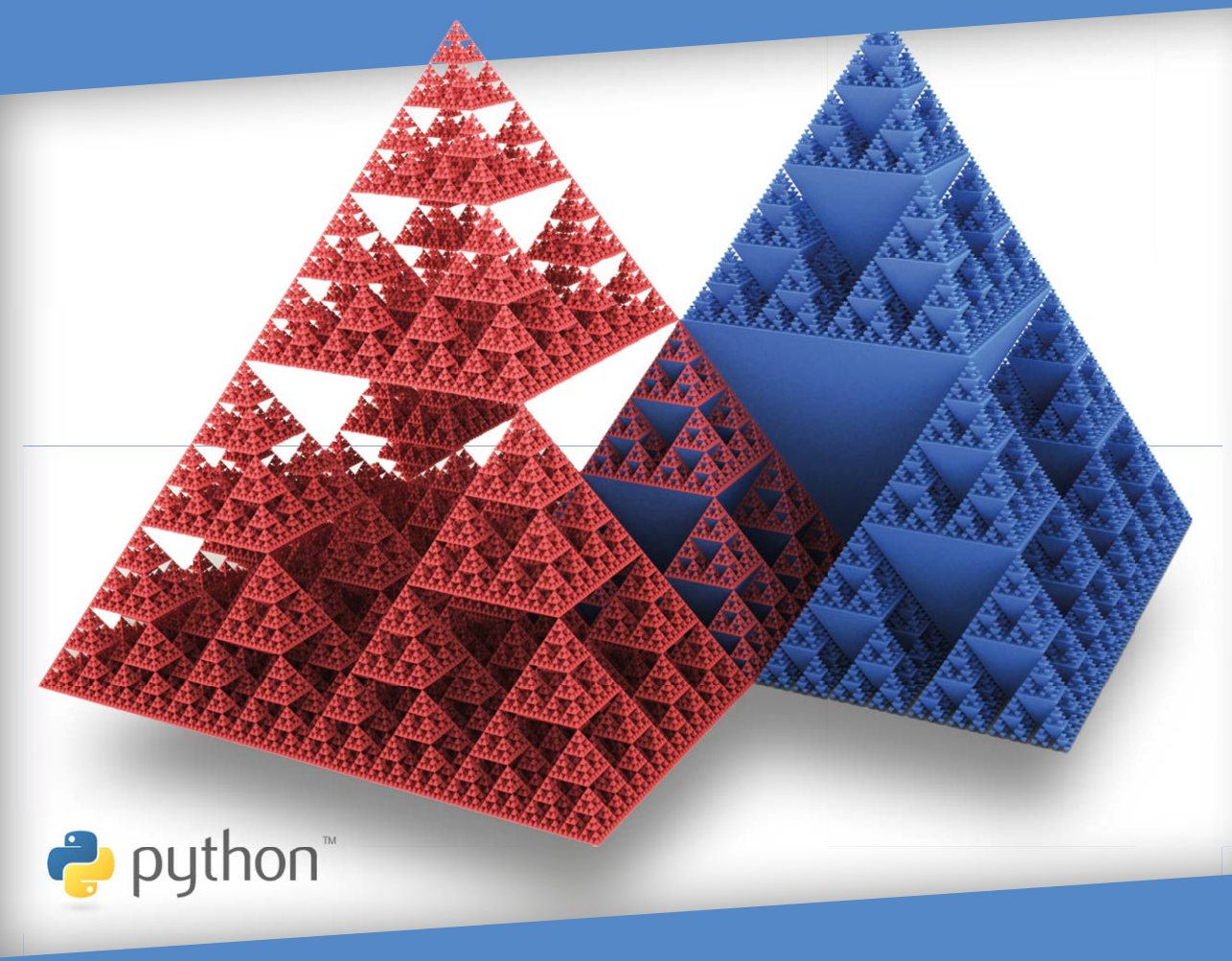
elektor books

Кикстарт на Python 3

Курс сверхбыстрого программирования



Ашвин Панджакар

elektor
design › share › sell

Кикстарт на Python 3

Курс сверхбыстрого программирования

Ашвин Панджакар

Глава 1. Введение в Python

Надеюсь, что вы просмотрели оглавление книги. Если нет, то я прошу вас сделать это, поскольку это даст читателям четкое представление о содержании этой главы. Если вы не совсем новичок в Python, эта глава может показаться вам очень простой. Однако если вы новичок в Python или компьютерном программировании, эта глава окажется для вас очень полезной.

В этой главе мы начнем наше путешествие с маленьких и простых шагов. В этой главе мы изучим следующие темы:

- История языка программирования Python.
- Установка Python на различные платформы.
- IDLE
- Скриптовый режим Python
- IDE Python
- Реализации и дистрибутивы Python.
- Индекс пакетов Python.

После этой главы мы освоим основы Python и запустим простые программы на Python.

1.1 История языка программирования Python

Python — это интерпретируемый язык программирования высокого уровня общего назначения. Он был создан с целью сделать код легко читаемым. Код Python можно охарактеризовать синтаксисом, подобным английскому языку. Ее легко прочитать и понять человеку, который только начал учиться программировать. Он заимствует множество функций из других языков программирования.

Язык программирования Python находится под сильным влиянием ABC, который был разработан в Центре Wiskunde & Informatica (CWI). Сам ABC находился под сильным влиянием SETL и ALGOL 68.

Основным автором Python является Гвидо фон Россум. Раньше он работал с языком программирования ABC в CWI. Сообщество Python присвоило ему титул **Доброжелательного диктатора на всю жизнь** (BDFL).

Идея языка программирования Python возникла в конце 1980-х годов как преемника языка программирования ABC. Гвидо также позаимствовал систему модулей у Модулы-3. Он начал реализацию языка в 1989 году. Версия 0.9.0 была опубликована на alt.sources в феврале 1991 года. Версия 1.0 была опубликована в 1994 году. Python 2.0 был выпущен в октябре 2000 года.

В декабре 2008 года была выпущена новая (и обратно несовместимая) версия языка программирования Python, известная как Python 3. 1 января 2020 года Python 2 поддерживается и больше не разрабатывается. Сейчас единственной версией является Python 3. Он находится в активной разработке и поддерживается Python Software Foundation. Мы будем использовать Python3 для всех демонстраций в книге. Всякий раз, когда я говорю о Python, я имею в виду Python 3.

1.2 Установка Python на различные платформы

В этом разделе мы узнаем, как установить Python 3 в Windows и Linux. Python поддерживает широкий спектр операционных систем. Однако наиболее часто используемые операционные системы для разработки на Python — это Windows и Linux. Поэтому я объясню Python 3 на примере этих двух платформ.

1.2.1 Установка в Linux

Почти все основные дистрибутивы Linux поставляются с установленными Python 2 и3. Интерпретатор Python 2 — это двоичный исполняемый файл Python, а интерпретатор Python 3 — это двоичный исполняемый файл **Python3**. Я использую вариант **Debian** Linux Raspberry Pi OS с Raspberry Pi 4 в качестве предпочитаемой платформы Linux для разработки на Python. Откройте эмулятор терминала вашего дистрибутива Linux и выполните следующую команду:

```
pi@raspberrypi:~ $ python3 -V
```

Он покажет следующий вывод:

```
Python 3.7.3
```

1.2.2 Установка в Windows

Установить Python на компьютер под управлением Windows очень легко и просто. Нам нужно посетить домашнюю страницу Python в Интернете, расположенную по адресу www.python.org. Если мы наведем указатель мыши на раздел **Downloads**, веб-сайт автоматически определит операционную систему и отобразит соответствующий файл для загрузки. Нажмите кнопку с надписью **Python 3.9.7**. Этот номер версии изменится в будущем, но процесс тот же. Как только вы нажмете кнопку, исполняемый установочный файл Windows будет загружен в каталог **Downloads** на вашем компьютере. Он определяет архитектуру вашего компьютера и загружает соответствующий файл. Например, у меня есть компьютер с 64-разрядной версией Windows x86. Он загрузил файл **python-3.9.7-amd64.exe.** Скриншот сайта www.python.org показан на рисунке 1-1:

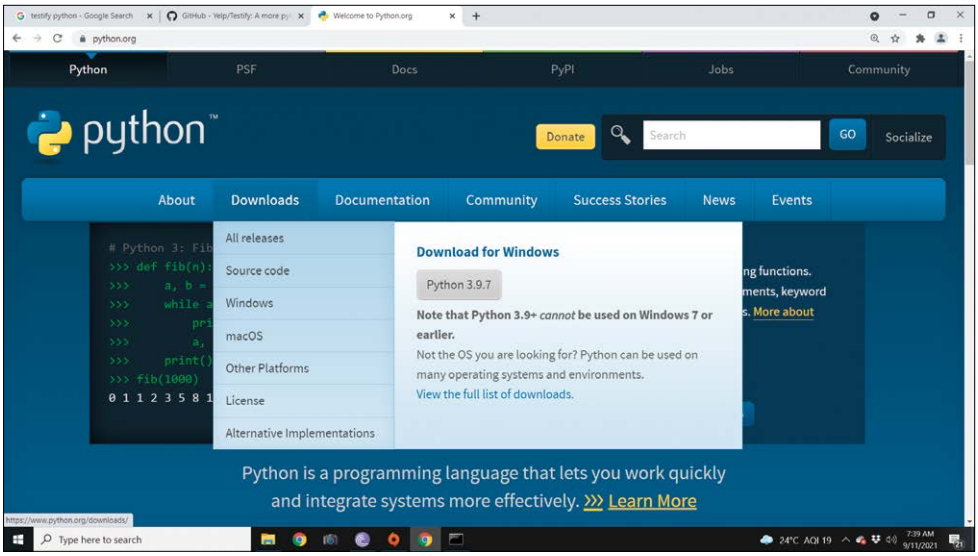


Рис.*1-1:* Загрузка установки *Python* для *Windows*

Теперь откройте установочный файл, и он запустит программу установки.

ПРИМЕЧАНИЕ: Вы можете открыть файл, щелкнув загруженный файл в браузере. Вы также можете определить физическое местоположение файла, воспользовавшись опцией загрузки в своем браузере. Он будет указывать на местоположение в каталоге **Downloads** вашего Windows.

Окно программы установки показано на скриншоте на рис.1-2:

Рис.*1-2:* Программа установки *Python*

Установите все флажки и нажмите кнопку **Customize installation**. Откроется следующий экран установки, как показано на рис.3:

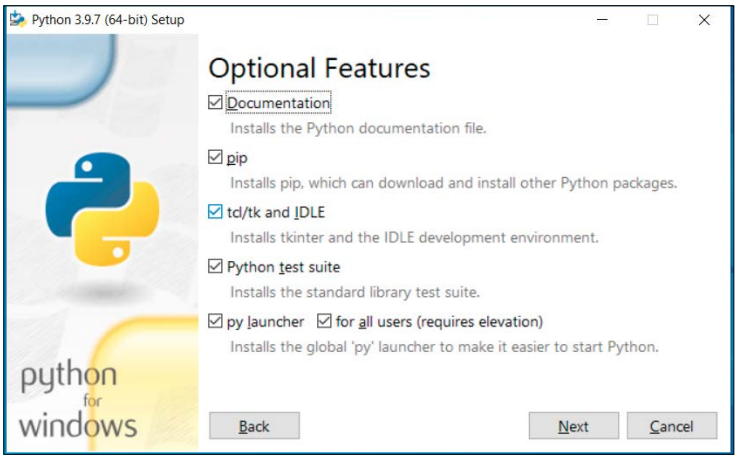


Рис.*1-3:* Варианты установки

Нажмите кнопку **Next**, и вы попадете на экран, показанный на рис.1-4:

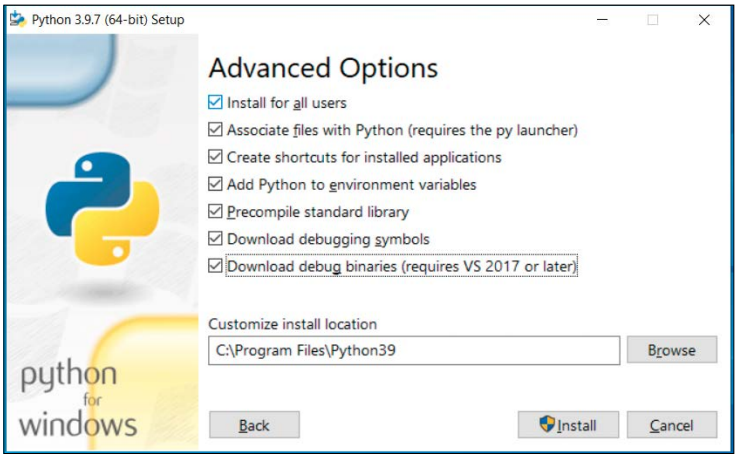


Рис.*1-4:* Варианты установки

Проверьте все параметры и нажмите кнопку `Install`. Затем он запросит учетные данные администратора. Введите учетные данные, и начнется установка Python и других соответствующих программ на ваш компьютер с Windows. После завершения установки появится следующий экран (рис. 1-5):

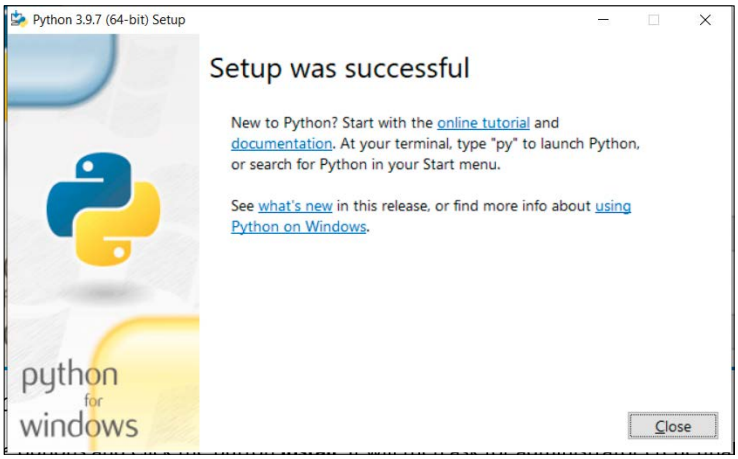


Рис.*1-5:* Сообщение об успешной установке

Прежде чем мы начнем праздновать, нам нужно проверить несколько вещей. Откройте командную строку Windows (cmd) и выполните следующую команду:

```
C:\Users\Ashwin>python -V
```

Результат следующий:

Python 3.9.7

Поздравляем! Мы установили Python 3 на ваш компьютер с Windows.

1.3 IDLE

Python Software Foundation разработал интегрированную среду разработки (IDE) для Python. Он получил название «IDLE», что означает «**Интегрированная среда разработки и обучения**». Он поставляется вместе с настройкой Python при установке в Windows. В дистрибутивах Linux его необходимо устанавливать отдельно. Для Debian и его производных выполните следующую команду в командной строке (эмулятор терминала):

```
pi@raspberrypi:~ $ sudo apt-get install idle -y
```

Он установит IDLE в ваш дистрибутив Linux.

Давайте поработаем с IDLE. Мы можем найти его, набрав IDLE в строке поиска Windows. Мы можем найти его в меню Linux. В Linux мы можем запустить его из командной строки, выполнив следующую команду:

```
pi@raspberrypi:~ $ idle &
```

Окно IDLE выглядит следующим образом (рис.1-6):

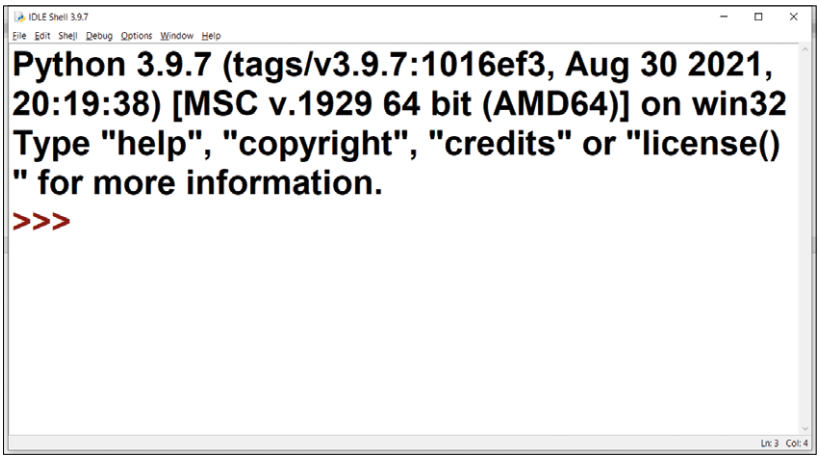


Рис.1-6: IDLE

В строке меню, в пункте меню **Options**, мы можем найти параметр **Configure IDLE**, где мы можем установить размер шрифта и другие детали (рис.1-7).

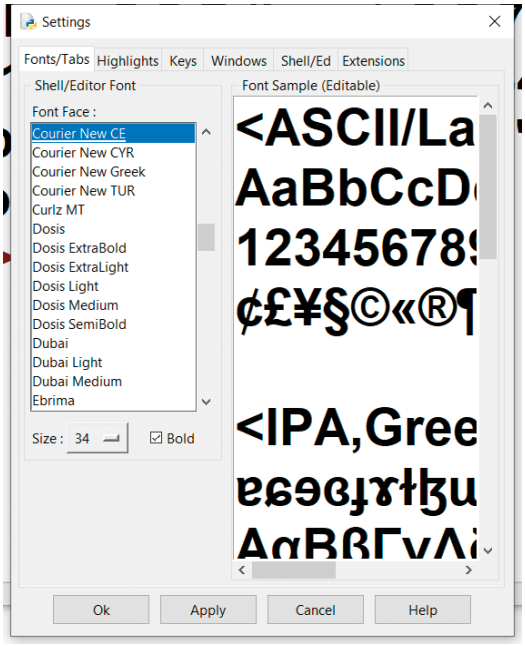


Рис.*1-7:* Варианты конфигурации *IDLE*

Измените шрифт и размер по вашему выбору и нажмите кнопки «**Apply** - Применить» и «OK» соответственно. Теперь давайте попробуем понять интерактивный режим Python. При вызове Windows IDLE отображает интерактивный режим. Мы можем использовать его для запуска операторов Python напрямую, без сохранения.

Операторы Python передаются непосредственно в интерпретатор, и результат немедленно отображается в том же окне. Если вы когда-либо работали с командной строкой ОС, то это почти то же самое. Интерактивный режим обычно используется для выполнения отдельных операторов или коротких блоков кода. Давайте попробуем запустить простой оператор:

```
>>> print("Hello World!")
```

Это дает следующий результат и печатает его в том же окне:

Hello World!

Курсор возвращается к приглашению и готов получить новую команду пользователя. Таким образом мы выполнили наш первый оператор **Python**. Мы можем выйти из интерпретатора, выполнив в интерпретаторе команду **exit()**. Альтернативно, мы можем нажать **CTRL + D**, чтобы выйти. Мы также можем вызвать интерактивный режим из командной строки, выполнив команды **python** Windows и **python3** в командной строке Linux.

