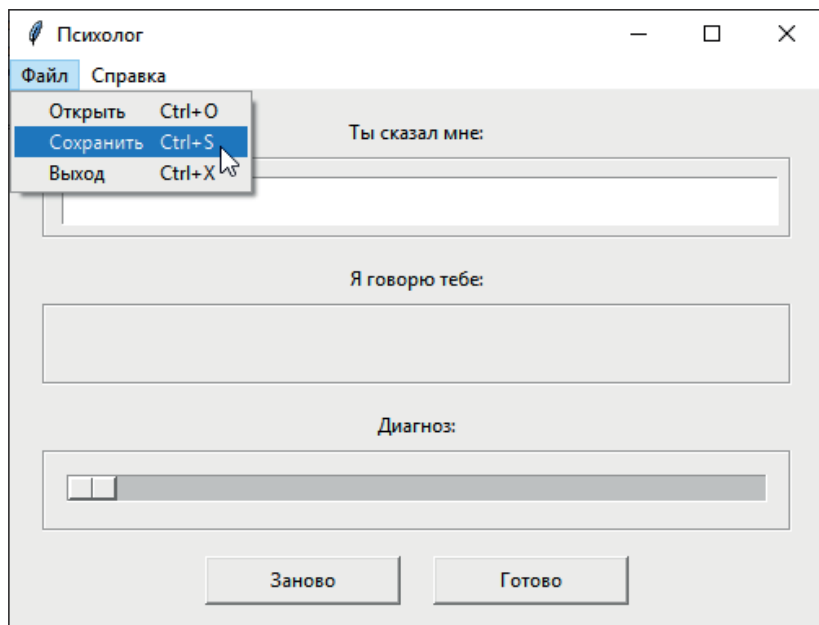


Рис. 10.6. Отображение сочетаний клавиш в меню

10

Подведение итогов

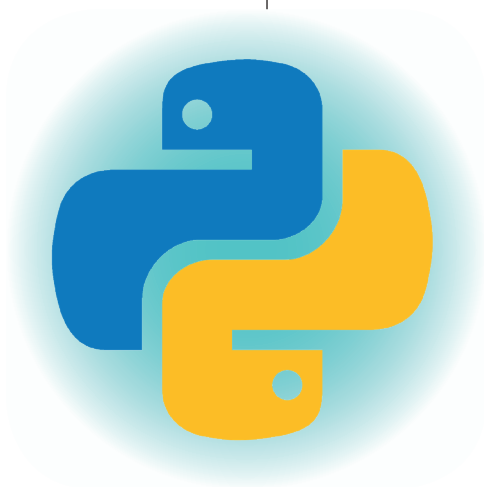
На этом мы заканчиваем работу над приложением-психологом. Конечно, программа все еще не идеальна, но я оставляю ее на тебя для продолжения экспериментов. Однако ты можешь и просто насладиться плодами своего труда. Давай посмотрим, что мы узнали:

Код	Описание
<code>add_cascade()</code>	Добавляет пункт в строку меню
<code>add_command()</code>	Добавляет команду в список меню
<code>label=</code>	Отображает текст в меню
<code>tearoff=</code>	Отображает/скрывает пунктирную линию в меню
<code>post()</code>	Позиционирует и открывает меню
<code>messagebox</code>	Подборка методов для всплывающих окон
<code>showinfo()</code>	Отображает окно с простым сообщением (для оповещения)
<code>askokcancel()</code>	Отображает окно сообщения с кнопками ОК и Отмена
<code>askyesno()</code>	Отображает окно сообщения с кнопками Да и Нет
<code>filedialog</code>	Подборка методов для диалоговых окон
<code>askopenfilename()</code>	Отображает диалоговое окно для открытия файла
<code>asksaveasfilename()</code>	Отображает диалоговое окно для сохранения файла
<code>filetypes=</code>	Ограничивает отображение указанных типов файлов
<code>protocol()</code>	Перехватывает управление событиями Windows
<code>destroy()</code>	Закрывает и удаляет окно
<code><Button-1> - <Button -3></code>	Кнопки мыши для обработки событий
<code><Control-x> ...</code>	Быстрое выполнение команды с помощью клавиши Ctrl
<code>accelerator</code>	Отображает в меню сочетание клавиш
<code>keysym</code>	События клавиатуры
<code>x_root, y_root</code>	Текущая позиция события

Несколько вопросов...

1. Из каких элементов ты собрал меню?
2. Как открыть простое диалоговое окно одним нажатием кнопки?

...и нет задач



11

Графика в Python

Хотя мы уже имели дело с графическими элементами в `tkinter`, но в основном это был текст. А можно ли рисовать в Python? Если этому научиться, ты узнаешь о некоторых интересных возможностях `tkinter`, с помощью которых можно отображать в окне довольно сложную графику. Это также зависит от того, как ты будешь использовать представленные методы.

Итак, в этой главе ты узнаешь:

- ⦿ как работать с холстом – компонентом `Canvas`;
- ⦿ как рисовать линии, прямоугольники и эллипсы;
- ⦿ как отображать в окне разноцветный текст;
- ⦿ как менять цвет и шрифт;
- ⦿ кое-что о черепашьей графике.

Точки и координаты

Когда дело доходит до графики, нужно кое-то пояснить о системе координат. Ты сейчас вспомнил о своем учителе математики, как он стоит у доски, рассказывая что-то об осях x и y . Поэтому я не буду слишком подробно объяснять эту тему.

Все, что ты видишь на мониторе, состоит из множества маленьких точек, называемых пикселями. Количество этих

11

пикселей называется разрешением. Ширина и высота изображения на мониторе могут превышать 2000 пикселей.

Чтобы увидеть изображение, на каждом компьютере есть так называемая видеокарта. Она способна обрабатывать изображения с несколькими миллионами цветов. Разрешение, которое ты видишь на экране, зависит не только от видеокарты. Разумеется, экран тоже должен быть достаточно технологичным, чтобы отображать высокое разрешение.

Чтобы компьютер мог знать, где разместить точку на экране, когда нужно отобразить какое-либо изображение, область вывода представляется как невидимая мелкая сетка. Каждая точка описывается двумя числами. Они указывают расстояние от фиксированной начальной точки. В Python (а также в большинстве графических редакторов) она находится в верхнем левом углу экрана (рис. 11.1).

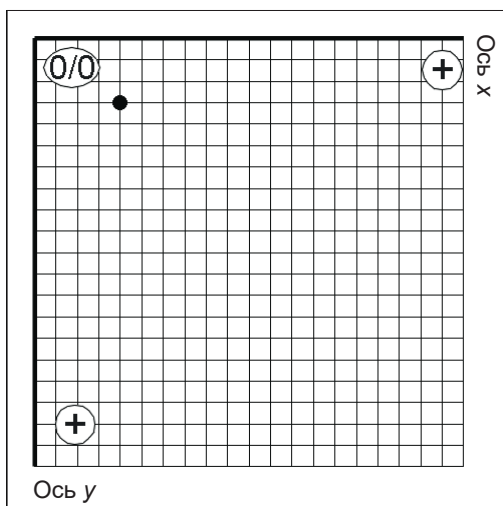


Рис. 11.1. Система координат на экране

Эта точка также называется *началом координат* и имеет координаты $x = 0$ и $y = 0$. Если отсчитывать от начала координат вправо, то будешь перемещаться по оси x . Также можно сказать про перемещение горизонтально или по горизонтальной оси. Но поскольку математики назвали ее осью x , компьютерщики так и оставили это название.

Если опускаться вниз от начала координат, ты будешь двигаться по оси y . Ее можно было бы назвать вертикальной осью, но не говори так: математики назвали ее осью y . (Кстати, жирная точка на рис. 11.1 находится в позиции (4 | 3), что можно выразить как $x = 4$ и $y = 3$.)

Вспомнил урок математики? Система координат с осями x и y выглядела совсем иначе! В центре было так называемое пересечение, где соединялись оси x и y (отсюда и пересечение координат) (рис. 11.2).

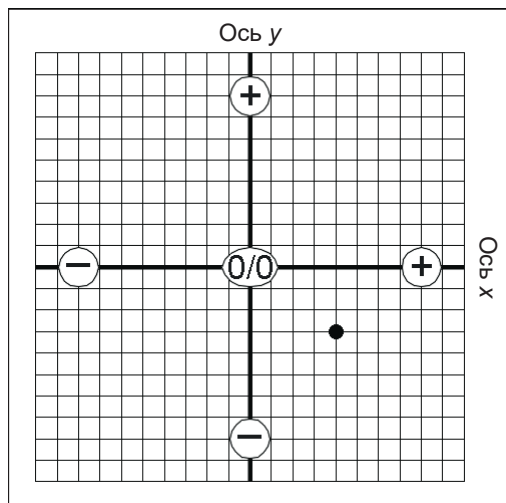


Рис. 11.2. Система координат из урока математики

(Опять же, жирная точка имеет координаты $(4 | 3)$, поэтому $x = 4, y = 3$.)

Компьютерщики все перевернули и сказали: зачем нам отрицательные значения осей? И вообще, для нас логичнее начинать так, как мы пишем: сверху слева.

Как и в математике, значения по оси x увеличиваются слева направо. Но, в отличие от математики, значения по оси y увеличиваются сверху вниз, а не наоборот.

Осталась лишь перевернутая буква «L» от математической координатной оси.

Кроме двух названных выше осей, существует также ось z . Она отвечает за глубину. Но где на экране глубина? Это как раз проблема оси z : на самом деле все фотографии являются двухмерными, т. е. используют только ширину (x) и высоту (y). А везде, где бы мы не находились, есть еще третье измерение – глубина (z) (рис. 11.3).



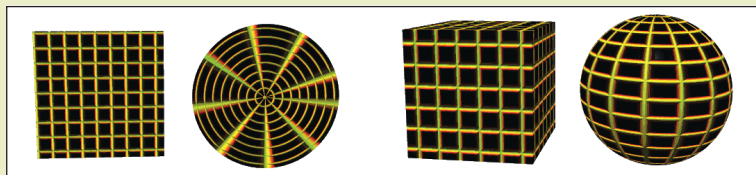


Рис. 11.3. Двухмерные и трехмерные объекты

Чтобы получить изображения, которые выглядят так, как объемные, есть специальные программы, которые определяют, как выглядит третье измерение в изображении. Новейшие видеокарты гарантируют, что на экране все будет выглядеть качественно и работать быстро. Поэтому сегодня, например, в компьютерных гонках машины и дороги действительно выглядят объемно, и с каждым движением руля экран приспособливается мгновенно.

Первое изображение

Хватит теории! Настало время перейти к практике! Итак, первая графическая программа. В качестве основы нам понадобится только окно с кнопкой. Первая попытка написать программу будет выглядеть так (\Rightarrow `graphic0.py`):

```
# Графика в Python
from tkinter import *

# Функция события
def buttonClick() :
    pass

# Основная программа
Window = Tk()
Window.title("Графика")
Window.config(width=500, height=330)
Button = Button(Window, text="Посмотреть!", command=buttonClick)
Button.place(x=190, y=150, width=120, height=30)
Window.mainloop()
```

Здесь у нас есть все необходимое для создания программы на основе библиотеки `tkinter`. Кнопка в центре уже настроена на функцию события, несмотря на то что оно еще не настроено. Окно пока все еще пустое (11.4).

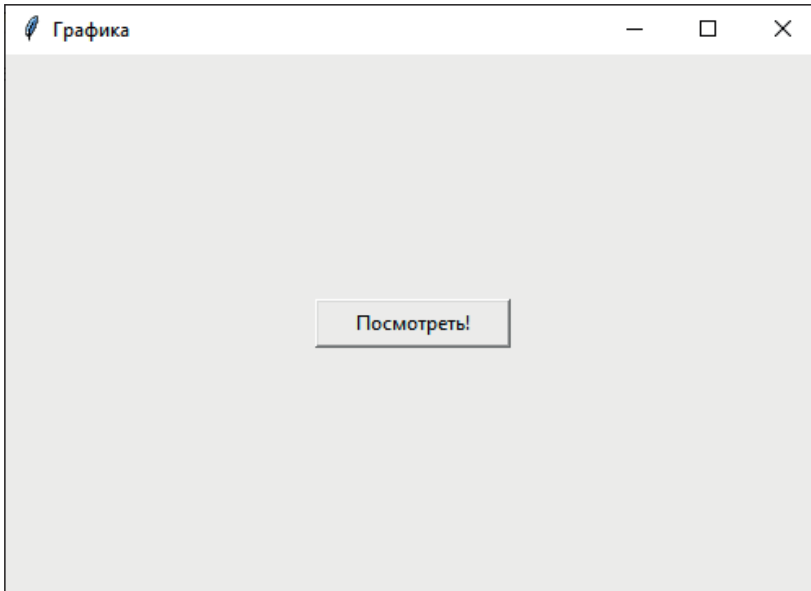


Рис. 11.4. Базовое приложение на основе библиотеки *tkinter*

Чтобы что-то нарисовать в окне, нам нужен компонент *Canvas*, что означает не что иное, как холст для рисования. Мы можем создать его и поместить в окно:

```
Graphic = Canvas(Window, width=500, height=330)
Graphic.pack()
```

Теперь у нас есть доступ к одному из методов, чтобы нарисовать, например, линии, прямоугольники или круги. Мы прописываем это в функцию, которая вызывается, когда мы нажимаем кнопку «Посмотреть!». Ниже представлен полный листинг программы (\Rightarrow *graphic1.py*):

```
# Графика в Python
from tkinter import *

# Размеры окна
Breadth = 500
Highness = 330

# Функция события
def buttonClick() :
    Graphic.create_rectangle(20, 20, Breadth-20, Highness-20)
    Graphic.create_oval(20, 20, Breadth-20, Highness-20)
    Graphic.create_line(Breadth/2, 20, Breadth/2, Highness-20)
    Graphic.create_line(20, Highness/2, Breadth-20, Highness/2)
```

11

```
# Основная программа
Window = Tk()
Window.title("Графика")
Window.config(width=Breadth, height=Highness)
Graphic = Canvas(Window, width=Breadth, height=Highness)
Graphic.pack()
Button = Button(Window, text="Посмотреть!", command=buttonClick)
Button.place(x=Breadth/2-60, y=Highness/2-15, width=120, height=30)
Window.mainloop()
```

Создай новый файл и введи указанный выше исходный код. Затем запусти программу и нажми кнопку «Посмотреть!» (рис. 11.5).

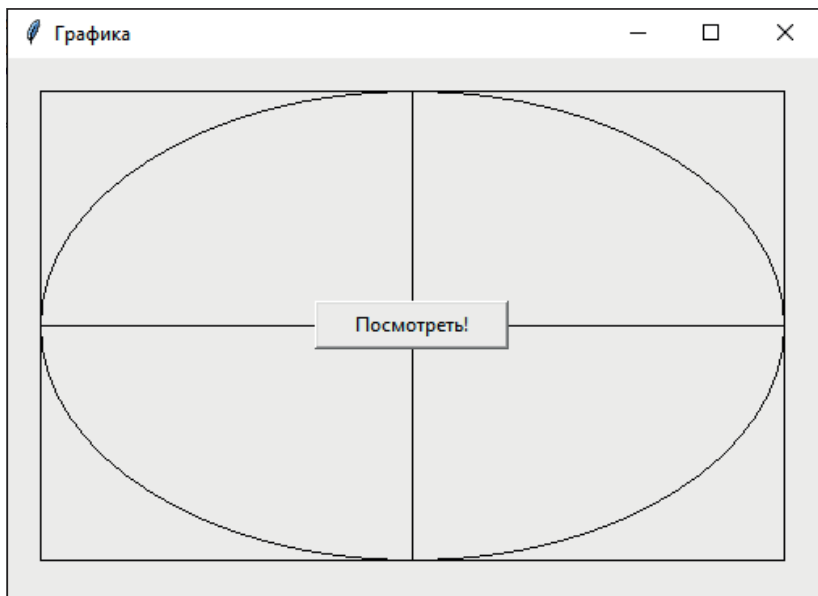


Рис. 11.5. Наша первая графическая программа

И теперь давай более подробно рассмотрим, какие объекты возникают в результате работы функции события. Появляются прямоугольник, эллипс и две линии, которые пересекаются. Все это выполняется с помощью объекта `canvas`. С помощью каких действий появляются эти объекты, ты можешь догадаться сам.

Метод `create_line()` создает линию между двумя точками:

```
create_line(xStart, yStart, xEnd, yEnd)
```

С помощью этого метода можно рисовать практически все виды графики. Когда дело касается прямоугольников или эллипсов, можно сделать это двумя другими способами:

```
create_rectangle(xLeft, yTop, xRight, yBottom)
create_oval(xLeft, yTop, xRight, yBottom)
```

Указываются координаты верхнего левого и правого нижнего углов. Они образуют (невидимую) прямоугольную область, в которую точно вписывается прямоугольник или эллипс (рис. 11.6).

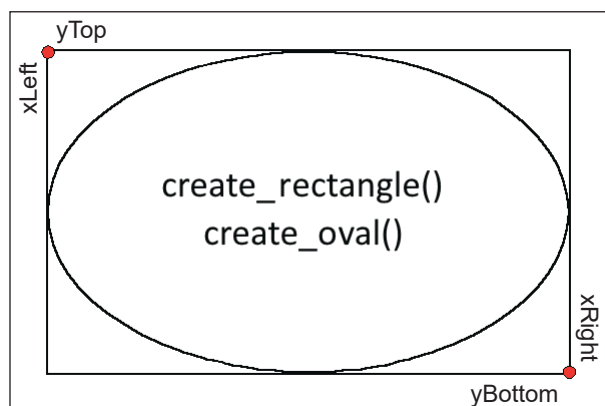


Рис. 11.6. Создание прямоугольника или эллипса

Как ты мог заметить, я не просто работаю с целыми числами в графической программе, но и создал две дополнительные переменные для ширины (Breadth) и высоты (Highness), которые затем я использую в соответствующих функциях. Преимущество: тебе не нужно изменять значения этих двух переменных, адаптация для методов происходит автоматически.

В области рисования (Graphic), как и в случае с экраном, верхний левый угол начинается с точки с координатами $x = 0$ и $y = 0$. Я учитываю поля, поэтому мы прибавляем дополнительные 20 пикселей. Разделив значение дополнительной переменной на 2, мы получаем ровно центр этой ширины или высоты. Так создается ровное пересечение без указания фиксированных координат:

```
Graphic.create_line(Breadth/2, 0, Breadth/2, Highness)
Graphic.create_line(0, Highness/2, Breadth, Highness/2)
```

11

Добавим цвета

Давай немного поэкспериментируем с методами Canvas, о которых вы узнали выше. Нам нужны цвета и помощь в их подборе. Подойдет код из предыдущей программы, нам лишь нужно изменить инструкции в функции события (\Rightarrow *graphic2.py*):

```
def buttonClick() :
    for Nr in range(0,84) :
        Dye = Color[random.randint(0,7)]
        Graphic.create_line(0, Nr*4, Breadth, Highness-Nr*4, fill=Dye)
        Graphic.create_line(Nr*6, 0, Breadth-Nr*6, Highness, fill=Dye)
```

Дополнительный параметр fill гарантирует, что линия также имеет другой цвет, отличный от черного по умолчанию. Откуда берутся другие цвета? Для этого мне пришлось создать список цветов в tkinter:

```
Color = ["gray", "black", "red", "green", \
        "blue", "cyan", "yellow", "magenta"]
```

Color – это набор цветов, из которого функция случайным образом выбирает один:

```
Dye = Color[random.randint(0,7)]
```

Конечно, модуль случайных чисел тоже должен быть импортирован:

```
import random
```

Ниже приведена табл. 11.1 с наиболее подходящими цветами (от белого я отказался, потому что фон и так белый):

Таблица 11.1. Константы цветов

Константа	Цвет	Константа	Цвет	Константа	Цвет
black	Черный	cyan	Голубой	red	Красный
gray	Серый	magenta	Пурпурный	green	Зеленый
white	Белый	yellow	Желтый	blue	Синий

- Отредактируй исходный код и запусти программу. Нажми кнопку **Посмотрим!** пару раз (ты также можешь изменить значения или формулы по своему усмот-

рению и посмотреть, что произойдет с линиями) (рис. 11.7).



Рис. 11.7. Программа обрела цвет

На большом экране все выглядит очень хорошо.

Углы и круги

Для следующей программы ты можешь смело использовать код предыдущей. Только функция события должна быть вновь изменена. Ниже показан измененный исходный код (\Rightarrow *graphic3.py*):

```
# Графика в Python
from tkinter import *
import random

# Размеры окна
Breadth = 500
Highness = 330

# Константы цвета
Color = ["gray", "black", "red", "green", \
         "blue", "cyan", "yellow", "magenta"]

# Функция события
```

```
def buttonClick() :
    for Nr in range(0,48) :
        Dye = Color[random.randint(0,7)]
        Graphic.create_rectangle(Nr*4, Nr*3, Breadth-Nr*4, \
                                Highness-Nr*3, fill=Dye)

# Основная программа
Window = Tk()
Window.title("Графика")
Window.config(width=Breadth, height=Highness)
Graphic = Canvas(Window, width=Breadth, height=Highness)
Graphic.pack()
Button = Button(Window, text="Посмотреть!", command=buttonClick)
Button.place(x=Breadth/2-60, y=Highness/2-15, width=120, height=30)
Window.mainloop()
```

Привожу снова полный листинг, чтобы ты не запутался, что изменилось. В программе создаются прямоугольники, и с каждой итерацией цикла `for` параметры и размеры изменяются соответствующим образом (рис. 11.8).

- Чтобы понять, как работает программа, посмотри, что произойдет, если ты изменишь некоторые значения.

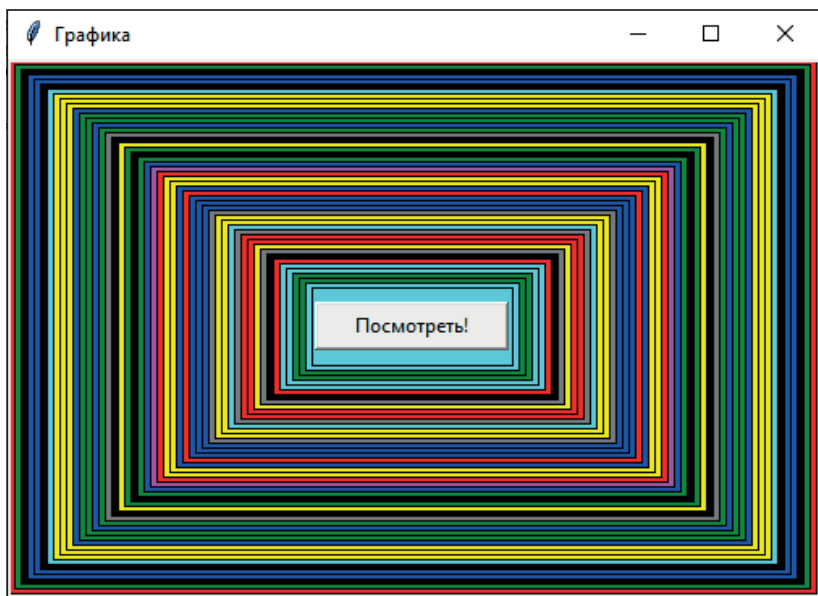


Рис. 11.8. Вариант программы с прямоугольниками

И сразу следующая версия программы, хотя ты не успел отдохнуть. Опять же, здесь изменена только функция события (\Rightarrow `graphic3a.py`):

```
# Функция события
def buttonClick() :
    for Nr in range(0,48) :
        Dye = Color[random.randint(0,7)]
        Graphic.create_oval(Nr*4, Nr*3, Breadth-Nr*4, \
                           Highness-Nr*3, fill=Dye)
```

- Измени программу и запусти ее (нажми кнопку несколько раз) (рис. 11.9).

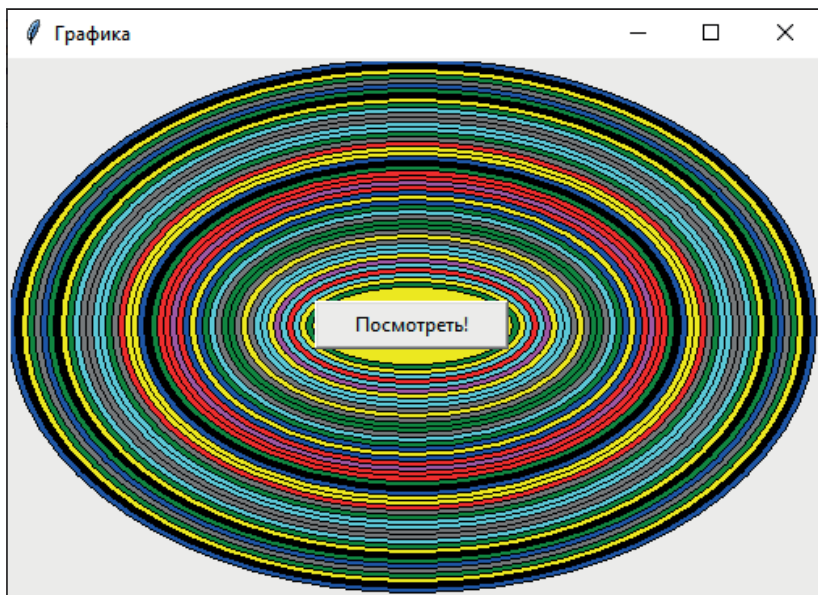


Рис. 11.9. Вариант программы с овалами

Благодаря следующей программе ты теперь знаешь, что Canvas поддерживает не только прямоугольники и круги, но и многоугольники (\Rightarrow *graphic4.py*):

```
def buttonClick() :
    Dye = Color[random.randint(0,7)]
    Point = [(20,Highness-20), (Breadth/2,20),
             (Breadth-20,Highness-20)]
    Graphic.create_polygon(Point, fill=Dye)
```

Здесь точки сначала объединяются в список, а затем метод `create_polygon()` соединяет эти точки друг с другом прямыми линиями. В данном случае получается (цветной) треугольник (рис. 11.10).

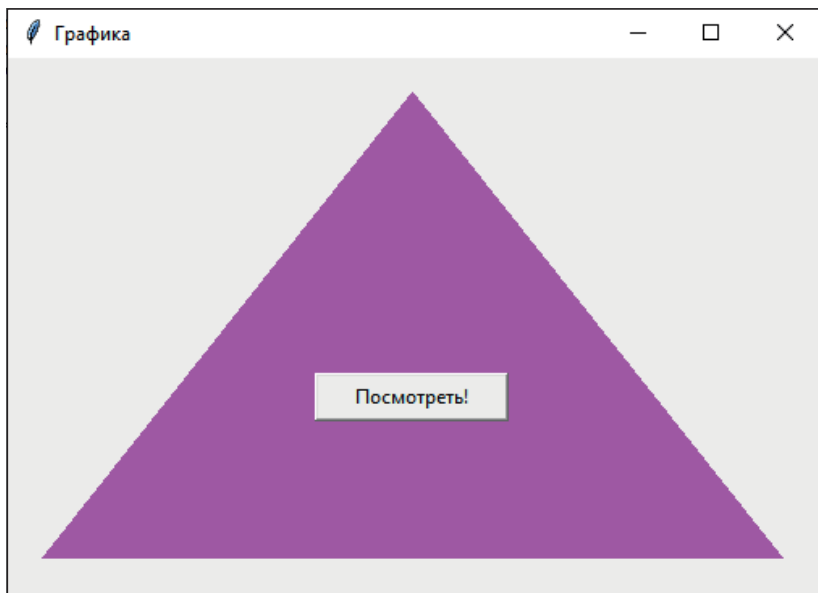


Рис. 11.10. Вариант программы с треугольником

Эксперименты с текстом

Объект класса `Canvas` может не только рисовать. Ты также можешь передать ему свои строки текста и попросить отобразить их. Ранее мы всегда передавали наш текст определенным компонентам, и они отображали его. И в самом окне отображается строка заголовка. С помощью `Canvas` можно поместить текст в любую область окна. Как это сделать, показано в следующей версии графической программы (\Rightarrow `graphic5.py`):

```
# Функция события
def buttonClick() :
    for Nr in range(0,32) :
        Dye = Color[random.randint(0,7)]
        Font = random.randint(10,30)
        Graphic.create_text(random.randint(0,Breadth), \
                           random.randint(0,Breadth), \
                           text="Привет!", fill=Dye, font=("Arial", Font))
```

`create_text()` – это имя метода, который содержит множество параметров. Я разбил их здесь на две строки (напоминаю об обратной косой черте, которая разделяет слишком длинные строки).

Здесь не только цвет, но и размер шрифта устанавливается случайным образом. Параметр `font` позволяет выбрать шрифт и кегль.

➤ Запусти эту версию графической программы (рис. 11.11).

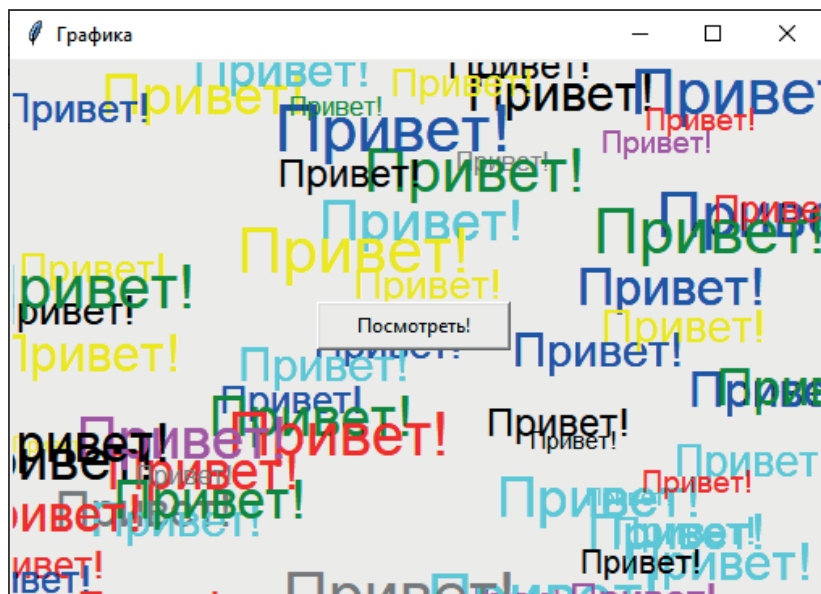


Рис. 11.11. Вариант программы с текстом

Звездное небо

Окунемся немного в романтику. Пусть звезды или что-то другое будет сверкать. Для этого мы должны установить черное фоновое небо. В этом случае объекту `Graphic` требуется другой параметр:

```
Graphic = Canvas(Window, width=Breadth, height=Highness,
background="black")
```

Примечание. Вместо слова `background` также можно использовать краткое имя параметра – `bg`.)

Кнопка не должна быть заслонять красочное зрелище, поэтому мы переместим ее к самому нижнему краю окна программы.

Теперь к функции события. Цикл `for` продолжает выполняться. Он генерирует 300 случайных значений для переменных `x`, `y` и `z` 300 раз, которые влияют на положение

11

и диаметр окружностей, создаваемых с помощью метода `create_oval()`. Поскольку фон черный, я заменяю черный цвет на белый в списке цветов.

Код изменился в нескольких местах, поэтому привожу его полностью (\Rightarrow *graphic6.py*):

```
# Графика в Python
from tkinter import *
import random

# Константы цвета
Color = ["white", "gray", "red", "green", \
         "blue", "cyan", "yellow", "magenta"]

# Функция события
def buttonClick() :
    for Nr in range(0,300) :
        x = random.randint(0,Breadth)
        y = random.randint(0,Highness)
        z = random.randint(2,15)
        Dye = Color[random.randint(0,7)]
        Graphic.create_oval(x,y, x+z, y+z, fill=Dye)

# Основная программа
Window = Tk()
Window.title("Графика")
Breadth = 500
Highness = 330
Window.config(width=Breadth, height=Highness)
Graphic = Canvas(Window, width=Breadth, height=Highness, \
                  background="black")
Graphic.pack()
Button = Button(Window, text="Посмотреть!", command=buttonClick)
Button.place(x=Breadth/2-60, y=Highness-40, width=120, height=30)
Window.mainloop()
```

- Внеси изменения в свою программу, затем запусти, и пусть звезды (или что-то еще) засверкают (рис. 11.12).