1.4 Скриптовый режим Python

Интерактивный режим Python хорош для однострочных операторов и небольших блоков кода. Однако он не сохраняет операторы как программы. Это можно сделать в режиме скриптов. В интерпретаторе Python IDLE в меню **File** в строке меню выберите **New File**. Введите сюда следующий код:

```
print("Hello World!")
```

В меню File в строке меню выберите **Save As**. Откроется окно «**Сохранить как**». Сохраните его в выбранном вами месте. IDLE автоматически добавляет расширение ***.py** в конец имени файла. Затем нажмите меню **Run** в строке меню и нажмите **Run Module.** Это приведет к выполнению программы и отображению результатов в окне интерактивного режима IDLE, как показано ниже (рис.1-8).

Рис/1-8: Вывод скрипта Python

Другой способ выполнить программу — запустить ее из командной строки. Чтобы запустить программу в командной строке, перейдите в каталог, в котором хранится программа, а затем выполните следующую команду в Windows:

```
C:\ Python Book Project\Code\Chapter01>python prog01.py
```

Это дает следующий результат:

```
Hello World!
```

В Linux команда выглядит следующим образом:

```
pi@raspberrypi:~ $ python3 prog01.py
```

Есть еще один способ запустить программу в Linux, выполнив следующую команду:

```
pi@raspberrypi:~ $ which python3
```

Она возвращает местоположение интерпретатора Python 3:

```
/usr/bin/python3
```

Теперь давайте добавим следующую строку кода в созданный нами файл кода Python:

```
#!/usr/bin/python3
```

Таким образом, весь файл кода будет выглядеть так:

```
#!/usr/bin/python3
print("Hello World!")
```

Теперь давайте изменим права доступа к файлу программы Python. Предполагая, что мы сохранили файл с именем **prog01.py**, выполните следующую команду:

```
pi@raspberrypi:~ $ chmod +x prog01.py
```

Таким образом мы изменили права доступа к файлу и сделали его исполняемым. Мы можем запустить файл следующим образом:

```
pi@raspberrypi:~ $ ./prog01.py
```

Обратите внимание, что мы добавили к имени файла префикс **./**.
Это дает следующий результат:

```
Hello World!
```

Мы добавили первую строку, чтобы оболочка операционной системы знала, какой интерпретатор использовать для запуска программы.

Вот несколько способов запуска программ Python.

1.5 IDE Python

До сих пор мы использовали IDLE для программирования на Python. Мы также можем использовать другие редакторы и IDE. В командной строке Linux мы можем использовать программы-редакторы, такие как vi, vim и nano. Редактор Thevi поставляется с большинством дистрибутивов Linux. Мы можем установить два других в Debian (и производные дистрибутивы), используя следующую команду:

```
pi@raspberrypi:~ $ sudo apt-get install vim nano -y
```

Мы также можем использовать текстовый редактор, например **Notepadon Windows** или **Leafpadon** Linux. Мы можем установить редактор **Leafpad** в Debian и других дистрибутивах, используя следующую команду:

```
pi@raspberrypi:~ $ sudo apt-get install leafpad -y
```

ОС Raspberry Pi (моя любимая производная от Debian) поставляется с Thonny, Geany и MuIDE. Мы можем установить их на другие производные Debian с помощью следующей команды:

```
pi@raspberrypi:~ $ sudo apt-get install thonny geany mu-editor -y
```

Если вам удобнее работать с **Eclipse**, есть хороший плагин, известный как **Pydev**. Его можно установить с **Eclipse Marketplace**.

1.6 Реализации и дистрибутивы Python

Программа, которая интерпретирует и запускает программу Python, называется интерпретатором Python. Программа, которая поставляется с Linux по умолчанию и предоставляется Python Software Foundation, известна как CPython. Другие организации создали интерпретаторы Python, соответствующие стандартам Python. Эти интерпретаторы известны как реализации Python. Подобно тому, как C и C++ имеют множество компиляторов, Python имеет множество реализаций интерпретаторов. На протяжении всей книги мы будем использовать стандартный интерпретатор CPython, который по умолчанию поставляется с Linux. Ниже приводится неполный список других популярных альтернативных реализаций интерпретатора Python:

1. IronPython
2. Jython
3. PyPy
4. Stackless Python
5. MicroPython

Многие организации объединяют интерпретатор Python по своему выбору со множеством модулей и библиотек и распространяют его. Эти пакеты известны как дистрибутивы Python. Мы можем получить список реализаций и дистрибутивов Python по следующим URL-адресам:

https://www.python.org/download/alternatives/
https://wiki.python.org/moin/PythonDistributions
https://wiki.python.org/moin/PythonImplementations

1.7 Индекс пакетов Python

Python поставляется с множеством библиотек. Это известно как философия Python «Batteries included - Батареи включены». Многие другие библиотеки разрабатываются многими сторонними разработчиками и организациями. В зависимости от вашего профиля работы эти библиотеки могут оказаться полезными для выполнения намеченных задач. Все сторонние библиотеки размещаются в индексе пакетов Python. Он находится по адресу https://pypi.org/. Мы можем найти библиотеку на этой странице.

В состав Python входит утилита **pip**. Пип — это обратная аббревиатура. Это означает, что расширение термина имеет сам термин. Pip означает, что **pip** устанавливает пакеты или **pip** устанавливает Python. Это инструмент управления пакетами, который устанавливает пакеты Python в виде утилиты командной строки. Мы можем проверить его версию, выполнив следующую команду в командной строке (cmd и эмулятор терминала) ОС:

```
pi@raspberrypi:~ $ pip3 -V
```

Она напечатает текущую установленную версию pip. Если мы хотим увидеть список установленных на данный момент пакетов, нам нужно запустить следующую команду:

```
pi@raspberrypi:~ $ pip3 list
```

Она вернет список всех уже установленных пакетов.

ПРИМЕЧАНИЕ: Все команды, связанные с pip, одинаковы в Windows и Linux.

Если мы хотим установить новую библиотеку, мы можем найти ее в PyPI. Мы можем установить новый пакет следующим образом:

```
pi@raspberrypi:~ $ pip3 install numpy
```

Он установит библиотеку NumPy на компьютер. Мы будем использовать эту утилиту на протяжении всей книги для установки необходимых сторонних пакетов.

Краткое содержание

В этой главе мы рассмотрели историю языка программирования Python и его установку в Windows и Linux. Мы также увидели, как использовать интерпретатор Python и как писать и выполнять сценарии Python различными способами. Мы сделали обзор IDLE, а также рассмотрели различные другие IDE для Python. Наконец, мы научились использовать **pip**, менеджер пакетов Python.

Следующая глава будет гораздо более практической. Мы научимся писать программы со встроенными структурами данных.

Глава 2 • Встроенные структуры данных

В предыдущей главе мы установили Python 3 на различные платформы и написали простую вводную программу и научились запускать ее различными способами. Также мы научились работать в режиме переводчика (интерактивном). Эта глава была вводной и не была слишком сложной по программированию.

Эта глава посвящена программированию (также известному как кодирование). Мы познакомимся с различными встроенными структурами данных в Python,где остановимся на следующих темах:

- IPython
- Список
- Кортеж
- Набор
- Словарь

После этой главы мы освоим IPython и встроенные структуры данных в Python.

2.1 IPython

IPython означает **интерактивную оболочку Python**, которая предоставляет нам больше возможностей, чем встроенная интерактивная оболочка Python. Нам нужно установить его отдельно, используя следующую команду:

```
pi@raspberrypi:~ $ pip3 install ipython
```

Команда одинакова для Windows и macOS. Если вы устанавливаете его в дистрибутив Linux (как я), вы можете увидеть следующее сообщение в журнале установки:

> **Скрипты iptest, iptest3, ipython и ipython3 устанавливаются в каталог /home/pi/.local/bin, которого нет в PATH.**
>
> **Рассмотрите возможность добавления этого каталога в PATH или, если вы предпочитаете скрыть это предупреждение, используйте --no-warn-script-location.**

Это означает, что нам нужно добавить указанный каталог в **~/.bashrcand~/.bash_profilefiles**. Нам нужно добавить следующую строку в оба файла (чтобы это работало для оболочек с входом и без входа):

```
PATH=$PATH:/home/pi/.local/bin
```

Аналогичное сообщение будет показано и для Windows. Нам также необходимо добавить путь к каталогу, указанному в журнале установки, к переменным **PATH** (обе переменные пользователя и системы) в Windows.

Как только мы изменим путь, нам придется закрыть и повторно вызвать утилиты командной строки операционных систем. После этого выполните следующую команду:

```
pi@raspberrypi:~$ ipython3
```

Это запустит IPython для Python 3 в командной строке. Команда одинакова для Windows и других платформ. На рис. 2-1 показан снимок экрана текущего сеанса IPython на настольном компьютере под управлением Windows:

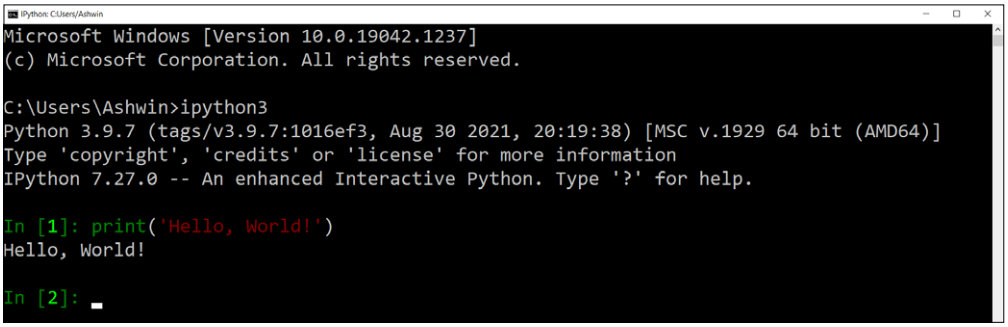


Рис.*2-1:* Выполняется сеанс *IPython*

Мы можем использовать IPython для написания небольших программ. Итак, поехали.

2.2 Списки

Мы можем хранить более одного значения в списках. Списки — это встроенная функция Python. При использовании списков нам не нужно ничего устанавливать или импортировать. Элементы списка заключаются в квадратные скобки и разделяются запятыми. Но списки ни в коем случае не являются линейными структурами данных, поскольку у нас также может быть список списков. Подробнее о списке списков мы поговорим позже, когда освоимся с основами.

Списки изменяемы, то есть мы можем их изменять. Давайте посмотрим несколько примеров списков и связанных с ними встроенных процедур. Откройте IPython в командной строке вашей операционной системы и следуйте коду:

```
In [1]: fruits = ['babana', 'pineapple', 'orange']
```

Он создаст список. Теперь мы можем вывести его на консоль двумя способами:

```
In [2]: fruits
```

Это дает следующий результат:

```
Out[2]: ['babana', 'pineapple', 'orange']
```

Мы также можем использовать встроенную функцию печати следующим образом:

```
In [3]: print(fruits)
```

Ниже приводится вывод:

```
['banana', 'pineapple', 'orange']
```

Список — это упорядоченная структура данных. Это означает, что элементы списков хранятся и извлекаются в определенном порядке. Мы можем использовать это в своих интересах для извлечения элементов списков. Самый первый элемент имеет индекс 0. Если размер списка равен n, последний элемент имеет индекс n-1. Это похоже на индексы массива в C, C++ и Java. Если вы раньше программировали на этих языках программирования, эта схема индексации покажется вам знакомой.

Мы можем получить элементы списков, используя комбинацию имени списка и индекса элемента. Ниже приведен пример:

```
In [4]: fruits[0]
Out[4]: 'banana'
In [5]: fruits[1]
Out[5]: 'pineapple'
In [6]: fruits[2]
Out[6]: 'orange'
```

Мы также можем использовать отрицательные индексы. -1 относится к последнему элементу, а -2 относится к последнему элементу кнопки. Ниже приведен пример:

```
In [7]: fruits[-1]
Out[7]: 'orange'
In [8]: fruits[-2]
Out[8]: 'pineapple'
```

Если мы попытаемся использовать неверный индекс, мы увидим следующие результаты:

```
In [9]: fruits[3]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-7ceeafd384d7> in <module>
----> 1 fruits[3]

IndexError: list index out of range
In [10]: fruits[-4]

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
```

```
<ipython-input-10-1cb2d66442ee> in <module>
----> 1 fruits[-4]

       IndexError: list index out of range
```

Мы можем получить длину списка следующим образом:

```
In [12]: len(fruits)
Out[12]: 3
```

Мы также можем увидеть тип данных списка следующим образом:

```
In [13]: type(fruits)
Out[13]: list
In [14]: print(type(fruits))
<class 'list'>
```

Как мы видим в выводе, класс переменной — список. Об этом мы подробно узнаем в четвертой главе книги.

Мы можем использовать конструктор list() для создания списка:

```
In [15]: fruits = list(('banana', 'pineapple', 'orange'))
In [16]: fruits
Out[16]: ['banana', 'pineapple', 'orange']
```

Мы можем получить диапазон элементов из списков следующим образом:

```
In [17]: SBC = ['Raspberry Pi', 'Orange Pi', 'Banana Pi', 'Banana Pro', 'NanoPi',
'Arduin
    ...: o Yun', 'Beaglebone']
In [18]: SBC[2:5]
Out[18]: ['Banana Pi', 'Banana Pro', 'NanoPi']
```

В этом примере мы извлекаем элементы с индексами 2, 3 и 4. Также рассмотрим следующий пример:

```
In [19]: SBC[2:]
Out[19]: ['Banana Pi', 'Banana Pro', 'NanoPi', 'Arduino Yun', 'Beaglebone']
```

Таким образом, мы можем получить элементы, начиная с индекса 2 и далее.

```
In [20]: SBC[:2]
Out[20]: ['Raspberry Pi', 'Orange Pi']
```

Таким образом, мы можем получить все элементы до индекса 2. Мы также можем использовать отрицательные индексы для получения нескольких элементов следующим образом:

```
In [21]: SBC[-4:-1]
Out[21]: ['Banana Pro', 'NanoPi', 'Arduino Yun']
```

Мы также можем использовать конструкцию **if** для проверки существования элемента в списке

```
In [23]: if 'Apple Pie' in SBC:
    ...:     print('Found')
    ...: else:
    ...:     print('Not Found')
    ...:
       Not Found
```

Мы можем изменить элемент в списке:

```
In [25]: SBC[0] = 'RPi 4B 8GB'
```

Мы также можем вставить элемент в список по индексу:

```
In [36]: SBC.insert(2, 'Test Board')
```

Элемент с индексом 2 в этом списке сдвигается на одну позицию вперед. То же самое и с остальными предметами.

Мы можем добавить элемент в список следующим образом:

```
In [38]: SBC.append('Test Board 1')
```

Он добавит элемент в конец списка. Мы также можем использовать операцию расширения со списками. Это добавляет один список в конец другого списка.

```
In [39]: list1 = [1, 2, 3]; list2 = [4, 5, 6];
In [40]: list1.extend(list2)
In [41]: list1
Out[41]: [1, 2, 3, 4, 5, 6]
```

Мы можем удалить элемент из списка следующим образом:

```
In [43]: SBC.remove('Test Board')
```

Мы можем использовать два разных подхода для удаления элемента по указанному индексу. Оба подхода продемонстрированы ниже:

```
In [44]: SBC.pop(0)
Out[44]: 'RPi 4B 8GB'
In [46]: del SBC[0]
```

Если мы не укажем индекс, он вытолкнет (то есть удалит и вернет) последний элемент:

```
In [47]: SBC.pop()
Out[47]: 'Test Board 1'
```

Мы можем удалить все элементы из списка следующим образом:

```
In [48]: SBC.clear()
```

Мы также можем удалить весь список,

```
In [49]: del SBC
```

Если мы попытаемся получить доступ к списку сейчас, он выдаст следующую ошибку:

```
In [50]: SBC
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-50-69ed78d7b4fc> in <module>
----> 1 SBC

NameError: name 'SBC' is not defined
```

Теперь мы поймем, как использовать списки с циклами. Создайте список следующим образом:

```
In [51]: fruits = ['apple', 'banana', 'cherry', 'pineapple', 'watermelon',
'papaya']
```

Мы можем использовать конструкцию цикла for следующим образом:

```
In [52]: for member in fruits:
    ...:     print(member)
    ...:
apple
banana
cherry
pineapple
watermelon
papaya
```

Следующий код также дает тот же результат:

```
In [53]: for i in range(len(fruits)):
    ...:     print(fruits[i])
    ...:
apple
banana
cherry
pineapple
watermelon
papaya
```

Мы также можем использовать цикл **while** следующим образом:

```
In [54]: i = 0

In [55]: while i < len(fruits):
    ...:     print(fruits[i])
    ...:     i = i + 1
    ...:
apple
banana
cherry
pineapple
watermelon
papaya
```

Прежде чем продолжить, я хочу рассказать об одной важной особенности. Мы работали с примерами многих списков. Большинство списков, с которыми мы работали, представляют собой списки строк символов. Пара — это списки чисел. У нас также могут быть списки других типов данных. Ниже приведены примеры:

```
In [56]: l1 = [1.2, 2.3, 3.4]

In [57]: l2 = ['a', 'b', 'c']

In [58]: l3 = [True, False, False, True, True, False]
```

Здесь мы создали списки вещественных чисел, символов и логических значений соответственно. Мы также можем создать список смешанных типов следующим образом:

```
In [59]: l4 = [1, 'Test', 'a', 1.2, True, False]
```

Мы можем отсортировать список следующим образом:

```
In [60]: fruits.sort()
In [61]: fruits
Out[61]: ['apple', 'banana', 'cherry', 'papaya', 'pineapple', 'watermelon']
```

Мы также можем отсортировать список в обратном порядке:

```
In [62]: fruits.sort(reverse = True)
In [63]: fruits
Out[63]: ['watermelon', 'pineapple', 'papaya', 'cherry', 'banana', 'apple']
```

В качестве упражнения отсортируйте числовые и логические списки.

Мы можем скопировать один список в другой следующим образом:

```
In [64]: newlist = fruits.copy()
```

Ранее мы видели, что процедура **extend()** может объединять два списка. Мы можем использовать оператор сложения (+) для объединения двух списков следующим образом:

```
In [65]: l1 + l2
Out[65]: [1.2, 2.3, 3.4, 'a', 'b', 'c']
```

Мы можем использовать оператор умножения со списками следующим образом:

```
In [66]: l1 * 3
Out[66]: [1.2, 2.3, 3.4, 1.2, 2.3, 3.4, 1.2, 2.3, 3.4]
```

2.3 Кортежи

Кортежи похожи на списки. При их создании нам приходится использовать скобки. Их отличие от списков в том, что они неизменяемы, то есть после создания их нельзя изменить. Давайте посмотрим на простой пример:

```
In [1]: fruits = ('apple', 'grape', 'mango')
In [2]: fruits
Out[2]: ('apple', 'grape', 'mango')
In [3]: print(type(fruits))
<class 'tuple'>
```

Индексация, цикл и конкатенация (оператор +) для кортежей аналогичны спискам. Поскольку кортежи неизменяемы, мы не можем напрямую изменять какую-либо информацию, хранящуюся в кортежах. Однако мы можем преобразовать их в списки, а затем присвоить измененный список любому кортежу. См. следующий пример:

```
In [4]: temp_list = list(fruits)

In [5]: temp_list.append('papaya')

In [6]: fruits = tuple(temp_list)

In [7]: fruits
Out[7]: ('apple', 'grape', 'mango', 'papaya')
```

В приведенном выше примере мы продемонстрировали использование процедуры **tuple()**. Таким образом, мы можем разумно использовать все процедуры списков для работы с кортежами.
Давайте посмотрим демонстрацию метода **count()** для подсчета того, сколько раз определенный элемент-член встречается в кортеже:

```
In [8]: test_tuple = (2, 3, 1, 3, 1, 4, 5, 6, 3, 6)
In [9]: x = test_tuple.count(3)
In [10]: print(x)
3
```

2.4 Наборы

Списки и кортежи представляют собой упорядоченные структуры данных, и оба допускают дублирование значений. Наборы отличаются от обоих, поскольку они неупорядочены и, следовательно, не допускают дублирования значений. Наборы определяются с помощью фигурных скобок. Ниже приведен пример простых наборов:

```
In [12]: set1 = {'apple', 'banana', 'orange'}
In [13]: set1
Out[13]: {'apple', 'banana', 'orange'}
In [14]: set2 = set(('apple', 'banana', 'orange'))
In [15]: set2
Out[15]: {'apple', 'banana', 'orange'}
In [16]: print(type(set1))
<class 'set'>
```

Мы не можем использовать индексы для извлечения элементов любого множества, поскольку множества неупорядочены. Но мы можем использовать конструкции цикла for и while. Попробуйте это в качестве упражнения.

Мы можем добавлять новые элементы с помощью подпрограммы `add()`:

```
In [17]: set1
Out[17]: {'apple', 'banana', 'orange'}
In [18]: set1.add('pineapple')
In [19]: set1
Out[19]: {'apple', 'banana', 'orange', 'pineapple'}
```

Мы можем использовать процедуры `remove()` или `discard()` для удаления элемента из любого списка следующим образом:

```
In [20]: set1.remove('banana')
In [21]: set1.discard('apple')
```

Обе процедуры вызывают ошибки, если мы пытаемся удалить несуществующие элементы.
Давайте посмотрим несколько установленных методов. Сначала мы увидим, как вычислить объединение двух множеств и создадим для этого наборы:

```
In [22]: set1 = {1, 2, 3, 4, 5}
In [23]: set2 = {3, 4, 5, 6, 7}
In [24]: set3 = set1.union(set2)
In [25]: set3
Out[25]: {1, 2, 3, 4, 5, 6, 7}
```

Здесь мы сохраняем объединение в новом наборе. Небольшой альтернативный подход сохраняет объединение в первом наборе следующим образом:

```
In [29]: set1.update(set2)
In [30]: set1
Out[30]: {1, 2, 3, 4, 5, 6, 7}
```

Мы также можем удалить все элементы набора:

```
In [31]: set3.clear()
```

Подпрограмма copy() работает аналогично list. Давайте посчитаем разницу:

```
In [32]: set3 = set1.difference(set2)
In [33]: set3
Out[33]: {1, 2}
```

Этот пример возвращает новый выходной набор. Мы можем удалить соответствующие элементы из одного из наборов, используя это:

```
In [34]: set1.difference_update(set2)
In [35]: set1
Out[35]: {1, 2}
```

Мы можем вычислить пересечение следующим образом:

```
In [37]: set3 = set1.intersection(set2)
In [38]: set3
Out[38]: {3, 4, 5}
```

Мы можем проверить, является ли набор подмножеством другого набора следующим образом:

```
In [39]: set2 = {1, 2, 3, 4, 5, 6, 7, 8}
In [40]: set1.issubset(set2)
Out[40]: True
```

Аналогичным образом мы можем проверить, является ли набор надмножеством другого набора:

```
In [41]: set2.issuperset(set1)
Out[41]: True
```

Также можно проверить, не пересекаются ли два множества (не имеют ли они общих элементов):

```
In [42]: set1 = {1, 2, 3}
In [43]: set2 = {4, 5, 6}
In [44]: set1.isdisjoint(set2)
Out[44]: True
```

Симметричная разность между двумя наборами вычисляется следующим образом:

```
In [45]: set1 = {1, 2, 3}
In [46]: set2 = {2, 3, 4}
In [47]: set3 = set1.symmetric_difference(set2)
In [48]: set3
Out[48]: {1, 4}
```

Объединение и пересечение с помощью операторов **|** и **s** вычисляется следующим образом:

```
In [49]: set1 | set2
Out[49]: {1, 2, 3, 4}
In [50]: set1 & set2
Out[50]: {2, 3}
```

2.5 Словари

Словари упорядочены, изменяемы и не допускают дублирования. Словари в Python3.6 неупорядочены. Словари из Python 3.7 заказаны. Элементы хранятся в словаре inkey:valuepairs, и на них можно ссылаться по имени ключа. Давайте создадим простой словарь:

```
In [52]: test_dict = { "fruit": "mango", "colors": ["red", "green", "yellow"]}
```

Мы можем получить доступ к элементам словаря, используя ключи:

```
In [53]: test_dict["fruit"]
Out[53]: 'mango'
```

```
In [54]: test_dict["colors"]
Out[54]: ['red', 'green', 'yellow']
```

Мы можем получить ключи и значения следующим образом:

```
In [55]: test_dict.keys()
Out[55]: dict_keys(['fruit', 'colors'])

In [56]: test_dict.values()
Out[56]: dict_values(['mango', ['red', 'green', 'yellow']])
```

Мы можем обновить значение следующим образом:

```
In [60]: test_dict.update({"fruit": "grapes"})
In [61]: test_dict
Out[61]: {'fruit': 'grapes', 'colors': ['red', 'green', 'yellow']}
```

Мы можем добавить в словарь следующее:

```
In [62]: test_dict["taste"] = ["sweet", "sour"]
In [63]: test_dict
Out[63]:
{'fruit': 'grapes',
 'colors': ['red', 'green', 'yellow'],
 'taste': ['sweet', 'sour']}
```

Мы также можем выталкивать элементы:

```
In [64]: test_dict.pop("colors")
Out[64]: ['red', 'green', 'yellow']
In [65]: test_dict
Out[65]: {'fruit': 'grapes', 'taste': ['sweet', 'sour']}
```

Мы также можем вытащить последний вставленный элемент:

```
In [66]: test_dict.popitem()
Out[66]: ('taste', ['sweet', 'sour'])
```

Мы также можем удалить элемент,

```
In [67]: del test_dict["fruit"]
In [68]: test_dict
Out[68]: {}
```

Мы также можем удалить словарь следующим образом:

```
In [69]: del test_dict
In [70]: test_dict
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-70-6651ddf27d40> in <module>
----> 1 test_dict

NameError: name 'test_dict' is not defined
```

Мы можем перебирать словари, используя конструкции цикла for и while. Попробуйте сделать это в качестве упражнения.

Краткое содержание

В этой главе мы изучили основы множеств, кортежей, списков и словарей в Python. В совокупности они называются коллекциями в Python.

В следующей главе мы изучим основы строк, функций и рекурсии.

Глава 3. Строки, функции и рекурсия

В предыдущей главе мы рассмотрели коллекции Python, которые включают списки, кортежи, наборы и словари. Мы также начали писать небольшие фрагменты кода. Мы изучили множество встроенных функций для коллекций. Мы также изучили консоль IPython.

В этой главе мы углубимся в программирование на Python. Мы также начнем использовать IDLE. Ниже приводится список тем, которые мы рассмотрим в этой главе:

- Строки в Python
- Функции
- Рекурсия
- Прямая и косвенная рекурсия.

После прочтения этой главы вы должны освоиться с концепциями и программированием строк и функций на Python.

3.1 Строки в Python

В предыдущих главах, работая с функцией **print()** и коллекциями (списками, кортежами, наборами и словарями), мы много работали со строками. Я не упомянул об этом, потому что хотел осветить это в отдельной главе, поскольку важно понять это с самых основ. В этом разделе мы подробно рассмотрим строки в Python: изучим основы и связанные с ними процедуры.

Давайте откроем IPython в командной строке и начнем обучение. Мы можем создать строку, используя пару одинарных или двойных кавычек. Ниже приведены примеры:

```
In [1]: 'Python'
Out[1]: 'Python'
In [2]: "Python"
Out[2]: 'Python'
```

Мы можем использовать пару одинарных или двойных кавычек, но не их комбинацию. Ниже приводится демонстрация:

```
In [3]: 'Python"
  File "<ipython-input-3-580e07628eb0>", line 1
    'Python"
            ^
Синтаксическая ошибка: EOL при сканировании строкового литерала
```

Данные в строках хранятся одинаково, независимо от того, используем ли мы пару одинарных или двойных кавычек. В следующей демонстрации сравниваются строки, созданные с использованием пары одинарных и двойных кавычек:

```
In [4]: 'Python' == "Python"
Out[4]: True
```

Как мы видим, результатом является логическое значение **True**. Это означает, что обе строки равны. Данные в кавычках чувствительны к регистру. Следующая демонстрация хорошо это объясняет:

```
In [5]: 'Python' == "python"
Out[5]: False
```

Она вернула значение False, поскольку обе строки не равны. Обратите внимание на первые символы в обеих строках. Их случаи не совпадают. Вот почему сравнение возвращает логическое значение **False**.

Мы уже видели, как печатать строки. Давайте посмотрим на это еще раз. Кроме того, мы можем сохранить строку в переменную и распечатать ее. Ниже приводится демонстрация:

```
In [6]: print('Python')
Python
In [7]: var1 = 'Python'
In [8]: print(var1)
Python
```

У нас также могут быть многострочные строки. Ниже приведены примеры:

```
In [9]: var2 = '''test string,
   ...: test string'''
In [10]: var3 = """test string,
    ...: test string"""
```

Мы можем рассматривать строки как массивы. Ниже приведен пример:

```
In [13]: var1[0]
Out[13]: 'H'
In [14]: var1[1]
Out[14]: 'e'
In [15]: var1[2]
Out[15]: 'l'
In [16]: var1[3]
Out[16]: 'l'
In [17]: var1[4]
Out[17]: 'o'
```

Мы также можем вычислить длину строки:

```
In [20]: len(var1)
Out[20]: 5
```

We can use loops to access the strings:

```
In [19]: for x in var1:
    ...:     print(x)
    ...:
H
e
l
l
o
```

В C и C++ есть тип данных, известный как **scharacter** - символ. Scharacter содержит один символ ASCII. В Python нет символьного типа данных. Он рассматривает один символ как строку одного размера.

Мы также можем проверить, является ли строка подстрокой другой строки. Ниже приведен пример:

```
In [21]: 'test' in 'This is a test.'
Out[21]: True
In [22]: 'test' not in 'This is a test.'
Out[22]: False
```

Мы также можем разрезать строки так же, как списки в Python:

```
In [23]: var1 = 'This is a test.'
In [24]: var1[2:]
Out[24]: 'is is a test.'
In [25]: var1[:2]
Out[25]: 'Th'
In [26]: var1[2:7]
Out[26]: 'is is'
```

Мы также можем использовать отрицательные индексы:

```
In [27]: var1[-2:]
Out[27]: 't.'
In [28]: var1[:-2]
Out[28]: 'This is a tes'
In [29]: var1[-4:-2]
Out[29]: 'es'
```

Мы можем изменить регистр буквенных символов в строке:

```
In [30]: var1 = 'MiXed CaSe'
In [31]: var1.upper()
Out[31]: 'MIXED CASE'
```

```
In [32]: var1.lower()
Out[32]: 'mixed case'
```

Мы можем удалить ненужные пробелы с обоих концов строк:

```
In [33]: var1 = ' whitespace   '
In [34]: var1.strip()
Out[34]: 'whitespace'
```

Также можем заменить символы в строке:

```
In [35]: var1 = 'love'
In [36]: var1.replace("o", "0")
Out[36]: 'l0ve'
```

И разделить строку вокруг символа следующим образом:

```
In [37]: var1 = 'a,b,c,d,e,f'
In [38]: var1.split(',')
Out[38]: ['a', 'b', 'c', 'd', 'e', 'f']
```

Мы также можем использовать первый символ строки с заглавной буквы:

```
In [39]: var1 = 'hello!'
In [40]: var1.capitalize()
Out[40]: 'Hello!'
```

Можем объединить две строки:

```
In [41]: var1 = "Hello, "
In [42]: var2 = "World!"
In [44]: var1 + var2
Out[44]: 'Hello, World!'
```

Мы не можем объединять числа со строками. В ответ он вернет ошибку:

```
In [45]: age = 35
In [46]: var1 = 'I am '
In [47]: var1 + age
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-5677c330dd7a> in <module>
----> 1 var1 + age

TypeError: можно объединить только строку (не «int») с строкой.
```

Нам нужно преобразовать числа в строки, а затем объединить их со строками:

```
In [48]: var1 + str(age)
Out[48]: 'I am 35'
```

Мы также можем отформатировать строку:

```
In [49]: var1 = 'I am {}'.format(35)
In [50]: var1
Out[50]: 'I am 35'
```

Вот еще один пример:

```
In [51]: var1 = 'I am {} and my brother is {}'.format(35, 30)
In [52]: var1
Out[52]: 'I am 35 and my brother is 30'
```

Давайте посмотрим, как использовать escape-символы в строке. Рассмотрим следующий пример, где нам приходится использовать двойные кавычки внутри двойных кавычек:

```
In [53]: var1 = "He said, "I am fine"."
  File "<ipython-input-53-50120a6e6e28>", line 1
    var1 = "He said, "I am fine"."
                      ^
SyntaxError: invalid syntax - SyntaxError: неверный синтаксис
```

Она выдает ошибку, потому что синтаксически это неправильно. Мы можем использовать escape-символ \ для использования двойных кавычек следующим образом:

```
In [54]: var1 = "He said, \"I am fine\"."
In [55]: var1
Out[55]: 'He said, "I am fine".'
```

3.2 Функции

Почти все языки программирования высокого уровня предусматривают повторно используемые блоки кода. Они известны как подпрограммы или функции. Функция — это именованный блок кода, который можно вызвать из другого блока кода. В этом разделе мы подробно изучим функции. Кроме того, до сих пор мы использовали консоль IPython. Теперь приступим к сохранению программ в скриптах. Рассмотрим следующую простую программу, демонстрирующую простую функцию:

```
prog00.py
def func1():
    print('Test')
```

```
func1()
```

В приведенном выше примере мы создаем простую функцию (в первых двух строках), которая печатает строку при вызове. В последнем случае мы вызываем функцию. При вызове функция выполняет запланированное действие. Запустите код и посмотрите результат.

Мы также можем вызвать эту функцию несколько раз:

```
prog00.py
def func1():
    print('Test')

func1()
func1()
```

Здесь мы вызываем функцию дважды. Она дважды напечатает сообщение в консоли. Теперь давайте рассмотрим более питонический способ определения и вызова функций. Взгляните на следующий код:

```
prog01.py
def func1():
    print('Test')

def main():
    print("This is the main() function block...")
    func1()

if __name__ == "__main__":
    main()
```

Этот код написан на Python. Сначала мы определим две функции. Вторая функция с именем main() призвана служить аналогом основной функции, которую мы пишем в программах на C или C++. Мы можем присвоить этой функции любое допустимое имя по нашему выбору (в качестве упражнения вы можете изменить имя функции в объявлении и вызове функции). Затем строка **if __name__ == "__main__":** проверяет, выполняем ли мы модуль напрямую или вызываем его из другой программы Python. Если мы выполним его напрямую, он запустит код, указанный под ним. Мы узнаем больше об этой функции, когда будем изучать пакеты в следующей главе. Запустите код и посмотрите результат. Это профессиональный и питонический способ организации файлов кода Python. Мы будем часто использовать этот шаблон для наших программ на протяжении всей книги.

Давайте посмотрим, как мы можем передавать значения в функции. В определении функции мы должны объявить переменную-заполнитель для передаваемого значения. Он известен как **параметр**. Фактическое значение, которое должно быть передано, называется **аргументом**. Давайте посмотрим на это в действии.

prog02.py
```
def print_msg(msg):
    print(msg)

def main():
    print_msg("Test String...")


if __name__ == "__main__":
    main()
```

У нас также может быть несколько аргументов:

prog03.py
```
def print_msg(msg, count):
    for i in range(count):
        print(msg)

def main():
    print_msg("Test String...", 5)

if __name__ == "__main__":
    main()
```

В этом примере мы принимаем два аргумента. Первый — это сообщение, а второй — количество раз, которое сообщение должно быть напечатано. Мы также можем иметь значение аргумента по умолчанию, как показано ниже:

prog04.py
```
def print_msg(msg, count=2):
    for i in range(count):
        print(msg)

def main():
    print_msg("Test String...")


if __name__ == "__main__":
    main()
```

В приведенном выше примере кода мы устанавливаем значение по умолчанию для последнего аргумента, равное 2. Если мы не передаем аргумент при вызове функции, она принимает значение по умолчанию и продолжает действовать соответствующим образом.

У нас также может быть функция, возвращающая значение. Давайте определим функцию, которая принимает два значения и возвращает их сумму. Кодировать легко:

prog05.py
```
def add(a, b):
    return a + b

def main():
    print(add( 3, 5))


if __name__ == "__main__":
    main()
```

Мы также можем написать функцию, которая возвращает несколько значений:

prog06.py
```
def compute(a, b):
    return (a + b, a-b)

def main():
    (r1, r2) = compute( 3, 5)
    print(r1)
    print(r2)

if __name__ == "__main__":
    main()
```

Как мы видим в приведенном выше примере, можно упаковать возвращаемые значения в один кортеж и вернуть его.

3.3 Рекурсия

Мы можем вызвать экземпляр функции из той же функции. Это известно как рекурсия. Давайте перепишем предыдущий пример, чтобы учесть рекурсию:

prog07.py
```
def print_msg(msg, count):
    if count != 0:
        print(msg)
        print_msg(msg, count-1)

def main():
    print_msg("Test String...", 6)

if __name__ == "__main__":
    main()
```

В рекурсии есть две основные вещи. Первое – это условия прекращения действия. В отсутствие этого рекурсия будет работать вечно. И вторая часть — это рекурсивный вызов, при котором функция вызывает саму себя. В приведенной выше программе мы используем условие завершения **if**. В большинстве примеров рекурсии для завершения используется условие **if**. Оператор **if** будет содержать условие или критерии прекращения действия. Напишем программу для вычисления факториала натурального числа:

prog08.py
```
def factorial(n):
   if n == 1:
       return n
   else:
       return n*factorial(n-1)

def main():
    print(factorial(5))

if __name__ == "__main__":
    main()
```

Мы также можем написать программу для вычисления n-го числа ряда Фибоначчи:

prog09.py
```
def fibo(n):
   if n <= 1:
       return n
   else:
       return(fibo(n-1) + fibo(n-2))

def main():
    print(fibo(15))

if __name__ == "__main__":
    main()
```

Это несколько реальных примеров рекурсии.

3.3.1 Косвенная рекурсия

Мы видели демонстрацию рекурсии. Это были примеры прямой рекурсии, в которой мы вызываем функцию внутри себя. Существует еще один метод рекурсии, известный как косвенная рекурсия. В простейшей форме косвенной рекурсии подпрограмма A вызывает подпрограмму B, а подпрограмма B вызывает подпрограмму A. Давайте рассмотрим программу, включающую косвенную рекурсию:

prog10.py
```
def ping(i):
    if i>0:
        print("Ping " + str(i))
```

```
        return pong(i-1)
    return 0

def pong(i):
    if i>0:
        print("Pong " + str(i))
        return ping(i-1)
    return 1

ping(30)
```

Выше приведен интересный пример косвенной рекурсии. Мы моделируем ходы игры в пинг-понг с помощью рекурсии. Запустите эту и все другие программы из этой главы. Большинство из них имеют простые выходные данные, поэтому я не показал их. Я также советую вам попробовать написать свои функции, чтобы получить детальное представление об этой концепции.