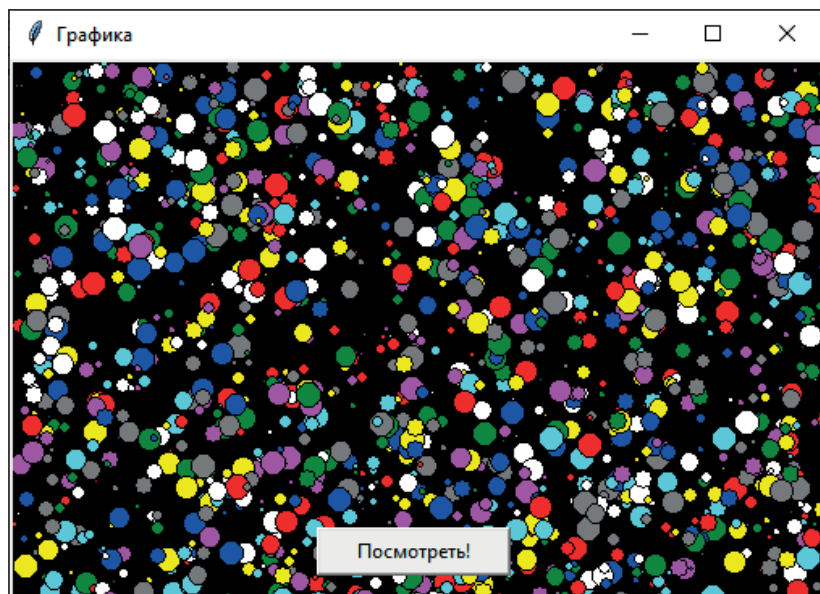


Рис. 11.12. Программа с разноцветными звездами

Сам себе художник

Ранее tkinter использовал объект Canvas, чтобы нарисовать что-то в окне. Но почему бы не попробовать нечто другое? Итак, давай создадим программу, в которой ты сможешь рисовать мышью самостоятельно. Конечно, мы также позовем на помощь модуль Canvas. Сначала мы должны убедиться (в основной программе), что существует связь между объектом Graphic и функцией события. Для этого мы используем метод bind:

```
Graphic.bind("<B1-Motion>", mouseDraw)
```

Так мы передаем событие <B1-Motion> функции mouseDraw().

Теперь, когда мы взяли в оборот другое событие, пришло время рассказать про еще несколько событий в Python – в табл. 11.2.





Таблица 11.2. События в Python

<Button-1>, <Button-3>	Нажата левая или правая кнопка мыши
<B1-Motion>, <B3-Motion>	Левая или правая кнопка мыши удерживается, мышь двигается
<Motion>	Мышь двигается
<Enter>, <Leave>	Указатель мыши находится в пределах компонента или вне
<Return>, <Escape>	Нажата клавиша Esc или нет
<key>	Нажата ли какая-либо клавиша (клавиатуры)

Конечно, есть еще много событий, почти для каждой клавиши на клавиатуре. Кроме того, они также используются для изменения состояния, размера и расположения компонента.

Итак, здесь мы имеем событие, которое происходит, когда мышь перемещается при удерживаемой левой кнопке мыши. В соответствии с тем, что должно при этом произойти, я назову новую функцию события `mouseDraw()`.

```
def mouseDraw(event):
    x = event.x - 1
    y = event.y - 1
    Graphic.create_oval( x, y, x+3, y+3)
```

Функция принимает событие как параметр с событием. Внутри функции переменные `x` и `y` назначаются координатами указателя мыши. В этот момент рисуется очень маленький круг, жирная точка.

У нас уже есть полный исходный код. Поэтому введи следующие строки и попытай удачу, например, написав свое имя (\Rightarrow `graphic7.py`):

```
# Рисование с помощью мыши
from tkinter import *

# Размеры окна
Breadth = 500
Highness = 330

# Функция события
def mouseDraw(event):
    x = event.x - 1
    y = event.y - 1
```

```
Graphic.create_oval( x, y, x+3, y+3)
```

```
# Основная программа
Window = Tk()
Window.title("Графика")
Window.config(width=Breadth, height=Highness)
Graphic = Canvas(Window, width=Breadth, height=Highness)
Graphic.bind("<B1-Motion>", mouseDraw)
Graphic.pack()
Anzeige = Label(Window, text = "Рисование мышью")
Anzeige.pack(side="bottom")
Window.mainloop()
```

Как видишь, я отказался от кнопки и заменил ее экранной подсказкой, чтобы сообщить тебе, что делать (рис. 11.13).



Рис. 11.13. Рисование в программе с помощью мыши

Черепашня графика

Наконец, я добрался до чего-то совершенно иного, также связанного с графикой, но где библиотека `tkinter` не принимает участия. Давай посвятим время медлительному животному, которое может быть воплощено в жизнь на Python.

11

Нет, это не какое-то выдуманное существо, оно действительно существует. Его не нужно кормить, но оно умеет даже рисовать, оставляя следы после себя. Посмотрим, как оно выглядит и движется (\Rightarrow *turtle1.py*):

```
# Черепашня графика
from turtle import *

# Настройка окна
Width, Height = 600, 400
Window = Screen()
Window.title("Черепашка")
Window.setup(width=Width, height=Height)

# Рисование
shape("turtle")
for step in range(0,4) :
    forward(150)
    left(90)
```

Можно было не настраивать окно, но я хотел бы сам выбрать, насколько оно будет большое и как называться. С помощью функции `Screen()` создается объект окна, а с помощью метода `setup()` устанавливаются размеры.



Новый модуль называется `turtle`. Его возможности основаны на `tkinter`. (Вот почему этот модуль интегрируется в модуль `turtle` с помощью команды `import`.) Уже при запуске окно открывается автоматически, но его размер задается системой.

Фактически будущая фигура представляет собой квадрат:

```
for step in range(0,4) :
    forward(150)
    left(90)
```

`Turtle` (черепаха на русском языке) перемещается строго на 150 пикселей вперед, оставляя линию за собой, затем она поворачивается на 90° влево и проезжает еще 150 пикселей. Это повторяется несколько раз, пока не будет нарисован квадрат.

Первоначальная позиция черепахи всегда находится в середине окна. И начальное направление указывает вверх. Это соответствует углу 0° .

Если тебе интересно, где останавливается черепаха, добавь эту строку в начале:

```
shape("turtle")
```

Затем запусти программу еще раз (рис. 11.14).

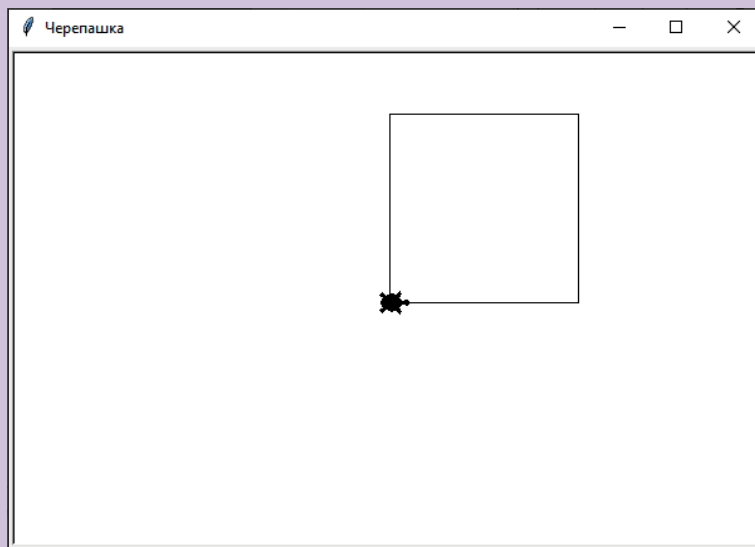


Рис. 11.14. Черепашка нарисовала квадрат



Конечно, кроме методов `forward()` и `left()`, есть также `back()` и `right()`. Я думаю, ты догадался, что означают два других метода. С помощью метода `home()` ты перемещаешь черепаху обратно в центр окна.

- Проверь код программы и запусти ее. Если ты хочешь изменить исходный код, создай свои методы в программе и попробуй реализовать то, что ты хочешь.

Это работает следующим образом: если ты импортируешь модуль `random`, черепашка будет двигаться по случайной траектории (\Rightarrow `turtle2.py`):

```
for step in range(0,10) :  
    forward(random.randint(10,100))  
    left(random.randint(60,90))
```

В принципе, рисование спирали – довольно простая в реализации задача (\Rightarrow `turtle3.py`) (рис. 11.15).

11

```
Path = 5
degree = 45
while degree > 28 :
    Path += 2
    degree -= 0.2
    left(degree)
    forward(Path)
```

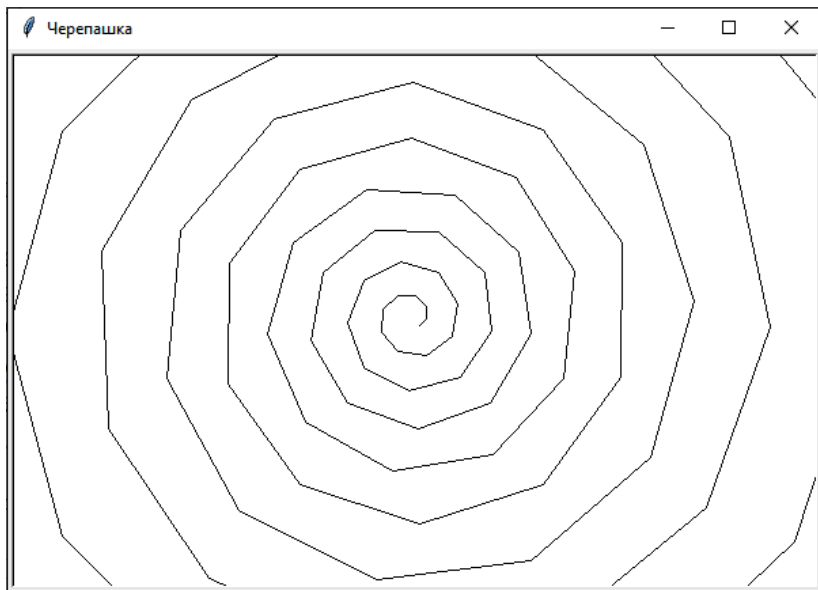


Рис. 11.15. Черепашка нарисовала спираль

Разумеется, есть возможность беспрепятственно перемещать черепаху. Это может понадобиться, например, если ты хочешь начинать рисовать не от центра, а где-то из другой позиции.

Функция `penup()` «поднимает перо», и дальнейшее движение становится невидимым. А функция `pendown()` «кладет перо» обратно на холст. С помощью функции `goto()` ты можешь переместить черепаху в нужную позицию.

Следующая программа сначала рисует прямоугольник, а затем круг в середине (\Rightarrow *turtle4.py*):

```
penup()
goto(-245,150)
pendown()
for step in range(0,2) :
    forward(480)
    right(90)
```

```
forward(290)
right(90)
# Circle
penup()
goto(-25,-140)
pendown()
circle(145)
```

И это далеко не все методы, которые доступны. Если ты хочешь узнать больше, то сможешь найти информацию на сайте docs.python.org/3/library/turtle.html.

Подведение итогов

Все было довольно красочно. И это была наиболее простая часть графических возможностей Python. Ты также узнал много новых методов модуля tkinter. Подведем итоги:

Canvas	Модуль реализует инструменты для создания графики
create_line()	Функция для рисования линий
create_rectangle()	Функция для рисования прямоугольников (квадратов)
create_oval()	Функция для рисования эллипсов (кругов)
create_polygon()	Функция для рисования (любых) многоугольников
create_text()	Функция для отображения текста (строк)
bind()	Связывает событие и функцию (напоминаю)
<Motion>	Отвечает за движение мыши
background=, bg=	Задаёт цвет фона
fill=	Задаёт цвет контура или заливки
font=	Задаёт шрифт и его размер
turtle	Модуль для черепаший графики
forward(), back()	Перемещает черепашку вперед/назад
left(), right()	Перемещает черепашку влево/вправо
home(), goto()	Перемещает черепашку в центр окна или в указанное положение
penup(), pendown()	Поднимает черепашку над холстом (рисует и не рисует соответственно)

Несколько вопросов...

1. Где находится начало системы координат на экране?
2. Методы рисования линий, прямоугольников и эллипсов используют четыре параметра. Чем они отличаются?

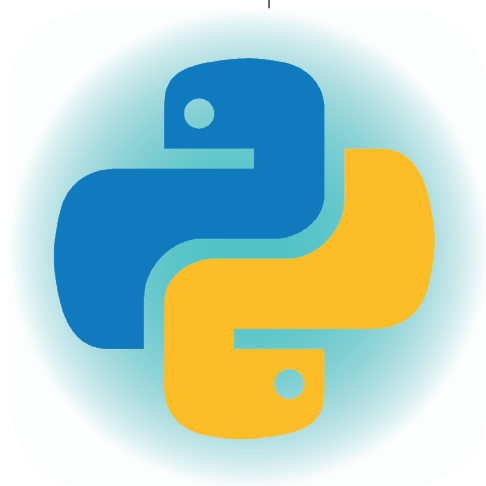
11

...и задач

1. В предпоследней графической программе (со сверкающими звездочками) замени их на маленькие точки.
2. В той же программе замени цветные точки квадратами.
3. Сможешь нарисовать домик?
4. Нарисуй спиральную паутину с помощью черепашки.

12

Создание анимации



Твои достижения с графикой в последней главе были неплохими. Но, может, будет лучше, если ты сможешь отобразить в программе загруженный графический элемент? И чтобы у тебя была возможность перемещать такую фигуру. Это то, о чем мы сейчас и поговорим.

Итак, в этой главе ты узнаешь:

- ⦿ как использовать внешние графические файлы с помощью модуля `Canvas`;
- ⦿ дополнительные приемы по работе с графикой;
- ⦿ как заставить фигуру появляться на экране и двигаться;
- ⦿ как создать собственный игровой модуль.

Начнем с круга

Начнем с простого графического объекта, круга. Он должен появляться после нажатия кнопки, а затем снова исчезать. Во-первых, нам нужно окно с тремя кнопками в нижней части экрана (рис. 12.1).

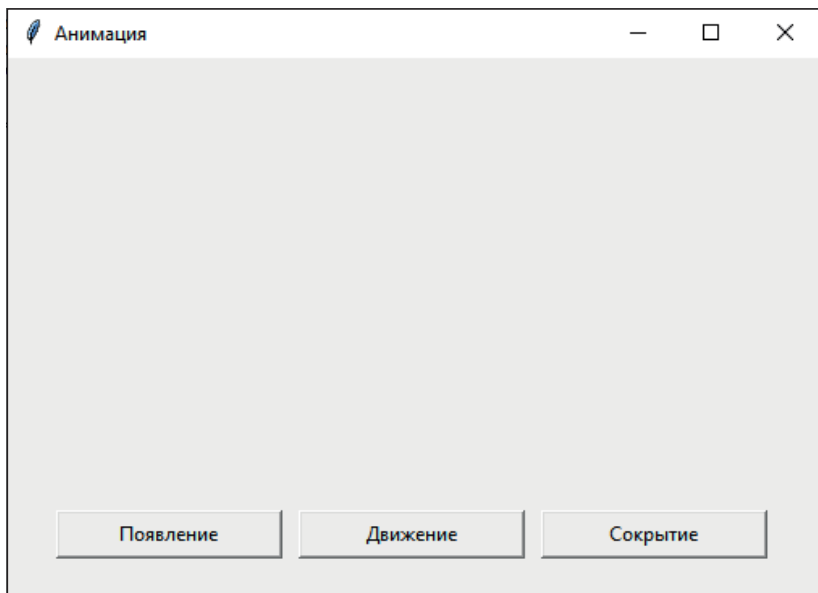


Рис. 12.1. Базовое приложение для проекта

Начнем с соответствующего исходного кода (\Rightarrow *movie0.py*):

```
# Анимация
from tkinter import *

# Размеры окна и текста
Width = 500
Height = 330
Mode = ["Появление", "Движение", "Скрытие"]

# Функция события
def showImage() :
    pass
def moveImage() :
    pass
def hideImage() :
    pass

# Основная программа
Window = Tk()
Window.title("Анимация")
Window.config(width=Width, height=Height)
Graphic = Canvas(Window, width=Width, height=Height)
Graphic.pack()
Knob = []
for Nr in range(0,3) :
    Knob.append(Button(Window, text=Mode[Nr]))
```

```
Knob[Nr].place(x=30+Nr*150, y=Height-50, width=140, height=30)
Knob[0].config(command=showImage)
Knob[1].config(command=moveImage)
Knob[2].config(command=hideImage)
Window.mainloop()
```

Конечно, нам по-прежнему нужен объект, который мы сможем передвигать в окне. Для этого сначала возьмем простой круг.

Его положение и размеры определены:

```
x, y, z = 20, 50, 200
```

Здесь я наконец использую возможность (как упоминалось выше при согласовании размера окна) Python, касающуюся создания сразу нескольких переменных и присваивания им значений.

Функция `showImage()` выглядит следующим образом (\Rightarrow *move1.py*):

```
def showImage() :
    global Circle
    Circle = Graphic.create_oval(x, y, x+z, y+z, fill="red")
```

Тут я использую глобальную переменную `Circle`, которой присваивается возвращаемое значение метода `create_oval()`. Таким образом, `Circle` является не объектом, а своего рода идентификационным номером. Это необходимо для двух других функций события.

Теперь после нажатия кнопки круг должен перемещаться вправо. Следующий метод выглядит так (\Rightarrow *move1.py*):

```
def moveImage() :
    global Circle
    for pos in range(20,Width-220,2) :
        Graphic.move(Circle, 2, 0)
        Graphic.update()
        Graphic.after(10)
```

В цикле круг теперь перемещается постепенно слева направо. Используется метод `move()`, который принимает размер шага для горизонтального и вертикального перемещений в дополнение к переменной `Circle`. В примере перемещение выполняется только слева направо, поэтому значение `y` равно 0.

12



Метод `update` гарантирует, что содержимое окна обновляется после прохождения каждого цикла. Но какой смысл в методе `after()`? Если ты не укажешь эту строку, то сразу увидишь эффект: круг будет двигаться слева направо с молниеносной скоростью, а если метод используется, движение выполняется несколько миллисекунд.

Почему бы нам не использовать модуль `time`, как ранее? По идее, тот же результат будет достигнут, например, с помощью `time.sleep(0,01)` вместо `after(10)`. Прием также работает, в принципе. Однако не рекомендуется использовать внешние функции времени, потому что `tkinter` – это полноценная система (пользовательский интерфейс). В программах, особенно крупных, внешний таймер может конфликтовать с внутренним управлением событиями посредством `mainloop()`. Вот почему мы используем метод `tkinter after()`.

Нам еще не хватает функции, с помощью которой круг будет исчезать (\Rightarrow *move1.py*):

```
def hideImage() :  
    global Circle  
    Graphic.delete(Circle)
```

Метод `delete()` удаляет круг из окна, затем его можно и нужно создать заново с помощью метода `create_oval()`.

- В приведенную выше программу добавь исходный код трех методов для кнопок `SHOW`, `MOVE` и `DISAPPEAR`. Затем запусти программу, отобрази круг, передвинь и удали его (рис. 12.2).



Рис. 12.2. Готовое приложение

Попробуй поэкспериментировать с разными значениями в качестве параметров методов `move()` и `after()`. (При значении 0 эффект торможения равен нулю, но если значения слишком велики, движение может стать бесконечным!)

Загружаем на холст изображение

Да, круг неплох, однако лучше сделать реальную картинку, например с фигуркой, которая сможет даже двигаться сама. Но для этого нам нужен метод, который способен обрабатывать внешние фотографии. Он доступен в `Canvas` под именем `create_Image()`, но сначала должен быть загружен файл изображения. Это можно сделать с помощью класса, отличного от `Canvas`:

```
Picture = PhotoImage(file="Bilder/Figure01.gif")
```

Опять же, `Picture` – это лишь идентификационный номер, который `PhotoImage()` назначает изображению после его загрузки, для чего параметру `file` передается имя файла.

В нашем случае файл `figur01.gif` находится в папке `Bilder`. В случае использования другой папки необходимо указать правильный путь.

12



Почему именно изображения с расширением *GIF*, как насчет *JPG* или *PNG*? В настоящее время *tkinter* позволяет использовать только изображения в формате *GIF*. Если ты хочешь применять другие форматы, необходим дополнительный модуль под названием *Python Imaging Library* (сокращенно *PIL*). Его можно загрузить с сайта pythonware.com/products/pil/.

К сожалению, этот модуль разработан только под версию *Python 2*, а в этой книге мы имеем дело с *Python 3*, но для внедрения поддержки есть модуль под названием *Pillow*, который доступен на сайте pypi.python.org/pypi/Pillow/.

Модуль должен быть установлен в папку *Python*. Затем он может быть интегрирован с *PIL* путем ввода инструкции `import Image`. На мой взгляд, такой прием сложнее, чем класс *PhotoImage*. Поэтому если есть возможность использовать изображения в формате *GIF*, не нужно маяться с модулями *PIL* и *Pillow*.

Как только изображение будет доступно, его можно создать и отобразить в объекте *Canvas*:

```
Figure = Graphic.create_image(x, y, image=Picture)
```

Следует отметить, что центр изображения указан здесь с помощью переменных *x* и *y*.

Поскольку переменная *Circle* была заменена фигурой, другие функции события также претерпевают (небольшие) изменения. Ниже представлен полный листинг новой версии программы (\Rightarrow *movie2.py*):

```
# Анимация
from tkinter import *

# Режим и текст
Width, Height = 600, 400
x, y = 120, 160
Mode = ["Появление", "Движение", "Скрытие"]

# Функция события
def showImage():
    global Figure
    global Picture
    Picture = PhotoImage(file="Bilder/Figure01.gif")
    Figure = Graphic.create_image(x, y, image=Picture)
def moveImage():
    global Figure
    for pos in range(20, Width-200, 2):
        Graphic.move(Figure, 2, 0)
```

```

Graphic.update()
Graphic.after(10)
def hideImage() :
    global Figure
    Graphic.delete(Figure)

# Основная программа
Window = Tk()
Window.title("Анимация")
Window.config(width=Width, height=Height)
Graphic = Canvas(Window, width=Width, height=Height)
Graphic.pack()
Knob = []
for Nr in range(0,3) :
    Knob.append(Button(Window, text=Mode[Nr]))
    Knob[Nr].place(x=80+Nr*150, y=Height-50, width=140, height=30)
Knob[0].config(command=showImage)
Knob[1].config(command=moveImage)
Knob[2].config(command=hideImage)
Window.mainloop()

```

Как видишь, обе используемые переменные (Picture и Figure) должны быть глобальными, иначе все это не работает.

- Измени исходный код и убедись, что у тебя есть соответствующее изображение. (Если хочешь, ты можешь использовать изображения из папки с примерами для этой книги.)
- Теперь запусти программу; на этот раз рисунок будет перемещаться в пределах окна (рис. 12.3).

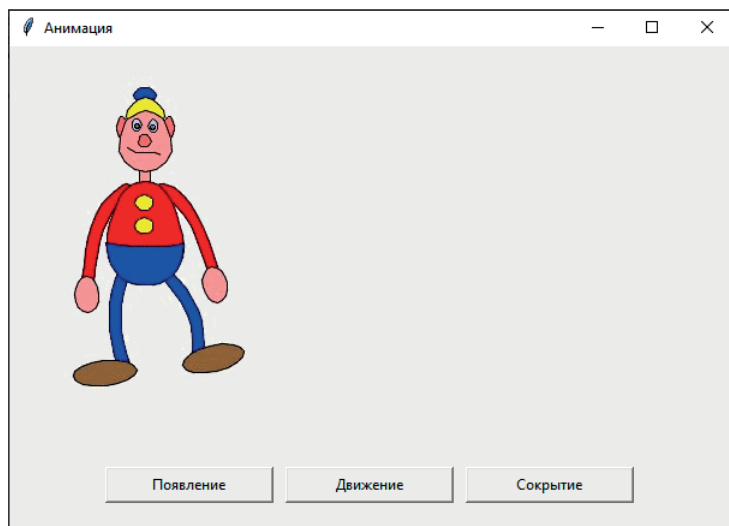


Рис. 12.3. Вместо круга – клоун

12

Коллекция изображений

Изображение перемещается, но без эффекта движения – в смысле ходьбы или бега. Это было не так трагично в случае с кругом, а здесь проблема очень заметна.

Для анимации прогулки в двух направлениях нам нужна коллекция из как минимум восьми изображений. Итак, сначала создадим список:

```
Picture = [0]
```

Первый элемент уже есть. Для удобства я начинаю отсчет с 1, поэтому использую в общей сложности 9 элементов (и оставлю нулевой неиспользованным) (рис. 12.4).

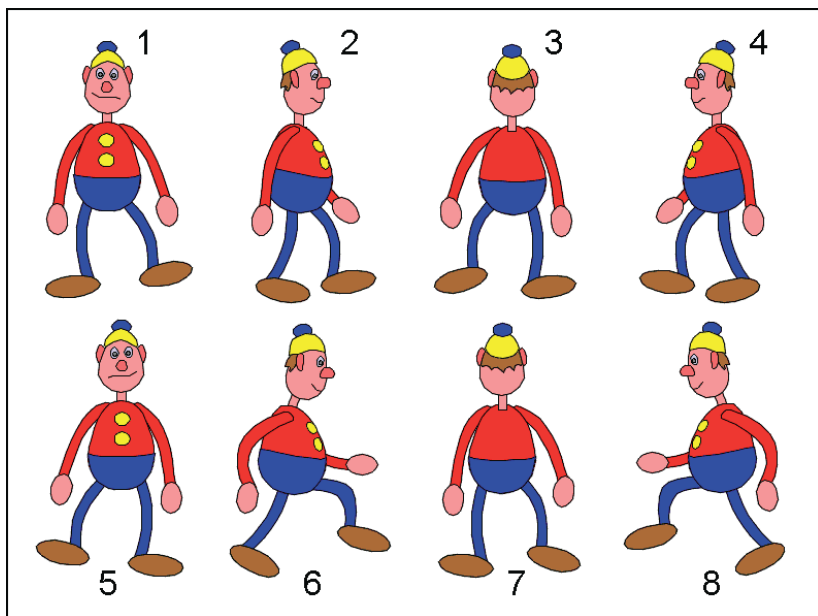


Рис. 12.4. Подборка изображений для проекта

Теперь нам нужно достаточное количество рисунков. Я пронумеровал свои файлы изображений с *figure01.gif* по *figure08.gif*.

Ты можешь, конечно, создать свою собственную коллекцию изображений. Все изображения должны иметь одинаковые размеры и должны быть сохранены в соответствующем порядке с числом от 1 до 8. Правила, которые нужно соблюдать, приведены в табл. 12.1.

Таблица 12.1. Изображения для проекта

Файл изображения	Позиция	Файл изображения	Позиция
<i>figure01.gif</i>	спереди	<i>figure05.gif</i>	спереди
<i>figure02.gif</i>	справа	<i>figure06.gif</i>	справа
<i>figure03.gif</i>	сзади	<i>figure07.gif</i>	сзади
<i>figure04.gif</i>	слева	<i>figure08.gif</i>	слева

Каждая фигура сохранена в двух разных положениях. Если показывать поочередно, например *figure02* и *figure06*, то похоже, что персонаж шагает вправо.

Загрузка коллекции изображений должна выполняться с помощью дополнительной функции, отличной от `showImage()`. Вместо одной инструкции нам нужен цикл, который должен выглядеть так:

```
for Nr in range(1,9) :
    Name = "Figure0"+str(Nr)+".gif"
    self.Picture.append(PhotoImage(file="Bilder/"+Name))
```

Во-первых, составляется имя файла, что можно сделать, используя его номер (который, конечно же, должен быть преобразован в строку). Затем элемент добавляется в список.

Но прежде чем мы определим другую функцию, мы должны сделать небольшую паузу. Мы уже говорили про классы в прошлых главах, но есть вероятность, что ты мог все забыть. Что мешает использовать класс в нашем анимационном проекте?

С моей точки зрения, ничего. Но как должен выглядеть новый класс? В любом случае в него входят три метода событий. Затем мы попытаемся суммировать все это и поместить туда инициализацию (\Rightarrow *movie3.py*):

```
# Класс Player
class Player :
    Picture = [0]
    def __init__(self, Graphic) :
        for Nr in range(1,9) :
            Name = "Figure0"+str(Nr)+".gif"
            self.Picture.append(PhotoImage(file="Bilder/"+Name))
    def showImage(self) :
        self.Figure = Graphic.create_image(x, y, image=self.Picture[1])
    def moveImage(self) :
        for pos in range(20,Width-200,2) :
            Graphic.move(self.Figure, 2, 0)
```

```
Graphic.update()
Graphic.after(10)
def hideImage(self) :
    Graphic.delete(self.Figure)
```

Новым является метод `__init__()`. Важно, чтобы каждый метод имел параметр `self`. Только так можно получить доступ к методам переменных класса.

Поскольку графический объект `Canvas` не относится к классу `Player`, он наследуется, когда экземпляр инициализируется или создается.

После того как все изображения были собраны в `__init__()`, функция `showImage()` отображает первый рисунок:

```
self.Figure = Graphic.create_image(x, y, image=self.Picture[1])
```

Разумеется, изменения коснутся и основной программы. Там изображение должно быть согласовано как объект:

```
Gameer = Player(Graphic)
```

Ее код также нуждается в сопоставлении методов событий, которые теперь относятся к классу `Player`:

```
Knob[0].config(command=Gameer.showImage)
Knob[1].config(command=Gameer.moveImage)
Knob[2].config(command=Gameer.hideImage)
```

Весь остальной код остается неизменным. Но, даже отредактировав исходный код, ты не заметишь видимых изменений после запуска программы. Изображения все еще не анимированы.

Класс `Player`

Прежде чем мы оживим персонажа, мы должны сделать класс более универсальным. Для этого мы сохраним его в новом файле, который я назову *mplayer.py*.

Первой строкой обязательно должна быть инструкция `import`:

```
from tkinter import *
```

Ниже вставляем скопированное полное определение класса `Player` (рис. 12.5).

В файле с основной программой ты, разумеется, можешь удалить соответствующий код. В верхней части файла с основной программой введи дополнительную инструкцию `import`:

```
from mplayer import *
```

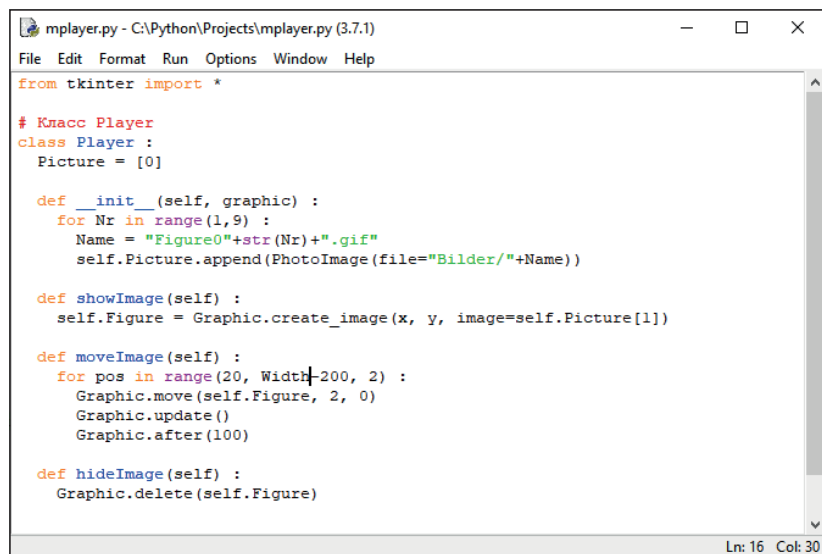
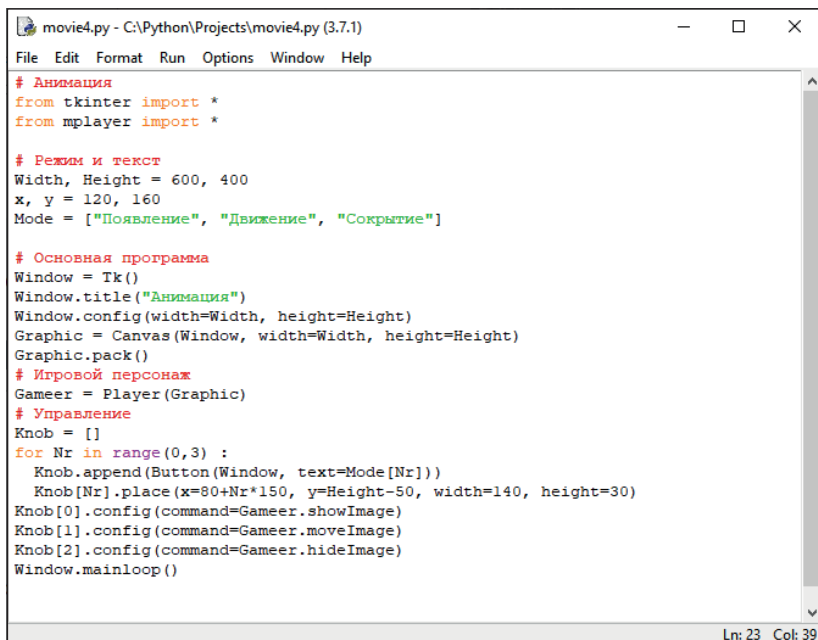


Рис. 12.5. Содержимое файла `mplayer.py`

Теперь программа стала намного лучше (рис. 12.6). Но работает ли она?



```

movie4.py - C:\Python\Projects\movie4.py (3.7.1)
File Edit Format Run Options Window Help

# Анимация
from tkinter import *
from mplayer import *

# Режим и текст
Width, Height = 600, 400
x, y = 120, 160
Mode = ["Появление", "Движение", "Скрытие"]

# Основная программа
Window = Tk()
Window.title("Анимация")
Window.config(width=Width, height=Height)
Graphic = Canvas(Window, width=Width, height=Height)
Graphic.pack()

# Игровой персонаж
Gameer = Player(Graphic)

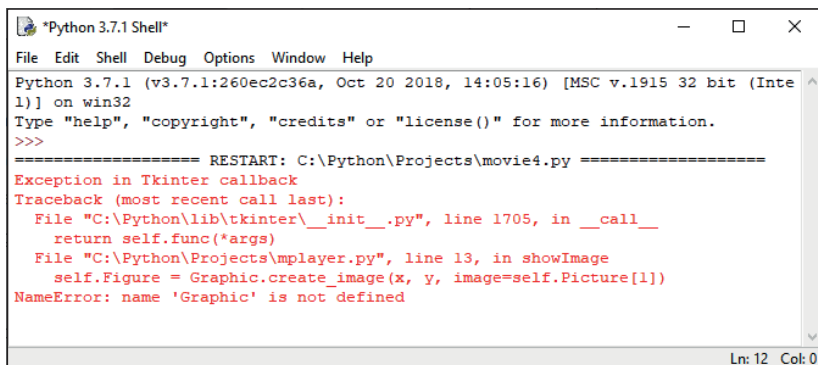
# Управление
Knob = []
for Nr in range(0,3) :
    Knob.append(Button(Window, text=Mode[Nr]))
    Knob[Nr].place(x=80+Nr*150, y=Height-50, width=140, height=30)
Knob[0].config(command=Gameer.showImage)
Knob[1].config(command=Gameer.moveImage)
Knob[2].config(command=Gameer.hideImage)
Window.mainloop()

```

Рис. 12.6. Содержимое файла *movie4.py*

Достаточно ли просто создать внешний класс и связать его с основной программой с помощью инструкции импорта? В принципе, да.