Краткое содержание

В этой главе мы изучили строки в Python. Затем мы перешли к рекурсии функций и косвенной рекурсии. Кроме того, в этой главе мы начали использовать редактор кода IDLE и режим сценариев Python. В этой главе было больше концепций и программирования. Мы продолжим эту тенденцию, и каждая последующая глава будет знакомить с темами возрастающей сложности. Кроме того, все последующие главы будут более тяжелыми с точки зрения кодирования.

В следующей главе мы изучим основы объектно-ориентированного программирования на Python 3. Мы рассмотрим такие важные концепции, как классы, наследование и обработка исключений.

Глава 4. Объектно-ориентированное программирование (ООП).

В предыдущей главе мы рассмотрели некоторые важные особенности языка программирования Python. Мы подробно рассмотрели функции и рекурсию. Мы снова рассмотрим концепцию рекурсии в следующих главах, когда будем рассматривать графические библиотеки в Python. Сейчас мы будем изучать ООП.

ООП является обязательным для любого профессионального программиста. Это очень подробная глава, и у нас будет много практических демонстраций программирования. Ниже приводится список тем, которые мы рассмотрим в этой главе:

- Объекты и классы
- В Python все является объектом
- Начало занятий
- Модули и пакеты
- Пакеты
- Наследование
- Больше наследства
- Модификаторы доступа в Python
- Полиморфизм
- Синтаксические ошибки
- Исключения
- Обработка исключений
- Вызов исключения
- Исключения, определяемые пользователем

4.1 Объекты и классы

Объектно-ориентированное программирование (сокращенно ООП) — это парадигма программирования. Объект в реальном мире — это то, что мы можем потрогать, увидеть, почувствовать и с чем взаимодействовать. Например, яблоко — это объект. Велосипед – это предмет. Дом – это объект. Точно так же и в мире разработки программного обеспечения у нас тоже есть объекты. Объект — это набор данных и связанного с ним поведения. Объектно-ориентированное программирование относится к стилю программирования, в котором программа проектируется и разрабатывается с использованием объектов и связанных с ними концепций. Давайте возьмем пример ОС с графическим интерфейсом. Здесь окно программы является объектом. Панель задач — это объект. Экранные приложения — это объекты. Объекты в стиле ООП аналогичны своим реальным аналогам — объектам физического мира. Теперь предположим, что есть два объекта: велосипед и автомобиль. Оба являются объектами. Однако их типы различны. Автомобиль принадлежит классу автомобилей, а велосипед принадлежит классу велосипедов. В ООП класс описывает объект. Другими словами, класс — это тип данных. Объекты — это переменные с типом данных класса. Давайте посмотрим пример простой программы с классом и объектом на Python. Это наша первая объектно-ориентированная программа на Python 3 (мы сознательно пишем ее (хотя мы использовали несколько объектно-ориентированных функций Python).

```
prog00.py

class Point:
    pass

p1 = Point()
print(p1)
```

Результат будет следующим:

```
<__main__.Point object at 0x0000025E29D658E0>
```

В приведенной выше программе мы создаем простой класс **Point** и объект **p1**. Это простейший пример класса и объекта.

4.1.1 В Python все является объектом

Python — действительно язык ООП. В Python почти всё является объектом. Все может иметь соответствующую документацию. У всего есть методы и атрибуты. Это признак настоящего объектно-ориентированного языка программирования. Давайте посмотрим, как мы можем это продемонстрировать. Откройте интерпретатор Python и запустите следующий код:

```
>>> a = 5
>>> print (a)
5
>>> print(type(a))
<class 'int'>
```

Когда я говорю **Почти все в** `Python` **является объектом**, я имею в виду именно это. Например, `rou-tineprint()` — это встроенный метод. Мы можем вывести его тип, выполнив следующую инструкцию в интерактивной подсказке:

```
>>> print(type(print))
<class 'builtin_function_or_method'>
```

Мы также можем просмотреть документацию по процедуре print(), выполнив следующий оператор:

```
>>> print(print.__doc__)
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

По умолчанию печатает значения в поток или в sys.stdout.

Необязательные аргументы ключевого слова:

file: файлоподобный объект (поток); по умолчанию используется текущий sys.stdout.

sep: строка, вставленная между значениями, по умолчанию — пробел.

end: строка, добавляемая после последнего значения, по умолчанию — новая строка.

flush: следует ли принудительно сбрасывать поток.

4.2 Начало работы с классами

В последнем разделе мы изучили основы занятий. В этом разделе мы увидим, как создать встроенную документацию в файле кода Python с помощью строк документации. Мы также подробно изучим членов класса. Для демонстрации мы будем использовать тот же класс Point. Мы будем продолжать добавлять новые функции в один и тот же файл кода.

4.2.1 Docstrings - Строки документации

Мы можем предоставить встроенную документацию для кода Python с помощью **Docstrings**. **Docstrings** Python означают строки документации Python. Это наиболее удобный способ предоставления документации для разработчиков. Ниже приведен простой пример строки документации:

```
prog00.py
'This is the DocString example'
class Point:
    'This class represent the data-structure Point'
    pass

p1 = Point()
print(p1)
print(p1.__doc__)
print(help(p1))
```

Запустите этот код (**prog01.py**), как показано в выводе.

Как мы обсуждали ранее и в этом примере, мы можем использовать встроенную процедуру **__doc__** для печати строк документации, если она доступна. Мы также можем использовать встроенную функцию **help()** для отображения строк документации.

4.2.2 Добавление атрибутов в класс

Если вы знакомы с Java или C++, мы объявляем переменную класса в определении класса. В Python нет такого понятия, как объявление переменной. Мы можем добавить атрибуты к классу Point, добавив переменную для каждой координаты следующим образом:

```
prog01.py
'This is the DocString example'
class Point:
    'This class represent the data-structure Point'
    pass

p1 = Point()
p1.x = 1
p1.y = 2
p1.z = 2
```

```
print(p1.x, p1.y, p1.z)
```

Запустите программу и посмотрите результат.

4.2.3 Добавление метода в класс

Мы можем добавить поведение в класс, добавив в него методы. Ниже приведен пример класса Point с собственными методами **Assign()** и **printPoint()**:

```
prog01.py
'This is the DocString example'
class Point:
    'This class represent the data-structure Point'

    def assign(self, x, y, z):
        'This assigns the value to the co-ordinates'
        self.x = x
        self.y = y
        self.z = z

    def printPoint(self):
        'This prints the values 0f co-ordinates'
        print(self.x, self.y, self.z)

p1 = Point()
p1.assign(0, 0, 0)
p1.printPoint()
```

Запустите приведенный выше код и проверьте вывод. Как мы видим, объявление метода мало чем отличается от объявления функции. Основное отличие состоит в том, что обязательно наличие параметра ссылки на себя. В этом случае в обоих методах он называется **self**. Мы можем назвать это как угодно. Однако я еще не видел другого имени для этой переменной. Другое различие между функцией и методом заключается в том, что метод всегда связан с классом. В приведенном выше коде метод **Assign()** присваивает значения координатам, а метод **printPoint()** печатает значения координат точки.

4.2.4 Метод инициализатора

Существует специальный метод Python для инициализации объектов: **__init__**. Мы можем использовать это для присвоения значений атрибутам объекта во время самого создания объекта. Ниже приведен пример этого:

```
prog01.py
'This is the DocString example'
class Point:
    'This class represent the data-structure Point'

    def __init__(self, x, y, z):
```

```
        'The initializer'
        self.assign(x, y, z)

    def assign(self, x, y, z):
        'This assigns the value to the co-ordinates'
        self.x = x
        self.y = y
        self.z = z

    def printPoint(self):
        'This prints the values 0f co-ordinates'
        print(self.x, self.y, self.z)

p1 = Point(0, 1, 2)
p1.printPoint()
```

Запустите приведенную выше программу и посмотрите результат.

4.2.5 Многострочные Docstrings в Python

Давайте посмотрим пример Docstrings в Python. Многострочные **Docstrings** - строки документации используются для распределения строк документации по нескольким строкам. Ниже приведен пример **Docstrings** документа:

prog01.py

```
'This is the DocString example'
class Point:
    """This class represent the data-structure Point
    This is an example of the multiline docstring...
    """

    def __init__(self, x, y, z):
        '''The initializer ---
        This initializes the object with the passed arguments
        '''
        self.assign(x, y, z)

    def assign(self, x, y, z):
        'This assigns the value to the co-ordinates'
        self.x = x
        self.y = y
        self.z = z

    def printPoint(self):
        'This prints the values 0f co-ordinates'
        print(self.x, self.y, self.z)
```

```
p1 = Point(0, 1, 2)
p1.printPoint()
```

Запустите программу и посмотрите результат.

4.3 Модули и пакеты

В этом разделе мы начнем с концепции **модульности** в Python. Мы рассмотрим модули и пакеты. Мы также напишем программы для демонстрации этих концепций.

4.3.1 Модули

В предыдущем разделе мы продемонстрировали примеры концепции Class в Python. У нас был класс с именем Point в файле кода Python prog02.py. Мы также можем иметь несколько классов в одном файле кода Python. Рассмотрим следующую программу:

prog02.py

```
class Class01:

    def __init__(self):
        print("Just created an object for Class01...")


class Class02:

    def __init__(self):
        print("Just created an object for Class02...")


o1 = Class01()
o2 = Class02()
```

В приведенном выше примере мы определяем два класса Class01 и Class02 в одном файле кода Python. Файлы кода Python также известны как модули Python. Предположим, у вас есть каталог, в котором вы создали кучу файлов кода Python, мы можем просто ссылаться на каждый файл кода как на модуль. Запустите приведенный выше пример и посмотрите результат. Кроме того, доступ к классам и функциям в модуле Python можно получить из другого модуля. Это известно как модульность. Мы можем сгруппировать связанные классы в модуле и при необходимости импортировать их в другие модули. Чтобы увидеть это в действии, создайте еще один модуль Python prog03 в том же каталоге. Ниже приведен код для него:

prog03.py

```
import prog02

o1 = prog02.Class01()
o2 = prog02.Class02()
```

Когда мы запускаем модуль **prog03**, мы получаем следующий вывод:

Just created an object for Class01...
Just created an object for Class02...
Just created an object for Class01...
Just created an object for Class02...

Можете ли вы догадаться, почему это происходит?? Операторы печатаются дважды, поскольку мы импортируем весь модуль. Итак, мы импортируем часть из prog02, где мы создаем объекты. Следовательно, объекты создаются дважды. Чтобы смягчить это, Python предлагает эффективный подход. Внесите следующие изменения в prog02.py:

```
prog02.py
class Class01:

    def __init__(self):
        print("Just created an object for Class01...")


class Class02:

    def __init__(self):
        print("Just created an object for Class02...")

if __name__ == "__main__":
    o1 = Class01()
    o2 = Class02()
```

Написание нашего основного кода под блоком, начинающимся с **if __name__ == "__main__"**: гарантирует, что он будет вызываться только тогда, когда модуль выполняется напрямую. Когда мы импортируем весь модуль в другой модуль, код с основной логикой не импортируется в другой модуль. В другой модуль импортируются только функции и классы. Если вы запустите оба модуля один за другим, вы увидите, что объекты создаются только один раз во время каждого запуска. Но что, если мы хотим импортировать и запустить основной логический код модуля по требованию в том модуле, куда он импортирован? Чтобы сделать это возможным, существует более питонический способ организации нашего кода. Перепишите модуль **prog02** следующим образом:

```
prog02.py
class Class01:

    def __init__(self):
        print("Just created an object for Class01...")


class Class02:
```

```
    def __init__(self):
        print("Just created an object for Class02...")

def main():
    o1 = Class01()
    o2 = Class02()

if __name__ == "__main__":
    main()
```

Также внесите изменения в модуль prog02 следующим образом:

```
prog03.py
import prog02
prog01.main()
```

В модуле **prog03** мы напрямую импортируем функцию **main()** модуля **prog02**. Теперь запустите модуль **prog03**, чтобы увидеть результат. Будет то же самое. Однако модуль **prog02** теперь более организован, чем раньше. Давайте внесем те же изменения в код **prog03** следующим образом:

```
prog03.py
import prog02


def main():
    prog02.main()


if __name__ == "__main__":
    main()
```

Запустите код еще раз, чтобы увидеть результат. Наконец, чтобы добавить большей ясности, мы изменим **prog02** следующим образом:

```
prog02.py
class Class01:

    def __init__(self):
        print("Just created an object for Class01...")


class Class02:

    def __init__(self):
        print("Just created an object for Class02...")
```

```
def main():
    o1 = Class01()
    o2 = Class02()

if __name__ == "__main__":
    print("Module prog01 is being run directly...")
    main()
else:
    print("Module prog02 has been imported in the current module...")
```

Запустите оба модуля и посмотрите результат. Что здесь случилось?? Мы заметили, что код перед оператором **elsest** выполняется, если мы запускаем модуль напрямую, а код после оператора **elsest** выполняется, когда мы импортируем весь модуль.

Давайте перейдем к пониманию другого способа импорта членов модуля. В самой ранней версии модуля prog03 у нас был следующий код:

```
prog03.py
import prog02
o1 = prog01.Class01()
o2 = prog01.Class02()
```

Здесь мы обратились к членам модуля **prog02** с помощью обозначения **prog2.member**. Это связано с тем, что оператор **import prog02** импортирует все элементы в модуль **prog03**. Существует еще один способ импорта членов модулей. Ниже приведен пример этого:

```
prog04.py
from prog02 import Class01, Class02

def main():
    o1 = Class01()
    o2 = Class02()

if __name__ == "__main__":
    main()
```

Благодаря синтаксису `<module> import <member>` нам не нужно использовать соглашение `<mod-ule>.<member>` в вызывающем модуле. Мы можем напрямую обращаться к членам импортированного модуля, как показано в приведенном выше примере. Запустите программу и посмотрите результат. В приведенной выше программе мы импортируем оба члена `prog02`. Измените приведенную выше программу, чтобы импортировать один член модуля `prog02`. Кроме того, добавьте часть `else` после вызова функции `main()`.

4.3.2 Пакеты

До сих пор мы видели, как организовать функции и классы Python в отдельных модулях. Мы можем организовать модули в пакеты. Давайте создадим пакет уже написанного кода. Создайте подкаталог в каталоге, в котором вы сохраняете все модули Python. Назовите подкаталог **mypackage** и переместите туда модуль **prog02**. Создайте пустой файл **__init__.py** и сохраните его в каталоге **mypackage**. Мы только что создали наш собственный пакет Python! Правильно. Создать пакет легко. Нам просто нужно поместить модули в каталог и создать в том же каталоге пустой файл **__init__.py**. Имя каталога, содержащего модули, является именем пакета. Теперь в исходном родительском каталоге создайте модуль Python **prog05** следующим образом:

```
prog05.py
import mypackage.prog02
def main():
    mypackage.prog02.main()

if __name__ == "__main__":
    main()
```

Теперь запустите модуль `prog05` и посмотрите результат. Обратите внимание, что мы используем нотацию `<package>.<module>.<member>` для доступа к члену пакета. Нам не нужно этого делать, если мы используем для импорта синтаксис `from <module> import <class>`.

В качестве упражнения напишите программу, импортировав модули **prog02** с синтаксисом **prog02**, **import Class01**, **Class02** и запустите код.
У нас даже может быть пакет внутри пакета. Для этого нам просто нужно создать еще один каталог в каталоге **mypackage** и добавить в него нужный файл модуля и пустой файл **__init__.py**.

В качестве упражнения создайте **подпакет** в **mypackage -** моем пакете. Импортируйте и используйте его в модуле в родительском каталоге.

4.4 Наследование

Современные языки программирования имеют разные механизмы повторного использования кода. Процедуры предоставляют отличный способ повторного использования кода. Функции и методы классов в Python — отличные примеры того, как можно добиться повторного использования кода с помощью процедур. Объектно-ориентированные языки программирования, такие как Python, также имеют другой подход к повторному использованию кода. Это известно как **наследование**. В Python есть механизм наследования на основе классов. Класс в Python может наследовать **атрибуты** и поведение другого класса. Базовый класс, от которого мы получаем другие классы, также известен как родительский класс или суперкласс. Класс, который наследует (или является производным или расширяется, имейте в виду, что эти термины используются как взаимозаменяемые в мире объектно-ориентированного программирования) атрибуты и поведение, известен как **дочерний класс** или **подкласс**.

4.4.1 Базовое наследование в Python

Технически все классы Python являются подклассами определенного встроенного класса, известного как **object**. Этот механизм позволяет Python одинаково обращаться с объектами. Итак, если вы создаете какой-либо класс в Python, это означает, что вы неявно используете наследование. Это неявно, потому что вам не нужно делать для этого каких-либо специальных положений в коде. Вы также можете явно получить свой собственный класс из встроенного класса **object** следующим образом:

```
prog01.py
class MyClass(object):
    pass
```

Код выше имеет тот же смысл и функциональность:

```
prog01.py
class MyClass:
    pass
```

Давайте определим собственный класс и создадим от него другой класс:

```
prog02.py
class Person:
    pass
class Employee:
    pass
def main():
    print(issubclass(Employee, Person))
if __name__ == "__main__":
    main()
```

Запустите приведенный выше код. Подпрограмма **issubclass()** возвращает **true**, если класс, упомянутый в первом аргументе, является подклассом класса, упомянутого во втором аргументе. Приведенная выше программа печатает **false**. Упс! Мы забыли вывести класс **Person** из класса **Employee** (сотрудник). Для этого измените код следующим образом:

```
prog02.py
class Person:
    pass
class Employee(Person):
    pass
def main():
    print(issubclass(Employee, Person))
    print(issubclass(Person, object))
    print(issubclass(Employee, object))
if __name__ == "__main__":
    main()
```

Вы, должно быть, заметили, что мы добавили еще два оператора в раздел **main()**. Запустите код. Он должен вывести **True** три раза.

4.4.2 Переопределение метода

Давайте добавим больше функциональности классу Person. Измените модуль prog01.py:

prog01.py

```
class Person:
    def __init__(self, first, last, age):
        self.firstname = first
        self.lastname = last
        self.age = age
    def __str__(self):
        return self.firstname + " " + self.lastname + ", " + str(self.age)
class Employee(Person):
    pass
def main():
    x = Person("Ashwin", "Pajankar", 31)
    print(x)
if __name__ == "__main__":
    main()
```

Мы должны определить поведение класса **Employee**. Поскольку мы унаследовали его от **Person**, мы можем повторно использовать методы класса **Person** для определения поведения класса **Employee**:

prog02.py

```
class Person:
    def __init__(self, first, last, age):
        self.firstname = first
        self.lastname = last
        self.age = age
    def __str__(self):
        return self.firstname + " " + self.lastname + ", " + str(self.age)
class Employee(Person):
    pass
def main():
    x = Person("Ashwin", "Pajankar", 31)
    print(x)
    y = Employee("James", "Bond", 32)
    print(y)
if __name__ == "__main__":
    main()
```

Как видите, класс **Employee** - Сотрудник наследует инициализатор и метод **__str__()** от класса **Person**. Однако мы хотим добавить к классу **Employee** еще один дополнительный атрибут, называемый **empno**. Нам также необходимо соответствующим образом изменить поведение метода **__str__()**. Следующий код делает это:

prog02.py