Но первый запуск приводит к ошибке, показанной на рис. 12.7, сразу после нажатия кнопки **Появление**.



```

Python 3.7.1 Shell
File Edit Shell Debug Options Window Help

Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python\Projects\movie4.py =====
Exception in Tkinter callback
Traceback (most recent call last):
  File "C:\Python\lib\tkinter\_init_.py", line 1705, in __call__
    return self.func(*args)
  File "C:\Python\Projects\mplayer.py", line 13, in showImage
    self.Figure = Graphic.create_image(x, y, image=self.Picture[1])
NameError: name 'Graphic' is not defined

```

Рис. 12.7. Ошибка после запуска программы

Объект `Graphic` не определен? Но этот объект был передан как параметр при создании экземпляра `Player`! Очевидно, что ошибка заключается в определении класса `Player`.

Первое, о чем я подумал, – это указать слово `self` перед `Graphic`. Но это приводит к новой ошибке: теперь Python жалуется, что `Graphic` не является атрибутом `Player`. Это можно изменить следующим образом:

```
def __init__(self, graphic) :  
    self.Graphic = graphic
```

Я чуть изменил имя второго параметра метода `__init__()`, при котором во время назначения будет создана новая переменная, которая теперь является атрибутом класса.

Сейчас нужно многое изменить. Все это можно найти в следующем листинге в определении класса `Player`:

```
from tkinter import *  
  
# Класс Player  
class Player :  
    Picture = [0]  
  
    def __init__(self, graphic) :  
        self.Graphic = graphic  
        for Nr in range(1,9) :  
            Name = "Figure0"+str(Nr)+".gif"  
            self.Picture.append(PhotoImage(file="Bilder/"+Name))  
  
    def showImage(self, x, y, Nr) :  
        self.Figure = self.Graphic.create_image(x, y, image=self.Picture[Nr])  
  
    def moveImage(self, von, bis) :  
        for pos in range(von, bis, 2) :  
            self.Graphic.move(self.Figure, 2, 0)  
            self.Graphic.update()  
            self.Graphic.after(100)  
  
    def hideImage(self) :  
        self.Graphic.delete(self.Figure)
```

➤ Напиши исходный код определения класса в новом файле.

В основной программе также есть улучшения. Это касается вызова двух методов – `showImage()` и `moveImage()`. Поскольку у них теперь больше параметров, а не только `self`, это необходимо учитывать и в соответствующих кнопках:

```
Knob[0].config(command=lambda : Gameer.showImage(x,y,1))
Knob[1].config(command=lambda : Gameer.moveImage(20,Width-200))
```

Еще раз, ключевое слово `lambda` должно использоваться как вспомогательное средство, чтобы вызываемые методы могли иметь параметры. Метод `showImage()` принимает позицию фигуры и номер изображения, а метод `moveImage()` – начальное и конечное значения по оси `x` (т. е. для перемещения по горизонтали).

Все работает?

Все работает потому, что мы наконец создали нормальную программу, в которой может быть использован внешний класс? Да, но это еще не все. Теперь нам надо позаботиться о том, чтобы этот класс смог анимировать персонажа.

Очевидно, что изменения в основном касаются метода `moveImage()`. Но есть еще несколько иных коррективов. В первую очередь нам нужна другая переменная в качестве атрибута класса:

```
PictureNr = 0
```

Она нужна, потому что для анимации мы должны переключаться между разными изображениями. В этом случае мы имеем дело с изображениями номер 2 и 6. Для метода `moveImage()` в цикле `for` это выглядит так:

```
if self.PictureNr == 2 :
    self.PictureNr = 6
else :
    self.PictureNr = 2
```

Конструкция `if` гарантирует, что номер изображения всегда сменяется с 2 на 6 и так далее, а затем изображение сбрасывается:

```
self.Graphic.itemconfig(self.Figure, \
                        image=self.Picture[self.PictureNr])
```

С помощью метода `itemconfig()` мы заменяем одно изображение другим. Таким образом, два рисунка отображаются поочередно – либо *figure02*, либо *figure06*. Имитируется анимация. Кроме того, изображение перемещается:

```
self.Graphic.move(self.Figure, 10, 0)
self.Graphic.update()
self.Graphic.after(100)
```

Я изменил размер шага для `move()`, а также добавил паузу, поэтому скорость смены изображений соответствует общему движению. В конце клоун должен снова смотреть вперед:

```
self.Graphic.itemconfig(self.Figure, image=self.Picture[1])
```

Ниже показан полный исходный код класса `Player` (\Rightarrow `mplayer.py`):

```
class Player :
    Picture = [0]
    PictureNr = 0

    def __init__(self, graphic) :
        self.Graphic = graphic
        for Nr in range(1,9) :
            Name = "Figure0"+str(Nr)+".gif"
            self.Picture.append(PhotoImage(file="Bilder/"+Name))

    def showImage(self, x, y, Nr) :
        self.Figure = self.Graphic.create_image(x, y, \
            image=self.Picture[Nr])

    def moveImage(self, von, bis) :
        for pos in range(von, bis, 10) :
            if self.PictureNr == 2 :
                self.PictureNr = 6
            else :
                self.PictureNr = 2
            self.Graphic.itemconfig(self.Figure, \
                image=self.Picture[self.PictureNr])
            self.Graphic.move(self.Figure, 10, 0)
            self.Graphic.update()
            self.Graphic.after(100)
            self.Graphic.itemconfig(self.Figure, image=self.Picture[1])

    def hideImage(self) :
        self.Graphic.delete(self.Figure)
```

- Оформи исходный код в соответствии с моими замечаниями, а затем запусти последнюю программу и нажми по очереди кнопки **Появление** и **Движение**. Посмотри, как фигура бродит в окне (рис. 12.8). Затем нажми кнопку **Соккрытие**.

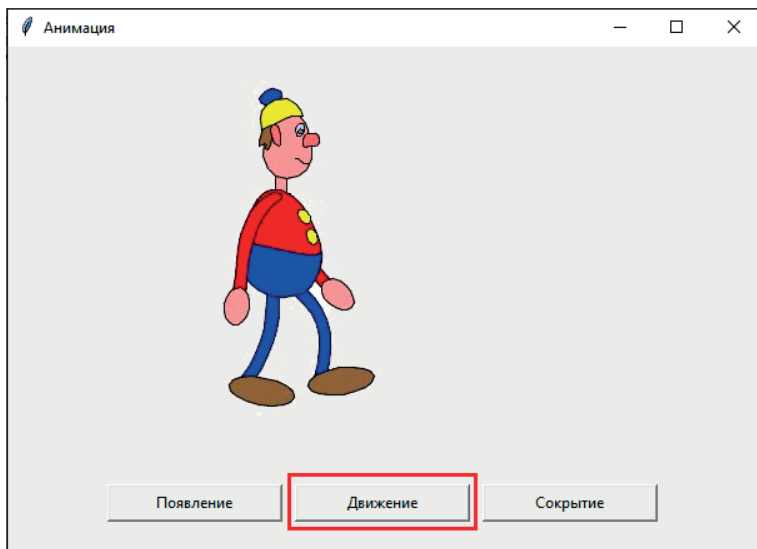


Рис. 12.8. Фигура клоуна бредет в окне

Повороты

В моем проекте есть место для другого метода, который я бы назвал `turnImage()`. Он заставляет фигуру поворачиваться вокруг своей оси.

Для этого сначала нужно дополнить окно еще одной кнопкой. Измени исходный код так, как показано ниже (\Rightarrow *movie5.py*):

```
# Анимация
from tkinter import *
from mplayerX import *

# Режим и текст
Width, Height = 600, 400
x, y = 120, 160
Mode = ["Появление", "Движение", "Поворот", "Скрытие"]

# Основная программа
Window = Tk()
Window.title("Анимация")
Window.config(width=Width, height=Height)
Graphic = Canvas(Window, width=Width, height=Height)
Graphic.pack()
# Игровой персонаж
Gameer = Player(Graphic)
# Управление
Knob = []
```

```
for Nr in range(0,4) :
    Knob.append(Button(Window, text=Mode[Nr]))
    Knob[Nr].place(x=30+Nr*135, y=Height-50, width=130, height=30)
Knob[0].config(command=lambda : Gameer.showImage(x,y,1))
Knob[1].config(command=lambda : Gameer.moveImage(20,Width-200))
Knob[2].config(command=Gameer.turnImage)
Knob[3].config(command=Gameer.hideImage)
Window.mainloop()
```

Нумерация кнопок немного меняется, но кнопка **Сокры-
тие** должна оставаться последней. Метод `turnImage()` не нуж-
дается в параметрах, поэтому его можно задействовать без
ключевого слова `lambda`.

Давай посмотрим, как мы можем заставить фигуру пово-
рачиваться (\Rightarrow *mplayerx.py*):

```
def turnImage(self) :
    for Nr in range(1, 5) :
        self.Graphic.itemconfig(self.Figure, image=self.Picture[Nr])
        self.Graphic.update()
        self.Graphic.after(200)
    self.Graphic.itemconfig(self.Figure, image=self.Picture[1])
```

В цикле изображения отображаются последовательно, одно
за другим. Похоже, что персонаж крутится вокруг своей оси
(рис. 12.9).

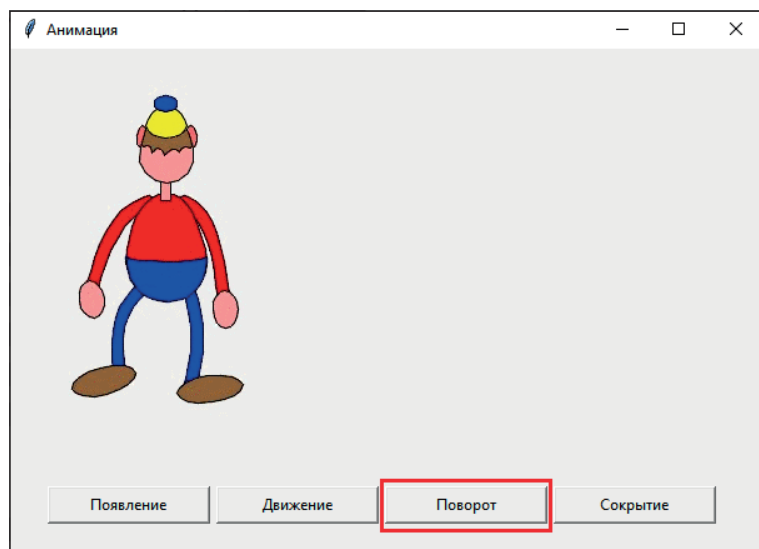


Рис. 12.9. Клоун крутится вокруг своей оси



Важно, чтобы фотографии были сохранены в порядке, указанном выше. Если ты используешь свои собственные изображения, обязательно обрати на этот момент внимание или измени код метода `turnImage()`.

Чтобы вращение не происходило очень быстро, после прерыва интервал увеличивается. В конце фигура возвращается в положение, обращенное к пользователю.

Исчезновение и появление

Есть еще кое-что, что меня беспокоит. Если ты нажмешь кнопку **Появление**, а затем **Движение**, появится фигура и начнет двигаться. Однако при повторном нажатии кнопки **Появление** появляется вторая фигура (рис. 12.10)!

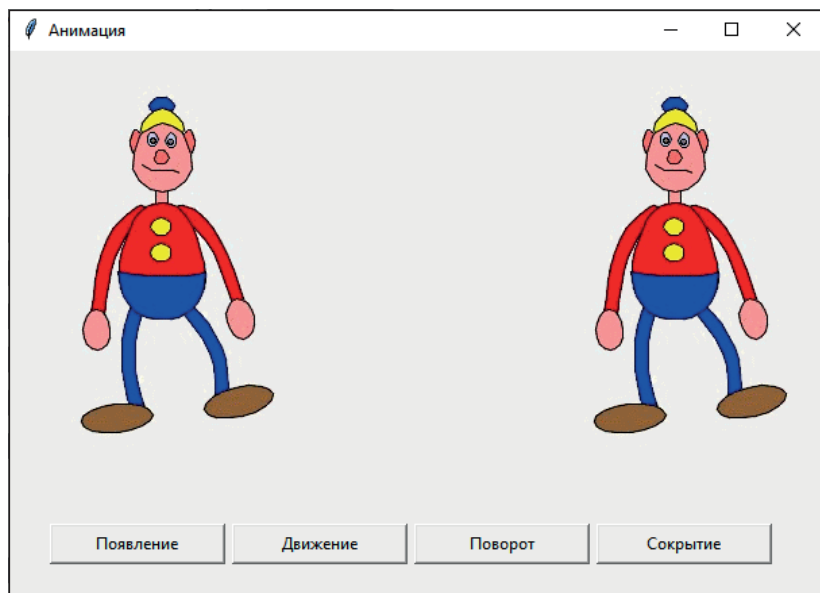


Рис. 12.10. Клонирование клоуна

Необходимо как-то предотвратить повторное генерирование фигуры до ее удаления нажатием кнопки **Скрытие**. Например, присвоив переменной `Figure` значение 0 в самом начале определения класса:

```
Figure = 0
```

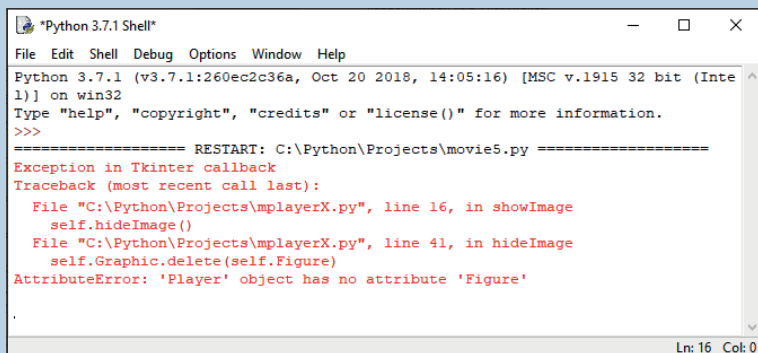
Теперь есть атрибут с именем `Figure`, но, разумеется, нет самого объекта фигуры. Тем не менее здесь также есть метод `hideImage()`, поэтому я дополню функцию `showImage()` (\Rightarrow `mplayerx.py`):

```
def showImage(self, x, y, Nr) :  
    self.hideImage()  
    self.Figure = self.Graphic.create_image(x, y, \  
        image=self.Picture[Nr])
```

Сначала существующая фигура удаляется, прежде чем будет создана снова. Это гарантирует, что никогда не появится двух клоунов одновременно, независимо от того, сколько раз вы нажмете **APPEAR**.

Главное, что существующая цифра удаляется или удаляется, а затем снова воссоздается. Это гарантирует, что никогда не будет двух символов одновременно, независимо от того, сколько раз ты нажал кнопку **Появление**.

А как быть, если функция `showImage()` вызывается в первый раз? В этом случае фигура еще не существует, чтобы она была удалена методом `hideImage()`. Догадка сразу же подтверждается в сообщении об ошибке, показанном на рис. 12.11.



```
*Python 3.7.1 Shell*  
File Edit Shell Debug Options Window Help  
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
===== RESTART: C:\Python\Projects\movie5.py =====  
Exception in Tkinter callback  
Traceback (most recent call last):  
  File "C:\Python\Projects\mplayerX.py", line 16, in showImage  
    self.hideImage()  
  File "C:\Python\Projects\mplayerX.py", line 41, in hideImage  
    self.Graphic.delete(self.Figure)  
AttributeError: 'Player' object has no attribute 'Figure'  
.  
Ln: 16 Col: 0
```

Рис. 12.11. У объекта нет атрибута

Чтобы этого сбоя не было, нам нужно создать переменную `Figure` в самом начале (см. рис. 12.12):

```
Figure = 0
```



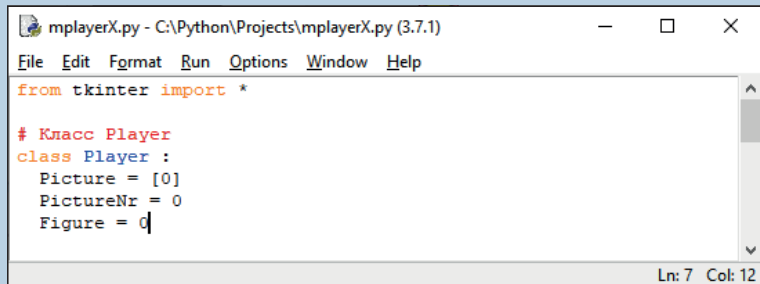


Рис. 12.12. Создаем переменную при объявлении класса

Есть еще одна нерешенная проблема. Она возникает, когда ты нажимаешь кнопку **Движение** несколько раз. Факт, что персонаж просто продолжает движение, не проблема, но в конце концов он просто исчезает.

Это можно предотвратить, отслеживая координаты фигуры. Для этого нужно знать текущее положение фигуры. Ты можешь получить его с помощью Canvas-метода `coords()`. Он возвращает координаты объекта в четырех значениях относительно вершин слева и справа. Нам нужна только одна строка:

```
self.x, self.y = self.Graphic.coords(self.Figure)
```

Точнее, нам нужен лишь `self.x`, потому что наша фигура движется только горизонтально. Нужно сделать так, чтобы фигура остановилась около правого края окна, оставаясь при этом полностью видимой:

```
if self.x <= bis+100 :
```

Здесь в ход идет «улучшенная» версия метода `moveImage()` (\Rightarrow *mplayer.py*):

```
def moveImage(self, von, bis) :
    for pos in range(von, bis, 10) :
        if self.PictureNr == 2 :
            self.PictureNr = 6
        else :
            self.PictureNr = 2
        self.Graphic.itemconfig(self.Figure, \
                                image=self.Picture[self.PictureNr])
        self.x, self.y = self.Graphic.coords(self.Figure)
        if self.x <= bis+100 :
```

```
self.Graphic.move(self.Figure, 10, 0)
self.Graphic.update()
self.Graphic.after(100)
self.Graphic.itemconfig(self.Figure, image=self.Picture[1])
```

Теперь фигура может перемещаться только в том случае, если ее позиция находится в диапазоне допустимых значений. В противном случае движение прекращается.

- Открой определение класса и измени его код. Затем убедись, что вместо класса `mplayer` импортируется класс `mplayerX`, и запусти программу. Программа работает так, как должно?

У меня теперь появляются сообщения об ошибках или предупреждения, только если в процессе работы программы я не вовремя нажимаю кнопки, особенно **Сокрытие**. Но программа не завершает работу, поэтому проблема может быть легко решена.

Подведение итогов

Вот мы и добрались до конца этой главы. Ты добавил изображения, а также анимировал фигуру и заставил ее двигаться. Давай посмотрим, что ты узнал о Python и tkinter:

<code>PhotoImage()</code>	Класс для загрузки файлов изображений
<code>create_Image()</code>	Создает объект изображения
<code>itemconfig()</code>	Принимает новое изображение (изменяет содержимое объекта изображения)
<code>update()</code>	Обновляет содержимое окна
<code>move()</code>	Перемещает объекты изображений
<code>after()</code>	Приостанавливает работу программы (собственный метод в tkinter)
<code>coords()</code>	Определяет (или задает) положение графического объекта
<code>delete()</code>	Удаляет объект (повторяюсь)
<code>file=</code>	Присваивает имя, например открываемого файла
<code>image()</code>	Изображение (из файла)

Вопрос...

1. Как добавить изображение на холст?

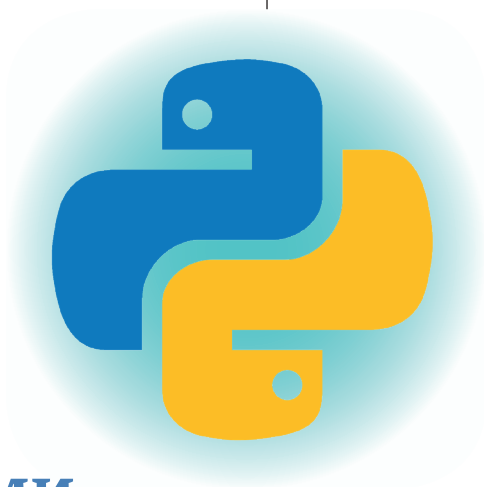
12

...и несколько задач

1. В первом примере этой главы смени класс `Circle` на `Sqaure`, и пусть квадратный объект блуждает по экрану.
2. В последней версии проекта перемести персонажа влево и назад вправо.

13

Игра с насекомыми



Существует множество возможностей программирования игр на Python. Нам даже не нужен сложный внешний модуль, чтобы превратить простое Python-приложение в настоящую графическую игру. Задачи из этой главы могут быть решены даже с помощью модуля `tkinter`. Но иногда подобный подход недостаточно элегантен, к тому же `tkinter` довольно ограничен, когда дело касается игрового программирования. Вот почему сейчас мы познакомимся с новым пакетом.

Итак, в этой главе ты узнаешь:

- ⦿ что такое пакет `pygame`;
- ⦿ кое-что о спрайтах;
- ⦿ как управлять объектами с помощью клавиш;
- ⦿ о вращении объектов;
- ⦿ об управлении границами поля.

Начало работы с Pygame

Пакет, который будет обсуждаться в этой главе, называется `Pygame`. Он состоит из нескольких модулей и основан на библиотеке для программирования игр под названием *SDL*. Эта аббревиатура расшифровывается как «*Simple DirectMedia Layer*» и переводится как «Простой мультимедийный

13

слой». Эта библиотека отвечает за управление графикой, звуком, клавиатурой и мышью. Пакет SDL изначально разрабатывался для других языков программирования, таких как С и С++, а Pygame предлагает соответствующие возможности и в Python.

Загрузка и установка этого пакета продемонстрированы в приложении А. Для моих проектов я поместил все модули в каталоге `C:\Python` во вложенные папки *include* и *Lib*. Необходимые модули должны импортироваться в соответствующую программу.

- Если у тебя уже установлен пакет `pygame`, создай новый пустой файл. Укажи в нем первую строку:

```
import pygame
```

Это необходимо, чтобы Pygame инициализировался со всеми его модулями. Далее нам нужно создать окно:

```
pygame.display.set_mode((600, 400))
```

Это выполняется с помощью метода `set_mode()`, который относится к классу `display`.



Двойные круглые скобки обязательны, поскольку метод `set_mode()` принимает значения не как два отдельных параметра, а как связанные значения в качестве параметра.

Указанных строк достаточно, чтобы создать окно. Однако программа еще не закончена. Для завершения необходима последняя строка:

```
pygame.quit()
```

Если ты запустишь эту программу, на мгновение откроется окно и тут же исчезнет. Для решения проблемы нам нужно что-то, способное заморозить окно, пока мы не закроем его сами. Для этого мы используем следующий цикл:

```
running = True
while running :
    for event in pygame.event.get() :
        if event.type == pygame.QUIT :
            running = False
```

Здесь вновь используется логическая переменная (или переменная переключения). Ранее такая переменная использовалась в программе-лотерее. Напомню, что логические переменные принимают только значения True или False, поэтому переключаются при смене значения. Логические переменные также хорошо подходят для создания условий.

В моем примере в логической переменной используется значение true. Пока оно остается таким, выполняется цикл while, внутри которого вложен еще один цикл, for:

```
for event in pygame.event.get() :
```

Каждое возникающее событие последовательно извлекается из цепочки событий, и проверяется, является ли оно событием pygame.QUIT. Оно означает: окно Pygame должно быть закрыто:

```
if event.type == pygame.QUIT :  
    running = False
```

Если это событие обнаруживается, присваивается значение False, а цикл while больше не выполняется. Только тогда выполняется pygame.quit(), программа завершает работу.

Если ты считаешь, что этот цикл похож на mainloop() из модуля tkinter, то ты прав. Разница в том, что хотя нам пришлось самостоятельно написать код этого цикла, мы можем определить, что находится в данном цикле.



Теперь мы соберем все вместе и создадим небольшую про-мо-программу (\Rightarrow game0.py):

```
# Импорт Pygame  
import pygame  
  
# Запуск Pygame, создание окна  
pygame.init()  
pygame.display.set_mode((600, 400))  
  
# Цикл событий  
running = True  
while running :  
    for event in pygame.event.get() :
```

```
if event.type == pygame.QUIT :  
    running = False  
  
# Завершение Pygame  
pygame.quit()
```

- Введи весь показанный код в своем файле, а затем запусти программу.

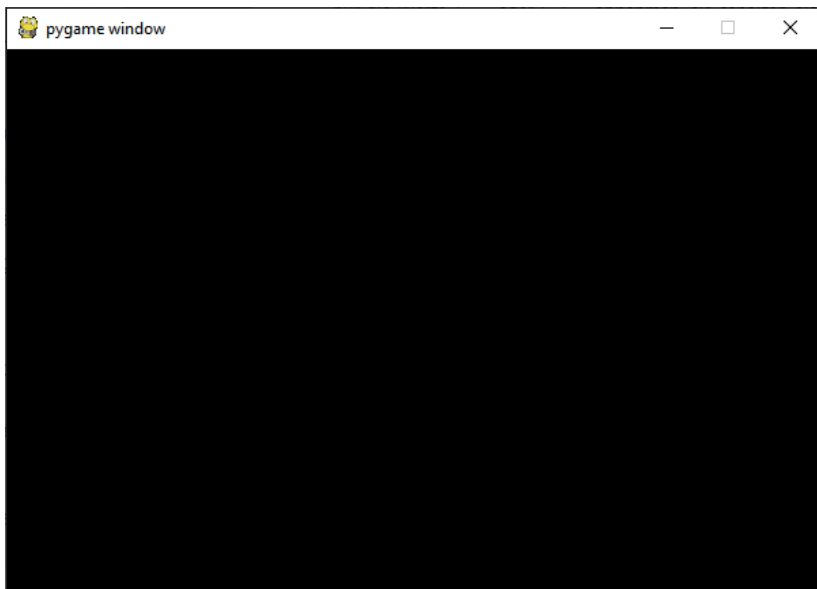


Рис. 13.1. Окно *pygame*

В отличие от *tkinter*, окно окрашено в черный цвет. Его, разумеется, можно изменить. Как и в случае с *tkinter*, ты можешь отображать в этом окне графику, но возможности этим не исчерпываются. Ты можешь запускать настоящие игры. И мы это сейчас рассмотрим.

Создаем в окне объект

Начнем с простого. Давай создадим круг, как мы это делали с *tkinter*. Программа выглядит следующим образом (⇒ *game1.py*):

```
# Импорт Pygame  
import pygame
```

```

# Определение цветов
Red = (255,0,0)
Green = (0,255,0)

# Запуск Pygame, создание окна
pygame.init()
Window = pygame.display.set_mode((600, 400))
Window.fill(Green)

# Цикл событий
running = True
while running :
    for event in pygame.event.get() :
        if event.type == pygame.QUIT :
            running = False
    # Рисование круга
    Circle = pygame.draw.circle(Window, Red, (300, 200), 100)
    pygame.display.update()

# Завершение Pygame
pygame.quit()

```

Чтобы использовать оттенки цвета в Pygame, нам нужно указывать их значения по модели RGB, как показано в табл. 13.1.

Таблица 13.1. Примеры цветов и их значения

Red = (255,0,0)	Красный
Green = (0,255,0)	Зеленый
Blue = (0,0,255)	Синий
Cyan = (0,255,255)	Бирюзовый
Magenta = (255,0,255)	Фиолетовый
Yellow = (255,255,0)	Желтый
White = (255,255,255)	Белый
Black = (0,0,0)	Черный

Любой оттенок получается указанием значений насыщенности трех основных цветов, красного, зеленого и синего. Допускаются значения в диапазоне от 0 до 255. Так можно создать любой из многих миллионов оттенков.

Метод `display.set_mode()` имеет возвращаемое значение, которое я присвоил переменной `Window`. Также мы создаем объект типа `Surface`. Это своего рода контейнер для получения и отображения изображений (аналогично компоненту `Canvas` в `tkinter`).

Вместо черного я заполню окно зеленым цветом:

```
Window.fill(Green)
```

Представление объекта содержится в цикле `while`. Сначала рисуется круг:

```
pygame.draw.circle(Window, Red, (300, 200), 100)
```

В данном примере параметры – это окно, в котором нужно создать круг, его цвет, центральная точка (пара значений) и радиус.

Затем этот метод нужно вызывать, иначе ничего не получится:

```
pygame.display.update()
```

Метод `update()` отвечает за обновление содержимого окна. Чтобы это происходило постоянно, его вызывает цикл `while`.



Думаю, сейчас стоит упомянуть о *рендеринге*: когда отображается графический объект, он *рендерится*. Объектом или сценой игры изначально является только (невидимая) модель, а при рендеринге она становится видна. Скорость рендеринга зависит от возможностей графической подсистемы компьютера.

Повторный вызов метода обновления в цикле необходим, так как сцена может постоянно меняться и необходимо отображать новое содержимое.

➤ Теперь проверь свою программу и запусти ее (рис. 13.2).

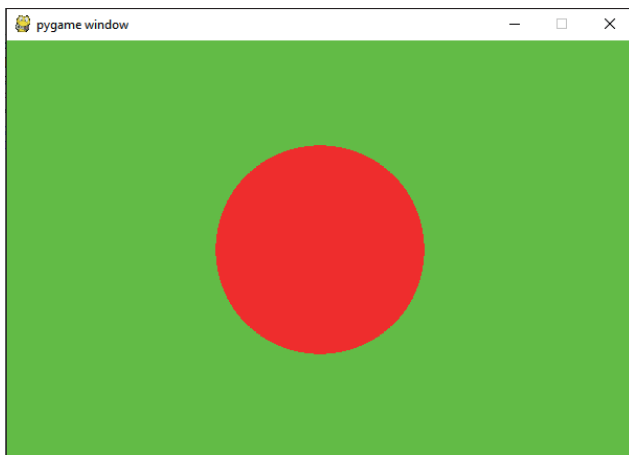


Рис. 13.2. Вид запущенной программы

Разумеется, Pygame способен рисовать и другие объекты, не только круги. Ты можешь попробовать, заменив слово `circle` одним из следующих значений:

```
pygame.draw.rect(Window, Blue, (100, 50, 400, 100))
pygame.draw.ellipse(Window, Yellow, (100, 250, 400, 100))
pygame.draw.line(Window, Black, (50,50), (550,350), 5)
```

Так ты можешь рисовать прямоугольники, эллипсы и линии. Параметры перечислены в табл. 13.2 (включая круг).

Таблица 13.2. Параметры методов для разных кругов

Метод	Параметры
<code>line()</code>	Окно, цвет, (начальная точка, конечная точка), толщина линии
<code>rect()</code>	Окно, цвет, (x, y, ширина, высота), толщина линии
<code>ellipse()</code>	Окно, цвет, (x, y, ширина, высота), толщина линии
<code>circle()</code>	Окно, цвет, (x, y, радиус, толщина линии)

Параметр «толщина линии» можно не указывать. В таком случае фигуры полностью заполняются цветом. Если указать значение толщины линии, например 1, будет нарисована только цветная граница фигуры без заливки.