```
class Person:
    def __init__(self, first, last, age):
        self.firstname = first
        self.lastname = last
        self.age = age
    def __str__(self):
        return (self.firstname + " " +
                self.lastname + ", " +
                str(self.age))
class Employee(Person):
    def __init__(self, first, last, age, empno):
        self.firstname = first
        self.lastname = last
        self.age = age
        self.empno = empno
    def __str__(self):
        return (self.firstname + " "
                + self.lastname + ", "
                + str(self.age) + ", "
                + str(self.empno))
def main():
    x = Person("Ashwin", "Pajankar", 31)
    print(x)
    y = Employee("James", "Bond", 32, 0x007)
    print(y)
if __name__ == "__main__":
    main()
```

В приведенном выше коде мы переопределяем определения методов **__init__() и __str__()** из класса **Person** в **Employee** в соответствии с нашими потребностями. Запустите код и посмотрите результат. Таким образом, мы можем переопределить любой метод базового класса в подклассе.

4.4.3 super()

В последнем примере мы увидели, как переопределить методы базового класса в подклассе. Есть еще один способ сделать это. См. приведенный ниже пример кода:

```
class Person:
    def __init__(self, first, last, age):
        self.firstname = first
        self.lastname = last
```

```
        self.age = age

    def __str__(self):
        return (self.firstname + " " +
                self.lastname + ", " +
                str(self.age))


class Employee(Person):
    def __init__(self, first, last, age, empno):
        super().__init__(first, last, age)
        self.empno = empno

    def __str__(self):
        return (super().__str__() + ", "
                + str(self.empno))

def main():
    x = Person("Ashwin", "Pajankar", 31)
    print(x)
    y = Employee("James", "Bond", 32, 0x007)
    print(y)

if __name__ == "__main__":
    main()
```

Как видите, мы использовали метод **super()**. Это специальный метод, который возвращает объект как экземпляр базового класса. Мы можем использовать **super()** в любом методе. Поскольку super возвращает объект как экземпляр базового класса, код **super().<method()>** вызывает соответствующий метод базового класса. Метод **super()** можно использовать внутри любого метода подкласса для вызова экземпляра родительского класса в качестве объекта. Его не обязательно вызывать в первой строке определения метода подкласса. Его можно вызвать в любом месте тела. Таким образом, мы также можем переписать метод **__init__()** Employee следующим образом:

```
    def __init__(self, first, last, age, empno):
        self.empno = empno
        super().__init__(first, last, age)
```

4.5 Больше наследования

В последнем разделе мы изучили базовое наследование, переопределение и метод **super()**. В этом разделе мы изучим множественное наследование, проблему ромба, модификаторы доступа, а также абстрактные классы и методы.

4.5.1 Множественное наследование

Когда класс является производным от более чем одного класса, этот механизм известен как множественное наследование. На следующей диаграмме это показано:

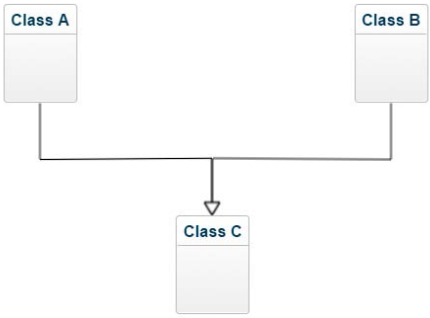


Рис.*4-1:* Множественное наследование

Давайте попробуем написать для этого простой код:

```
prog01.py
class A:
    pass


class B:
    pass


class C(A, B):
    pass


def main():
    pass


if __name__ == "__main__":
    main()
```

Множественное наследование используется в тех случаях, когда нам нужно, чтобы атрибуты и поведение более чем одного класса были производными в одном классе.

4.5.2 Method Resolution order

Рассмотрим следующий пример:

prog02.py

```
class A:
    def m(self):
        print('This is m() from Class A.')
class B:
    def m(self):
        print('This is m() from Class B.')
class C(A, B):
    pass
def main():
    x = C()
    x.m()
if __name__ == "__main__":
    main()
```

Когда мы запускаем код, результат выглядит следующим образом:

```
This is m() from Class A.
```

Это связано с порядком, в котором класс C является производным от родительских классов. Если мы изменим код класса C следующим образом:

```
class C(B, A):
    pass
```

Результат

```
This is m() from Class B.
```

Механизм разрешения производных методов подкласса известен как порядок разрешения методов.

4.6 Абстрактный класс и метод

Класс с объектом, который не создан, называется абстрактным классом. Кроме того, метод, объявленный без реализации, называется абстрактным методом. Абстрактный класс может иметь или не иметь абстрактный метод. В Python мы явно объявляем класс и метод как абстрактные. Явно объявленный абстрактный класс может быть только подклассом. Во многих ситуациях нам нужно иметь класс и методы только для того, чтобы они были производными и переопределенными соответственно. Ниже приведен идеальный пример из реальной жизни:

prog03.py

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def move(self):
        pass
```

```
class Human(Animal):
    def move(self):
        print("I Walk.")
class Snake(Animal):
    def move(self):
        print("I Crawl.")
def main():
    a = Human()
    b = Snake()
    a.move()
    b.move()
if __name__ == "__main__":
    main()
```

Чтобы сделать абстрактный класс явным, нам нужно получить его от встроенного класса ABC. Если мы хотим явно абстрагировать методы класса, нам нужно использовать метод **Decorator@abstract** с методом, который нужно сделать абстрактным. Запустите приведенный выше код и посмотрите результат.

4.7 Модификаторы доступа в Python

Во многих языках программирования, таких как C++ и Java, существует концепция **Access Control** для членов классов. В Python не существует строгих мер по контролю доступа члена извне класса.

Если вы не хотите, чтобы к методу или переменной класса был доступен извне, вы можете упомянуть об этом в **docstrings** - строке документации класса. Другой способ сообщить другим, что им не следует обращаться к переменной или методу, предназначенному для внутреннего использования, — поставить перед переменной знак подчеркивания. Другой человек, который изменяет код или использует его путем импорта, поймет, что переменная или метод предназначены только для внутреннего использования. Тем не менее, он по-прежнему может получить к ней доступ извне. Другой способ настоятельно рекомендовать другим не обращаться к переменной или методу извне — использовать механизм искажения имен. Для этого нам нужно поставить перед методом или переменной двойное подчеркивание. Тогда доступ к нему можно будет получить только с помощью специального синтаксиса, который продемонстрирован в программе ниже:

prog04.py

```
class A:
    def __init__(self):
        self.a = "Public"
        self._b = "Internal use"
        self.__c = "Name Mangling in action"
def main():
    x = A()
    print(x.a)
    print(x._b)
    print(x.__c)
if __name__ == "__main__":
    main()
```

В выходных данных отображаются значения первых двух атрибутов. Для третьего атрибута отображается ошибка, содержащая сообщение:

AttributeError: 'A' object has no attribute '__c'

Чтобы увидеть значение атрибута **__c**, внесите следующие изменения в последнюю функцию **print()**:

```
print(x._A__c)
```

Вывод следующий:

Public
Internal use
Name Mangling in action

4.8 Полиморфизм

В последнем разделе мы изучили расширенное наследование и модификаторы доступа в Python. В этом разделе мы будем изучать полиморфизм. Полиморфизм означает способность принимать различные формы. С точки зрения языков программирования это означает предоставление единого интерфейса для объектов разных типов. Большинство объектно-ориентированных языков программирования допускают различные степени полиморфизма. Мы изучали переопределение в предыдущем разделе. Это форма полиморфизма. Итак, мы уже изучили тип полиморфизма в Python 3. В этом разделе мы сначала изучим перегрузку методов, а затем перегрузку операторов, которая подпадает под концепцию полиморфизма.

4.8.1 Перегрузка метода

Когда метод может быть вызван с разным количеством аргументов, это называется перегрузкой метода. В языках программирования, таких как C++, мы можем иметь несколько определений функций-членов класса. Однако Python не позволяет этого, поскольку мы знаем, что все в Python является объектом. Чтобы сделать это возможным, мы используем методы с аргументами по умолчанию. Пример следующий:

prog01.py

```
class A:

    def method01(self, i=None):
        if i is None:
            print("Sequence 01")
        else:
            print("Sequence 02")

def main():
    obj1 = A()
    obj1.method01()
    obj1.method01(5)
```

```
if __name__ == "__main__":
    main()
```

Запустите приведенный выше код и посмотрите результат.

4.8.2 Перегрузка оператора

Операторы оперируют операндами и выполняют различные операции. Поскольку мы знаем, что в Python все является объектом, все операнды, с которыми работают операторы в Python, являются объектами. Операции и результаты операций операторов над встроенными объектами в Python уже четко определены в Python. Мы можем возложить на операторов дополнительную ответственность за объекты пользовательских классов. Эта концепция известна как перегрузка операторов. Ниже приведен простой пример оператора сложения:

prog02.py

```
class Point:

    def __init__(self, x, y, z):
        self.assign(x, y, z)

    def assign(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def printPoint(self):
        print(self.x, self.y, self.z)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        z = self.z + other.z
        return Point(x, y, z)

    def __str__(self):
        return("({0},{1},{2})".format(self.x, self.y, self.z))


def main():
    p1 = Point(1, 2, 3)
    p2 = Point(4, 5, 6)
    print(p1 + p2)


if __name__ == "__main__":
    main()
```

Запустите код и проверьте вывод. Когда мы выполняем в коде операцию **p1 + p2**, Python вызывает **p1.__add__(p2)**, который, в свою очередь, вызывает **Point.__add__(p1,p2)**. Точно так же мы можем перегрузить и другие операторы. Специальные функции, необходимые для реализации двоичных операций, приведены в таблице ниже:

Operator	Special Function
+	object.add(self, other)
-	object.sub(self, other)
*	object.mul(self, other)
//	object.floordiv(self, other)
/	object.truediv(self, other)
%	object.mod(self, other)
**	object.pow(self, other[, modulo])
<<	object.lshift(self, other)
>>	object.rshift(self, other)
&	object.and(self, other)
^	object.xor(self, other)
|	object.or(self, other)

Ниже представлена таблица расширенных операций:

Operator	Special Function
+=	object.add(self, other)
-=	object.sub(self, other)
*=	object.mul(self, other)
//=	object.floordiv(self, other)
/=	object.truediv(self, other)
%=	object.mod(self, other)
**=	object.pow(self, other[, modulo])
<<=	object.lshift(self, other)
>>=	object.rshift(self, other)
&=	object.and(self, other)
^=	object.xor(self, other)
|=	object.or(self, other)

This table is for the unary operators:

Operator	Special Function
+	object.pos(self)
-	object.neg(self)
abs()	object.abs(self)
~	object.invert(self)
complex()	object.complex(self)
int()	object.int(self)
long()	object.long(self)
float()	object.float(self)

oct()	object.oct(self)
hex()	object.hex(self)

Эта таблица предназначена для операторов сравнения:

Operator	**Special Function**
<	object.lt(self, other)
<=	object.le(self, other)
==	object.eq(self, other)
!=	object.ne(self, other)
>=	object.ge(self, other)
>	object.gt(self, other)

4.9 Синтаксические ошибки

Когда мы пишем программы на Python (или на любом другом языке программирования, если уж на то пошло), мы обычно не делаем их правильными с первого раза. Здесь вступают в обсуждение термины **Errors** и **Exceptions** - Исключения. В этой главе мы начнем с ошибок и исключений в Python.

```
>>> print("Hello)

SyntaxError: EOL while scanning string literal
```

В приведенном выше операторе **print()** мы забыли добавить **"** после строки Hello. Это синтаксическая ошибка. Поэтому интерпретатор Python выделил ее, выдав **SyntaxError**.

4.10 Исключения

Мы говорим, что ошибки синтаксиса/синтаксического анализа обрабатываются с помощью **SyntaxError.** Несмотря на то, что этот оператор верен с точки зрения синтаксиса, во время его выполнения может возникнуть ошибка (не связанная с синтаксисом). Ошибки, обнаруженные во время выполнения, называются исключениями. Рассмотрим следующие операторы и их выполнение в интерпретаторе:

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
>>> '1' + 1
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    '1' + 1
TypeError: can only concatenate str (not "int") to str
>>> a = 8 + b
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    a = 8 + b
```

```
NameError: name 'b' is not defined
```

Последняя строка результатов выполнения каждого оператора показывает, что с ним не так. Это демонстрирует, что исключение неявно вызывается всякий раз, когда во время выполнения возникает ошибка. Исключения, определенные в библиотеке Python, известны как встроенные исключения.

https://docs.python.org/3/library/exceptions.html#bltin-exceptions содержит все перечисленные встроенные исключения.

4.10.1 Обработка исключений

Теперь мы знаем, что интерпретатор автоматически генерирует исключение, если во время выполнения обнаруживается ошибка. Рассмотрим следующий фрагмент кода:

prog01.py

```
def main():
    a = 1/0
    print("DEBUG:  We are here...")
if __name__ == "__main__":
    main()
```

Когда мы выполним это, обратите внимание, что интерпретатор встречает следующее исключение в строке = 1/0.

ZeroDivisionError: division by zero

При обнаружении этого исключения операторы, следующие за оператором, в котором обнаружено исключение, не выполняются. Вот как исключения обрабатываются по умолчанию в Python. Однако Python имеет лучшие возможности для обработки исключений. Мы можем поместить код в блок try, где мы, скорее всего, встретим исключение, и логику его обработки в блоке исключений следующим образом:

prog01.py

```
def main():
    try:
        a = 1/0
        print("DEBUG:  We are here...")
    except Exception:
        print("Exception Occured")

if __name__ == "__main__":
    main()
```

Запустите код. Он будет вызывать блок исключений при возникновении исключения, а не резко завершать работу. Обратите внимание, что код после оператора, в котором произошло исключение, не выполняется.

В блоке исключений **Exception** является базовым классом для всех встроенных исключений Python. В последующих частях этой главы мы также изучим пользовательские исключения, которые будут производными от класса **Exception**.

Давайте изменим код и добавим еще немного кода в функцию **main()**, которая не является частью блоков **try** или кроме.

prog01.py
```
def main():
    try:
        a = 1/0
        print("DEBUG:  We are here...")
    except Exception:
        print("Exception Occured")
    print("This line will be executed...")

if __name__ == "__main__":
    main()
```

При выполнении вы заметите, что строка за пределами блока **try** и **except** выполняется, несмотря на возникновение исключения.

4.10.2 Обработка исключений по типам

Когда мы запускаем программу, мы можем столкнуться с исключениями нескольких типов. Мы можем обеспечить обработку различных типов исключений. Простой пример выглядит следующим образом:

prog02.py
```
def main():
    try:
        a = 1/1
    except ZeroDivisionError as err:
        print("Error: {0}".format(err))
    except TypeError as err:
        print("Error: {0}".format(err))
    except Exception as err:
        print("Error: {0}".format(err))
if __name__ == "__main__":
    main()
```

В приведенном выше коде есть блоки исключений для обработки **ZeroDivisionError** и **TypeError**. Если встречается какое-либо другое неожиданное исключение, оно обрабатывается последним блоком исключений, который является универсальным блоком **except**. Обратите внимание, что общий блок всегда должен быть последним блоком кроме (как показано в приведенном выше коде). Если это самый первый блок исключений, то при возникновении какого-либо исключения всегда будет выполняться общий блок обработки исключений. Это связано с тем, что класс Exception является базовым классом для всех исключений и имеет приоритет.

В качестве упражнения перепишите приведенную выше программу, включив в качестве первого блока обработки исключений исключение.

Как мы знаем, в Python все является объектом. SyntaxError также является типом исключения. Это особый тип исключения, который не может быть обработан в блоке исключений.

4.10.3 Блок else

Мы можем добавить блок *else* в код после блока исключений. Если в блоке try не обнаружено ошибок, выполняется блок else. Следующая программа демонстрирует это:

```
prog02.py
def main():
    try:
        a = 1/1
    except ZeroDivisionError as err:
        print("Error: {0}".format(err))
    except TypeError as err:
        print("Error: {0}".format(err))
    except Exception as err:
        print("Error: {0}".format(err))
    else:
        print("This line will be executed...")

if __name__ == "__main__":
    main()
```

4.10.4 Вызов исключения

Мы знаем, что исключение автоматически возникает при возникновении ошибки во время выполнения. Мы можем явно и намеренно вызвать исключение, используя оператор повышения. Следующий код демонстрирует это:

```
prog03.py
def main():
    try:
        raise Exception("Exception has been raised!")
    except Exception as err:
        print("Error: {0}".format(err))
    else:
        print("This line will be executed...")


if __name__ == "__main__":
    main()
```

Результат следующий:

Error: Exception has been raised! - Ошибка: возникло исключение!

4.10.5 finally clause

finally — это предложение для оператора try, который всегда выполняется на выходе. Это означает, что по сути это Clean Up Clause - пункт об очистке. Он всегда выполняется в конце предложения try, независимо от того, возникло ли исключение в операторе try. Если какое-либо исключение не обрабатывается в блоке-исключении, оно в конечном итоге вызывается повторно. Ниже приведен пример того же:

```
prog04.py
def divide(x, y):
     try:
         result = x / y
     except ZeroDivisionError:
         print("division by zero!")
     else:
         print("result is", result)
     finally:
         print("executing finally clause")

def main():
    divide(2, 1)
    divide("2", "1")


if __name__ == "__main__":
    main()
```

Ниже приводится вывод:

```
result is 2.0
executing finally clause
executing finally clause
Traceback (most recent call last):
   File "C:\Users\Ashwin\Google Drive\Elektor\Python Book Project\
Code\Chapter04\Exceptions\prog04.py", line 17, in <module>
   main()
   File "C:\Users\Ashwin\Google Drive\Elektor\Python Book Project\
Code\Chapter04\Exceptions\prog04.py", line 13, in main
   divide("2", "1")
   File "C:\Users\Ashwin\Google Drive\Elektor\Python Book Project\
Code\Chapter04\Exceptions\prog04.py", line 3, in divide
   result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Как мы видим, в приведенном выше коде не предусмотрена обработка исключений TypeError.
Итак, предложение **finally** поднимает этот вопрос еще раз.

4.10.6 Пользовательские исключения

Мы можем определить собственные исключения, производные от класса Exception. Обычно в модуле мы определяем один базовый класс, производный от Exception, и получаем от него все остальные классы исключений. Пример этого показан ниже:

prog05.py

```
class Error(Exception):
   pass


class ValueTooSmallError(Error):
   pass


class ValueTooLargeError(Error):
   pass


def main():
    number = 10
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        else:
            print("Perfect!")
    except ValueTooSmallError:
        print("This value is too small!")
    except ValueTooLargeError:
        print("This value is too large!")


if __name__ == "__main__":
    main()
```

Пожалуйста, запустите приведенную выше программу и посмотрите результат.
В приведенной выше программе класс Error унаследован от встроенного класса Exception. Мы наследуем больше подклассов от класса Error.

Краткое содержание

Мы подробно изучили парадигму объектно-ориентированного программирования. Теперь нам очень удобно использовать ООП при использовании Python. В следующих главах мы будем широко использовать ООП, поэтому я описал его в этой специальной главе как можно раньше.

Следующая глава посвящена структурам данных с использованием Python. Это также будет длинная и подробная глава, в которой будут использованы все концепции, изученные в предыдущих главах, для реализации и использования структур данных.