Если ты взглянешь на окно программы с кругом (или другой фигурой), то увидишь основные игровые элементы: игровое поле и фигуру. Объект `window` – это игровое поле, на котором может выполняться все игровое действие. А объект `Circle` – это игровой персонаж, только очень примитивный:

```
Circle = pygame.draw.circle(Window, Red, (300, 200), 100)
```

То есть можно сказать, что ты можешь поиграть с кругом (объектом `Circle`). Объект становится настоящим игровым персонажем, когда начинает двигаться. К этому мы вернемся чуть позже.

Насекомое в качестве персонажа

Прежде всего мы должны заменить круг чем-то более «симпатичным» (и поменьше размером). Для этого в Pygame есть класс `Sprite`. Графический объект этого типа предоставляет собой прямоугольную область, которая изначально пуста. Ты можешь поместить в нее любую графику и даже загрузить изображение.

13



Спрайт подобен плитке, которая изначально прозрачна. Ты можешь «приклеить» спрайт к шаблону в загруженном графическом файле. Так ты получишь так называемую текстуру.

Текстура также может изображать, например, животное или человека (или его часть, например лицо).

Лучший способ определить класс для нашего нового персонажа показан ниже (\Rightarrow *game2.py*):

```
class Player(pg.sprite.Sprite) :
    def __init__(self) :
        super().__init__()
        self.image = pg.image.load("Bilder\Insekt1.png")
```

Этот класс получен из спрайта Pygame. В скобках обычно указывается строка `pygame.sprite.Sprite`, но я воспользовался возможностью сократить имя модуля.

Если ты тоже так хочешь поступить, измени строку импорта следующим образом:

```
import pygame as pg
```

Почему бы нам просто не написать инструкцию импорта, которую мы знаем по `tkinter`?

```
from pygame import *
```

Разумеется, эта инструкция работает, но такой подход не всегда целесообразен. При разработке сложных программ со многими импортируемыми модулями (а игры могут быть действительно большими) всегда полезно знать, какой метод к какому модулю относится. Нередко существуют одинаковые методы, которые нужно уметь отличать. Поэтому гейм-дизайнеры обычно рекомендуют импортировать Pygame так:

```
import pygame
```

Чтобы ты мог сокращать часто используемые (длинные) слова, ты можешь воспользоваться строкой

```
import pygame as pg
```

И ты увидишь, насколько этот подход удобен, потому что Pygame в основном используется для программирования игр.



Вернемся к классу `Player`. На данный момент есть только один метод для инициализации спрайта. Во-первых, он ссылается на тот же метод родительского класса:

```
super().__init__()
```

Для этого используется вспомогательное слово `super`. Так ты передашь спрайту все характеристики от `Rugame`. Добавьте к ним собственные, например ссылку на графический файл, которой спрайт должен быть «заполнен»:

```
self.image = pg.image.load("Bilder\Insekt1.png")
```

Метод `image.load()` загружает файл изображения, расположенный в подпапке *Bilder*. Ты уже знаком с похожим методом, изученным в главах про `tkinter`.

Разумеется, изображение с указанным именем должно находиться в папке, сообщение об ошибке не появится, если файл не будет найден. Я выбрал изображение жука, которого позже заставлю ползать по игровому полю.

➤ Прежде всего найди подходящее изображение и скопируй его в папку *Bilder* своего `Rugame`-проекта.

Ты должен знать, что `Rugame` поддерживает больше форматов изображений, чем `tkinter`. Здесь можно не ограничиваться GIF, и поэтому я рекомендую формат PNG, который позволяет сделать фон рисунка прозрачным.

Как сделать фон прозрачным? Если изображение сохранено в формате, отличном от PNG, тебе нужно открыть его в графическом редакторе, например `paint.net` (бесплатно) или `Photoshop` (который наверняка у тебя установлен) и сохранить в нем рисунок в виде файла формата PNG.

В программе `paint.net` выбери инструмент «Волшебная палочка» и щелкни мышью по фону, чтобы его область была выделена цветом и «бегущими муравьями». Затем нажми клавишу **Del**, и фон станет клетчатым, обозначая прозрачность (рис. 13.3).



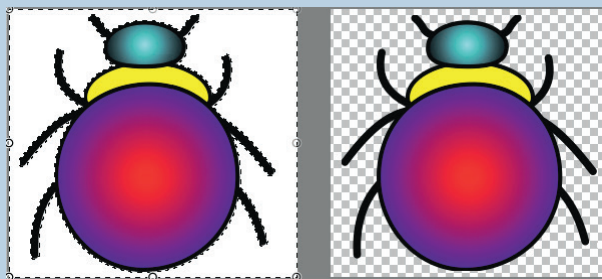
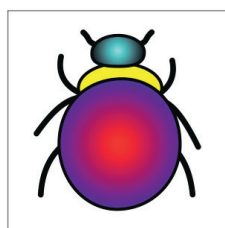


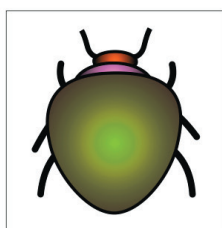
Рис. 13.3. Жук на белом фоне (слева) и прозрачном (справа)

В программе Photoshop процесс аналогичен: инструмент тоже называется «Волшебная палочка», и щелчка мышью по фону и нажатия клавиши **Del** достаточно, чтобы сделать фон прозрачным.

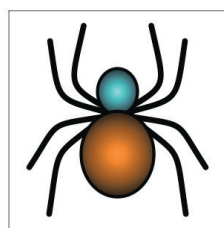
Ты также можешь использовать файлы примеров из папки, доступной по адресу dmkpress.com (рис. 13.4).



insect1.png



insect2.png



insect3.png

Рис. 13.4. Рисунки жуков для проекта

Все необходимые файлы расположены в папке *bilder*.

После того как класс `Player` готов, мы можем сразу его использовать. Итак, давай создадим объект прямо сейчас:

```
Figure = Player()
```

Как спрайт отобразить в окне? Взгляни на следующую строку:

```
Window.blit(Figure.image, (250, 150))
```

Последний параметр используется для указания позиции нашего будущего игрового персонажа. Первый принимает изображение спрайта, которое будет отображаться.

В этом примере персонаж, он же жук, располагается в центре окна. Думаю, тебе понятно, что вызов метода `blit` находится в цикле `while`.

Перестрой свою программу в соответствии со следующим листингом (\Rightarrow *game2.py*):

```
# Pygame-графика
import pygame as pg

# Класс Player
class Player(pg.sprite.Sprite) :
    def __init__(self) :
        super().__init__()
        self.image = pg.image.load("Bilder\Insekt1.png")

# Определение цветов
Green = (0,255,0)

# Запуск Pygame, создание игры
pg.init()
Window = pg.display.set_mode((600, 400))
Window.fill(Green)

# Создание персонажа
Figure = Player()

# Цикл событий
running = True
while running :
    for event in pg.event.get() :
        if event.type == pg.QUIT :
            running = False
    # Позиционирование спрайта в окне
    Window.blit(Figure.image, (250, 150))
    pg.display.update()

# Завершение Pygame
pg.quit()
```

Запусти программу, и ты увидишь спрайт в центре окна (рис. 13.5).

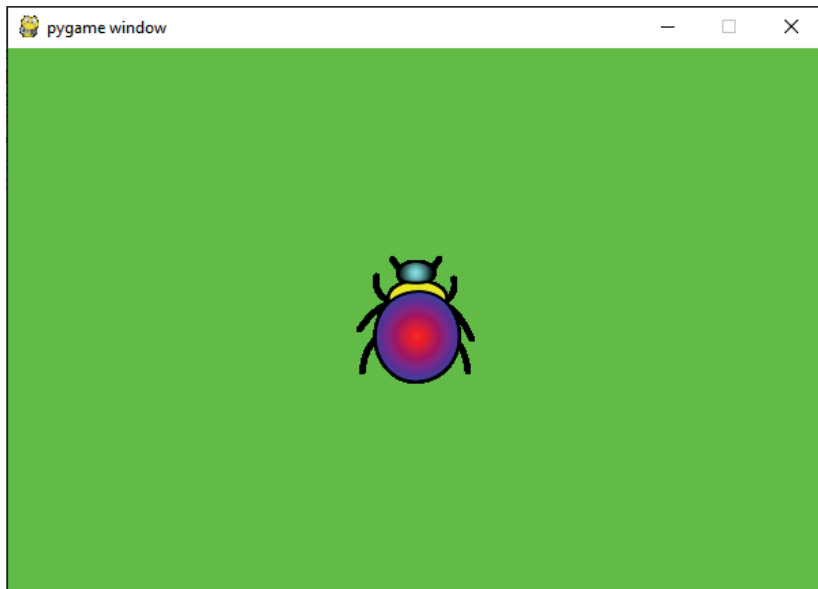


Рис. 13.5. Наш милый жучок

Как видишь, фон изображения жука прозрачен. Если бы это было не так, тогда ты увидел бы белый фон, как показано на рис. 13.6 (слева).

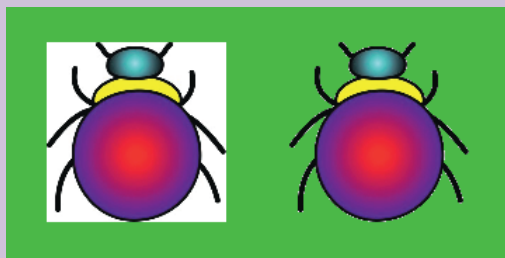


Рис. 13.6. Изображение жука с непрозрачным фоном (слева) и прозрачным (справа)

Слева показан жук из файла формата GIF или JPG, а справа – из PNG.



Управление персонажем

Зеленый лужок – это игровое поле, а жук – игровой персонаж. Мы должны сейчас как-то заставить его двигаться. Лучший способ управления – использовать клавиши со стрелками.

Для этого нам нужно создать следующее событие в цикле `for`:

```
if event.type == pg.KEYDOWN :
```

Теперь важно сопоставить действие с нажатой клавишей (в коде выше – «Стрелка вниз» – `Keydown`). Но что именно должно произойти, если ты нажмешь одну из клавиш со стрелками? Жук (т. е. спрайт) должен изменить свое положение. Во-первых, нам понадобятся две переменные, которые отвечают за позицию персонажа. Для этого мы расширим определение класса `Player` (\Rightarrow *buggy1*):

```
class Player(pg.sprite.Sprite) :  
    def __init__(self, xPos=0, yPos=0) :  
        super().__init__()  
        self.image = pg.image.load("Bilder\Insekt1.png")  
        self.x, self.y = xPos, yPos
```

Я переименовал проект в *buggy* в честь английского слова «баг», т. е. «ошибка».



Чтобы создать объект `Player`, воспользуемся уже знакомым способом:

```
Figure = Player()
```

Затем атрибуты объекта `Player` устанавливаем равными 0. Параметры `xPos` и `yPos` опциональны, ты можешь их использовать, а потом переданные значения передаются как положение по осям *x* и *y*. В нашем случае мы используем следующую строку кода:

```
Figure = Player(250, 150)
```

Теперь атрибуты имеют новые значения. И они увеличиваются или уменьшаются в зависимости от нажатой клавиши:

```
if event.key == pg.K_LEFT :  
    Figure.x -= 5  
if event.key == pg.K_RIGHT :  
    Figure.x += 5  
if event.key == pg.K_UP :
```

```
Figure.y -= 5
if event.key == pg.K_DOWN :
    Figure.y += 5
```

Теперь вызов метода `blit` изменяется следующим образом:

```
Window.blit(Figure.image, (Figure.x, Figure.y))
```

К сожалению, по-прежнему есть, по крайней мере, два упущения, которые ты увидишь, если запустишь программу:

- жук перемещается на 5 пикселей в одном направлении, но тебе нужно отпускать и вновь нажимать клавишу для каждого последующего движения;
- спрайт оставляет след, как показано на рис. 13.7.

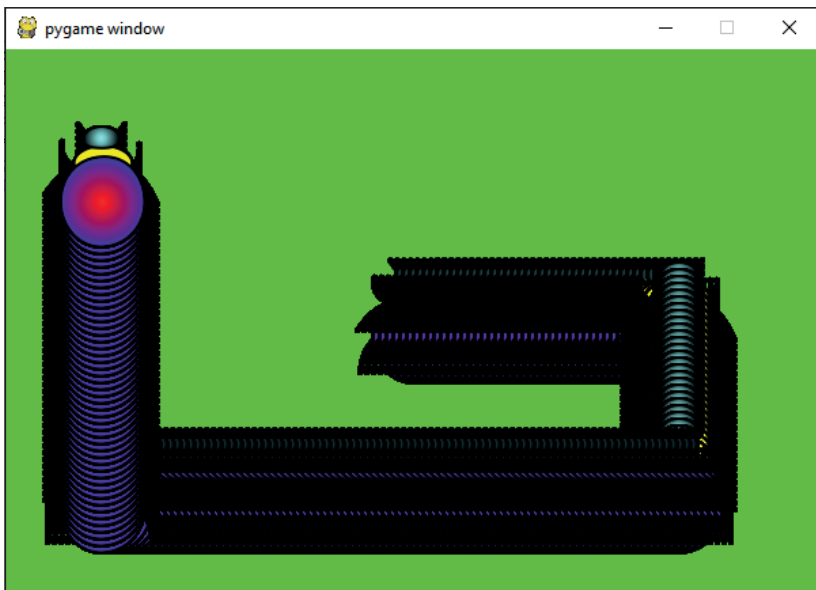


Рис. 13.7. Во время движения жук копируется

Последнее упущение можно легко исправить, переместив показанную ниже строку:

```
Window.fill(Green)
```

Теперь жук может перемещаться свободно, но нам по-прежнему нужен вариант повторения события при удержании клавиши. За это отвечает следующая инструкция:

```
pg.key.set_repeat(20,20)
```

Эта строка должна быть вверху, желательно прямо под кодом `pg.init().set_repeat()`. Она определяет продолжительность повторения и интервал между двумя повторениями (поэкспериментируй с другими значениями!).

Теперь ты можешь создать рабочую программу. Для этого напиши следующий исходный код (\Rightarrow *buggy1.py*):

```
# Pygame-графика
import pygame as pg

# Класс Player
class Player(pg.sprite.Sprite) :
    def __init__(self, xPos=0, yPos=0) :
        super().__init__()
        self.image = pg.image.load("Bilder\Insekt1.png")
        self.x, self.y = xPos, yPos

# Установка начальных значений
Green = (0,255,0)

# Запуск Pygame, создание игрового поля и персонажа
pg.init()
pg.key.set_repeat(20,20)
Window = pg.display.set_mode((600, 400))
Figure = Player(250, 150)

# Цикл событий
running = True
while running :
    for event in pg.event.get() :
        if event.type == pg.QUIT :
            running = False
    # Установка клавиш
    if event.type == pg.KEYDOWN :
        if event.key == pg.K_LEFT :
            Figure.x -= 5
        if event.key == pg.K_RIGHT :
            Figure.x += 5
        if event.key == pg.K_UP :
            Figure.y -= 5
        if event.key == pg.K_DOWN :
            Figure.y += 5

# Положение спрайта в окне (новое)
Window.fill(Green)
Window.blit(Figure.image, (Figure.x, Figure.y))
```

```
pg.display.update()
```

```
# Завершение Pygame  
pg.quit()
```

- Запусти программу и нажимай клавиши со стрелками для перемещения жука (рис. 13.8).

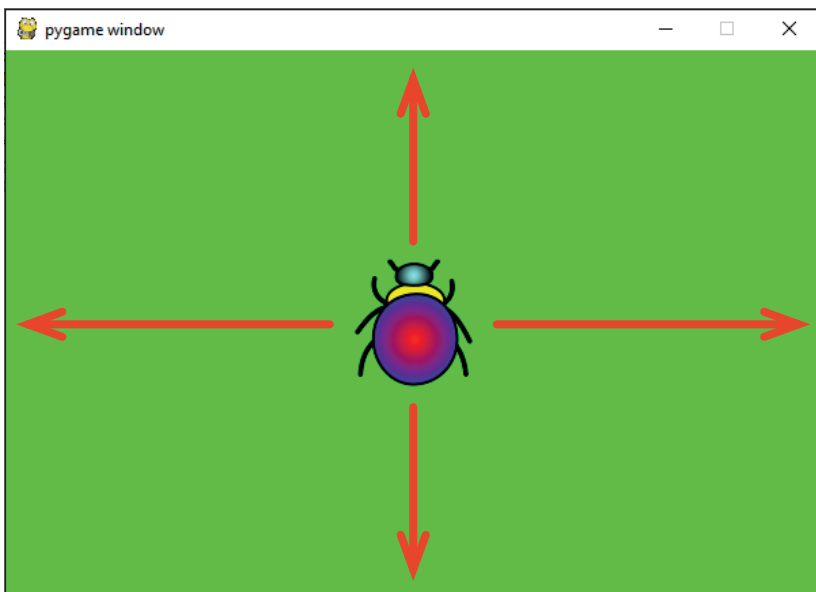


Рис. 13.8. Жук двигается в четырех направлениях

Существует интересная альтернатива части кода, отвечающей за обработку нажатий клавиш (\Rightarrow *buggy1a.py*):

```
keys = pg.key.get_pressed()  
if keys[pg.K_LEFT]:  
    Figure.x -= 5  
if keys[pg.K_RIGHT]:  
    Figure.x += 5  
if keys[pg.K_UP]:  
    Figure.y -= 5  
if keys[pg.K_DOWN]:  
    Figure.y += 5
```

В этом коде нажатые клавиши собираются в список и обрабатываются, т. е. ты также можешь одновременно нажать две клавиши со стрелками для перемещения жука по диагонали.



Поворот персонажа

Теперь ты можешь перемещать своего персонажа, жука, в горизонтальном и вертикальном направлениях. Что вполне естественно. Но меня беспокоит положение жука. Он продолжает смотреть вверх, независимо, в каком направлении движется. Это как-то неестественно, хотя, может, некоторые насекомые и бегают так.

Чтобы решить эту проблему для нашего жука, нам нужно научиться поворачивать спрайт. За это отвечает функция `rotate()`, которая принимает в качестве параметра значение угла (в градусах).

Функция `rotate()` относится к группе `transform`. Существует несколько других полезных методов, показанных в табл. 13.3.

Таблица 13.3. Методы трансформации

<code>transform.scale</code> (файл, (ширина, высота))	Изменение размера изображения
<code>transform.flip</code> (файл, xBool, yBool)	Зеркальное отражение по горизонтальной или вертикальной оси (true = да, false = нет)
<code>transform.rotate</code> (файл, угол)	Поворот изображения



Давай поместим функцию вращения в новый метод для класса `Player` (\Rightarrow `buggy2.py`):

```
def rotate(self, degree):
    self.Bild = pg.transform.rotate(self.image, degree)
```

Я использовал новый атрибут, который принимает существующее изображение, а затем поворачивает его, сохраняя оригинал (рис. 13.9).

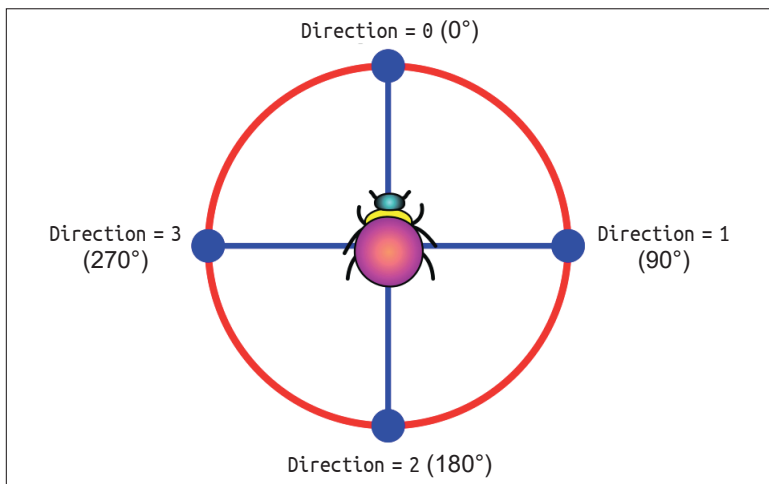


Рис. 13.9. Атрибут для поворота жука

Вращение выполняется с шагом в 90° . Значение глобальной переменной `Direction` устанавливается равным 0 при запуске программы (рис. 13.10).

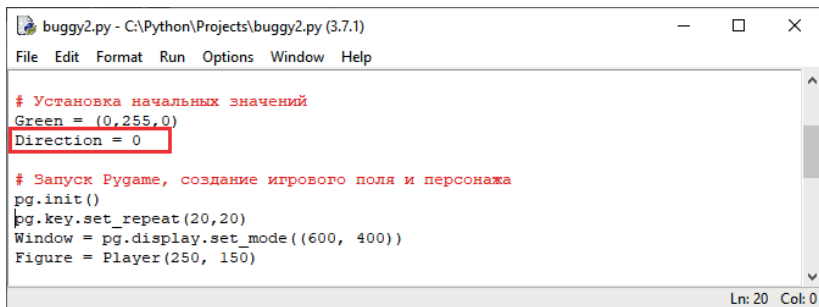


Рис. 13.10. Исходное значение переменной `Direction`

Затем она принимает значения от 0 до 3, если ты разместишь инструкции следующим образом (\Rightarrow `buggy2.py`):

```
if event.key == pg.K_LEFT :
    Direction = 1
    Figure.x -= 5
if event.key == pg.K_RIGHT :
    Direction = 3
    Figure.x += 5
if event.key == pg.K_UP :
    Direction = 0
    Figure.y -= 5
```

```
if event.key == pg.K_DOWN :  
    Direction = 2  
    Figure.y += 5
```

Далее рисунок поворачивается соответствующим образом, чтобы жук теперь смотрел в ту сторону, куда он ползет:

```
Figure.rotate(Direction*90)
```

Чтобы все заработало, требуется внести еще несколько изменений. Первое относится к классу Player (\Rightarrow *buggy2.py*):

```
def __init__(self, xPos=0, yPos=0) :  
    super().__init__()   
    self.image = pg.image.load("Bilder\Insekt1.png")  
    self.x, self.y = xPos, yPos  
    self.Bild = self.image
```

После того как изображение было создано, оригинал остается нетронутым, а с этого момента используется только копия. Поэтому вызов метода blit также изменяется (рис. 13.11):

```
Window.blit(Figure.Bild, (Figure.x, Figure.y))
```

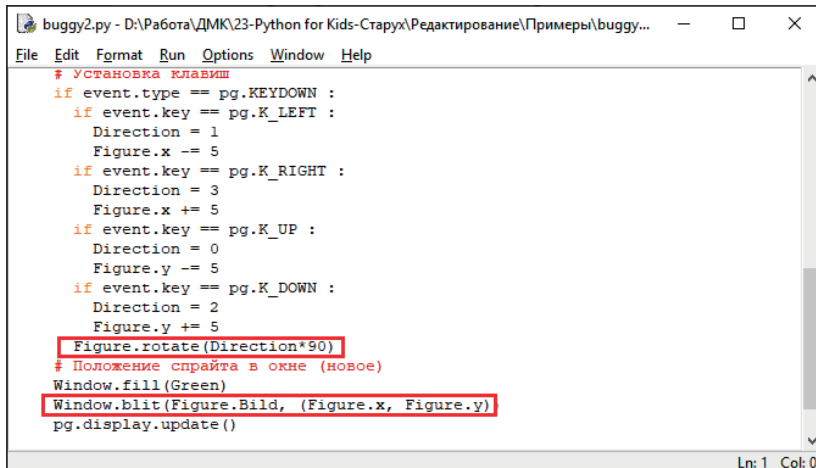


Рис. 13.11. Изменения в коде программы



Мы не вращаем оригинал по двум причинам. Во-первых, мы можем каждый раз начинать вращение с одного и того же исходного угла.

Во-вторых, любое вращение может привести к потере качества изображения. То же самое относится к масштабированию. Если потом использовать результат такого преобразования, качество изображения может ухудшаться все больше и больше. Это нежелательно, поэтому лучше сохранить оригинал нетронутым.

- Запусти программу и поуправляй жуком (рис. 13.12). Правильно ли он поворачивается?

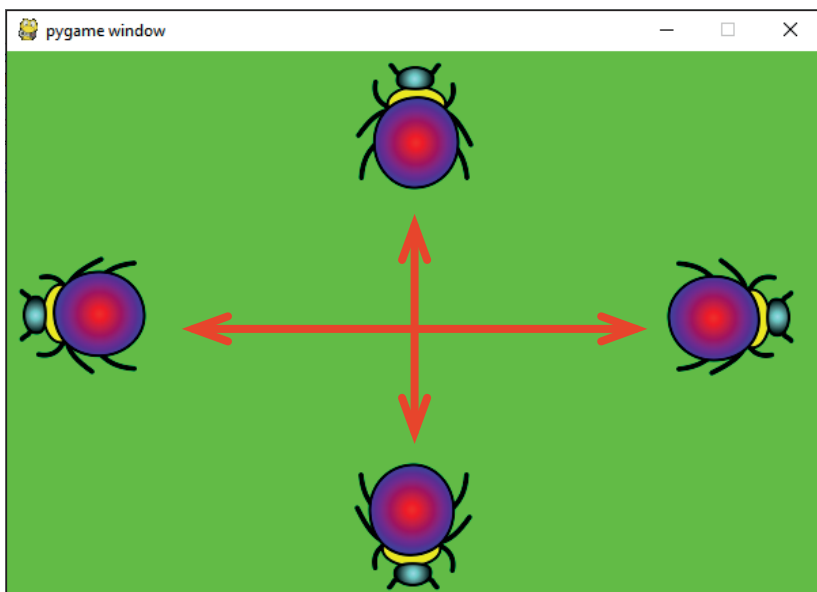


Рис. 13.12. Теперь жук поворачивается «лицом» по направлению движения

Отслеживание границ игрового поля

Обратимся теперь к теме «покидание поля»: если ты хочешь, чтобы жук оставался в пределах видимости, нужны ограничения и соответствующие управляющие конструкции.

Прежде всего нам нужны размеры игрового поля, которые я сейчас откорректирую:

```
xMax, yMax = 600, 400
```

Конечно, ты можешь использовать другие значения. Эти переменные могут быть использованы при создании окна и персонажа:

```
Window = pg.display.set_mode((xMax, yMax))  
Figure = Player(xMax/2-50, yMax/2-50)
```

Но и для наших границ мы можем использовать переменные `xMax` и `yMax`. При этом мы проверяем позицию жука после каждого «шага», чтобы убедиться, что он все еще находится в игровом поле (\Rightarrow *buggy3.py*):

```
if event.key == pg.K_LEFT :  
    Direction = 1  
    if Figure.x > 0 :  
        Figure.x -= 5  
if event.key == pg.K_RIGHT :  
    Direction = 3  
    if Figure.x < xMax-100 :  
        Figure.x += 5  
if event.key == pg.K_UP :  
    Direction = 0  
    if Figure.y > 0 :  
        Figure.y -= 5  
if event.key == pg.K_DOWN :  
    Direction = 2  
    if Figure.y < yMax-100 :  
        Figure.y += 5
```

- Измени исходный код соответствующим образом, затем запусти программу и начни управлять жуком в пределах окна программы.

Возможно, у тебя не получится использовать значение 100, поэтому придется изменить его, чтобы соответствовать спрайту в программе. У меня прямоугольник с изображением жука имеет толщину границы 100 пикселей.

А если ты предпочитаешь, чтобы жук останавливался на некотором расстоянии от края окна программы, тогда также нужно использовать другое значение вместо 0.



13

Подведение итогов

Проект еще не закончен, он станет настоящей игрой, но не сейчас. Уже в следующей главе мы продолжим. В этой главе ты научился управлять объектом с помощью клавиатуры и удерживать жука в пределах окна программы. Вот что ты узнал нового о Pygame:

<code>pygame</code>	Пакет для графики и игрового программирования
<code>init()</code>	Запуск pygame, инициализация всех модулей/коллекций
<code>quit()</code>	Выход из pygame
<code>display</code>	Группа методов для управления окном и экраном
<code>set_mode()</code>	Установка окна просмотра (в т. ч. и размера)
<code>update()</code>	Обновление окна
<code>Surface</code>	Класс для отображения изображений
<code>blit()</code>	«Перерисовка» изображения (и, таким образом, перекрытие исходного)
<code>fill()</code>	Заполнение области выбранным цветом
<code>draw</code>	Методы для рисования графических форм
<code>circle()</code>	Создание круга
<code>Ellipse()</code>	Создание эллипса
<code>line()</code>	Создание прямой линии
<code>rect()</code>	Создание прямоугольника
<code>sprite</code>	Набор методов для спрайтов
<code>Sprite</code>	Класс спрайта
<code>image.load()</code>	Загрузка файла изображения
<code>transform</code>	Методы трансформации, среди прочего, для поворота и масштабирования объектов
<code>flip()</code>	Зеркальное отражение объекта
<code>rotate()</code>	Вращение объекта
<code>scale()</code>	Увеличение/уменьшение объекта
<code>key.get_pressed()</code>	Регистрация событий нажатия клавиш
<code>key.set_pressed()</code>	Повтор нажатий клавиш

А также и в самом Python есть некоторые вещи, с которыми ты столкнулся впервые:

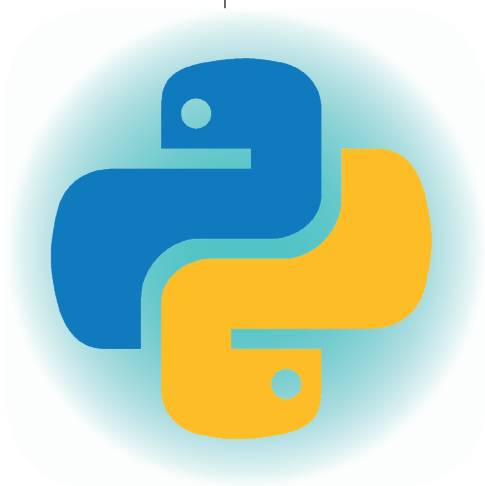
<code>super</code>	Вызов родительского метода
<code>event.type</code>	Тип события
<code>import-as</code>	Импорт модуля под другим именем

Несколько вопросов...

1. Почему нужно использовать изображения с прозрачным фоном для спрайтов?
2. Может ли `event.type` использоваться для обработки нажатий клавиш с буквами?

...и задача

1. Сделай так, чтобы жук появлялся на противоположной стороне окна, а не просто останавливался на краю.



14

Как раздавить жука

В предыдущей главе мы использовали клавиши со стрелками для перемещения жука по полю. В этой главе мы научимся применять для этого мышь, помимо клавиатуры. А еще мы расширим правила: мы разрешим насекомому свободно перемещаться без какого-либо контроля с нашей стороны.

Итак, в этой главе ты узнаешь:

- ⦿ как определить положение указателя мыши;
- ⦿ кое-что о Пифагоре и арктангенсах;
- ⦿ как перемещать фигуру под углом;
- ⦿ как «ловить» объекты мышью;
- ⦿ как создать еще один модуль `Player`.

Разбираемся с мышью

До сих пор жук управлялся нажатием одной из клавиш, но он также может ползти туда, где мы щелкнули мышью. В нашем следующем проекте нам не нужно управление с клавиатуры. Поэтому ты можешь создать новый документ и скопировать исходный код, но, если очень хочешь, также можешь добавить код, отвечающий за управление с помощью мыши, в предыдущий проект.

Давай возьмем код предыдущей программы и удалим все события, связанные с клавиатурой. Теперь речь пойдет о щелчках мыши. Положение указателя мыши должно определяться:

```
if event.type == pg.MOUSEBUTTONDOWN :  
    (xPos, yPos) = pg.mouse.get_pos()  
    Figure.x = xPos - 50  
    Figure.y = yPos - 50
```

Обязательное условие – модуль `pygame` импортируется следующим образом:

```
import pygame as pg
```

Так как нажатие кнопки мыши – это событие (`Mousebuttondown`), то пара значений определяется с помощью метода `mouse.get_pos()`. Так как параметры `xPos` и `yPos` указаны в круглых скобках, они образуют пару переменных, значения которых с корректировкой присваиваются переменным `Figure.x` и `Figure.y`, чтобы спрайт жука находился чуть ниже указателя мыши.

Такое обозначение называется *вектором* – структурой, содержащей два (и более) числа. Это могут быть, например, координаты местоположения или длина и ширина прямоугольной области. Кроме того, группу из четырех значений, например, встречающихся при обозначении параметров прямоугольников и эллипсов, ты тоже можешь, по сути, называть вектором, но обычно ее называют типом `Rect` (от англ. *rectangle* – прямоугольник).



- Введи следующий код или измени старый. Затем запусти программу и щелкни где-нибудь в окне (\Rightarrow *buggy4.py*):

```
import pygame as pg  
  
# Класс Player  
class Player(pg.sprite.Sprite) :  
    def __init__(self, xPos=0, yPos=0) :  
        super().__init__()   
        self.image = pg.image.load("Bilder\Insekt1.png")  
        self.x, self.y = xPos, yPos  
        self.Bild = self.image  
    def rotate(self, degree) :  
        self.Bild = pg.transform.rotate(self.image, degree)
```

```
# Установка начальных значений
Green = (0,255,0)
xMax, yMax = 600, 400

# Запуск Pygame, создание игрового поля и персонажа
pg.init()
pg.key.set_repeat(20,20)
Window = pg.display.set_mode((xMax, yMax))
Figure = Player(xMax/2-50, yMax/2-50)

# Цикл событий
running = True
while running :
    for event in pg.event.get() :
        if event.type == pg.QUIT :
            running = False
    # Запрос мыши
    if event.type == pg.MOUSEBUTTONDOWN :
        (xPos, yPos) = pg.mouse.get_pos()
        Figure.x = xPos - 50
        Figure.y = yPos - 50

# Положение спрайта в окне (новое)
Window.fill(Green)
Window.blit(Figure.Bild, (Figure.x, Figure.y))
pg.display.update()

# Завершение Pygame
pg.quit()
```

Я оставил метод, отвечающий за направление жука, потому что он понадобится нам позже.

Никуда без математики

Пока все идет нормально. Или нет. Потому что бедное насекомое не должно постоянно только прыгать. Мы хотим, чтобы жук приползал к месту назначения (и не торопился). Чтобы достичь этого, нам нужно определить путь, точнее: вычислить. Вот почему здесь понадобится математика. Сначала мы определяем расстояние от позиции жука до позиции щелчка мыши (рис. 14.1).

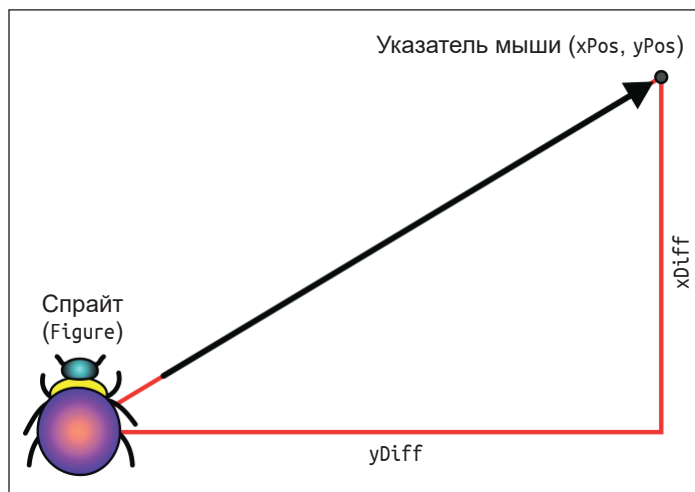


Рис. 14.1. Траектория движения жука

Однако это невозможно сделать сразу, сначала нам нужно выяснить разницу между координатами x и y фигуры и указателем мыши:

```
xDiff = xPos - Figure.x - 50  
yDiff = yPos - Figure.y - 50
```

Ты можешь подумать, что значение переменной можно записать, скажем, так: $\text{Figure.x} - \text{xPos}$. И все равно значение будет то отрицательным, то положительным. Это не важно, потому что впоследствии нам понадобится возводить эти значения в квадрат, а в этом случае результат всегда положительный.



Переменные xDiff и yDiff определяют количество пикселей, на которое объект класса `Player` должен передвигаться горизонтально (по оси x) и вертикально (по оси y). Реальный путь, которому он следует, определяется диагональю – самой длинной стороной в этом прямоугольном треугольнике.

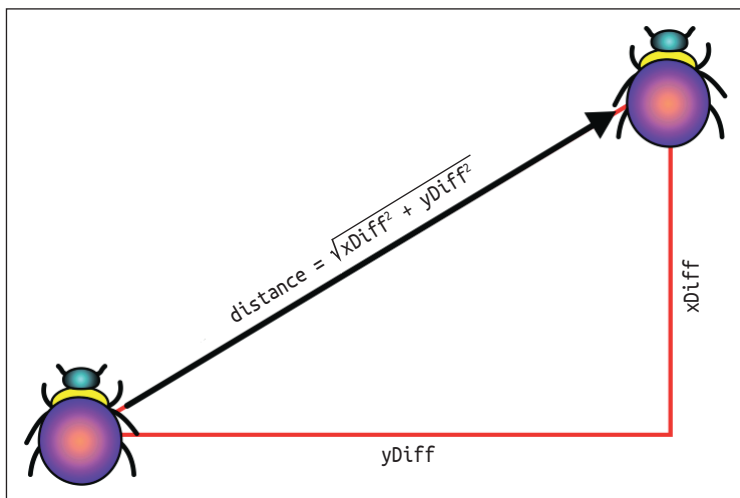


Рис. 14.2. Вычисление дистанции

Чтобы вычислить дистанцию, мы используем формулу, выведенную греческим математиком Пифагором. Для этого нам сначала нужны квадраты расстояний `xDiff` и `yDiff`, которые мы получаем следующим образом:

```
xDiff*xDiff
```

и

```
yDiff*yDiff
```

Эти два значения складываются.

```
xDiff*xDiff + yDiff*yDiff
```

И затем из суммы извлекается квадратный корень. Для этого мы используем функцию `sqrt()`:

```
distance = math.sqrt(xDiff*xDiff + yDiff*yDiff)
```

Поскольку эта математическая функция находится в отдельном модуле, мы должны сначала импортировать его. Мы сделаем это, чтобы ты мог использовать указанное выше уравнение:

```
from math import *
```

Благодаря этому можно сократить вышеприведенное уравнение так:

```
distance = sqrt(xDiff*xDiff + yDiff*yDiff)
```

Прежде чем отправить жука в путь, мы должны сначала выровнять его. Поэтому нам нужно определить подходящий угол. Опять же, математическая функция, которую может предложить Python (в модуле Math), поможет нам:

```
degree = atan2(-yDiff, xDiff)
```

Слово `atan` – аббревиатура от `arcus tangens`, арктангенс, угловая функция. Сюда же относятся синусы, косинусы и тангенсы. А `atan2()` – специальная форма функции, которая может принимать оба значения в качестве параметров. С помощью нее вычисляется угол. Важно, чтобы сначала передавалось значение по оси `y`, а затем `x`. Здесь нужно указать значение по оси `y` с минусом перед ним.

К сожалению, в результате получается угол не в градусах, а в так называемых радианах.

Если тебе нужны подробности, радианы задействуют не градусы как единицу измерения, а значение «пи» (π) (примерно 3,14). Таким образом, 360° соответствуют 2π , 180° – π , 90° – $\pi/2$.



С помощью показанной ниже функции мы преобразуем значение:

```
degree = degrees(degree) - 90
```

В этой строке еще вычитаются 90° . Только тогда направление будет правильным.

Важно, чтобы обе переменные получили начальные значения при запуске, иначе программа не будет работать:

```
distance = 0  
degree = 0
```



Выравнивание жука выполняется следующим образом:

```
Figure.rotate(degree)
```

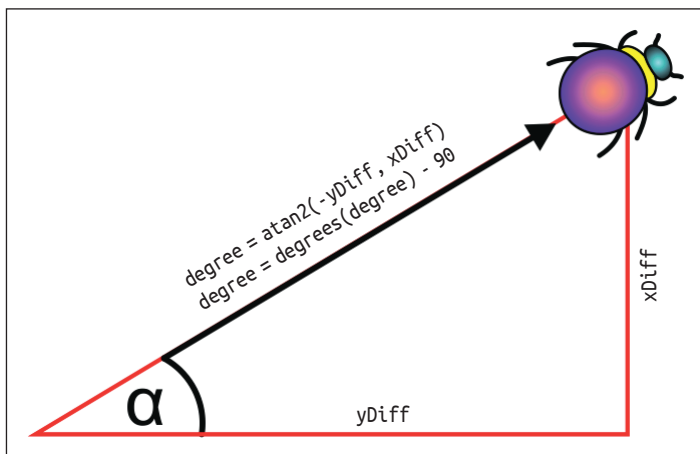


Рис. 14.3. Вычисление траектории движения жука

Теперь мы должны переместить нашего жука, чтобы он начал ползать. Для этого мы дополним класс `Player` методом перемещения (\Rightarrow `buggy5.py`):

```
def move(self, distance, xx, yy) :
    self.x += xx
    self.y += yy
    distance -= 1
    return distance
```

Параметры, которые принимает эта функция, – это шаги перемещения жука по горизонтали и вертикали (что приводит к движениям по диагонали):

```
self.x += xx
self.y += yy
```

Затем значение дистанции уменьшается на 1 и возвращается:

```
distance -= 1
return distance
```

И вот как выглядит вызов функции:

```
distance = Figure.move(distance, xDiff, yDiff)
```

Но прежде должны быть определены параметры `xDiff` и `yDiff`:

```
xDiff /= distance
yDiff /= distance
```

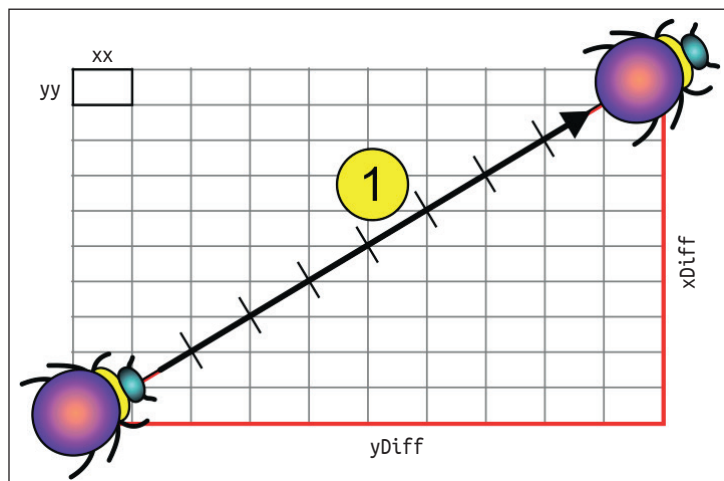


Рис. 14.4. Траектория жука вычислена

Таким образом у нас используются замечательные маленькие шажочки, которые мы передаем методу, отвечающему за перемещение. Соответствующий вызов находится вне цикла `for`. И так будет происходить множество раз, пока персонаж не окажется достаточно близко от позиции щелчка мышью:

```
if distance > 5 :
    distance = Figure.move(distance, xDiff, yDiff)
    pg.time.delay(5)
```

Кроме того, я тут же определил задержку: метод `time.delay()` создает (небольшую) паузу для отклика программы. Мне хватило 5 миллисекунд, но ты можешь попробовать другие значения.

Собираем все вместе

Чтобы ты мог проанализировать и внести изменения, я вновь привожу листинг целиком (\Rightarrow *BUGGY5.PY*):

```
import pygame as pg
from math import *
```

```

# Класс Player
class Player(pg.sprite.Sprite) :
    def __init__(self, xPos=0, yPos=0) :
        super().__init__()
        self.image = pg.image.load("Bilder\Insekt1.png")
        self.x, self.y = xPos, yPos
        self.Bild = self.image
    def rotate(self, degree) :
        self.Bild = pg.transform.rotate(self.image, degree)
    def move(self, distance, xx, yy) :
        self.x += xx
        self.y += yy
        distance -= 1
        return distance

# Установка начальных значений
Green = (0,255,0)
xMax, yMax = 600, 400
distance = 0
degree = 0

# Запуск Pygame, создание игрового поля и персонажа
pg.init()
pg.key.set_repeat(20,20)
Window = pg.display.set_mode((xMax, yMax))
Figure = Player(xMax/2-50, yMax/2-50)

# Цикл событий
running = True
while running :
    for event in pg.event.get() :
        if event.type == pg.QUIT :
            running = False
    # Запрос мыши
    if event.type == pg.MOUSEBUTTONDOWN :
        (xPos, yPos) = pg.mouse.get_pos()
        ##--
        xDiff = xPos - Figure.x - 50
        yDiff = yPos - Figure.y - 50
        distance = sqrt(xDiff*xDiff + yDiff*yDiff)
        degree = atan2(-yDiff, xDiff)
        degree = degrees(degree) - 90
        xDiff /= distance
        yDiff /= distance
        Figure.rotate(degree)

# Задержка движения
if distance > 5 :
    distance = Figure.move(distance, xDiff, yDiff)
    pg.time.delay(5)

```

```
# Положение спрайта в окне (новое)
Window.fill(Green)
Window.blit(Figure.Bild, (Figure.x, Figure.y))
pg.display.update()

# Завершение Pygame
pg.quit()
```

- Доработай свою программу и запусти ее. Теперь жук начнет ползать.

На мой взгляд, получилось неплохо. Но было бы лучше, если бы жук всегда оставался полностью видимым на поле. И ты сможешь решить эту проблему ближе к концу главы.

Свободное ползание

Сейчас мы предоставим немного свободы нашей программе. Как только программа запустится, жук начнет ползать самостоятельно.

Теперь вы умеете перемещать жука, но всегда в одном направлении. А как насчет генерации случайных чисел?

Давай попробуем. Начнем с импорта модуля Random:

```
import random
```

Ниже, непосредственно перед циклом `while`, мы сгенерируем два случайных значения переменных `xStep` и `yStep`:

```
xStep = random.randint(0,2)
yStep = random.randint(0,2)
```

С помощью метода `random.randint()` мы получаем число в диапазоне от 0 до 1. Но если мы оставим все так, как есть, жук будет либо всегда ползти в одном направлении, либо стоять на месте. Поэтому мы должны сделать, чтобы нули становились реальными значениями шага:

```
if xStep == 0 :
    xStep = -1
if yStep == 0 :
    yStep = -1
```

Теперь переменные `xStep` и `yStep` могут иметь четыре комбинации для направлений, показанные в табл. 14.1 и на рис. 14.5.

Таблица 14.1. Значения переменных xStep и yStep

	Левый верхний	Левый нижний	Правый верхний	Правый нижний
xStep	-1	-1	+1	+1
yStep	-1	+1	-1	+1

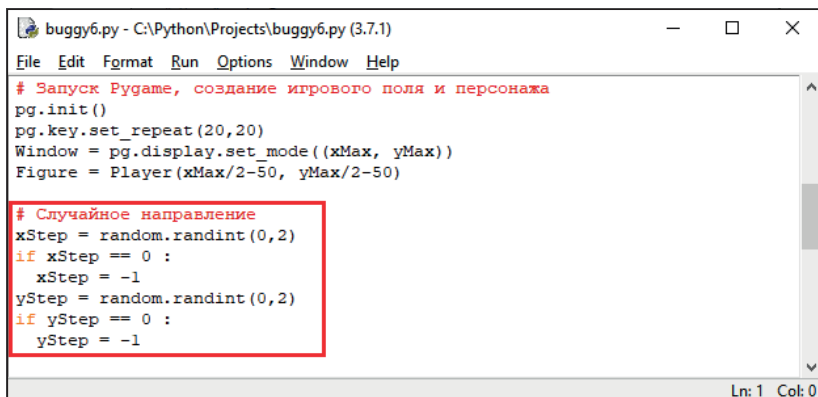


Рис. 14.5. Значения переменных xStep и yStep

Прежде чем мы позволим жуку ползти, мы должны установить пределы его движения. Как это работает, ты узнал из прошлой главы. Но здесь код выглядит по-другому (\Rightarrow *buggy6.py*):

```

if (Figure.x < 0) or (Figure.x > xMax-110) :
    xStep = -xStep
if (Figure.y < 0) or (Figure.y > yMax-110) :
    yStep = -yStep
  
```

На первый взгляд сложно.

- Если из-за шага жук уходит слишком далеко влево или вправо, меняем размер шага по оси x .
- Если из-за шага жук уходит слишком далеко вверх или вниз, меняем размер шага по оси y .

«Слишком далеко» означает математически меньше (<) либо больше (>). А изменение шага означает, что жук поворачивается и ползет обратно к границе поля. Как только жук выходит на границу поля, он просто меняет свое направление (рис. 14.6).

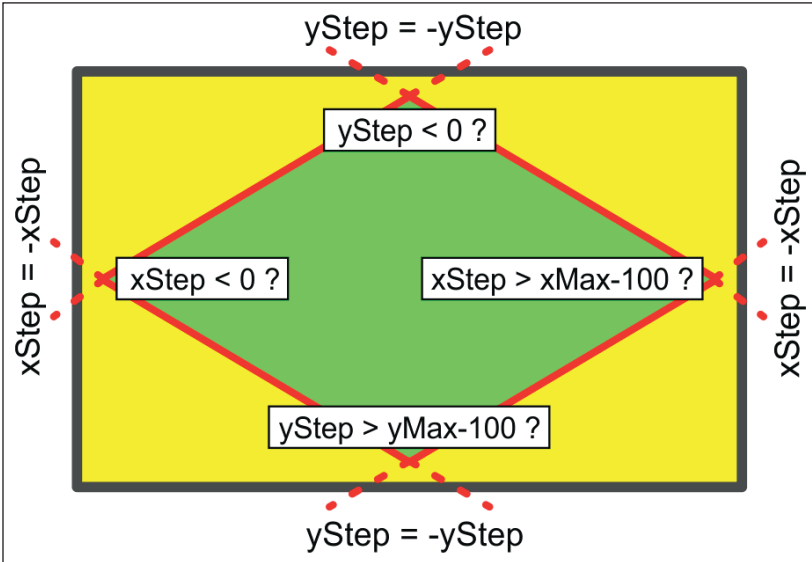


Рис. 14.6. Схема вычислений движений жука

Изменение направления $xStep$ и для $yStep$ всегда должно выполняться в двух случаях, поэтому есть два условия, связанных оператором `or` (ИЛИ). Ты уже видел его в главе 3. В табл. 14.2 напомним, в чем заключается разница между операторами `И` (`and`) и `ИЛИ` (`or`).

Таблица 14.2. Разница между операторами `И` (`and`) и `ИЛИ` (`or`)

Условие1 <code>or</code> Условие2	Одно из условий должно быть выполнено
Условие1 <code>and</code> Условие2	Все условия должны быть выполнены

Теперь значения могут быть переданы методу `move`. Но поскольку мы не располагаем расстоянием в виде параметра (и нам это не нужно), я использую метод с другим именем (который, разумеется, мне еще нужно определить):

```
Figure.step(xStep, yStep)
pg.time.delay(5)
```

Ниже показано определение метода `step` (\Rightarrow *buggy6.py*):

```
def step(self, xx, yy) :
    self.x += xx
    self.y += yy
```

Перед нами, так сказать, уменьшенная версия метода `move`. Конечно, мы также сохраним это в классе `Player`, на всякий случай.

Если ты сейчас запустишь программу, будет казаться, что перемещаемый жук всегда смотрит только в одном направлении. Поэтому мы должны задействовать поворот. Для этого сначала определим угол, на этот раз с помощью значений переменных `xStep` и `yStep`, затем повернем (\Rightarrow *buggy6.py*):

```
degree = atan2(-yStep, xStep)
degree = degrees(degree) - 90
Figure.rotate(degree)
```

Итак, эта версия программы завершена.

- Удали все под событием мыши, где вычисляются расстояние и угол. Затем добавь изученный исходный код (\Rightarrow *buggy6.py*) (рис. 14.7).

```
buggy6.py - C:\Python\Projects\buggy6.py (3.7.1)
File Edit Format Run Options Window Help

# Цикл событий
running = True
while running :
    for event in pg.event.get() :
        if event.type == pg.QUIT :
            running = False
    # Запрос мыши
    if event.type == pg.MOUSEBUTTONDOWN :
        (xPos, yPos) = pg.mouse.get_pos()

    # Ограничение движения
    if (Figure.x < 0) or (Figure.x > xMax-110) :
        xStep = -xStep
    if (Figure.y < 0) or (Figure.y > yMax-110) :
        yStep = -yStep

    # Определение направления движения
    degree = atan2(-yStep, xStep)
    degree = degrees(degree) - 90
    Figure.rotate(degree)

    # Задержка движения
    Figure.step(xStep, yStep)
    pg.time.delay(5)
```

Рис. 14.7. Новый код

- Запусти программу и проверь ее. Ты не можешь влиять на жука. Если повороты тебя не устраивают, измени их.

Тапком по виртуальному жуку

Наблюдая за бегущим жуком, не хотел бы ты наступить на него? Более жестко: нет ли желания поймать и раздавить насекомое?

Конечно, живого жука нам жалко, и нужно найти другое насекомое. Я выбрал виртуального жука, которого легко найти и раздавить.

В моей коллекции насекомых (которую ты можешь скачать с сайта dmkpress.com) есть файл *insect2.png*. Это версия нераздавленного жука. Также возьми файл *insect2x.png*, который демонстрирует результат «сдавливания» (рис. 14.8). (Оба они расположены в папке *bilder*.)

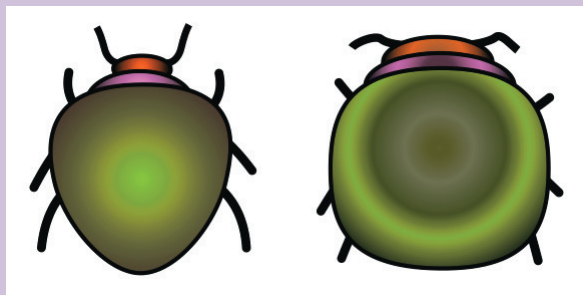


Рис. 14.8. Файлы *insect2.png* и *insect2x.png*

Но ты также можешь найти любых уродливых паразитов в интернете, если захочешь.



Чтобы заменить жука на другого, не нужно делать сложных манипуляций. Но, помимо этого, нам понадобится еще один рисунок. Мы займемся им позже. Во-первых, давай вернемся в игру, и на этот раз насекомое не должно автоматически бежать туда, где ты щелкнешь мышью, потому что на этот раз это будет опасно для его маленькой жизни.

Как проверить, попала ли мышь в жука? В игре известно положение указателя мыши и жука, и их координаты должны быть идентичны (рис. 14.9).

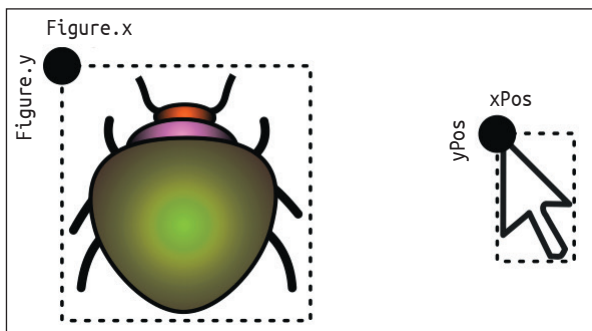


Рис. 14.9. Координаты жука и указателя мыши

В частности, речь идет о проверке того, что указатель мыши находится внутри прямоугольника, где находится жук. Итак, нам нужны четыре условия:

```
if event.type == pg.MOUSEBUTTONDOWN :
    (xPos, yPos) = pg.mouse.get_pos()
    if (xPos > Figure.x) and (xPos < Figure.x + 100) \
       and (yPos > Figure.y) and (yPos < Figure.y + 100) :
        print ("тык!")
```

(Обязательно распредели длинные условия для конструкции `if` на несколько строк!)

Я специально написал здесь инструкцию `print`, чтобы ты мог видеть слово «тык!» в окне, когда щелкаешь мышью по жуку.



Все условия связаны оператором И (`and`). Он используется потому, что только когда *все* условия выполнены, жук раздавливается (рис. 14.10). В этом случае еще говорят об *обнаружении столкновений*, поэтому конструкцию `if` можно назвать монитором столкновений. (Подробнее об этом в следующей главе.)

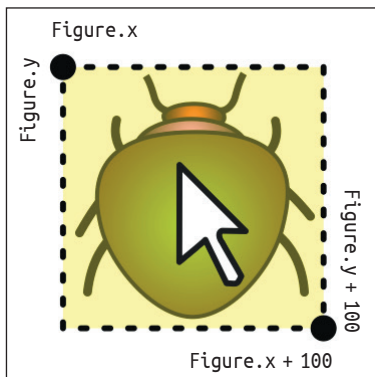


Рис. 14.10. Только при соблюдении всех условий инструкции в конструкции выполняются

Теперь нам нужен метод `Player` для гибели раздавленного насекомого. Он может выглядеть так (\Rightarrow *buggy7.py*):

```
def destroy(self) :
    self.image = pg.image.load("Bilder\Insekt2X.png")
    self.Bild = self.image
    self.isKilled = True
```

Сначала загружается изображение раздавленного насекомого, которое затем становится атрибутом `Bild` (для последующего отображения). И наконец, мне нужна еще одна переменная, `isKilled = True`. В методе `init` ей должно присваиваться значение `false` (\Rightarrow *buggy7.py*):

```
def __init__(self, xPos=0, yPos=0) :
    super().__init__()
    self.image = pg.image.load("Bilder\Insekt2.png")
    self.x, self.y = xPos, yPos
    self.Bild = self.image
    self.isKilled = False
```

Последнее изменение влияет на область, в которой вызывается метод `step`. Таким образом мы не позволяем мертвому насекомому продолжать передвигаться:

```
if not Figure.isKilled :
    Figure.step(xStep, yStep)
    pg.time.delay(5)
```

Управление классами

На самом деле ты уже можешь обновить свою программу. Но я хотел бы сделать еще одно изменение, которое не повлияет на ее работу, но позволит улучшить ее. В конце концов, возможно, что ты захочешь расширить класс `Player`, добавив некоторые атрибуты и методы. Потому что игра с жуками явно может дорабатываться.

Кроме того, ты можешь использовать этого персонажа в других подобных играх. Это хорошая идея – поместить его код в отдельный файл, листинг которого выглядит следующим образом (\Rightarrow *bplayer.py*):

```
import pygame as pg

# Класс Player
class Player(pg.sprite.Sprite) :
```

```

def __init__(self, xPos=0, yPos=0) :
    super().__init__()
    self.image = pg.image.load("Bilder\Insekt2.png")
    self.x, self.y = xPos, yPos
    self.Bild = self.image
    self.isKilled = False
def rotate(self, degree) :
    self.Bild = pg.transform.rotate(self.image, degree)
def move(self, distance, xx, yy) :
    self.x += xx
    self.y += yy
    distance -= 1
    return distance
def step(self, xx, yy) :
    self.x += xx
    self.y += yy
def destroy(self) :
    self.image = pg.image.load("Bilder\Insekt2X.png")
    self.Bild = self.image
    self.isKilled = True

```

- Создай новый файл и скопируй в него все определения класса последней версии программы. Затем сохрани оба файла. (Если ты хочешь, то также можешь скачать этот файл на сайте dmkpress.com.)



Мы не первый раз создаем отдельный файл класса. Вероятно, ты помнишь файл *mplayer.py*, в котором мы ранее поместили класс *Player*. Однако та программа была основана на библиотеке *tkinter*, поэтому тот класс нельзя использовать с *Pygame*. Отмечу, что может быть очень много классов для игры – так же, как существует множество разных игровых персонажей.

Теперь основной файл стал немного «худее». Приведу также полный листинг для основной части программы (\Rightarrow *buggy8.py*):

```

import pygame as pg
import random
from math import *
from bplayer import *

# Установка начальных значений
Green = (0,255,0)
xMax, yMax = 600, 400
degree = 0

```

```

# Запуск Pygame, создание игрового поля и персонажа
pg.init()
pg.key.set_repeat(20,20)
pg.display.set_caption("Моя игра")
Window = pg.display.set_mode((xMax, yMax))
Figure = Player(xMax/2-50, yMax/2-50)

# Случайное направление
xStep = random.randint(0,2)
if xStep == 0 :
    xStep = -1
yStep = random.randint(0,2)
if yStep == 0 :
    yStep = -1

# Цикл событий
running = True
while running :
    for event in pg.event.get() :
        if event.type == pg.QUIT :
            running = False
        # Запрос мыши
        if event.type == pg.MOUSEBUTTONDOWN :
            (xPos, yPos) = pg.mouse.get_pos()
            if (xPos > Figure.x) and (xPos < Figure.x + 100) \
                and (yPos > Figure.y) and (yPos < Figure.y + 100) :
                Figure.destroy()

# Ограничение управления
if (Figure.x < 0) or (Figure.x > xMax-110) :
    xStep = -xStep
if (Figure.y < 0) or (Figure.y > yMax-110) :
    yStep = -yStep

# Определение направления движения
degree = atan2(-yStep, xStep)
degree = degrees(degree) - 90
Figure.rotate(degree)

# Задержка движения
if not Figure.isKilled :
    Figure.step(xStep, yStep)
    pg.time.delay(5)

# Положение спрайта в окне (новое)
Window.fill(Green)
Window.blit(Figure.Bild, (Figure.x, Figure.y))
pg.display.update()

# Завершение Pygame
pg.quit()

```

- Измени свою предыдущую программу (или используй файлы с сайта dmkpress.com). Затем запусти программу и попробуй прихлопнуть жука. Если скорость высоковата, измени значение в функции `delay()` (рис. 14.11).

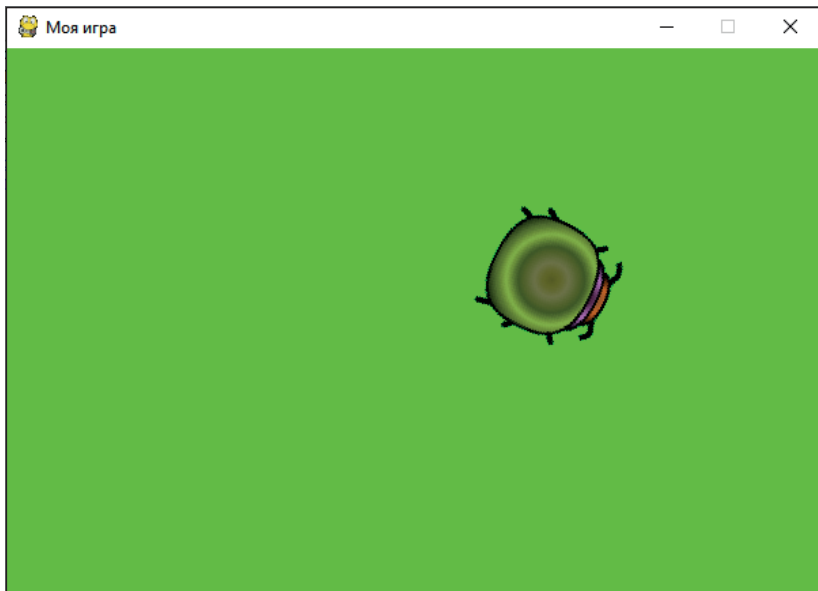


Рис. 14.11. Виртуальный паразит пойман!

В самом конце в код было добавлено нечто новое, о чем, конечно, я должен был упомянуть раньше. Но я не вижу особой необходимости присваивать окну определенный заголовок. Вот как я это сделал:

```
pg.display.set_caption("Моя игра")
```

Метод `set_caption()` берет указанный тобой текст и помещает его в строку заголовка окна.

Подведение итогов

Теперь у нас наконец есть (опять) игра. Графика, правда, не крутая, и процесс примитивен, но зато с проектом, с которым можно экспериментировать и который можно расширить. Вот что ты узнал новенького о Pygame:

<code>set_caption()</code>	Устанавливает заголовок окна
<code>Mouse.get_pos()</code>	Определяет позицию указателя мыши
<code>time</code>	Модуль времени Pygame
<code>delay()</code>	Приостанавливает игру на указанное время

Также ты познакомился с новым модулем Python и некоторыми его методами:

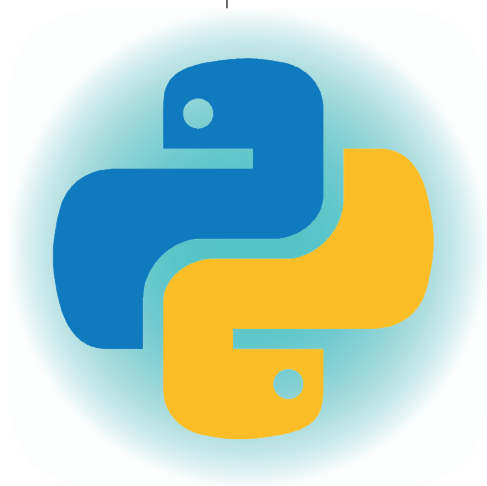
<code>Math</code>	Модуль для математических формул и вычислений
<code>atan2()</code>	Определяет угол между двумя фрагментами пути
<code>degrees()</code>	Преобразует значения из радиан в градусы
<code>sqrt()</code>	Извлекает квадратный корень из числа

Несколько вопросов...

1. Можно ли управлять жуком с помощью мыши и клавиатуры?
2. С помощью какого модуля ты можешь получить доступ к разным математическим функциям?