Глава 5 • Структуры данных

В предыдущей главе мы подробно изучили объектно-ориентированное программирование. Теперь нам должен быть удобен этот стиль программирования.

В этой главе мы используем знания объектно-ориентированного программирования для создания структур данных и работы с ними. Мы рассмотрим самые популярные и наиболее часто используемые линейные структуры данных с помощью языка программирования Python. Ниже приводится список полезных тем о структурах данных, которые мы рассмотрим в этой главе:

- Введение в структуры данных
- Блокнот Jupyter
- Связанные списки
- Двусвязный список
- Стек
- Очередь
- Двусторонние очереди
- Круговая очередь

После этой главы мы сможем писать программы для структур данных и их приложений.

5.1 Введение в структуры данных

Структура данных — это специализированный способ организации, хранения, обработки и извлечения данных. Существует множество структур данных. Концепция структур данных возникла еще до Python. Многие структуры данных, которые мы рассмотрим в этой главе, были разработаны с учетом ограничений таких языков программирования, как C, C++ и Java. Большинство языков программирования оснащены базовыми структурами данных, такими как массивы. Python имеет встроенную поддержку улучшенных версий и более универсальных массивов, известных как коллекции. Python также поддерживает строки. Мы уже изучили все встроенные структуры данных.

В этой главе мы сосредоточимся на самых популярных и наиболее используемых структурах данных. Многие сторонние библиотеки Python уже реализуют это. Однако мы хотим научиться реализовывать их все самостоятельно с нуля. Поэтому для всех этих структур данных мы напишем свой код.

5.1.1 Блокнот Jupyter

В этой главе мы собираемся использовать веб-среду для демонстрации программ. Эта среда известна как блокнот Jupyter, и прежде чем изучать реальные темы, мы изучим ее вкратце.

Мы использовали интерактивный режим интерпретатора Python, а также IDLE. Интерактивный режим обеспечивает быструю обратную связь, но не сохраняет программы. IDLE предлагает функции сохранения, но не обеспечивает быструю обратную связь по коду. Прежде чем работать с блокнотом Jupyter, мне хотелось иметь такую функциональность в специальном редакторе. Jupyter Notebook выполняет эту задачу. Он предлагает веб-среду для Python и многих других языков программирования, таких как R, Julia и GNU Octave.

Он обеспечивает быструю обратную связь по выполнению кода, и мы можем писать наш код небольшими фрагментами в ячейках веб-редактора. Давайте установим его на вашу платформу, используя следующую команду в командной строке вашей ОС:

```
pip3 install jupyter
pip3 install notebook
```

В командной строке перейдите в каталог, в котором вы хотите сохранить свои записные книжки, и выполните следующую команду:

```
jupyter notebook
```

Он запустит сервер ноутбука в командной строке и отобразит журнал. Вы можете найти URL в журнале. Если вы скопируете и вставите его в выбранный вами браузер и посетите веб-страницу, вы увидите домашнюю страницу блокнота Jupyter. Кроме того, когда вы запускаете сервер с помощью приведенной выше команды, он автоматически запускает домашнюю страницу блокнота Jupyter. Если вы случайно закроете вкладку браузера или окно, показывающее домашнюю страницу Jupyter, вы всегда можете получить URL-адрес из командной строки входа в систему.

Домашняя страница покажет вам все файлы и каталоги в текущем каталоге. Когда сервер Jupyter запускается, он создает сеанс, и до конца этого времени каталог, из которого мы запустили сервер, становится корневой отправной точкой всего сеанса, пока он жив. Мы можем просматривать файлы и каталоги в этом каталоге из веб-интерфейса, но не можем посетить родительский каталог этого каталога или любой другой каталог, который не является подкаталогом текущего каталога, из которого мы запустили серверный процесс. В этом интерфейсе есть кнопка с надписью «New». При нажатии отображаются параметры создания нового подсеанса для Python. Он также имеет параметры для создания новой папки, сеанса терминала и текстового файла. Вы также можете изучить эти варианты, если хотите. Они полезны при работе над проектами. Кроме того, на той же домашней странице есть вкладка с надписью «Running». Здесь показаны все запущенные ноутбуки и сеансы терминалов. А пока нажмите кнопку «New» и создайте блокнот Python. Откроется веб-интерфейс в новой вкладке. Эта вкладка обладает всеми функциями настольной IDE, такой как IDLE. Также будет пустая ячейка, в которую вы сможете написать одну строку или блок кода и запустить его. Вывод отображается на той же странице, когда мы выполняем код. Существует раскрывающийся список, в котором можно установить значение текущей ячейки. Мы можем установить ячейку как код или уценку. Ячейки кода используются для запуска кода. Ячейки Markdown используются для сохранения элементов форматированного текста. Изучите все опции в меню интерфейса. Это довольно интуитивно понятно для нового пользователя с небольшим опытом программирования. В начале главы или раздела я упомяну, что использую блокнот Jupyter. Мы будем использовать его довольно часто.

5.2 Связанные списки

Связанные списки — это динамические структуры данных, в которые мы можем легко вставлять и удалять данные. Элементы данных не хранятся в смежных местах памяти (например, в массивах) в связанном списке. Связанный список состоит из узлов. Узел состоит из двух частей: данных и части указателя. Указатель указывает на следующий узел.

Первый узел связанного списка называется **Head Node**, и если он пуст, то и список пуст. Давайте создадим новый блокнот Jupyter и добавим следующий код:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Это специальный класс для узлов. Давайте создадим класс для связанных списков:

```
class LinkedList:
    def __init__(self):
        self.head = None
```

Мы можем создать пустой связанный список, создать узлы и назначить узлы списку:

```
llist = LinkedList()
llist.head = Node(1)
second = Node(2)
third = Node(3)
llist.head.next = second;
second.next = third;
```

Он создает связанный список. Ниже приводится соглашение, которому следуют большинство людей для визуального представления связанного списка:

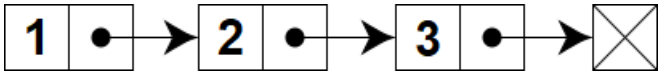


Рис.*5-1:* Связанный список

Точнее, это называется односвязным списком. Существуют и другие типы списков.

Давайте посмотрим, как мы можем пройти по нему и распечатать часть данных всех узлов последовательно, начиная с головного узла. Измените ячейку, в которой мы определили связанный список

```
class LinkedList:
    def __init__(self):
        self.head = None
    def printList(self):
        temp = self.head
        while (temp):
            print (temp.data)
            temp = temp.next
```

Мы идем до тех пор, пока не встретим последний узел, где временная переменная становится None. Поскольку мы переопределили определение класса, повторно выполните ячейку, в которой мы создаем связанный список, и запустите следующий код в той же или другой ячейке:

```
llist.printList()
```

Он распечатает часть данных связанного списка.

Давайте научимся вставлять новый узел в начало списка. Добавьте следующий метод в класс связанного списка:

```
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node
```

Измените ячейку, которая создает связанный список:

```
llist = LinkedList()
llist.head = Node(1)
second = Node(2)
third = Node(3)
llist.head.next = second;
second.next = third;
llist.push(0)
llist.printList()
```

Запустите ячейку и посмотрите результат. Аналогичным образом мы можем написать метод, который может добавлять узел в конец (операция добавления). Добавьте следующий код в класс связанного списка:

```
    def append(self, new_data):
        new_node = Node(new_data)
        if self.head is None:
            self.head = new_node
            return
        last = self.head
        while (last.next):
            last = last.next
        last.next =  new_node
```

Измените последнюю ячейку следующим образом:

```
llist = LinkedList()
llist.head = Node(1)
second = Node(2)
third = Node(3)
```

```
llist.head.next = second;
second.next = third;
llist.push(0)
llist.append(4)
llist.printList()
```

Теперь давайте напишем метод, принимающий аргументы. Если аргумент совпадает с частью данных любого узла в связанном списке, этот узел удаляется. Ограничение состоит в том, что если несколько узлов имеют один и тот же элемент в части данных, метод удаляет только первый из них. Ниже приводится метод:

```
    def deleteNode(self, key):
        temp = self.head
        if (temp is not None):
            if (temp.data == key):
                self.head = temp.next
                temp = None
                return
        while(temp is not None):
            if temp.data == key:
                break
            prev = temp
            temp = temp.next
        if(temp == None):
            return
        prev.next = temp.next
        temp = None
```

Мы можем проверить, работает ли он, внеся следующие изменения в код драйвера:

```
llist = LinkedList()
llist.head = Node(1)
second = Node(2)
third = Node(3)
llist.head.next = second;
second.next = third;
llist.push(0)
llist.append(4)
llist.printList()
llist.deleteNode(2)
llist.printList()
```

Мы можем удалить весь связанный список, удалив все узлы один за другим:

```
    def deleteList(self):
        current = self.head
        while current:
            prev = current.next
            del current.data
            current = prev
```

Давайте изменим код драйвера:

```
llist = LinkedList()
llist.head = Node(1)
second = Node(2)
third = Node(3)
llist.head.next = second;
second.next = third;
llist.push(0)
llist.append(4)
llist.printList()
llist.deleteNode(2)
llist.printList()
llist.deleteList()
llist.printList()
```

Последняя строка выдает ошибку, поскольку список уже удален.

Мы также можем пройти по списку и увеличить переменную-счетчик. В конце концов мы можем найти длину связанного списка. Код прост:

```
    def findLength(self):
        temp = self.head
        count = 0
        while (temp):
            count += 1
            temp = temp.next
        return count
```

Измените код драйвера:

```
llist = LinkedList()
llist.head = Node(1)
second = Node(2)
third = Node(3)
llist.head.next = second;
second.next = third;
```

```
llist.push(0)
llist.append(4)
#llist.printList()
print(llist.findLength())
```

Он напечатает длину списка. У нас даже может быть рекурсивный метод:

```
    def findLengthRec(self, node):
        if (not node):
            return 0
        else:
            return 1 + self.findLengthRec(node.next)
```

Измените код драйвера:

```
llist = LinkedList()
llist.head = Node(1)
second = Node(2)
third = Node(3)
llist.head.next = second;
second.next = third;
llist.push(0)
llist.append(4)
#llist.printList()
print(llist.findLength())
print(llist.findLengthRec(llist.head))
```

Мы также можем написать метод для поиска элемента в списке. Метод возвращает **True**, если элемент найден в списке, иначе он возвращает **False**:

```
    def search(self, x):
        current = self.head
        while current != None:
            if current.data == x:
                return True
            current = current.next
        return False
```

Код драйвера можно изменить следующим образом:

```
llist = LinkedList()
llist.head = Node(1)
second = Node(2)
third = Node(3)
llist.head.next = second;
second.next = third;
```

```
llist.push(0)
llist.append(4)
#llist.printList()
print(llist.search(3))
print(llist.search(6))
```

Структура данных, которую мы только что изучили, известна как **Singly Linked List** - односвязный список. Это потому, что у него есть только одна ссылка (или указатель), указывающая вперед.

5.2.1 Двусвязный список

Теперь мы знакомы с концепцией односвязного списка. Давайте посмотрим на расширенную структуру данных (или, скорее, на улучшение односвязного списка). Мы можем создать связанный список таким образом, чтобы в нем было две ссылки (указателя). Таким образом, мы можем пройти по списку в обе стороны. Давайте определим структуру данных узла:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
```

Мы можем определить класс как двусвязный список:

```
class DoublyLinkedList:

    def __init__(self):
        self.head = None
```

Мы можем добавить метод для помещения элемента в начало списка:

```
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        if self.head is not None:
            self.head.prev = new_node
        self.head = new_node
```

Также можно написать метод для добавления списка:

```
    def append(self, new_data):
        new_node = Node(new_data)
        if self.head is None:
            self.head = new_node
            return
        last = self.head
        while last.next:
```

```
            last = last.next
        last.next = new_node
        new_node.prev = last
        return
```

Мы можем пройти по списку:

```
    def printList(self, node):
        print ("Traversal in forward direction")
        while node:
            print(node.data)
            last = node
            node = node.next

        print ("Traversal in reverse direction")
        while last:
            print(last.data)
            last = last.prev
```

Давайте напишем код драйвера, чтобы продемонстрировать использование этих функций:

```
llist = DoublyLinkedList()
llist.push(1)
llist.append(2)
llist.append(3)
llist.printList(llist.head)
```

Приведенный выше код создает двусвязный список, как показано ниже:

Рис.*5-2:* Двусвязный список

5.3 Стек

Стек — это линейная структура данных, открытая с одного конца и закрытая с другого. Это означает, что программный доступ к нему возможен только с одного конца. Мы можем вставить и извлечь элемент с одного и того же конца. Операция вставки элемента в стек называется **Push**, а операция удаления элемента из стека называется **Pop**.

Существует множество способов реализации стека в Python, и мы рассмотрим многие из них.

Давайте посмотрим, как создать стек со списком в Python:

```
stack = []
stack.append('a')
```

Он создает пустой стек и помещает в него элемент. Давайте распечатаем содержимое стека и посмотрим результат:

```
print(stack)
```

Давайте добавим еще несколько элементов:

```
stack.append('b')
stack.append('c')
print(stack)
```

Мы можем извлечь элементы стека следующим образом:

```
print(stack.pop())
print(stack.pop())
print(stack.pop())
```

Давайте распечатаем стек:

```
print(stack)
```

Он покажет пустой список. Если мы попытаемся извлечь элемент из пустого стека, он выдаст исключение:

```
print(stack.pop())
```

Давайте реализуем стек с модулем **Deque** на Python:

```
from queue import LifoQueue

stack = LifoQueue(maxsize=5)

print("Current number of element: ", stack.qsize())

for i in range(0, 5):
    stack.put(i)
    print("Element  Inserted : " + str(i))

print("\nCurrent number of element: ", stack.qsize())
print("\nFull: ", stack.full())
print("Empty: ", stack.empty())
```

```
print('\nElements popped from the stack')
for i in range(0, 5):
    print(stack.get())

print("\nEmpty: ", stack.empty())
print("Full: ", stack.full())
```

Результат следующий:

```
Current number of
element:  0Element  Inserted : 0Element
Inserted : 1
Element  Inserted : 2
Element  Inserted : 3
Element  Inserted : 4

Current number of element:  5

Full:  True
Empty:  False
```

Элементы извлечены из стека

```
4
3
2
1
0

Empty:  True
Full:  False
```

Мы также можем определить стек более питоническим способом, используя модуль **Deque**:

```
from collections import deque
class Stack:

    def __init__(self):
        self.stack = deque()

    def isEmpty(self):
        if len(self.stack) == 0:
            return True
        else:
            return False

    def length(self):
```

```
        return len(self.stack)

    def top(self):
        return self.stack[-1]

    def push(self, x):
        self.x = x
        self.stack.append(x)

    def pop(self):
        self.stack.pop()
```

Давайте напишем код драйвера и научимся переворачивать строку со стеком. Мы читаем и помещаем символы строки в стек, а затем извлекаем их и создаем новую строку с выдвинутыми элементами следующим образом:

```
str1 = "Test_string"
n = len(str1)
stack = Stack()
for i in range(0, n):
    stack.push(str1[i])
reverse = ""
while not stack.isEmpty():
    reverse = reverse + stack.pop()
print(reverse)
```

Мы также можем реализовать стек, используя связанный список. Выполните это как упражнение.

5.4 Очередь

Очереди — это линейные структуры данных, которые позволяют вставлять данные с одного конца и удалять с другого. В этом отличие от стеков, которые работают по принципу «последним пришел — первым вышел» (**LIFO**). Очереди имеют формат «первым пришел — первым вышел» (**FIFO**). Добавление элемента в очередь называется **enqueue** - постановкой в очередь, а удаление элемента — **dequeue** - удалением из очереди. Мы можем реализовать очереди различными способами. Давайте реализуем их со списками в Python. Давайте посмотрим код:

```
queue = []
queue.append('a')
print(queue)
Let's add more elements to the queue:
queue.append('b')
queue.append('c')
print(queue)
```

Мы можем удалить несколько элементов:

```
print(queue.pop(0))
print(queue.pop(0))
print(queue)
```

Теперь очередь пуста, и попытка удалить из нее еще один элемент вызовет исключение:

```
print(queue.pop(0))
```

Мы также можем реализовать очереди с помощью класса **synchronized queue** в Python:

```
from queue import Queue
q = Queue(maxsize=3)
q.qsize()
```

При выполнении приведенный выше код печатает текущее количество элементов в очереди. Мы можем добавить несколько пунктов следующим образом:

```
q.put('a')
q.put('b')
q.put('c')
print(q.full())
```

Мы можем удалить элементы следующим образом:

```
print(q.get())
print(q.get())
print(q.get())
print(q.empty())
```

Мы также можем реализовать это с помощью **adeque**:

```
from collections import deque
q = deque()
```

Добавим к этому несколько элементов:

```
q.append('a')
q.append('b')
q.append('c')

print("Contents of the queue")
print(q)
```

Результат следующий:

Contents of the queue
deque(['a', 'b', 'c'])

Мы можем удалить элементы:

```
print(q.popleft())
print(q.popleft())
print(q.popleft())

print("\nAn empty Queue: ")
print(q)
```

Результат следующий:

a
b
c

An empty Queue:

deque([])

В предыдущем разделе я просил вас реализовать стек со связанным списком. Мы можем сделать то же самое и для очередей. Здесь мы научимся это реализовывать. Если вы написали программу для стека со связным списком, вы увидите, что мы можем изменить ту же программу в соответствии с нашими целями. Давайте напишем код:

```
class Node:

    def __init__(self, data):
        self.data = data
        self.next = None
```

Давайте определим очередь, используя этот узел:

```
class Queue:

    def __init__(self):
        self.front = self.rear = None

    def isEmpty(self):
        return self.front == None

    # Method to add an item to the queue
    def EnQueue(self, item):
```

```
        temp = Node(item)

        if self.rear == None:
            self.front = self.rear = temp
            return
        self.rear.next = temp
        self.rear = temp

    def DeQueue(self):

        if self.isEmpty():
            return
        temp = self.front
        self.front = temp.next

        if(self.front == None):
            self.rear = None
```

Наконец, мы можем написать код драйвера:

```
q = Queue()
q.EnQueue(1)
q.EnQueue(2)
q.DeQueue()
q.DeQueue()
q.EnQueue(3)
q.EnQueue(4)
q.EnQueue(5)
q.DeQueue()
```

Давайте напечатаем элементы в начале и конце очереди:

```
print("\nThe front of the Queue : " + str(q.front.data))
print("\nThe rear of the Queue : " + str(q.rear.data))
```

Он производит следующий вывод:

The front of the Queue : 4
The rear of the Queue : 5

5.4.1 Двусторонние очереди

Мы уже использовали класс deque для реализации простой односторонней очереди. Это двусторонняя очередь, в которую мы можем вставлять и удалять данные с обоих концов. Давайте воспользуемся им так, как оно было задумано, и напишем пример программы, чтобы продемонстрировать его двусторонние возможности:

```
import collections
deq = collections.deque([10, 20, 30])
print (deq)
```

Мы также можем добавить несколько элементов с одного конца:

```
deq.append(40)
print (deq)
```

Мы можем добавить элемент с другого конца:

```
deq.appendleft(0)
print (deq)
```

Мы можем удалить элемент с конца:

```
deq.pop()
print (deq)
```

Мы также можем удалить элемент с другого конца:

```
deq.popleft()
print (deq)
```

5.4.2 Круговая очередь

Циклическая очередь — это структура данных, которая также известна под названиями циклический буфер, циклический буфер или кольцевой буфер. Он представлен в виде связного кольца. Он имеет указатели начала и конца.
Но реальная память никогда не организуется в кольцо (по крайней мере, физически). Итак, мы используем линейные структуры данных, чтобы продемонстрировать это. Ниже приведено концептуальное представление кольцевого буфера:

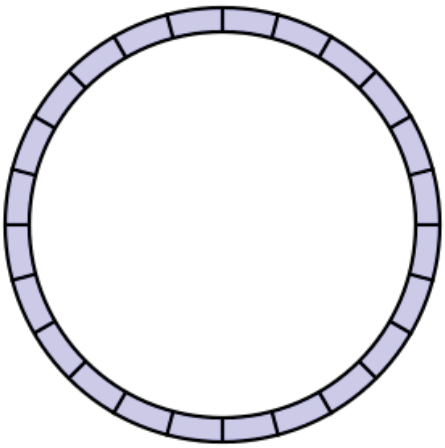


Рис.*5-3:* Круговой буфер

Давайте определим циклический буфер и связанные с ним операции:

```
class CircularQueue():

    def __init__(self, size):
        self.size = size
        self.queue = [None for i in range(size)]
        self.front = self.rear = -1

    def enqueue(self, data):

        if ((self.rear + 1) % self.size == self.front):
            print("The Circular Queue is full...")

        elif (self.front == -1):
            self.front = 0
            self.rear = 0
            self.queue[self.rear] = data
        else:
            self.rear = (self.rear + 1) % self.size
            self.queue[self.rear] = data

    def dequeue(self):
        if (self.front == -1):
            print ("The Circular Queue is empty...")
        elif (self.front == self.rear):
            temp=self.queue[self.front]
            self.front = -1
            self.rear = -1
            return temp
        else:
            temp = self.queue[self.front]
            self.front = (self.front + 1) % self.size
            return temp

    def show(self):
        if(self.front == -1):
            print("The Circular Queue is Empty...")
        elif (self.rear >= self.front):
            print("Elements in the circular queue are: ")
            for i in range(self.front, self.rear + 1):
                print(self.queue[i])
            print ()
        else:
            print ("Elements in the Circular Queue are: ")
            for i in range(self.front, self.size):
```

```
                print(self.queue[i])
            for i in range(0, self.rear + 1):
                print(self.queue[i])

        if ((self.rear + 1) % self.size == self.front):
            print("The Circular Queue is full...")
```

Давайте напишем программу-драйвер для использования этого:

```
cq = CircularQueue(5)
cq.enqueue(1)
cq.enqueue(2)
cq.enqueue(3)
cq.enqueue(4)
cq.show()
print ("Dequed value = ", cq.dequeue())
cq.show()
cq.enqueue(5)
cq.enqueue(6)
cq.enqueue(7)
print ("Dequed value = ", cq.dequeue())
cq.show()
```

Вывод следующий:

Elements in the circular queue are:

1
2
3
4

Dequed value = 1

Elements in the circular queue are:

2
3
4

The Circular Queue is full...
Dequed value = 2

Elements in the Circular Queue are:

3
4

5
6

Мы также можем определить циклическую очередь со связанным списком:

```
class Node:
    def __init__(self):
        self.data = None
        self.link = None
```

Давайте определим класс для циклической связанной очереди:

```
class Queue:
    def __init__(self):
        front = None
        rear = None
```

Мы можем написать метод вне определения класса, чтобы добавить элемент в эту циклическую очередь:

```
def enQueue(q, value):
    temp = Node()
    temp.data = value
    if (q.front == None):
        q.front = temp
    else:
        q.rear.link = temp
    q.rear = temp
    q.rear.link = q.front
```

Мы можем написать еще один метод, который извлекает элемент из очереди:

```
def deQueue(q):
    if (q.front == None):
        print("The circular queue is empty")
        return -999999999999

    value = None
    if (q.front == q.rear):
        value = q.front.data
        q.front = None
        q.rear = None
    else:
        temp = q.front
        value = temp.data
        q.front = q.front.link
        q.rear.link = q.front
```

```
    return value
```

Следующая функция показывает элементы очереди:

```
def show(q):
    temp = q.front
    print("The elements in the Circular Queue are: ")
    while (temp.link != q.front):
        print(temp.data)
        temp = temp.link
    print(temp.data)
```

Код драйвера для демонстрации всех вышеперечисленных функций выглядит следующим образом:

```
q = Queue()
q.front = q.rear = None

enQueue(q, 1)
enQueue(q, 2)
enQueue(q, 3)
show(q)

print("Dequed value = ", deQueue(q))
print("Dequed value = ", deQueue(q))
show(q)

enQueue(q, 4)
enQueue(q, 5)
show(q)
```

Вот вывод кода:

The elements in the Circular Queue are: - Элементами круговой очереди являются:
1
2
3
Dequed value = 1
Dequed value = 2

The elements in the Circular Queue are:

3

The elements in the Circular Queue are:

3
4
5

Краткое содержание

В этой главе мы исследовали традиционные линейные структуры данных. Теперь нам комфортно работать со стеками, очередями и связанными списками.

Следующая глава будет очень интересной и захватывающей: мы изучим графику черепах и узнаем множество рецептов черепах. Если вы креативны и ищете приключений в области графики, вам понравится следующая глава.

Глава 6 • Черепашья графика

В предыдущей главе мы исследовали различные структуры данных и продемонстрировали их с помощью программирования на Python.

В этой главе мы будем использовать встроенную библиотеку черепах в Python для рисования привлекательных графических фигур. В следующем списке представлены темы, которые мы рассмотрим в этой главе:

- История черепахи
- Начиная
- Рецепты с черепахой
- Визуализация рекурсии
- Несколько черепах.

После этой главы нам будет удобно рисовать фигуры с помощью черепахи.

6.1 История черепахи

Черепахи — это класс образовательных роботов, созданных в конце 1940-х годов под руководством исследователя Уильяма Грея Уолтера. Они используются при преподавании информатики и машиностроения. Эти роботы спроектированы так, чтобы находиться низко над землей. Они имеют очень маленький радиус поворота для более точного управления направлением. Иногда они также оснащаются датчиками.

Logo — это образовательный язык программирования, разработанный Уолли Фёрзейгом, Сеймуром Папертом и Синтией Соломон. Черепашья графика — одна из особенностей языка программирования логотипов. Для рисования на экране или листе бумаги он использует экранную или физическую черепаху соответственно.

Язык программирования Python также имеет библиотеку для графики Turtle. В этой главе мы подробно рассмотрим эту библиотеку.

6.2 Начало работы

Мы можем начать с установки необходимых библиотек. Установите Tkinterlibrary, используя следующую команду:

```
pi@pi-desktop:~$ sudo apt -y install python3-tk
```

Библиотеке Turtle нужен Tkinter. Turtle предустановлен в большинстве дистрибутивов Python.

Если в вашей установке Python нет Turtle, ее можно установить с помощью следующей команды:

```
pi@pi-desktop:~$ pip3 install PythonTurtle
```

Давайте запустим Python в интерактивном режиме и импортируем черепаху:

```
>>> import turtle as Turtle
```

Он импортирует модуль с псевдонимом Turtle. Теперь выполните следующий оператор:

```
>>> dir(Turtle)
```

Он возвращает следующий список:

```
['Canvas', 'Pen', 'RawPen', 'RawTurtle', 'Screen', 'ScrolledCanvas',
'Shape', 'TK', 'TNavigator', 'TPen', 'Tbuffer', 'Terminator', 'Turtle',
'TurtleGraphicsError', 'TurtleScreen', 'TurtleScreenBase', 'Vec2D', '_CFG',
'_LANGUAGE', '_Root', '_Screen', '_TurtleImage', '__all__', '__builtins__',
'__cached__', '__doc__', '__file__', '__forwardmethods', '__func_body', '__
loader__', '__methodDict', '__methods', '__name__', '__package__', '__spec__',
'__stringBody', '_alias_list', '_make_global_funcs', '_screen_docrevise', '_
tg_classes', '_tg_screen_functions', '_tg_turtle_functions', '_tg_utilities',
'_turtle_docrevise', '_ver', 'addshape', 'back', 'backward', 'begin_fill',
'begin_poly', 'bgcolor', 'bgpic', 'bk', 'bye', 'circle', 'clear', 'clearscreen',
'clearstamp', 'clearstamps', 'clone', 'color', 'colormode', 'config_dict',
'deepcopy', 'degrees', 'delay', 'distance', 'done', 'dot', 'down', 'end_
fill', 'end_poly', 'exitonclick', 'fd', 'fillcolor', 'filling', 'forward',
'get_poly', 'get_shapepoly', 'getcanvas', 'getmethparlist', 'getpen',
'getscreen', 'getshapes', 'getturtle', 'goto', 'heading', 'hideturtle',
'home', 'ht', 'inspect', 'isdown', 'isfile', 'isvisible', 'join', 'left',
'listen', 'lt', 'mainloop', 'math', 'mode', 'numinput', 'onclick', 'ondrag',
'onkey', 'onkeypress', 'onkeyrelease', 'onrelease', 'onscreenclick',
'ontimer', 'pd', 'pen', 'pencolor', 'pendown', 'pensize', 'penup', 'pos',
'position', 'pu', 'radians', 'read_docstrings', 'readconfig', 'register_shape',
'reset', 'resetscreen', 'resizemode', 'right', 'rt', 'screensize', 'seth',
'setheading', 'setpos', 'setposition', 'settiltangle', 'setundobuffer', 'setup',
'setworldcoordinates', 'setx', 'sety', 'shape', 'shapesize', 'shapetransform',
'shearfactor', 'showturtle', 'simpledialog', 'speed', 'split', 'st', 'stamp',
'sys', 'textinput', 'tilt', 'tiltangle', 'time', 'title', 'towards', 'tracer',
'turtles', 'turtlesize', 'types', 'undo', 'undobufferentries', 'up', 'update',
'width', 'window_height', 'window_width', 'write', 'write_docstringdict', 'xcor',
'ycor']
```

Это атрибуты и методы, доступные в библиотеке черепах. Поскольку мы импортировали библиотеку под псевдонимом **Turtle,** мы можем использовать **Turtle** для их объектно-ориентированного вызова. Если мы хотим узнать больше о каком-либо из этих атрибутов или методов, мы можем использовать функцию **help().**

```
>>> help(Turtle.fd)
```

Откроется страница справки в стиле UNIX, которую можно закрыть, нажав клавишу **q** на клавиатуре.

6.3 Изучение методов Turtle

Давайте рассмотрим методы черепах. Мы знаем положение черепахи:

```
>>> Turtle.position()
```

Оно показывает следующий вывод:

```
(0.00,0.00)
```

Это означает, что Turtle находится в начале холста (0,0). При выполнении оператора также открывается графическое окно, в котором Turtle представлена в виде стрелки, указывающей правильное направление с нашей точки зрения. Это показано на следующем рисунке (рис. 6-1):

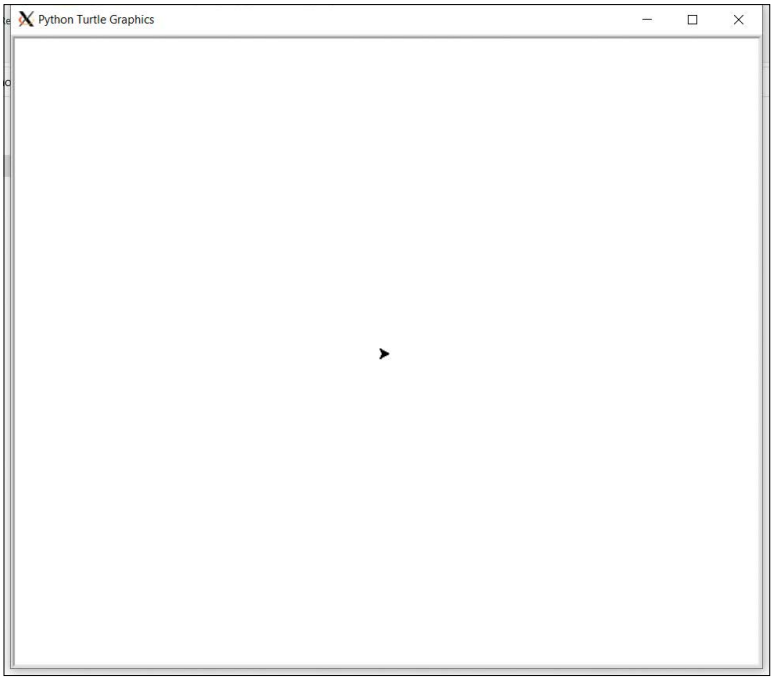


Рис.*6-1:* Черепаха в начале координат *(0,0)*

Мы можем переместить **Turtle** вперед с помощью метода **forward()** или **fd()**.

```
>>> Turtle.forward(25)
```

Or

```
>>> Turtle.fd(25)
```

Если мы снова запустим оператор **Position()**, мы увидим текущие координаты Черепах следующим образом:

```
>>> Turtle.position()
```

Результат следующий: (25.00,0.00)

Точно так же мы можем пройти назад любым из следующих методов:

```
>>> Turtle.backward(10)
>>> Turtle.bk (10)
>>> Turtle.back (10)
```

Методы **left()** или **lt()** и **right()** или **rt()** используются для поворота черепахи влево и вправо на заданный угол. По умолчанию угол указывается в градусах, но мы также можем установить его в радианах с помощью **radians()**. Мы можем вернуть его в градусы с помощью процедуры **degrees()**. По умолчанию мы видим стрелку на экране. Но мы можем изменить его на другие формы с помощью функции **shape()**. Допустимые аргументы: «"arrow», «Turtle», «circle», «square», «triangle» и «" classic». Давайте изменим форму следующим образом:

```
>>> Turtle.shape("turtle")
```

Теперь давайте учимся дальше, создавая несколько небольших рецептов. Остальные функции мы изучим по мере необходимости их использования. До сих пор мы работали в интерактивном режиме. Давайте теперь начнем создавать файл скрипта.

6.4 Рецепты с Turtle

Нарисуем квадрат с Turtle. Откройте IDLE и создайте файл. Напишите в нем следующий код:

prog00.py

```
import turtle as Turtle
import time

Turtle.forward(150)
Turtle.left(90)
Turtle.forward(150)
Turtle.left(90)
Turtle.forward(150)
Turtle.left(90)
Turtle.forward(150)
Turtle.left(90)
time.sleep(10)
Turtle.bye()
```

Приведенная выше программа нарисует квадрат и подождет 10 секунд, прежде чем закрыть окно.

Мы можем написать это в более питоническом стиле следующим образом:

prog01.py
```
import turtle as Turtle
import time

for i in range(4):
    Turtle.forward(150)
    Turtle.left(90)

time.sleep(10)
Turtle.bye()
```

Запустите программу и посмотрите результат. Давайте добавим к нему больше возможностей:

prog02.py
```
import turtle as Turtle
import time

def square(length):
    for i in range(4):
        Turtle.forward(length)
        Turtle.left(90)

if __name__ == "__main__":
    square(150)
    time.sleep(10)
    Turtle.bye()
```

Мы можем рисовать круги:

prog03.py
```
import turtle as Turtle

Turtle.circle(50)
Turtle.circle(-50)
```

Результат показан на рис.6-2.

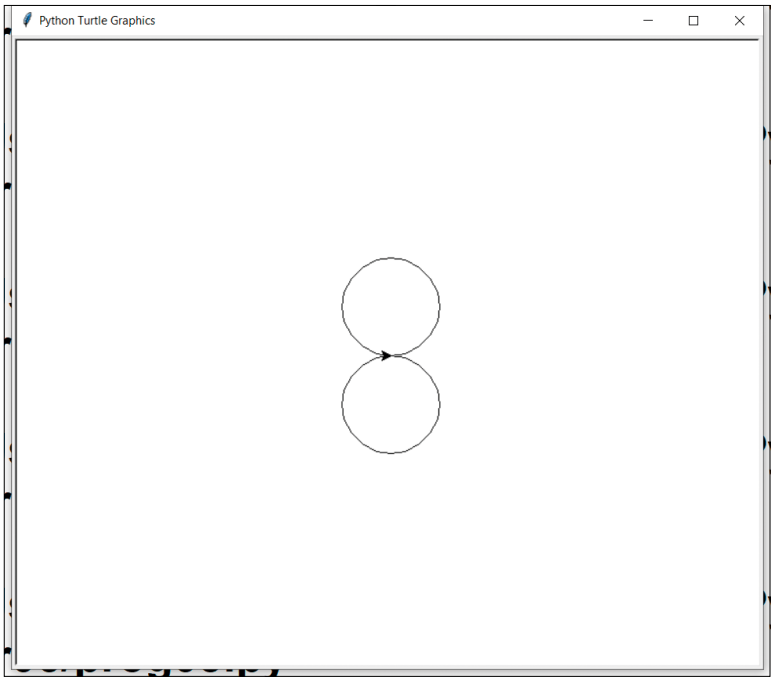


Рис.*6-2:* Рисование кругов

Аргумент, передаваемый подпрограмме Circle(), определяет радиус и направление круга. Давайте воспользуемся этой подпрограммой в сочетании с другими подпрограммами, чтобы создать красивую диаграмму:

prog04.py

```
import turtle as Turtle

Turtle.color("green")
for angle in range(0, 360, 10):
    Turtle.seth(angle)
    Turtle.circle(100)
```

В приведенной выше программе (**prog04.py**) мы используем функцию **color()** для установки цвета рисунка. Мы также используем функцию **seth()**, чтобы установить направление Черепахи. Выполнение кода занимает довольно много времени, и по завершении он создает красивый шаблон, показанный на рис. 6-3.

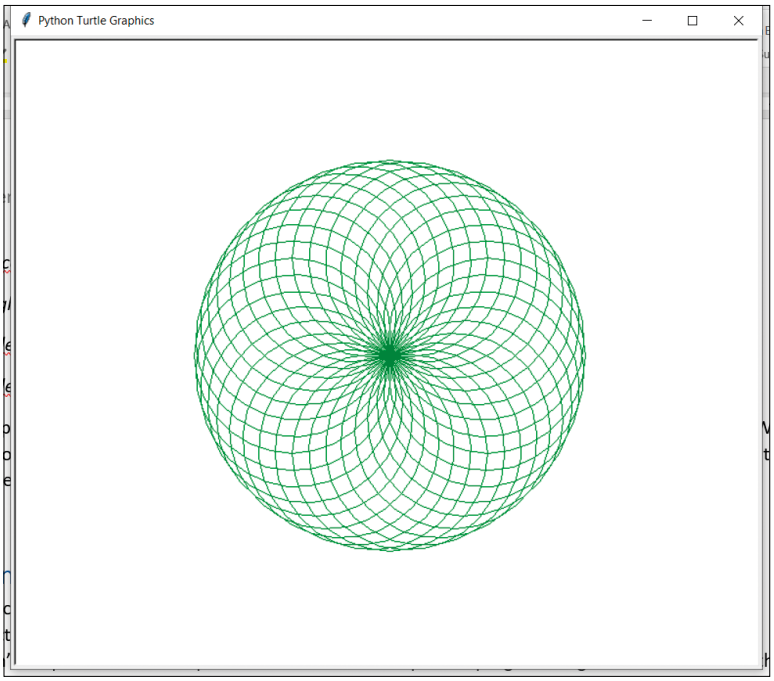


Рис.*6-3:* Рисование красивого узора из кругов.

Мы также можем нарисовать круг, не используя встроенную функцию **Circle()**. Логика в том, что мы двигаем Черепаху вперед на одну позицию, а затем делаем поворот на угол в один градус, делаем это 360 раз и получаем круг. Вот код:

prog05.py