...и задача

1. Сделай так, чтобы жук, управляемый мышью, оставался в пределах игрового поля.



15

Уклониться или проиграть

Как насчет нового игрового проекта? Для него мы можем взять и адаптировать класс `Player` из последней главы. Смысл игры отличается от предыдущего проекта, и поэтому нам понадобится в некоторых местах изменить класс `Player`.

На самом деле здесь есть два персонажа: один пытается поймать другого. Пока другой успевает уклоняться, игра продолжается. Поскольку оба они преследуют разные задачи, мы должны определить здесь два класса.

Итак, в этой главе ты узнаешь:

- ⊙ как написать новую игру и новый класс;
- ⊙ как использовать две разные фигуры;
- ⊙ кое-что об обнаружении столкновений.

Новый персонаж

Начнем с сюжета игры: персонаж стоит в одном месте на поле, объекты летают или катятся к нему, и он должен уклоняться, приседая или подпрыгивая (рис. 15.1). После столкновения игра заканчивается.

Я использовал для этого проекта название *Dodger*, потому что с английского это слово переводится как «Ловкач».

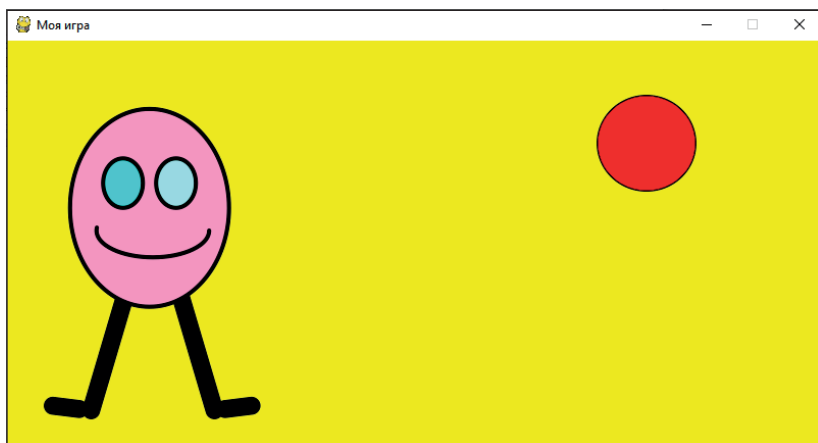


Рис. 15.1. Интерфейс игры «Ловкач»

На этот раз ты видишь игру не сверху, а сбоку. Таким образом, здесь игровое поле является скорее фоном. Я оформил его так же просто, как в предыдущем проекте, используя только один сплошной цвет.

У меня игрок находится слева (это персонаж, которым нужно управлять), а мяч справа перемещается к игроку. Для этого я определил две «высоты»: одна выше и одна ниже центральной линии (рис. 15.2).

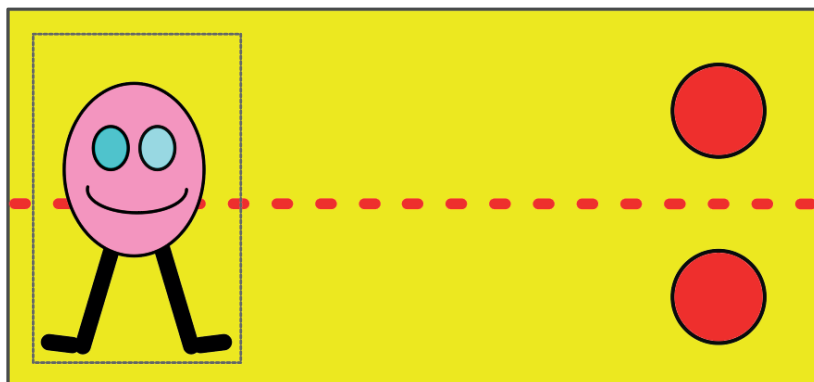


Рис. 15.2. Две «высоты» в игре

Стоящая фигура окружена невидимым прямоугольником, который должен иметь такую высоту, чтобы позже

в него вписалась фигура в прыжке. В общем, я создал три изображения для моего игрового объекта, как показано на рис. 15.3.

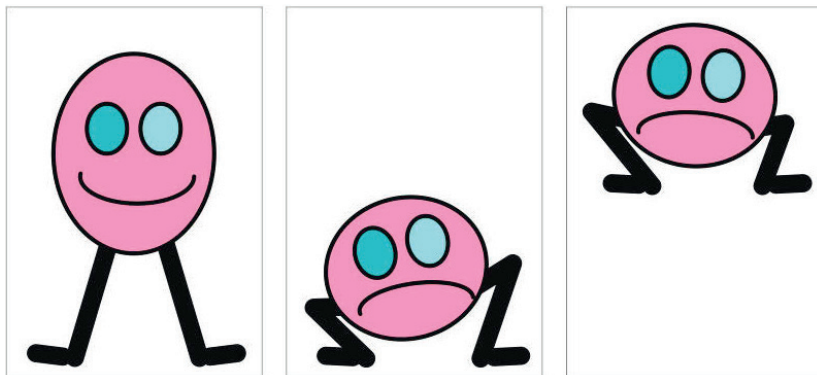


Рис. 15.3. Три изображения игрового персонажа



Ты можешь взять эти рисунки, если загрузишь примеры с сайта dmkpress.com (они также находятся в папке *Bilder*). Или если ты прирожденный художник, то можешь нарисовать собственные фигуры. Они должны иметь прозрачный фон и быть сохранены в формате PNG.

Преимущество этих трех изображений: персонажу в реальности не нужно двигаться, спрайт должен лишь сменить изображение. Это означает, что только мяч перемещается в этой игре. И мы посвятим новый ему класс чуть позже.



Опять же, если ты хочешь избежать написания большого количества исходного кода, то можешь загрузить файлы из предыдущего проекта.

Эта игра отличается, потому что есть разница между ползающими и односторонними жуками и фигурой, которая должна избегать препятствий. И класс `Player` здесь тоже сильно отличается для адаптации к типу игры.

Взгляни на мой первый вариант класса `Player` (\Rightarrow `dplayer.py`):

```
class Player(pg.sprite.Sprite) :
    image = []
    def __init__(self, xPos=0, yPos=0) :
        super().__init__()
```

```
self.image.append(pg.image.load("Bilder\DoStand.png"))
self.image.append(pg.image.load("Bilder\DoDuck.png"))
self.image.append(pg.image.load("Bilder\DoJump.png"))
self.x, self.y = xPos, yPos
self.Bild = self.image[0]
self.isHit = False
self.Status = 0
```

Загрузи файл *bplayer.py* и сохрани его под именем *dplayer.py*. Затем добавь изображения и скорректируй определение метода *init*.

Некоторые моменты знакомы, а другие совсем нет. Я полностью опустил другие методы здесь (но они находятся в листинге *dplayer.py*). Давай подробнее рассмотрим код класса.

Сначала создается пустой список изображений:

```
image = []
```

В методе *init* три файла изображений загружаются и вставляются в список:

```
self.image.append(pg.image.load("Bilder\DoStand.png"))
self.image.append(pg.image.load("Bilder\DoDuck.png"))
self.image.append(pg.image.load("Bilder\DoJump.png"))
```

Я присвоил имена трем изображениям, которые ты видел ранее в этой главе.

Здесь, как и в прошлом проекте, я непосредственно указываю имена файлов изображений. Ты также можешь использовать имена в списке параметров, если в дальнейшем планируешь использовать класс *Player* для разных игровых персонажей.



Затем определяется положение фигуры и переменной *Bild* присваивается первое изображение:

```
self.x, self.y = xPos, yPos
self.Bild = self.image[0]
```

Далее появляются два новых атрибута. Один из них – переменная переключения, которая определяет, попал ли в персонажа объект (например, мяч). Исходное значение, конечно, *False*:

```
self.isHit = False
```

В другой переменной хранится текущее положение фигуры. Номер также указывает номер отображаемого изображения:

```
self.Status = 0
```

Используются следующие значения:

Status	Действие	Файл с изображением
0	Фигура стоит	<i>dostand.png</i>
1	Фигура присела	<i>doduck.png</i>
2	Фигура подпрыгнула	<i>dojump.png</i>

Стоять, приседать и подпрыгивать

Затем нам нужен метод, который влияет на состояние персонажа, то есть определяет, какое изображение должно отображаться, – в зависимости от того, перемещается ли игрок в настоящее время – приседает или подпрыгивает:

```
def setState(self, Nr) :
    self.Status = Nr
    self.Bild = self.image[Nr]
```

Давай пока отставим это в сторону. Лучше посмотрим, чем занимается наш персонаж на поле. Поэтому мы перейдем к основной программе. Ниже представлен ее полный листинг (⇒ *dodger1.py*):

```
# Ловкач
import pygame as pg
import random
from dplayer import *

# Установка начальных значений
Yellow = (255,255,0)
xMax, yMax = 800, 400

# Запуск Pygame, создание игрового поля и персонажа
pg.init()
pg.key.set_repeat(20,20)
pg.display.set_caption("Моя игра")
Window = pg.display.set_mode((xMax, yMax))
```

```
Figure = Player(20,30)

# Цикл событий
running = True
while running :
    for event in pg.event.get() :
        if event.type == pg.QUIT :
            running = False

    # Установка клавиш
    if event.type == pg.KEYDOWN :
        if event.key == pg.K_UP :
            Figure.setState(2)
        if event.key == pg.K_DOWN :
            Figure.setState(1)
        if event.key == pg.K_RETURN :
            Figure.setState(0)

# Позиционирование спрайта в окне
Window.fill(Yellow)
Window.blit(Figure.Bild, (Figure.x, Figure.y))
pg.display.update()

# Завершение Pygame
pg.quit()
```

- Напиши все строки или используй исходный код из загруженного файла (dmkpress.com). Запусти программу и нажимай клавиши ↑, ↓ и **Enter**.

После нажатия соответствующей клавиши игрок приседает, прыгает или просто стоит на месте. Все в твоих руках. Для позиции «стоять» я использовал дополнительную клавишу. Однако для нормальной игры персонаж должен автоматически возвращаться в стоячее положение, допуская нажатие только клавиш ↑ или ↓.

К сожалению, это не так-то просто: автоматическое возвращение из прыжка или приседания должно происходить спустя некоторое время.

```
if event.key == pg.K_UP :
    Figure.setState(2)
    getTime(True)
if event.key == pg.K_DOWN :
    Figure.setState(1)
    getTime(True)
```

Задержка во времени бесполезна, если ты совершаешь следующий вызов метода `setState(0)` без каких-либо усло-

вий. Должно быть примерно так: когда прошло определенное время, нужно вернуться к позиции «стоять».

Поэтому нам нужно что-то вроде таймера, который стартует после каждого прыжка или приседания, а затем сбрасывается. Во-первых, нам нужна глобальная переменная, которая фиксирует время запуска. Мы присваиваем ей значение 0 в начале программы:

```
Start = 0
```

Я помещаю фактический таймер в функцию, которая выглядит следующим образом (\Rightarrow *dodger2.py*):

```
def getTime(Reset) :  
    global Start  
    if Reset :  
        Start = pg.time.get_ticks()  
    Diff = pg.time.get_ticks() - Start  
    return Diff
```

Reset – это переменная переключения. Если она имеет значение False, тогда рассчитывается и возвращается разница между временем начала и текущим временем:

```
Diff = pg.time.get_ticks() - Start  
return Diff
```

Если Reset имеет значение True – текущее время в момент запуска (и, таким образом, таймер сбрасывается), в результате чего Diff, естественно, возвращает 0. Определение времени (в миллисекундах) выполняется методом `time.get_ticks()` (рис. 15.4).

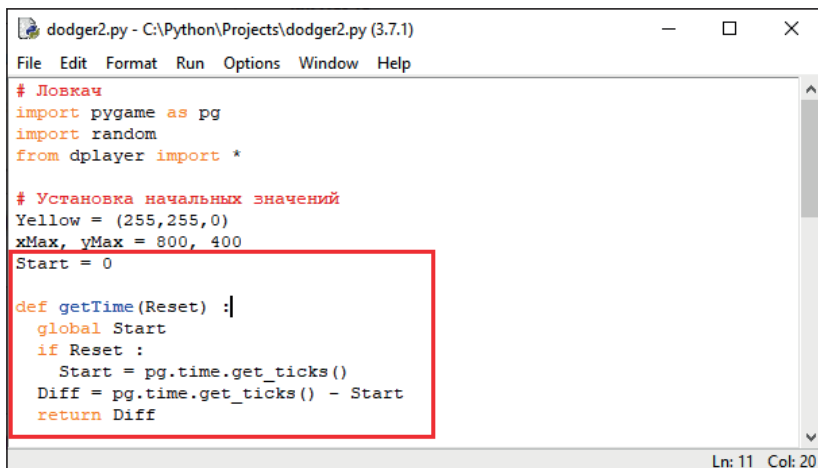


Рис. 15.4. Создание таймера

Новая функция теперь вставлена несколько раз в цикл `while`. Сначала обрабатываем две клавиши со стрелками (\Rightarrow `dodger2.py`):

```
if event.type == pg.KEYDOWN :
    if event.key == pg.K_UP :
        Figure.setState(2)
        getTime(True)
    if event.key == pg.K_DOWN :
        Figure.setState(1)
        getTime(True)
```

Каждый раз, когда ты нажимаешь клавишу \uparrow или \downarrow , таймер сбрасывается.

Вне цикла `for` мы фиксируем время, прошедшее с момента последнего нажатия клавиши со стрелкой, используя возвращаемое значение функции `getTime()`:

```
Time = getTime(False)
if Time > 1000 :
    Figure.setState(0)
```

И если прошло больше 1000 миллисекунд, тогда персонаж переходит в режим стояния, а также если ты отпустил соответствующую клавишу со стрелкой. Вот одно из мест, где ты можешь впоследствии повлиять на то, насколько сложным будет маневр персонажа.

- Напиши исходный код переменной `start`, определи функцию `getTime` и вызови ее. Затем запусти игру и попробуй, насколько хорошо персонаж выполняет упражнения по гимнастике.

Класс Thing

В конце концов, игрок уже знает, как он может справиться с предстоящими опасностями. Но до сих пор он приседает и прыгает под принуждением. Поэтому нам нужно как-то заставить его уворачиваться. И для этого нужен новый класс.

Я использую здесь мяч (точнее, круг, который ты можешь себе представить, это мяч или нечто еще опасное). Но это также может быть что-то угловатое, т. е. некая «вещь», поэтому я просто называю новый класс `Thing`.

Что должен делать мяч? Он должен появляться в правой части поля и двигаться влево к игроку. Затем он должен снова оказаться в правой части. И на этот раз или сверху, или снизу.

Давай посмотрим, как такой класс может выглядеть (\Rightarrow *dthing.py*):

```
import pygame as pg
import random

# Объект
class Thing(pg.sprite.Sprite) :
    def __init__(self, name, xPos=0, yPos=0) :
        super().__init__()
        self.image = pg.image.load(name)
        self.x, self.y = xPos, yPos
        self.Bild = self.image
        self.Width = self.Bild.get_width()
        self.Height = self.Bild.get_height()
        self.rect = self.Bild.get_rect()
    def setPosition(self, xPos, yPos, updown) :
        self.x = xPos
        if updown :
            y0yU = random.randint(0,1)
            self.y = y0yU * yPos + self.Height/2
        else :
            self.y = yPos
    def move(self, xx, yy) :
        self.x += xx
        self.y += yy
```

```
def controlRestart(self, xx, yy) :
    if self.x < 0 :
        y0yU = random.randint(0,1)
        self.x = xx - self.Width/2
        self.y = y0yU * yy + self.Height/2
        return True
    else :
        return False
```

- Создай новый файл и вставь исходный код, затем сохрани файл. (Конечно, ты также можешь использовать другое имя, а не *dthing.py*.)

Давай рассмотрим весь код построчно. Метод `init` принимает имя файла изображения, позволяя тебе выбрать, какой объект будет перемещаться:

```
def __init__(self, name, xPos=0, yPos=0) :
```

Внутри метода сначала загружается изображение, затем определяется положение спрайта:

```
self.image = pg.image.load(name)
self.x, self.y = xPos, yPos
```

После создания экземпляра изображения в `Bild` нам по-прежнему нужны ширина и высота спрайта – но об этом позже:

```
self.Bild = self.image
self.Width = self.Bild.get_width()
self.Height = self.Bild.get_height()
```

Другой метод гарантирует размещение объекта случайным образом выше или ниже средней линии. Метод `setPosition()` принимает сведения о расположении и переменную переключения:

```
def setPosition(self, xPos, yPos, updown) :
```

Сначала определяется горизонтальное положение (*x*):

```
self.x = xPos
```

Для оси *y* код выглядит немного иначе, так как значение зависит от переменной `updown`. Если ей присвоено значение `False`, то значение *y* оказывается равным *x*:

```
self.y = yPos
```

Если переменной `updown` присвоено значение `True`, то случайное число 0 или 1 будет генерироваться так, чтобы мяч не всегда стартовал из одного и того же места:

```
y0yU = random.randint(0,1)
```

В этом случае центральная линия объекта должна быть передана `yPos`. Тогда ты сможешь установить, будет ли мяч наводиться вверху или катиться вниз:

```
self.y = y0yU * yPos + self.Height/2
```

Давай перейдем к методу, который перемещает объект:

```
def move(self, xx, yy) :  
    self.x += xx  
    self.y += yy
```

Я мало что скажу о нем, потому что ты уже знаешь несколько способов перемещения.

Остался последний метод – `controlRestart()`. Он принимает в качестве параметра позицию, в которую объект должен быть перемещен и из которой мяч должен быть выпущен в дальнейшем:

```
def controlRestart(self, xx, yy) :
```

Как только мяч оказывается в левой части игрового поля ($x < 0$), его следует вернуть обратно на правый край:

```
self.x = xx - self.Width/2
```

Если ты укажешь `xMax` в качестве параметра для `xx`, половина ширины спрайта будет вычтена.

Разумеется, высота прыжка мяча должна быть определена случайным образом:

```
y0yU = random.randint(0,1)  
self.y = y0yU * yy + self.Height/2
```

Здесь тоже есть возвращаемое значение:

```
return True
```

Если мяч находится в области движения, т. е. двигается справа налево, то это возвращаемое значение должно быть False:

```
else :  
    return False
```

True означает, что мяч перезапускается, а False – что он в пути.

Учим персонажа уклоняться

Как насчет основной программы? Помимо того факта, что там должна быть добавлена инструкция импорта класса Thing, есть также следующие изменения.

Сначала создаем объект мяча непосредственно после Player (⇒ *dodger3.py*):

```
Ball = Thing("Bilder/ball1.png")
```

Кроме того, мы ставим этот мяч на край поля (рис. 15.5).

```
Ball.setPosition(xMax-50, yMax/2, True)
```

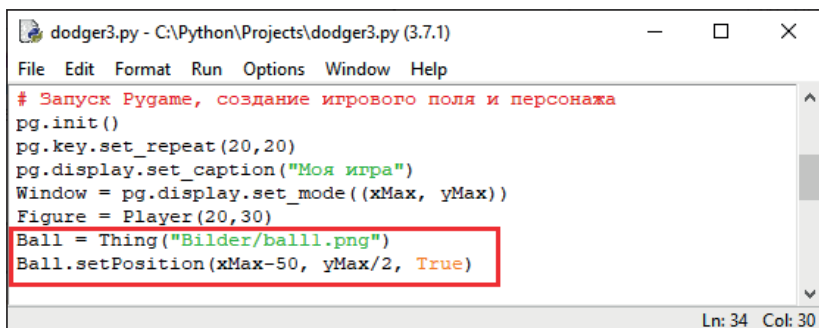


Рис. 15.5. Размещение мяча

Можно было указать позицию создания объекта мяча, но только метод `setPosition()` позволяет начать движение случайным образом вверх или вниз поля.



В цикле `while` после инструкций, касающихся времени, указаны инструкции для управления перемещением мяча справа налево (рис. 15.6).

```
Ball.move(-1, 0)
Ball.controlRestart(xMax-50, yMax/2)
```

И последнее, но не менее важное: ты также должен убедиться, что летающий или катящийся мяч отображается (рис. 15.6). Итак, вот еще один вызов `blit`:

```
Window.blit(Ball.Bild, (Ball.x, Ball.y))
```

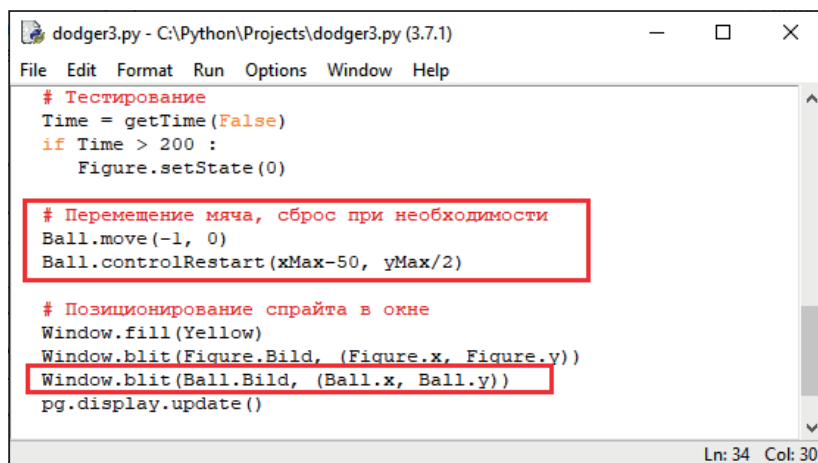


Рис. 15.6. Добавленные инструкции

- Напиши показанный исходный код, затем запусти программу.

В зависимости от случайного значения мяч будет наполовину отображаться вверху или внизу в правой части окна, а затем двигаться к персонажу.

- Старайся избегать попадания мяча. Получается? Кажется слишком легким? Увеличь значение первого параметра в методе `move`.

Независимо от того, уклонился ты или нет, мяч мчится мимо, а затем снова появляется справа и вновь катится. Для визуализации неудачного уклонения, чтобы был какой-то эффект, нам нужно обнаруживать столкновения.

Но как именно это должно выглядеть? Стоит ли персонаж на месте, приседает или вскакивает, всегда есть контакт между этими двумя объектами.

Давай взглянем в табл. 15.1, чтобы увидеть, какие варианты есть, когда объекты Ball и Player встречаются:

Расположение Состояние	DoStand	DoDuck	DoJump
Мяч вверх	врезается	пролетает мимо	врезается
Мяч вниз	врезается	врезается	пролетает мимо

Можно увидеть (только) два случая, когда уклонение работает (рис. 15.7):

- 1) когда мяч летит вверх, а игрок приседает;
- 2) когда мяч катится вниз, а игрок подпрыгивает.

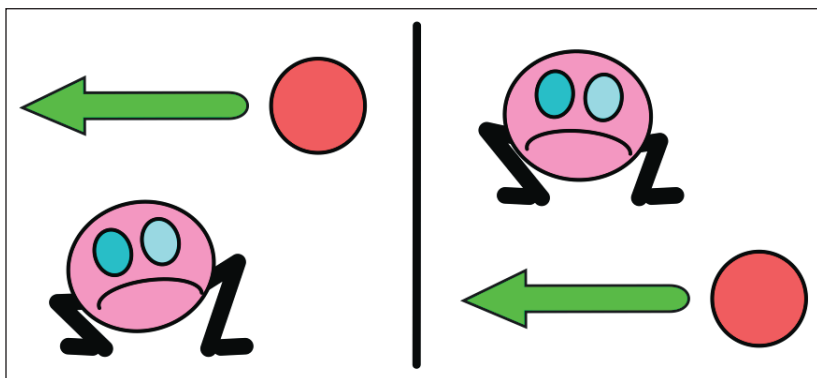


Рис. 15.7. Два варианта успешного уклонения

В случае ошибки (столкновения) игра заканчивается: в игрока попал мяч, и тот проигрывает. Итак, нам нужны две функции управления: одна для столкновения и одна для уклонения. Давай воспользуемся методом `dogde` (\Rightarrow `dplayer.py`):

```
def dodge(self, yPos, yMiddle) :
    if (yPos < yMiddle and self.Status == 1) \
    or (yPos > yMiddle and self.Status == 2) :
        return True
    else :
        return False
```

Параметр `yPos` информирует о высоте приближаемого мяча (или подкатывающегося), а `yMiddle` определяет центральную линию: если мяч летит выше нее, то поможет только приседание (`Status == 1`), а если мяч катится ниже, тогда нуж-

но присесть (`Status == 2`). Если уклонение прошло успешно, метод возвращает `True`, в противном случае – `False`.

Вызов этого метода имеет смысл только при наличии контакта между прямоугольником фигуры и поверхностью мяча. Как насчет конструкции `if` в духе того, как мы сделали с жуком в предыдущей главе?

Здесь нам не требуется столько условий. Важно лишь соприкосновение двух поверхностей. Нужно только проверить, находится ли мяч достаточно близко, чтобы перекрыть персонажа:

```
if (Ball.x < Figure.x+150)
```

Я определил вертикальную ограничительную линию, которую мяч должен пересечь, чтобы засчитать удар (рис. 15.8). (Ты должен поэкспериментировать со значением, которое добавляешь к переменной `Figure.x`.)

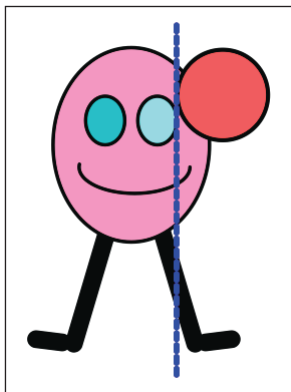


Рис. 15.8. Невидимая вертикальная линия, достигнув которой мячом, можно засчитывать проигрыш

Вот теперь в игру вступает атрибут `isHit`, который принимает значение `False` при создании персонажа, и `True`, если мяч коснулся линии ограничения, а персонаж не уклонился. Получается двойная условная конструкция `if`:

```
if (Ball.x < Figure.x+150) :  
    if not Figure.dodge(Ball.y, yMax/2) :  
        Figure.isHit = True
```

Для того чтобы мяч остановился, необходим следующий код:

```
if not Figure.isHit :  
    Ball.move(-1, 0)  
    Ball.controlRestart(xMax-50, yMax/2)
```

Основная программа

Чтобы ничего не потерялось, а именно все изменения и дополнения в исходном коде, я вновь привожу его целиком (\Rightarrow *dodger4.py*):

```
import pygame as pg  
import random  
from dplayer import *  
from dthing import *  
  
# Установка начальных значений  
Yellow = (255,255,0)  
xMax, yMax = 800, 400  
Start = 0  
  
# Таймер  
def getTime(Reset) :  
    global Start  
    if Reset :  
        Start = pg.time.get_ticks()  
    Diff = pg.time.get_ticks() - Start  
    return Diff  
  
# Запуск Pygame, создание игрового поля и персонажа  
pg.init()  
pg.key.set_repeat(20,20)  
pg.display.set_caption("Моя игра")  
Window = pg.display.set_mode((xMax, yMax))  
Figure = Player(20,30)  
Ball = Thing("Bilder/ball1.png")  
Ball.setPosition(xMax-50, yMax/2, True)  
  
# Цикл событий  
running = True  
while running :  
    for event in pg.event.get() :  
        if event.type == pg.QUIT :  
            running = False  
  
    # Установка клавиш  
    if event.type == pg.KEYDOWN :  
        if event.key == pg.K_UP :  
            Figure.setState(2)
```

```
        getTime(True)
        if event.key == pg.K_DOWN :
            Figure.setState(1)
            getTime(True)

# Тестирование
Time = getTime(False)
if Time > 200 :
    Figure.setState(0)

# Перемещение мяча, сброс при необходимости
if not Figure.isHit :
    Ball.move(-1, 0)
    Ball.controlRestart(xMax-50, yMax/2)
# Проверка, находится ли мяч в игровой зоне
if (Ball.x < Figure.x+150) :
    # Если игрок проиграл, игра заканчивается
    if not Figure.dodge(Ball.y, yMax/2) :
        Figure.isHit = True

# Позиционирование спрайта в окне
Window.fill(Yellow)
Window.blit(Figure.Bild, (Figure.x, Figure.y))
Window.blit(Ball.Bild, (Ball.x, Ball.y))
pg.display.update()

# Завершение Pygame
pg.quit()
```

- Набери исходный код и запусти программу. Если игра слишком проста для тебя, просто ускорь ее с помощью метода `move`.

Подведение итогов

Теперь у тебя есть еще одна игра. Такой проект никогда не завершится, ты всегда сможешь найти что-то, что можно улучшить. Здесь я использовал только один новый метод `pygame`, который ты узнал:

```
time.get_ticks()  Получение текущего времени (в миллисекундах)
```

Нет вопросов...

...и одна задача

1. В папке с примерами для этой книги (dmkpress.com) есть файл с побитым персонажем (рис. 15.9). Убедись, что при завершении игры появляется соответствующее изображение.

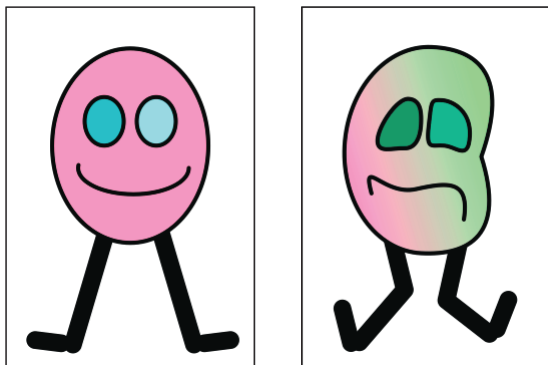
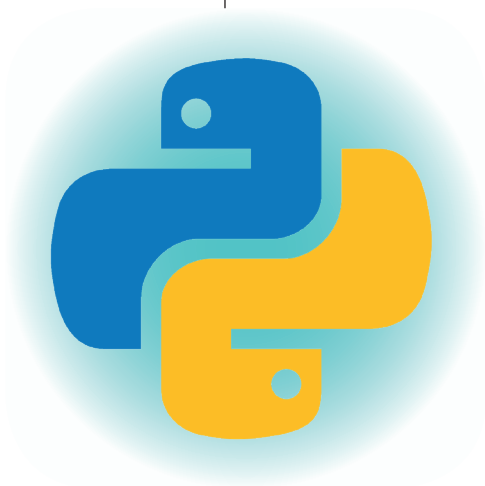


Рис. 15.9. Персонаж после попадания мяча



16

Пора развлекаться

В последних нескольких главах ты запрограммировал две игры, в которые уже можешь играть. Но было бы намного интереснее, если бы ты мог набирать очки, когда убиваешь жука или уклоняешься от мяча. Вот почему мы должны заняться этим в данной главе.

Кроме того, играть с одним жуком не так сложно и интересно, как с целой кучей насекомых. Давай наплодим их в этой главе.

Итак, в этой главе ты узнаешь:

- ⊙ как отображать текст в окне программы;
- ⊙ как создать новый класс (Game);
- ⊙ как получать и набирать очки;
- ⊙ как охотиться на более чем одного жука;
- ⊙ как использовать таймер игрового времени.

Игровой счет

Чтобы отобразить текст в окне Pygame, тебе понадобится так называемая «вставка», на которой будет отображаться текст. В Pygame нужный класс называется Surface. Итак, мы создаем соответствующий объект:

```
Text = pg.Surface((300,50))
Text.fill(Yellow)
```

В итоге получается не что иное, как прямоугольная область, которая на самом деле черная, но с помощью заливки мы делаем ее желтой, чтобы она (пока) не была видна на поле. Затем давай определим функцию для отображения текста (\Rightarrow *dodger5.py*):

```
def showMessage(text, Color) :
    global Text
    Font = pg.font.SysFont("arial", 48)
    Text = Font.render(text, True, Color)
```

Сначала мы должны указать шрифт, лучше системный. Я выбрал шрифт Arial с размером 48. (Попробуй другие шрифты и размеры.)

Теперь текст рендерится, и ты можешь его увидеть. Кроме того, ты можешь выбрать цвет текста.

Рендерится? Что это? Визуализация (отображение на экране) графического контента называется рендерингом. Это может быть, среди прочего, изображение или текст. Для многих объектов вызова `blit()` достаточно, чтобы сделать их видимыми, но в нашем случае текст, который должен отображаться, сначала должен быть «вставлен» в объект `Surface`.



Таким образом, теперь мы можем отображать очки, которые игрок получает, когда успешно уклоняется от мяча. Для этого нам нужна другая функция (\Rightarrow *dodger5.py*):

```
def setScore(num) :
    global Score
    Score += num
    showMessage("Счет: " + str(Score), Blue)
```

Глобальной переменной `Score`, которая здесь используется, присваивается значение 0 в начале программы:

```
Score = 0
```

Количество баллов, на которое должен увеличиться счет, принимается в качестве параметра. Затем функция `showMessage()` используется для отображения результата синим

цветом. Разумеется, цвет также должен быть определен заранее:

```
Blue = (0,0,255)
```

- Дополни исходный код своей программы Dodger кодом двух функций и связанных с ними переменных, как показано на рис. 16.1.

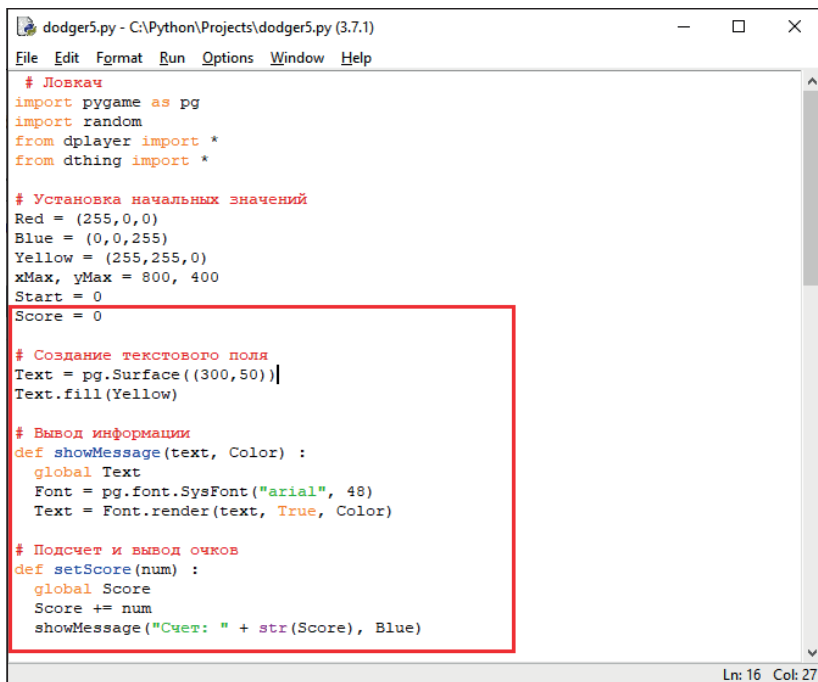


Рис. 16.1. Добавленные инструкции

Теперь давай посмотрим, где мы можем привести новые функции в игру. С одной стороны, когда игрок уклонился, счет очков должен быть увеличен:

```
if not Figure.isHit :
    Ball.move(-1, 0)
    if Ball.controlRestart(xMax-50, yMax/2) :
        setScore(1)
```

Если мяч в персонажа не попал, мяч может двигаться дальше и запускаться снова, если находится слева, то есть в области игрока. Напомню, что метод `controlRestart()` имеет возвращаемое значение. Потому что мы можем воспользоваться этим.

Если мяч был успешно запущен и не попал в игрока – должны увеличиться очки, которые сразу отображаются. Если хочешь, ты можешь быть более щедрым и начислять, например, по десять очков. Я скупой, со мной игрок получает только одно очко за каждое уклонение.

Чтобы ты мог видеть область со счетом, нам нужен еще один метод `blit`:

```
Window.blit(Text, (xMax/2, 10))
```

Я разместил эту инструкцию первой, перед другими методами `blit` (рис. 16.2).

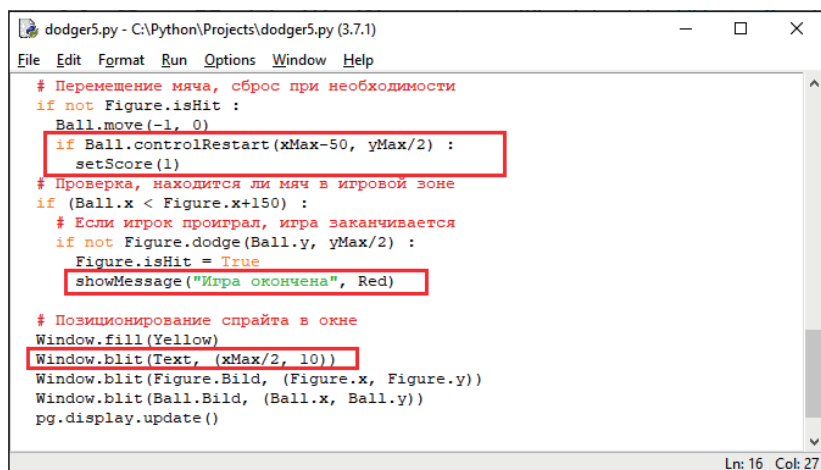


Рис. 16.2. Добавление кода, отвечающего за счет

Если ты внимательно посмотришь на рисунок, то увидишь, что я добавил текст, уведомляющий об окончании игры. Эта строка находится непосредственно под той, в которой `isHit` получает значение `True`:

```
if not Figure.dodge(Ball.y, yMax/2) :
    Figure.isHit = True
    showMessage("Игра окончена", Red)
```

Если в объект `Player` попал мяч, вместо счета отображается текст «Игра окончена».

- Дополни исходный код, запусти программу и набирай очки (\Rightarrow `dodge5.py`).

Класс Game

Теперь исходный код нашего «Ловкача» значительно увеличился, хотя мы сохранили инструкции для игрока и мяча в разные файлы. Давай посмотрим, нет ли еще чего-нибудь лишнего, чтобы уменьшить размер основной программы.

Прежде всего я думаю обо всех функциях, которые мы здесь объявили. А что, если мы создадим новый класс и превратим эти функции в методы данного класса? Давай попробуем.

- Создай новый файл и присвой ему имя *game.py* (это мой вариант). В исходном коде *dodger5.py* выдели код между двумя глобальными переменными и последней функцией *getTime()* (рис. 16.3).

```
# Установка начальных значений
Red = (255,0,0)
Blue = (0,0,255)
Yellow = (255,255,0)
xMax, yMax = 800, 400
Start = 0
Score = 0

# Создание текстового поля
Text = pg.Surface((300,50))
Text.fill(Yellow)

# Вывод информации
def showMessage(text, Color) :
    global Text
    Font = pg.font.SysFont("arial", 48)
    Text = Font.render(text, True, Color)

# Подсчет и вывод очков
def setScore(num) :
    global Score
    Score += num
    showMessage("Счет: " + str(Score), Blue)

# Таймер
def getTime(Reset) :
    global Start
    if Reset :
        Start = pg.time.get_ticks()
    Diff = pg.time.get_ticks() - Start
    return Diff

# Запуск Pygame, создание игрового поля и персонажа
pg.init()
pg.key.set_repeat(20,20)
pg.display.set_caption("Моя игра")
window = pg.display.set_mode((xMax, yMax))
```

Рис. 16.3. Выделенный фрагмент кода, который станет классом

- Скопируй код в новый файл, а затем удали его из исходного файла (или вырежи и вставь).

Я назвал новый класс `Game`, потому что там объявляются методы, касающиеся игры. Ты можешь постепенно расширять класс, когда придумаешь еще несколько (хороших) методов. Прежде чем мы начнем работу над методами, самая первая строка кода в `game.py` должна быть такой:

```
import pygame as pg
```

Затем напишем объявление класса:

```
class Game :
```

Давай перейдем к методам. Вот они в одном листинге (\Rightarrow `game.py`):

```
# Запуск и создание текстового поля
def __init__(self, Color) :
    self.Start = 0
    self.Score = 0
    self.Text = pg.Surface((300,50))
    self.Text.fill(Color)

# Вывод информации
def showMessage(self, text, Color) :
    self.Font = pg.font.SysFont("arial", 48)
    self.Text = self.Font.render(text, True, Color)

# Подсчет и вывод очков
def setScore(self, num, Color) :
    self.Score += num
    self.showMessage("Счет: " + str(self.Score), Color)

# Таймер
def getTime(self, Reset) :
    if Reset :
        self.Start = pg.time.get_ticks()
        self.Diff = pg.time.get_ticks() - self.Start
    return self.Diff

# Счет и время
def showAll(self, num, Color) :
    self.Score += num
    ptext = " | Счет: " + str(self.Score)
    ztext = "Время: " + str(int(self.Diff/1000))
    self.showMessage(ztext+ptext, Color)
```

Как видишь, различия и не так велики. Новым является метод `init`, в котором генерируется текстовое поле, которое

16

затем может служить для отображения всех видов информации и сообщений. Метод `setScore` нуждается в другом параметре, чтобы отображаемый цвет можно было выбирать свободно.

Давай использовать наш новый модуль в основной программе, код которой выглядит следующим образом (\Rightarrow *dodger6.py*):

```
# Ловкач
import pygame as pg
import random
from dplayer import *
from dthing import *
from game import *

# Установка начальных значений
Red = (255,0,0)
Blue = (0,0,255)
Yellow = (255,255,0)
xMax, yMax = 800, 400

# Запуск Pygame, создание игровых элементов
pg.init()
pg.key.set_repeat(20,20)
pg.display.set_caption("Моя игра")
Window = pg.display.set_mode((xMax, yMax))
Figure = Player(20,30)
Ball = Thing("Bilder/ball1.png")
Ball.setPosition(xMax-50, yMax/2, True)
Game = Game(Yellow)

# Цикл событий
running = True
while running :
    for event in pg.event.get() :
        if event.type == pg.QUIT :
            running = False

    # Установка клавиш
    if event.type == pg.KEYDOWN :
        if event.key == pg.K_UP :
            Figure.setState(2)
            Game.getTime(True)
        if event.key == pg.K_DOWN :
            Figure.setState(1)
            Game.getTime(True)

# Тестирование
Time = Game.getTime(False)
```

```
if Time > 200 :
    Figure.setState(0)

# Перемещение мяча, сброс при необходимости
if not Figure.isHit :
    Ball.move(-1, 0)
    if Ball.controlRestart(xMax-50, yMax/2) :
        Game.setScore(1, Blue)
# Проверка, находится ли мяч в игровой зоне
if (Ball.x < Figure.x+150) :
    # Если игрок проиграл, игра заканчивается
    if not Figure.dodge(Ball.y, yMax/2) :
        Figure.isHit = True
        Game.showMessage("Игра окончена", Red)

# Позиционирование спрайта в окне
Window.fill(Yellow)
Window.blit(Game.Text, (xMax/2, 10))
Window.blit(Figure.Bild, (Figure.x, Figure.y))
Window.blit(Ball.Bild, (Ball.x, Ball.y))
pg.display.update()

# Завершение Pygame
pg.quit()
```

Я назвал новую игру Arcade следующим образом:

```
Arcade = Game(Yellow)
```

В самой игре вызываются методы `Arcade.getTime()`, `Arcade.setScore()` и `Arcade.showMessage()` (иногда в нескольких местах). Обрати внимание, что даже в инструкции `blit` параметр `Text` теперь принадлежит `Arcade`.

- Измени исходный код и убедись, что программа выполняет те же действия, что и предыдущая.

Ошибка при сборке программы

Теперь вернемся к предыдущей игре с жуком. Как насчет того, чтобы там тоже установить подсчет набранных очков? Собственно, зачем, спросишь ты: если жук мертв, игра окончена. Правильно, но что, если мы преследуем не одного, а множество жуков, движущихся в разных направлениях? И у тебя есть ограниченное количество времени, чтобы выследить их всех? Тогда подсчет очков имеет смысл.

Давай возьмем наш последний проект с жуком и посмотрим, как сделать несколько жуков из одного. Для этого мы объявляем переменную в начале, значение которой ты можешь изменить по желанию:

```
bugMax = 5
```

Далее нам нужен (пустой) список:

```
Figure = []
```

Затем мы заполняем его с помощью цикла `for` насекомыми (\Rightarrow *buggy9.py*):

```
for Nr in range(0,bugMax) :  
    xPos = random.randint(100,xMax-100)  
    yPos = random.randint(50,yMax-100)  
    Figure.append(Player(xPos, yPos))
```

Здесь я использовал два случайных значения для расположения, чтобы жуки появлялись в разных местах каждый раз, когда запускается игра. В моем случае должно быть пять жуков. Нам также нужны инструкции для их направления, чтобы жуки не двигались в одном направлении. Для этого создадим еще два списка:

```
xStep = []  
yStep = []
```

Затем мы упаковываем определение случайного начального направления в цикл `for` (\Rightarrow *buggy9.py*):

```
for Nr in range(0,bugMax) :  
    xStep.append(random.randint(0,2))  
    if xStep[Nr] == 0 :  
        xStep[Nr] = -1  
    yStep.append(random.randint(0,2))  
    if yStep[Nr] == 0 :  
        yStep[Nr] = -1
```

В этом случае цикл должен проверять каждого жука отдельно после щелчка мышью, был ли он раздавлен:

```
if event.type == pg.MOUSEBUTTONDOWN :  
    (xPos, yPos) = pg.mouse.get_pos()  
    for Nr in range(0,bugMax) :
```

```
if (xPos > Figure[Nr].x) and (xPos < Figure[Nr].x + 100) \
and (yPos > Figure[Nr].y) and (yPos < Figure[Nr].y + 100) :
    Figure[Nr].destroy()
```

При попадании жук «сплющивается» (и переменной `isKilled` присваивается значение `true`).

Теперь отслеживание столкновений должно выполняться для каждого жука (\Rightarrow *buggy9.py*):

```
for Nr in range(0,bugMax) :
    if (Figure[Nr].x < 0) or (Figure[Nr].x > xMax-110) :
        xStep[Nr] = -xStep[Nr]
    if (Figure[Nr].y < 0) or (Figure[Nr].y > yMax-110) :
        yStep[Nr] = -yStep[Nr]
```

То же самое относится к ориентации жука по направлению движения:

```
for Nr in range(0,bugMax) :
    degree = atan2(-yStep[Nr], xStep[Nr])
    degree = degrees(degree) - 90
    Figure[Nr].rotate(degree)
```

Но это еще не все, движение настраивается также для каждого маленького насекомого:

```
for Nr in range(0,bugMax) :
    if not Figure[Nr].isKilled :
        Figure[Nr].step(xStep[Nr]*2, yStep[Nr]*2)
pg.time.delay(5)
```

Я установил задержку после цикла `for`, иначе игра будет ускоряться, в зависимости от того, сколько жуков будет убито. (Если хочешь, добавь инструкцию `delay` в конструкцию `if` и измени значение времени.)

Теперь нужно убедиться, что все жуки отображаются. Это делается в этом последнем цикле (\Rightarrow *buggy9.py*):

```
for Nr in range(0,bugMax) :
    Window.blit(Figure[Nr].Bild, (Figure[Nr].x, Figure[Nr].y))
```

- Дополни свой проект указанным кодом. Лучше всего в том порядке, как он указан в тексте главы. Или используй файлы примеров для книги. Затем запусти программу и проверь, что все работает (рис. 16.4).

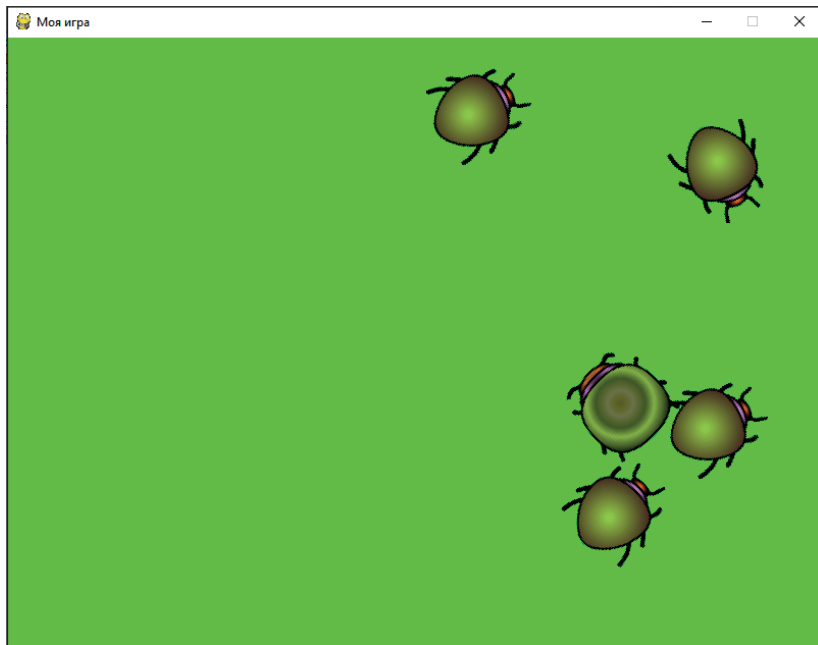


Рис. 16.4. Жуки заполнили экран

Очки при попадании

Давай перейдем к игровым очкам. Поскольку сейчас это того стоит, я предлагаю, чтобы за каждого раздавленного жука игрок получал 50 или даже 100 очков. И если ты позже сделаешь поле значительно больше и позволишь на нем бегать большому количеству жуков, то будет возможность набирать много очков.

Как хорошо, что мы определили и использовали класс `Game` в последней версии «Ловкача». Мы можем использовать его и здесь. Для начала импортируем модуль `Game`:

```
from game import *
```

Затем мы объявляем игровой объект (на этот раз с зеленым фоном):

```
Arcade = Game(Green)
```

Как только мы убиваем жука, метод `setScore()` вступает в действие:

```
Arcade.setScore(50, Blue)
Figure[Nr].destroy()
```

Разумеется, синий цвет должен был быть предварительно определен:

```
Blue = (0,0,255)
```

Чтобы очки отображались, эта строка должна находиться в нижней части листинга (\Rightarrow *buggy10.py*):

```
Window.blit(Arcade.Text, (xMax/3, 10))
```

- Напиши эти строки и попытайся набрать очки за каждое попадание.

Ты заработал больше, чем ожидал? Конечно, потому что если ты щелкал мышью по мертвому жуку, очки все равно зачислялись. Но это не входило в наши планы, и мы должны исправить сей недочет.

Это проще, чем ожидалось: мы проверяем, убили ли мы жука, если да – начисляются очки (\Rightarrow *buggy10.py*):

```
if not Figure[Nr].isKilled :
    Arcade.setScore(50, Blue)
Figure[Nr].destroy()
```

Теперь ошибка устранена.

- Настрой свою программу и протестируй ее до тех пор, пока не будут убиты все жуки. Достигнут ли предел?

Теперь ты всегда сможешь достичь наивысшего балла, но реальная привлекательность этой игры должна заключаться в том, что ты должен приложить в ней немного больше усилий. Может быть, ограничить время? Давай изменим программу так, чтобы у игрока было только определенное количество времени, чтобы убить всех жуков или как можно большее количество за отведенное время. Для этого сначала нужно запустить таймер (\Rightarrow *buggy11.py*):

```
Arcade.getTime(True)
```

Конечно, это должно происходить за пределами цикла `while`. Затем нам нужна переменная, которая постоянно записывает текущее время в этом цикле, по которой мы сможем контролировать время прохождения игры:

```
Time = Arcade.getTime(False)
```

Будет полезно для игрока, если время будет отображаться. Но поскольку у нас нет подходящего метода, мы должны дополнить класс `Game` (\Rightarrow *game.py*):

```
def showAll(self, num, Color) :  
    self.Score += num  
    ptext = " | Счет: " + str(self.Score)  
    ztext = "Время: " + str(int(self.Diff/1000))  
    self.showMessage(ztext+ptext, Color)
```

Метод `showAll()` суммирует отображение времени и набранных очков:

```
Arcade.showAll(0, Blue)
```

То есть никакие очки не добавляются, это касается только отображения изменения времени.

Также ниже следует заменить вызов метода `setScore()` (\Rightarrow *buggy11.py*):

```
if not Figure[Nr].isKilled :  
    Arcade.showAll(50, Blue)
```

После окончания игрового времени следующее сообщение выводит набранное количество очков:

```
if Time > bugMax*1500 :  
    Arcade.showMessage("Игра окончена", Red)  
    running = False
```

При этом я сделал игровое время зависимым от количества существующих жуков. Сначала появляется сообщение «Игра окончена», а затем переменной `running` присваивается значение `False` (рис. 16.5).

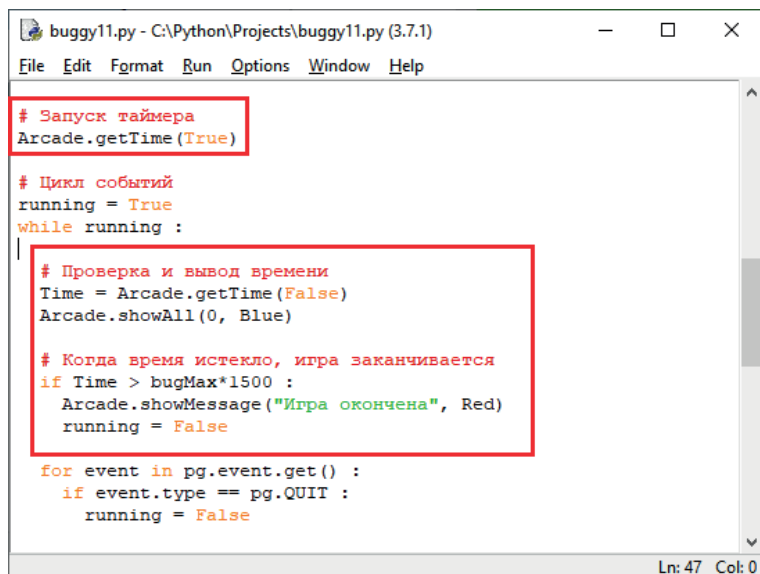


Рис. 16.5. Внесенные в код изменения

Чтобы после выхода из цикла `while` игра не заканчивалась слишком внезапно, мы должны установить небольшую паузу:

```

pg.time.delay(1500)
pg.quit()
    
```

А теперь добавим метод `showAll` в класс `Game`. Затем ты можешь играть в свою игру, но на этот раз будет сложнее поймать всех жуков (рис. 16.6).



Рис. 16.6. Ура! Победа!

Очень удобно, что ты можешь скачать все примеры, а не просто набирать их, на сайте dmkpress.com.

Подведение итогов и заключение

К сожалению, мы подошли к концу книги. В предыдущих главах о Python и Pygame, безусловно, многие возможности не были описаны. Но у тебя уже есть более чем прочные основы, и еще можно улучшить рассмотренные программы или даже разработать новые.

В этой главе было не так много нового, но все же несколько слов мы добавили в твой словарь:

Surface	Класс для отображения текста и графики
Font	Класс для шрифтов
render	Рендеринг текста

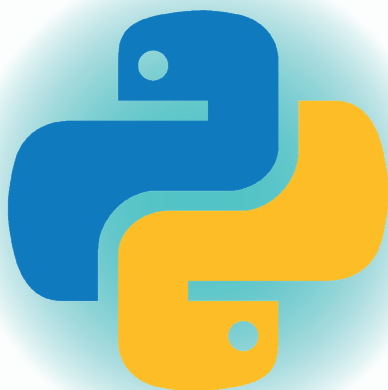
Конечно, что еще можно сделать с помощью Python, например, по следующим адресам в интернете:

- www.pygame.org/docs/ – здесь ты найдешь учебники по Pygame;
- docs.python.org/3/ – исчерпывающая документация по Python;
- wiki.python.org/moin/FrontPage – ты узнаешь еще больше о Python, посетив этот сайт.

Все эти сайты на английском языке.

A

Установка Python



Среду Python установить очень просто. Все, что нужно сделать, – это нажать пару кнопок для запуска процесса инсталляции. В случае затруднений ты можешь попросить помощи у взрослых.

Сначала тебе нужно загрузить последнюю версию Python. Для этого выполни следующие действия.

1. Запусти веб-браузер (например, Google Chrome или Microsoft Edge) и перейди на страницу www.python.org/downloads/ (рис. А.1).

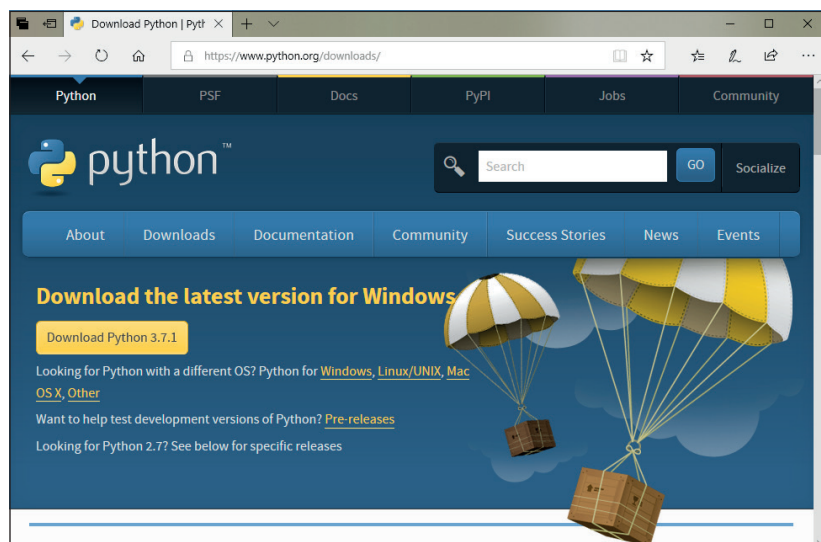


Рис. А.1. Страница с дистрибутивами Python

А

Здесь доступны две различные ветви дистрибутива Python. В этой книге используется последняя версия ветви Python 3.

- Щелкни по ссылке для скачивания последней версии дистрибутива ветви Python 3. (Когда я писал эту книгу, актуальной была версия 3.7.1.)
- Теперь найди и открой окно папки, в которую скачался файл. По умолчанию это папка **Загрузки** (рис. А.2).

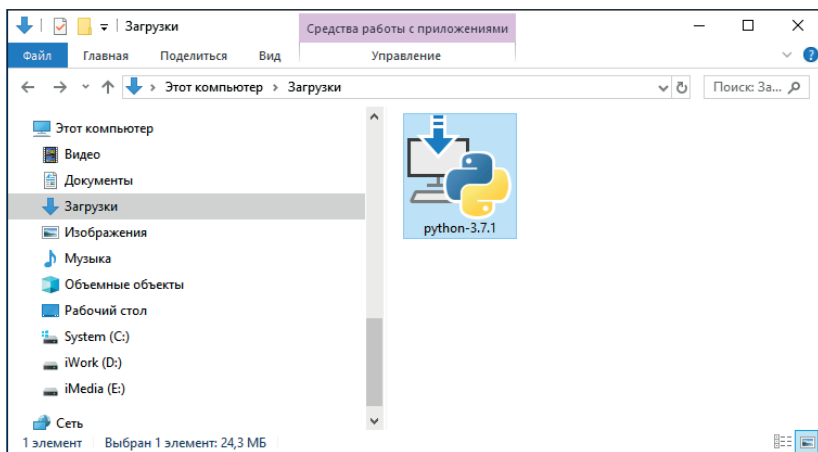


Рис. А.2. Содержимое папки **Загрузки**

- Дважды щелкни мышью по значку скачанного файла. Запустится программа установки (рис. А.3).

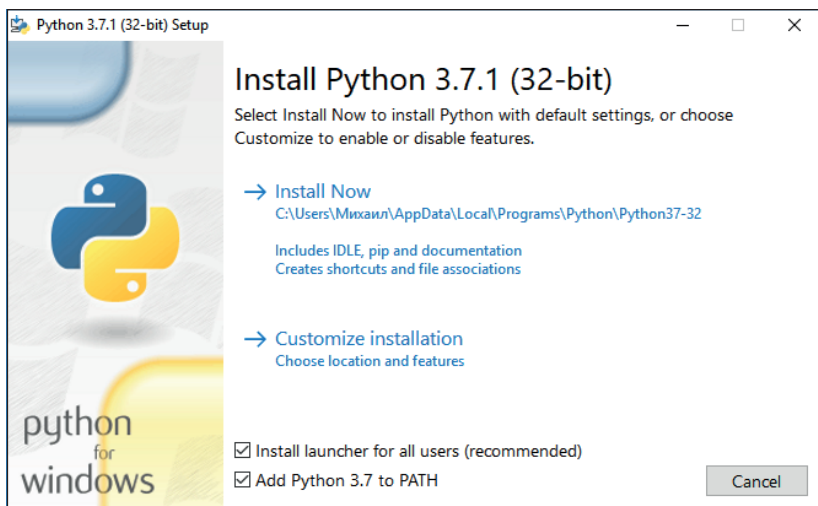


Рис. А.3. Окно мастера установки Python

5. Перед запуском процесса установки убедись, что установлены флажки **Install launcher for all users** (Установить запуск для всех пользователей) и **Add Python 3.7 to PATH** (Добавить Python 3.7 к пути PATH).
6. Теперь выбери пункт **Customize installation** (Настройка установки), потому что надо указать путь установки Python (рис. А.4).

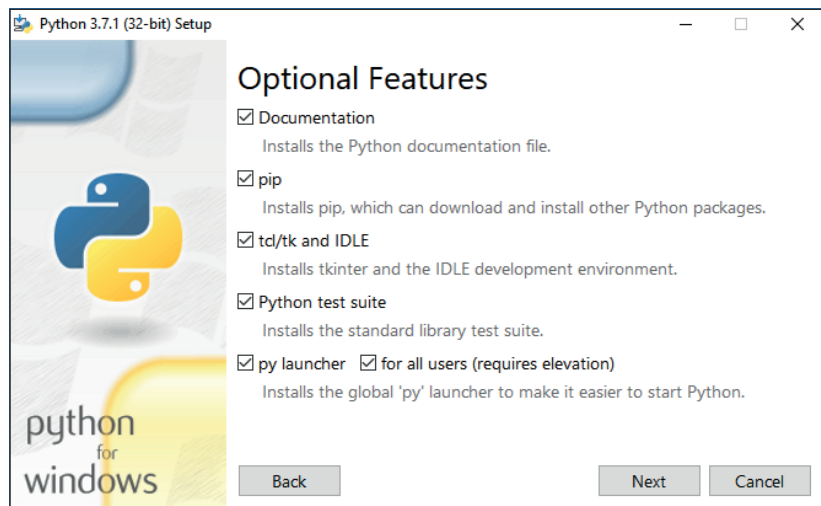


Рис. А.4. Настройка процесса установки

7. В следующем окне не надо ничего менять. Просто нажми кнопку **Next** (Далее), чтобы установить все указанные компоненты (рис. А.5).

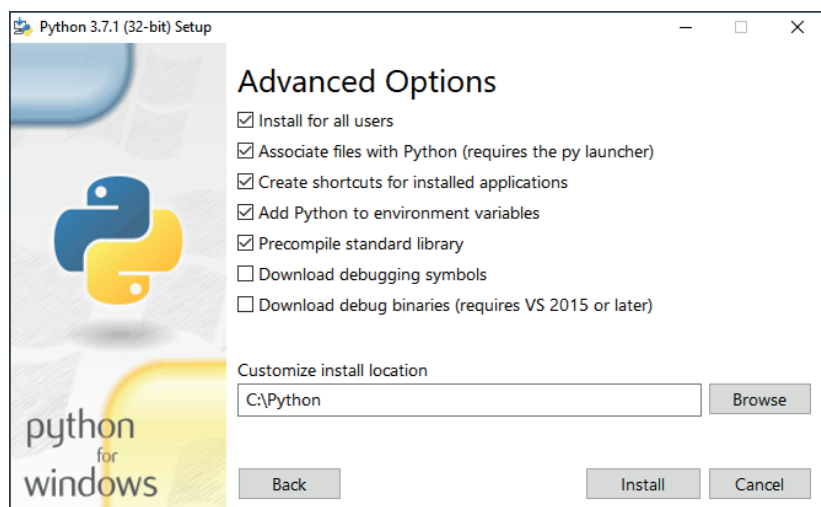


Рис. А.5. Окно выбора дополнительных компонентов

А

8. В следующем окне установи флажок **Install launcher for all users** (Установить запуск для всех пользователей).

Теперь надо указать, в какую папку должен быть установлен Python. Если этого не сделать, файлы будут скопированы в папку `C:\Users\твое_имя\AppData`.

9. Укажи путь `C:\Python` (как у меня на рисунке). Или нажми кнопку **Browse** (Обзор), чтобы выбрать папку, в которую должен быть установлен Python. Нажми кнопку **Install** (Установить).

Начнется установка. Теперь наберись терпения и немного подожди – процесс установки займет некоторое время. За процессом установки можно следить с помощью индикатора выполнения.

Наконец, ты увидишь сообщение о том, что процесс установки успешно завершен (рис. А.6).

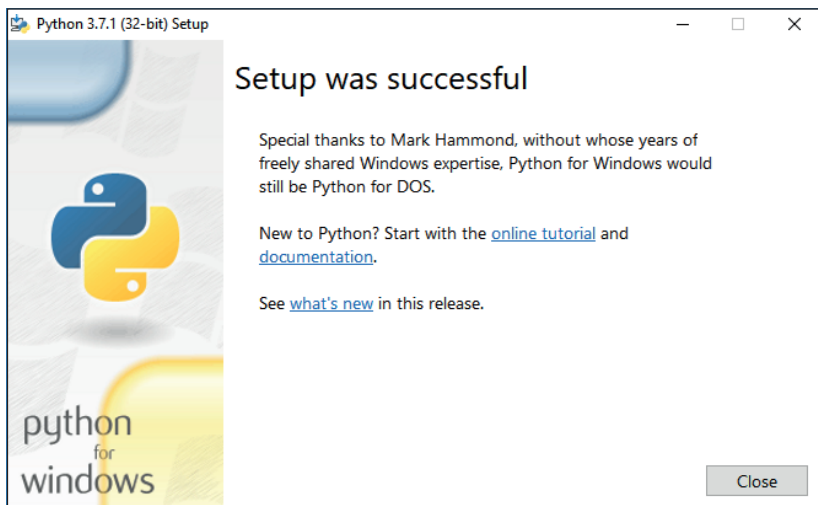


Рис. А.6. Процесс установки завершен

10. Нажми кнопку **Close** (Заккрыть).

На странице www.python.org ты всегда можешь скачать последнюю версию Python, а также просмотреть документацию и другие материалы.

Установка библиотеки Pygame, вариант 1

Поскольку библиотека Pygame не входит в состав дистрибутива Python, нужно установить ее отдельно. Давай это сделаем.

1. Запусти веб-браузер и перейди на страницу [pygame.org/download.shtml](https://www.pygame.org/download.shtml) (рис. А.7).

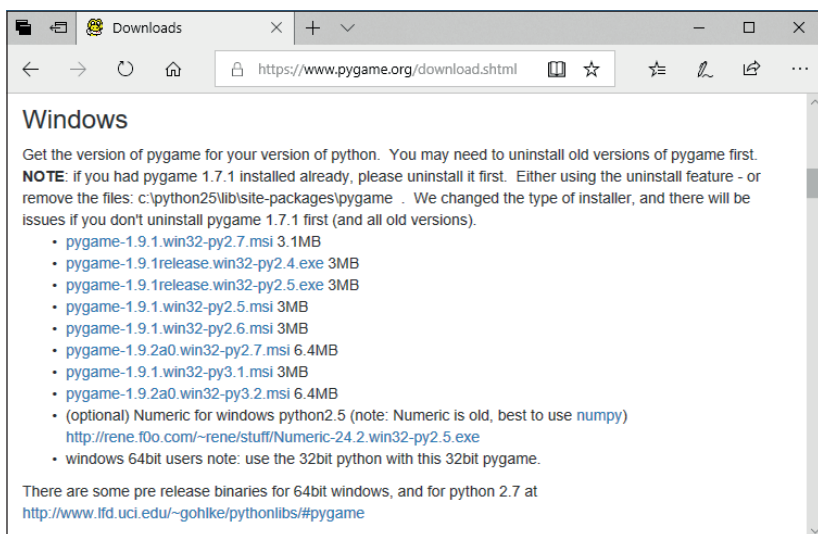


Рис. А.7. Страница загрузки дистрибутива Pygame

2. Прокрути содержимое страницы вниз до списка Windows. Выбери предпоследнюю ссылку – нужна версия, подходящая для Python 3.

Вероятно, на момент чтения книги появятся более новые версии. Поскольку Python непрерывно развивается, тебе также может потребоваться загрузить и установить несколько дополнительных файлов, которые работают с твоей версией Python.



3. В папке **Загрузки** ты увидишь значок загруженного файла (рис. А.8). В его имени содержится слово *pygame*.

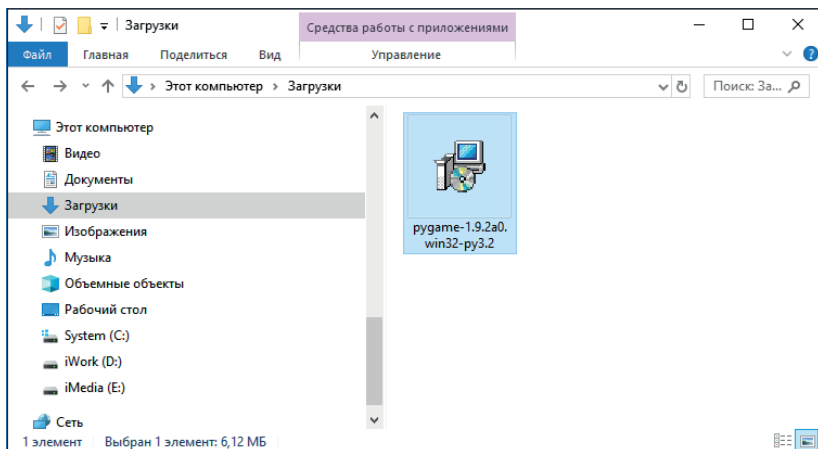


Рис. А.8. Файл установки Pygame

4. Дважды щелкни мышью по этому значку, чтобы начать установку.



Ты также можешь переместить программу установки в папку *Python* и перейти в нее, а затем начать установку.

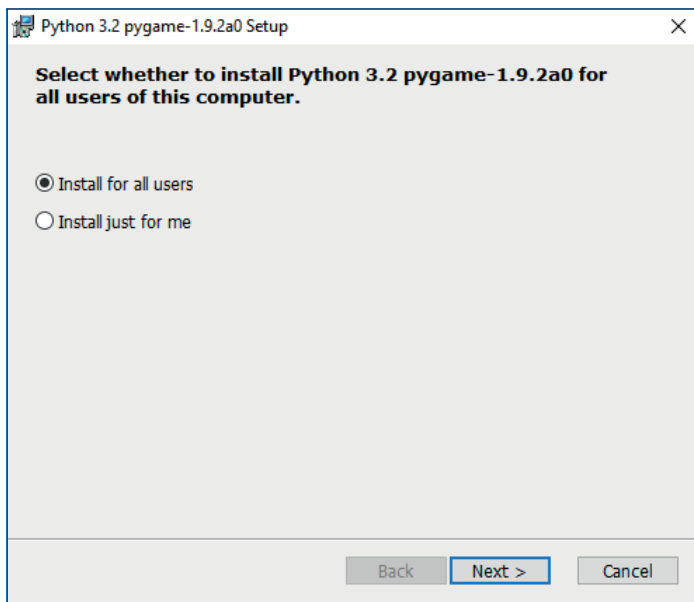


Рис. А.9. Выбор пользователей, которым будет доступна установленная программа

- Установи переключатель в положение **Install for all users** (Установить для всех пользователей) (рис. А.9), а затем нажми кнопку **Next** (Далее).

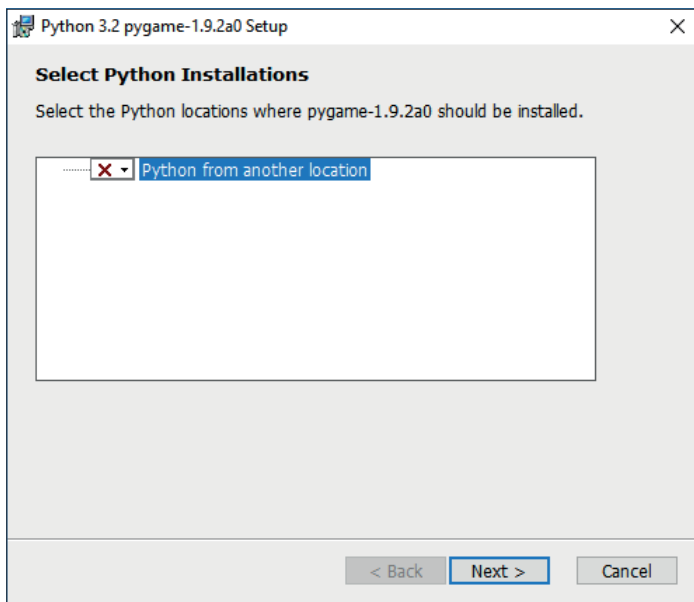


Рис. А.10. Окно выбора расположения Python

- В следующем окне ты можешь сразу нажать кнопку **Next** (Далее), если программа установки определила, куда установлен Python (рис. А.10). Если это не так, открой небольшое меню, щелкнув мышью по пункту **Python from another location** (Python в другом расположении).

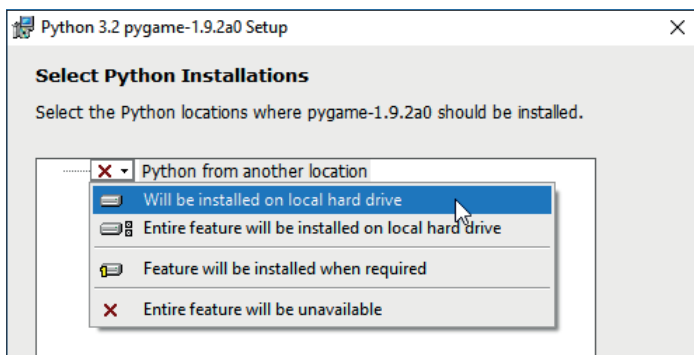


Рис. А.11. Выбор папки установки Python

А

7. Выбери первый пункт, чтобы установить Pygame на локальный жесткий диск (рис. А.11).

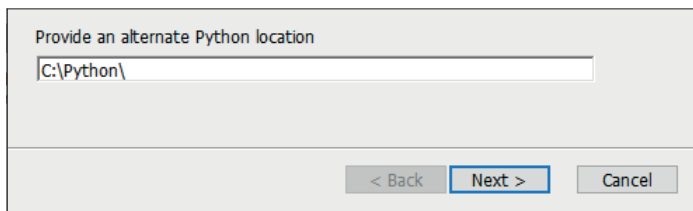


Рис. А.12. Указание пути к папке с Python

8. Теперь укажи путь к папке, в которую установлен Python (рис. А.12). Затем нажми кнопку **Next** (Далее).

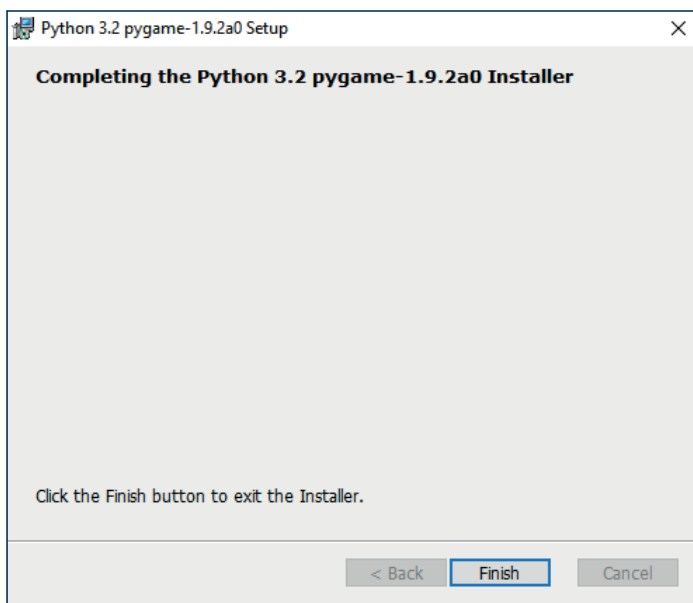


Рис. А.13. Завершение установки Pygame

9. Когда появится окно, показанное на рис. А.13, ты сможешь завершить установку, нажав кнопку **Finish** (Завершить).

Установка библиотеки Pygame, вариант 2

Первый способ установки работает почти всегда, но, к сожалению, может случиться так, что модуль Pygame не импортируется в программу Python, потому что отсутствуют необходимые файлы. В этом случае есть альтернативный способ.

Прежде чем приступить, нужно удалить установленный пакет Pygame. Это можно сделать, запустив программу установки еще раз и установив переключатель в положение, показанное на рис. А.14, а затем нажав кнопку **Finish** (Завершить).

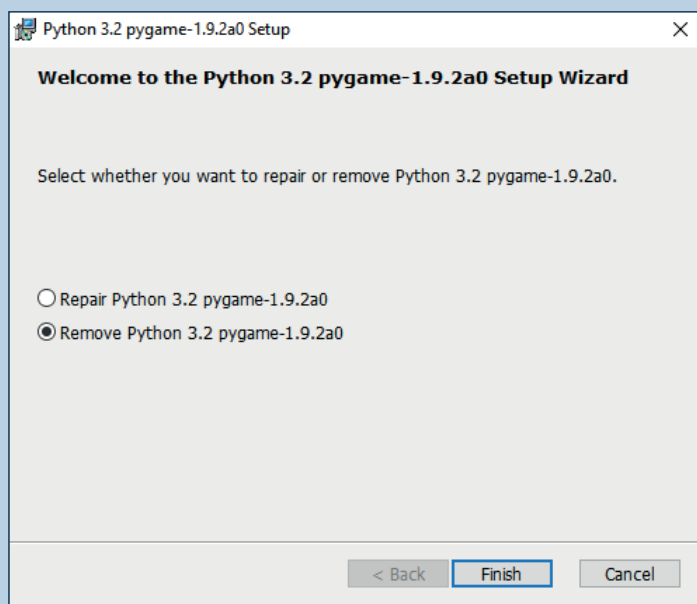


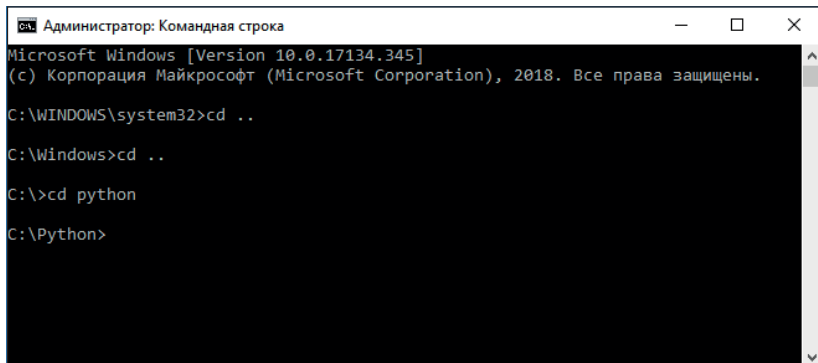
Рис. А.14. Удаление Pygame



Теперь Pygame можно установить.

Открой оболочку командной строки Windows (например, выбрав соответствующий пункт в меню **Пуск** (Start) или введя буквы **CMD** в окне поиска).

А



```
Администратор: Командная строка
Microsoft Windows [Version 10.0.17134.345]
(c) Корпорация Майкрософт (Microsoft Corporation), 2018. Все права защищены.

C:\WINDOWS\system32>cd ..

C:\Windows>cd ..

C:\>cd python

C:\Python>
```

Рис. А.15. Оболочка командной строки

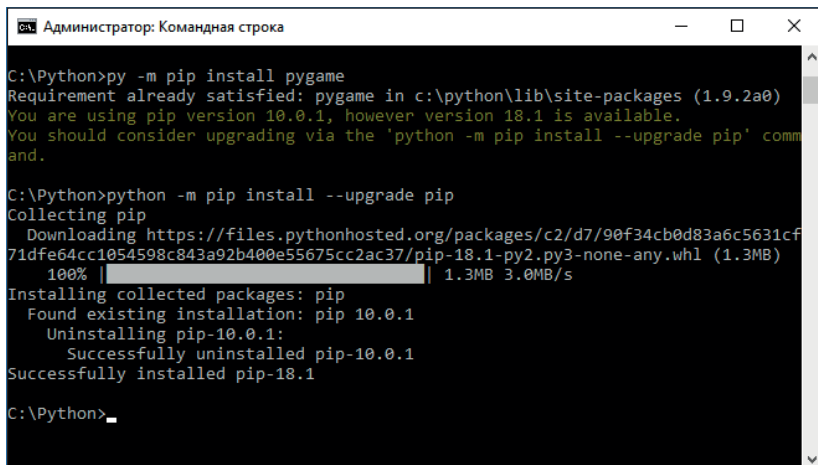
С помощью команды `cd` перейди в каталог `C:\Python` (рис. А.15). Затем выполни следующую команду:

```
py -m pip install pygame
```

Теперь дистрибутив Pygame, соответствующий твоей версии Python, будет загружен из интернета и распакован в каталоги `C:\Python\Lib` и `C:\Python\include`.



Если в оболочке командной строки появится запрос на обновление модуля `pip`, как показано на рис. А.16, выполни команду `python -m pip install --upgrade pip`.



```
Администратор: Командная строка

C:\Python>py -m pip install pygame
Requirement already satisfied: pygame in c:\python\lib\site-packages (1.9.2a0)
You are using pip version 10.0.1, however version 18.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

C:\Python>python -m pip install --upgrade pip
Collecting pip
  Downloading https://files.pythonhosted.org/packages/c2/d7/90f34cb0d83a6c5631cf71dfe64cc1054598c843a92b400e55675cc2ac37/pip-18.1-py2.py3-none-any.whl (1.3MB)
    100% |#####| 1.3MB 3.0MB/s
Installing collected packages: pip
  Found existing installation: pip 10.0.1
    Uninstalling pip-10.0.1:
      Successfully uninstalled pip-10.0.1
  Successfully installed pip-18.1

C:\Python>
```

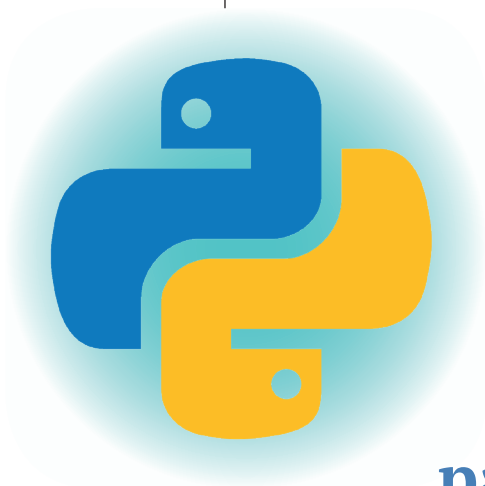
Рис. А.16. Установка Pygame в оболочке командной строки

Теперь ты можешь закрыть окно оболочки командной строки, Pygame установлен.

Файлы примеров

Если ты хочешь использовать файлы примеров из этой книги, то можешь скачать их на странице dmkpress.com.

Затем ты можешь разместить их в папке на жестком диске. Я скопировал эти файлы в папку `C:\Python\Projects`. (В папку `C:\Python` я ранее установил сам Python и модуль Pygame.)



Б

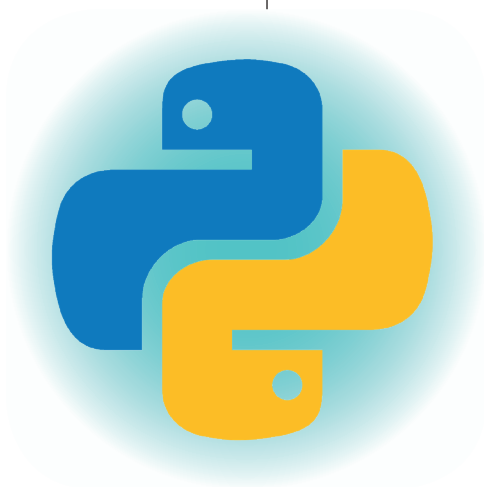
Решение распространенных проблем

Поиск ошибок раздражает многих разработчиков программ. К тому же самые сложные проблемы обычно настолько хорошо скрыты, что начинаешь сомневаться в том, что их вообще найдешь когда-нибудь. Во многих случаях следующий небольшой контрольный список поможет определить проблему:

- Не пропущены ли такие «мелочи», как запятая, точка, двоеточие?
- Все блоки кода программы (например, `if`, `while`, `for`, `try`, `def`) закрыты?
- Указаны ли двоеточия в конце строк условий и круглые скобки после имен функций, даже у не имеющих параметров?
- Выполняются ли условия в инструкциях `if` и `while`?
- Присвоены ли значения переменным и параметрам, обрабатываемым далее?
- Указано ли слово `self` в определении класса, перед атрибутом или в списке параметров?
- Совпадают ли типы данных слева и справа от оператора присваивания (`=`)?
- Не перепутаны ли операторы присваивания (`=`) и равенства (`==`)?

В

Ответы на вопросы и задачи



Глава 1

Вопросы

- 1 Для ввода и вывода информации в Python используются функции `print()` и `input()`. Но это не единственные функции с таким предназначением, о других ты узнаешь позже.
- 2 Интерпретатор переводит каждую строку программы в машинный язык для компьютера по отдельности, а компилятор – программу целиком.
- 3 Только на первый взгляд: инструкция присвоения, например `Number = 2 * 3`, похожа на уравнение, но код `Number = Number + 1` – это также инструкция присвоения.

Задачи

Нет задач.

Глава 2

Вопросы

- 1 Оператор с одинарным символом равенства (`=`) отвечает за присваивание, а с двойным (`==`) – за сравнение.

В

- 2 Фрагмент программы может быть упрощен с помощью инструкции `else`:

```
if Number == 0 :  
    print("Нет результата");  
else :  
    print(1/Number);
```

Задачи

- 1 См. файл *reciprocal.py*.
- 2 См. файл *password.py*.

Глава 3

Вопросы

- 1 Это операторы сравнения, является ли число меньше (<) или больше (>) указанного, причем указанное число не учитывается. Если ты хочешь включить указанное число в сравнение, используй операторы <= или >= соответственно. Например: условие `x < 5` выполняется (истинно), если `x` равен 1, 2, 3 или 4; а условие `x <= 5` выполняется, если `x` равен 1, 2, 3, 4 или 5.
- 2 Инструкция `Value + 1` увеличивает значение на 1, `Value - 1` — вычитает 1, а `Value * 2` — удваивает значение.
- 3 Случайное число для игральной кости генерируется следующей инструкцией:

```
Dice = random.randint(1,6)
```

Задачи

- 1 Дополнение `end = ""` можно увидеть в программах *mathe1a.py*, *mathe2a.py* и *mathe3a.py*.
- 2 Программа *grade2.py* поддерживает 10-балльную систему оценок.
- 3 Пример можно увидеть в файле *age.py*.

Глава 4

Вопросы

- 1 С помощью функции `sleep()` модуля `time` компьютер делает паузу, указанную в секундах. В функции `time.sleep(2)` используется, например, две полные секунды, а в `time.sleep(0,1)` — одна десятая секунды.

- 2 Начнется бесконечный цикл, потому что капитал никогда не станет желаемого значения.
- 3 Все три символа невидимы на экране: пробел – это маленький промежуток, который выглядит как один пустой символ (). Новая строка – это, так сказать, горизонтальный промежуток (между двумя строками). А пустая строка – это пустой текст ("").

Задачи

- 1 Программа называется *mathe5.py*.
- 2 Твой компьютер вычислит в программе *guess5.py* число, которое ты задумал, только если ты его не обманешь.
- 3 Аналогично, но компьютер сам с собой играет в программе *guess6.py*. Обмануть не получится.

Глава 5

Вопросы

- 1 Глобальные переменные применяются и существуют везде в программе, локальные переменные – только внутри блока инструкций (функция, условная конструкция).
- 2 Переменные могут использоваться как параметры, а также как фиксированные значения (константы).

Задачи

- 1 Версия программы со строками сохранена в файле *swar3.py*.
- 2 В файле *summe.py* введи, например, 10, тогда получишь сумму всех целых чисел от 1 до 10.
- 3 В файле *mean.py* введи, например, 10, тогда получишь среднее значение целых чисел от 1 до 10.
- 4 Нет примеров для этой задачи. Реши ее сам: в каких своих проектах ты захочешь использовать функции. Может быть, *Hello* или *Mathe*?

Глава 6

Вопросы

- 1 Переменные, используемые в классе, называются атрибутами, а функции, определенные в классе, называются методами.

В

- 2 Инкапсуляция означает, что несколько элементов объединяется в один класс. В этом случае доступ извне возможен только через ссылку на объект, который был создан как экземпляр этого класса.
- 3 Закрытые элементы могут использоваться только внутри объекта или класса, а защищенные элементы – во всех производных классах.

Задачи

Нет задач.

Глава 7

Вопросы

- 1 Пока что тебе известны компоненты `Label` и `Button`, а также `Tk` для класса окна и `messagebox` для окна сообщения.
- 2 Параметр `command` используется для связи компонента (например, кнопки) с функцией события.

Задачи

- 1 Ты найдешь предлагаемое решение в файле *horoskop1.py*.

Глава 8

Вопросы

- 1 Когда происходит событие `<<ListBoxSelect>>`, запись в списке активируется или выбирается. Функция связанного события оценивает результат `curselection()`, чтобы определить номер выбранного пункта.
- 2 Переключатели используют общую переменную (типа `IntVar`), с помощью которой можно определить номер позиции переключателя. Каждый флажок получает свою собственную переменную, определяющую, установлен он или нет.

Задачи

- 1 1/2/3. Решения можно найти в файлах *horoskop2.py* – *horoskop4.py*.

Глава 9

Вопросы

- 1 Компонент `Label` может отображать что-либо только при вводе чего-либо в компонент `Entry`.
- 2 Текстовый файл открывается с помощью метода `open()`. В зависимости от режима ("r" или "w") содержимое файла можно прочитать или записать что-либо в файл. Также это может быть выполнено с помощью методов `read()` и `write()`.

Задачи

- 1 Это уже сделано в файле *psych5.py*.
- 2 Такой ползунковый регулятор можно найти в файле *percent.py*.

Глава 10

Вопросы

- 1 Сначала создай экземпляр класса `Menu`. Это может быть строка меню, если это выпадающее меню. Функции `add_cascade()` и `add_command()` добавляют пункты в строку и команды в меню.
- 2 Для создания диалогового окна используй метод `messagebox`. Нажав кнопку, ты открываешь окно при вызове функции `showinfo()` в соответствующей функции события.

Задачи

Нет задач.

Глава 11

Вопросы

- 1 Начало системы координат на экране находится в верхнем левом углу и имеет координаты $x = 0$ и $y = 0$.
- 2 При рисовании линии, которая всегда является прямой, задаются координаты начальной и конечной точек. Для прямоугольников и эллипсов углы указываются слева сверху и справа внизу. В эту невидимую область должен быть вписан прямоугольник или эллипс.

В

Задачи

- 1 Решение можно найти в файле *graphic6a.py*.
- 2 Программа с разноцветными квадратами называется *graphic6b.py*.
- 3 Попытку нарисовать дом я предпринял в программе *house1.py*.

Глава 12

Вопросы

- 1 Сначала должен быть загружен файл изображения, затем создается соответствующий объект изображения:

```
Picture = PhotoImage (file = "Image.gif")  
Object = Graphic.create_image (x, y, image = Bild)
```

Задачи

- 1 Квадрат появляется и перемещается в программе *movie1A.py*.
- 2 Решение этой задачи в файлах *movie6.py* и *mplayerY.py*.

Глава 13

Вопросы

- 1 Поскольку спрайт часто перемещается по игровому полю, имеющему цвет или текстуру, рекомендуется использовать изображения с прозрачным фоном, чтобы ты не видел этот фон на поле.
- 2 Ты можешь обрабатывать с помощью `event.type` нажатия любых клавиш, даже их сочетаний. Более подробную информацию можно найти в документации Pygame на сайте www.pygame.org/docs/ref/key.html.

Задачи

- 1 В программе *buggy3a.py* жук «перебегает» на другую сторону.

Глава 14

Вопросы

- 1 Разумеется, ты можешь управлять персонажем с помощью мыши и клавиатуры сразу. Тебе просто нужно

оставить оба события в одной программе (как, например, в *buggy5B.py*).

- 2 С помощью модуля `math` у тебя в распоряжении есть множество математических функций.

Задачи

- 1 Решение задачи ты найдешь в файле *buggy5A.py*.

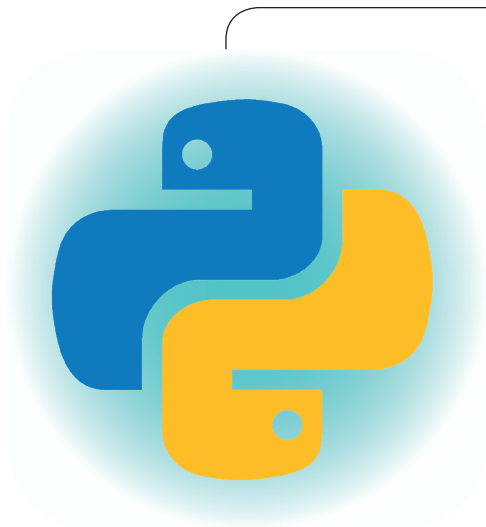
Глава 15

Вопросы

Нет вопросов.

Задачи

- 1 В файле *dodger4A.py* ты найдешь решение задачи.



Предметный указатель

Символы

`__init__`, 117

А

`and`, 63

С

`Canvas`, 212

загрузка изображения, 227

текст, 212

`command`

параметр, 139

Ф

`for` (цикл), 80

Н

`help()`, команда, 22

І

`IDLE`, 14

интерфейс, 27

команды, 38

скачивание, 14

создание ярлыка, 27

`input()`, функция, 26, 34

О

`or`, 63

Р

`PIL`, 228

`Pillow`, 228

`print()`, функция, 26, 34

`pygame`, 245

математические

вычисления, 271

управление объектом, 257

`Python`

всплывающее окно, 195

выполнение

программы, 32

границы окна, 264

завершение работы, 37

запись в файл, 183

запись файлов, 177

запуск, 19

история, 12

контекстное меню, 195

координаты, 201
математические
вычисления, 271
настройка, 19
открытие файла, 34
поворот объектов, 261
преимущества, 14
приглашение, 21
работа с цветом, 204
рисование мышью, 215
словарь, 94
сложение, 23
создание файла, 31
создание ярлыка, 20
сохранение файла, 32
сочетания клавиш, 198
спрайт, 251
текст, 23
управление объектом, 256
цветной текст, 212
числа, 23
чтение файлов, 177

S

self, 117

T

tkinter

entry, 172
slider, 176
всплывающее окно, 195
диалоговое окно, 145, 190
заголовок окна, 145
импорт, 136
инструкция grid, 142
инструкция pack, 137
инструкция place, 142
использование
классов, 145
кнопки, 152
компоненты, 136
контекстное меню, 195

координаты, 201
меню, 187
модуль, 135
переключатели, 157
ползунковый
регулятор, 173
работа с цветом, 204
рисование линий, 204
рисование мышью, 215
сетка, 142
создание рамки, 164
сочетания клавиш, 198
списки, 155
упаковка, 137
флажки, 160
цветной текст, 212
turtle, модуль, 218

A

Анимация, 223
Арифметические
операции, 50

Б

Блок инструкций, 43

В

Вектор, 269
Всплывающие окна, 194
Вычисления в Python, 48

Г

Генерация чисел, 63

Д

Деление на ноль, 54
Диалоговое окно
создание, 190
с помощью Python, 145
Дробные числа, 53

В

И

Изображение
 вращение, 238
 движение, 236
 исчезновение, 240
 обработка группы, 230
 появление, 240
Инкапсуляция, 114
Интерпретатор, 13, 22
Интерфейс, разметка, 142
Исключения, 55
 обработка, 55
Итерация, 74

К

Класс
 Player, 232
 внешний, 230
 дочерний, 121
 производный, 121
 родительский, 121
 управление, 281
 экземпляр, 114
Ключевое слово not, 90
Ключевые слова, 58
Код, 13
Команда
 continue, 74
 break, 73
Комментарии, 49
Компилятор, 13
Компоненты, 136
 tkinter, 136
Конструкция, 43
 for, 79
 if, 39
 if else, 44
 try except, 55
Контекстные меню, 194

Л

Логическая переменная, 89
Лямбда-выражения, 149

М

Машинный язык, 23
Меню, 186
 сочетания клавиш, 198
Методы трансформации, 261
Модуль, 125
 random, 65
 импорт, 65, 127
 создание, 125

Н

Наследование, 120

О

Обмен значений, 104
Обнаружение
 столкновений, 282, 299
Объект
 защищенный, 134
 приватный, 130
 публичный, 130
Оператор
 присваивания, 25
 сравнения, 41, 64
Операция сложения, 23
Отступ в коде, 43
Ошибка «функция
 не определена», 95

П

Палиндром, 90
Параметр, 24
 функции, 100
Переменная, 25
 глобальная, 98
 логическая, 89
 локальная, 98
Преобразование строк
 в целые числа, 51
Преобразование
 типов, 51
 программы, 55

Приглашение, 21

Программирование, 12

Р

Рендеринг, 250

С

Связывающие
операторы, 63

Смена изображений, 281

Сортировка чисел, 108

Среда разработки, 13, 14

Строка, 34

Т

Текстовый редактор, 13

У

Условия объединения, 61

Условная конструкция, 43

Ф

Формула, 35

Функция

append(), 88, 110, 114, 152

display, 137

exit(), 73

input(), 91

playGame(), 97

randint(), 65

range(), 88

sleep(), 78

str(), 72

Ц

Цикл

for, 80

while, 67

итерация, 74

Ч

Черепашья графика, 217

Число с плавающей
запятой, 53

Я

Язык программирования, 12

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planet.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@aliants-kniga.ru**.

Ханс-Георг Шуман

Python для детей

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Райтман М. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 27,95. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**