```
import turtle as Turtle
count = 0
while(count < 360):
    Turtle.forward(1)
    Turtle.left(1)
    count = count + 1
```

Запустите программу и посмотрите результат.

Мы также можем установить цвет фона. Напишем программу случайного блуждания В этом примере фон установлен черным.

prog06.py

```
import turtle as Turtle
import random

Turtle.speed(10)
Turtle.bgcolor('Black')
```

```
turns = 1000
distance = 20

for x in range(turns):
    right=Turtle.right(random.randint(0, 360))
    left=Turtle.left(random.randint(0, 360))
    Turtle.color(random.choice(['Blue', 'Red', 'Green',
                               'Cyan', 'Magenta', 'Pink', 'Violet']))
    random.choice([right,left])
    Turtle.fd(distance)
```

Она выдает следующий результат (рис. 6-4):

Рис.*6-4:* Рисунок на черном фоне.

Иногда выполнение программ занимает много времени. Мы можем управлять этим, изменяя скорость черепахи. Для этого мы можем использовать функцию **Speed()**. Скорость может варьироваться от 0 до 10. Ниже показано сопоставление строк скорости со значениями. По умолчанию установлено «normal».

- "fastest" corresponds to 0 «самый быстрый» соответствует 0
- "fast" corresponds to 10 «быстро» соответствует 10
- "normal" corresponds to 6 «нормальный» соответствует 6
- "slow" corresponds to 3 «медленный» соответствует 3
- "slowest" corresponds to 1 «самый медленный» соответствует 1

Давайте сделаем еще несколько интересных рисунков. Мы использовали круги (**prog04.py**), чтобы нарисовать красивый узор (рис.6-3). Мы можем использовать круги, чтобы нарисовать еще более сложные узоры:

```
prog07.py
import turtle as Turtle
Turtle.speed(10)
Turtle.bgcolor('Black')
colors=['Red', 'Yellow', 'Purple',
        'Cyan', 'Orange', 'Pink']
for x in range(100):
    Turtle.circle(x)
    Turtle.color(colors[x%6])
    Turtle.left(60)
```

Результат показан на рис.6-5.

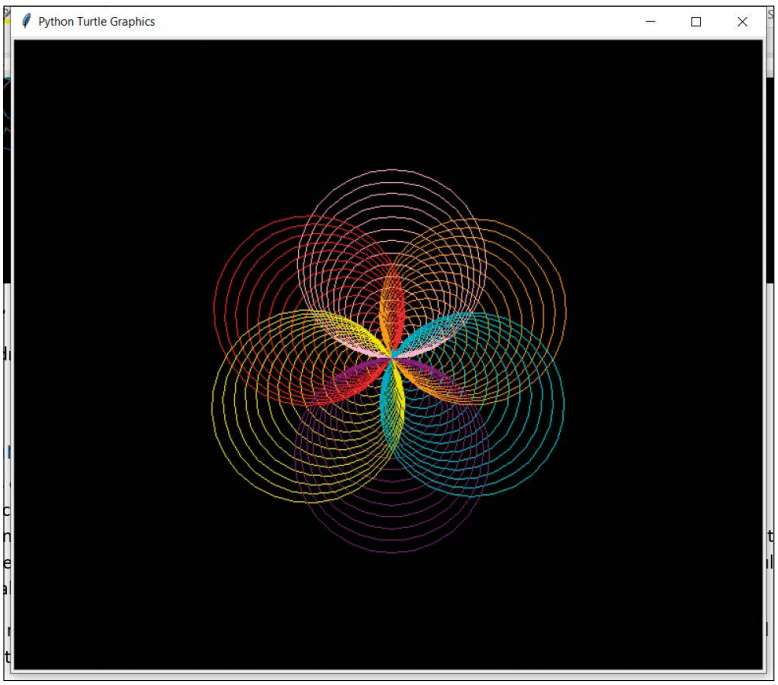


Рис.*6-5:* Цветочные узоры

Мы можем нарисовать линии, образующие красочный квадратный узор:

```
prog08.py
import turtle as Turtle
Turtle.speed(0)
Turtle.bgcolor('Black')
colors=['Red','Yellow','Pink','Orange']
```

```
for x in range(300):
    Turtle.color(colors[x%4])
    Turtle.forward(x)
    Turtle.left(90)
```

Результат показан на рис.6-6.

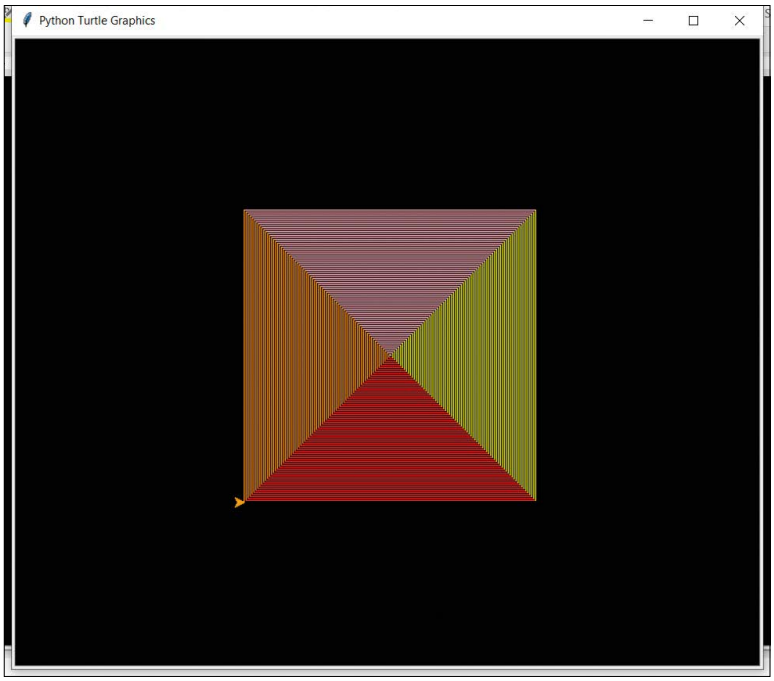


Рис.*6-6:* Квадратные узоры

Если вы не заметили, программа работает быстрее. Это потому, что мы увеличили скорость Черепахи и сделали ее самой быстрой. Мы также можем добавить́ больше цветов на дисплей:

prog09.py

```
import turtle as Turtle
Turtle.speed(0)
Turtle.bgcolor('Black')
colors=['Red', 'Yellow', 'Pink', 'Orange',
        'Blue', 'Green', 'Cyan', 'White']
for x in range(300):
    Turtle.color(colors[x%8])
    Turtle.forward(x)
    Turtle.left(90)
```

Мы также можем нарисовать линии случайным цветом из списка.

```
prog10.py
import turtle as Turtle
import random
Turtle.speed(0)
Turtle.bgcolor('Black')
colors=['Red', 'Yellow', 'Pink', 'Orange',
        'Blue', 'Green', 'Cyan', 'White']
for x in range(300):
    Turtle.color(colors[random.randint(0, 7)])
    Turtle.forward(x)
    Turtle.left(90)
```

Запустите обе программы и посмотрите результат.

Мы также можем нарисовать красивый рисунок шестиугольного узора:

```
prog11.py
import turtle as Turtle

Turtle.bgcolor("black")
colors=["Red","White","Cyan","Yellow","Green","Orange"]

for x in range(300):
    Turtle.color(colors[x%6])
    Turtle.fd(x)
    Turtle.left(59)
```

Результат показан на рис.6-7.

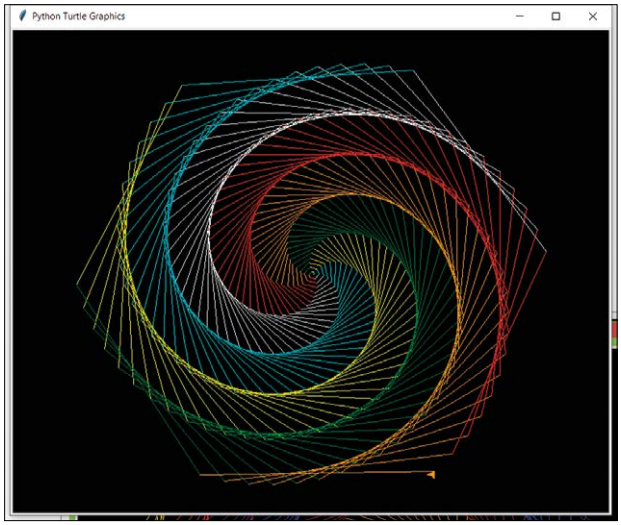


Рис.*6-7:* Чертеж шестиугольного узора.

Мы также можем заполнять фигуры цветами. Прежде чем закончить этот раздел, давайте посмотрим пару примеров. Мы будем использовать процедуры **fillcolor()**,**begin_fill()** и **end_fill()**. Ниже приведен простой пример.

```
prog12.py
Turtle.fillcolor('red')
Turtle.begin_fill()
Turtle.forward(100)
Turtle.left(120)
Turtle.forward(100)
Turtle.left(120)
Turtle.forward(100)
Turtle.left(120)
Turtle.end_fill()
```

Он нарисует треугольник и заполнит его красным цветом. Давайте посмотрим еще один простой пример:

```
prog13.py
import turtle as Turtle
Turtle.fillcolor('Orange')
Turtle.begin_fill()
for count in range(4):
    Turtle.forward(100)
    Turtle.left(90)
Turtle.end_fill()
```

И приведенная выше программа заполняет квадрат оранжевым цветом.

6.5 Визуализация рекурсии

О рекурсии мы узнали в третьей главе. Мы знаем, что вызов функции изнутри себя называется прямой рекурсией. Мы знаем, что есть две важные части рекурсивной функции. Первый — это критерии завершения рекурсии. Второй — рекурсивный вызов. В наших более ранних рекурсивных функциях мы выводили вывод на консоль. Теперь вместо текстового вывода у нас будет графический вывод. Начнем с простой программы:

```
prog00.py
import turtle as Turtle
def zigzag(size, length):
    if size > 0:
        Turtle.left(45)
        Turtle.forward(length)
        Turtle.right(90)
        Turtle.forward(2*length)
        Turtle.left(90)
        Turtle.forward(length)
        Turtle.right(45)
```

```
        zigzag(size-1, length)
if __name__ == "__main__":
    zigzag(5, 50)
```

В файле **prog00.py** мы видим, что у нас есть критерий завершения, который проверяет, больше ли один из переданных аргументов нуля. У нас есть только один рекурсивный вызов, который передает уменьшенный аргумент одному из параметров. Не обязательно должен быть только один рекурсивный вызов. Также может быть несколько рекурсивных вызовов. Мы будем изучать эти возможности в дальнейшем. Но сейчас мы рассмотрим несколько фигур, которые вызывают рекурсивную функцию только один раз. Ниже (рис. 6-8) показаны выходные данные:

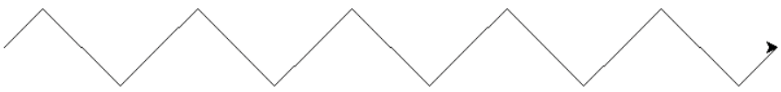


Рис.*6-8:* Вывод вызова функции *Zigzag*

Давайте внесем некоторые изменения в ту часть, где мы вызываем рекурсивную функцию из основной части программы:

```
zigzag(3, 70)
```

Мы можем наблюдать измененный вывод:

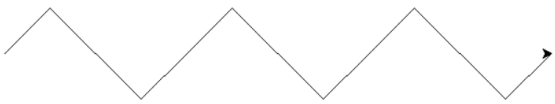


Рис.*6-9:* Вывод вызова функции *Zigzag* с измененными аргументами

Это была одна из самых простых рекурсивных фигур, которые мы можем нарисовать. Нарисуем более сложную фигуру: спираль. Давайте сначала посмотрим код. Потом я объясню это подробно.

prog01.py

```
import turtle as Turtle
def spiral(sideLen, angle, scaleFactor, minLength):
    if sideLen >= minLength:
        Turtle.forward(sideLen)
        Turtle.left(angle)
        spiral(sideLen*scaleFactor, angle,
               scaleFactor, minLength)
if __name__ == "__main__":
    Turtle.speed(0)
    spiral(200, 120, 0.9, 20)
```

Эта программа (**prog01.py**) рисует различные спиралевидные формы на основе аргументов, которые мы передаем рекурсивной функции. Критерием завершения является то, что длина стороны спирали должна быть больше минимальной длины. Если да, то перемещаем Turtle вперед и поворачиваем ее влево.

Затем мы делаем рекурсивный вызов так, что последующая часть спирали является уменьшенной версией предыдущей части. Давайте посмотрим результат:

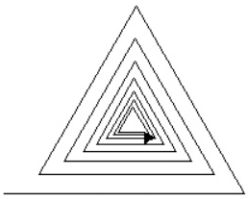


Рис.*6-10:* Треугольная спираль.

Из него можно сделать любую спираль правильной формы. Например, мы можем изменить угол на 90, чтобы изменить рекурсивный вызов:

```
spiral(200, 90, 0.9, 20)
```

Это результат:

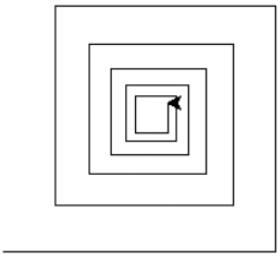


Рис.*6-11:* Квадратная спираль.

Мы можем изменить основной вызов пятиугольной спирали:

```
spiral(200, 72, 0.9, 20)
```

Это результат:

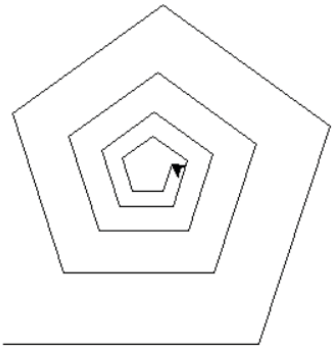


Рис.*6-12:* Пятиугольная спираль.

Наконец, мы можем создать шестиугольную спираль, используя следующий код:

```
spiral(200, 60, 0.9, 20)
```

Вывод следующий:

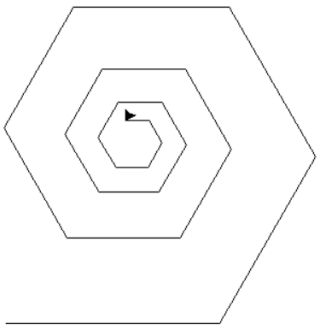


Рис.*6-13:* Шестиугольная спираль.

В качестве упражнения вызовите функцию с другим набором аргументов.

```
spiral(200, 45, 0.9, 20)
spiral(200, 40, 0.9, 20)
spiral(200, 36, 0.9, 20)
spiral(200, 30, 0.9, 20)
spiral(200, 24, 0.9, 20)
spiral(200, 20, 0.9, 20)
spiral(200, 18, 0.9, 20)
```

Ранее в третьей главе мы узнали, как написать программу для рядов Фибоначчи. Мы напечатали числа Фибоначчи на консоли. Здесь мы научимся визуально представлять Дерево Фибоначчи. Вместо печати будем рисовать Черепахой:

```
prog02.py
import turtle as Turtle
def drawfib(n, len_ang):
    Turtle.forward(2 * len_ang)
    if n == 0 or n == 1:
        pass
    else:
        Turtle.left(len_ang)
        drawfib(n - 1, len_ang)
        Turtle.right(2 * len_ang)
        drawfib(n - 2, len_ang)
        Turtle.left(len_ang)
    Turtle.backward(2 * len_ang)
if __name__ == "__main__":
```

```
    Turtle.left(90)
    Turtle.speed(0)
    drawfib(7, 20)
```

Как мы видим, код тот же, и мы только заменяем операторы **print()** движением Turtle. Дерево Фибоначчи является примером рекурсии, когда мы вызываем функцию рекурсивно на одном и том же уровне более одного раза (ровно два раза). Она производит следующее (рис.6-14):

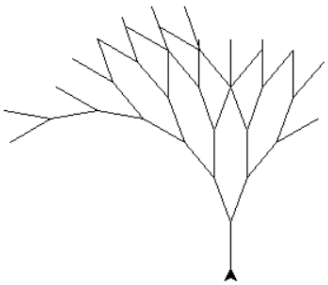


Рис.*6-14:* Дерево Фибоначчи.

Мы можем изменить аргументы вызова функции в основном разделе, чтобы получить разные выходные данные. Ниже приведены несколько примеров:

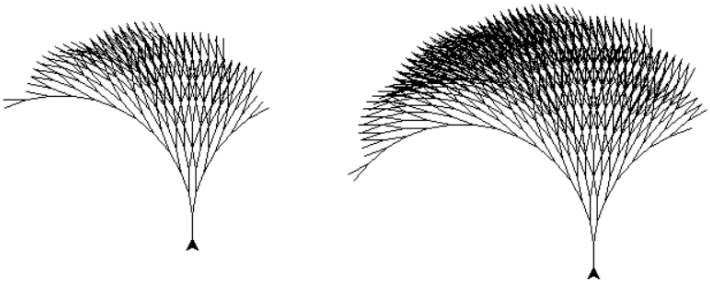


Рис.*6-15:* Деревья Фибоначчи с различными аргументами для вызова функции в основном разделе.

В качестве упражнения вызовите функцию с другим набором аргументов.

Также мы можем нарисовать часть снежинки Коха:

prog03.py

```
import turtle as Turtle
def koch_snowflake(length, depth):
    if depth == 0:
        Turtle.forward(length)
    else:
        length = length / 3
```

```
        depth = depth - 1
        Turtle.color('Blue')
        koch_snowflake(length, depth)
        Turtle.right(60)
        Turtle.color('Orange')
        koch_snowflake(length, depth)
        Turtle.left(120)
        Turtle.color('Red')
        koch_snowflake(length, depth)
        Turtle.right(60)
        Turtle.color('Green')
        koch_snowflake(length, depth)
Turtle.speed(10)
koch_snowflake(500, 4)
```

У нас в коде есть три рекурсивных вызова функций (**prog03.py**).

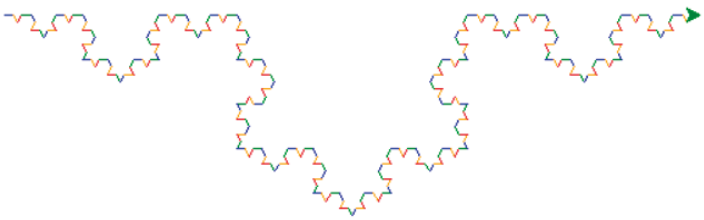


Рис.6-16: Часть снежинки Коха.

Мы также можем нарисовать снежинку целиком, изменив приведенный выше код (**prog03.py**). Нам нужно добавить вызов функции в основной раздел в цикле:

prog04.py

```
import turtle as Turtle
import random
def koch_snowflake(length, depth):
    if depth == 0:
        Turtle.forward(length)
    else:
        length = length / 3
        depth = depth - 1
        Turtle.color(colors[random.randint(0, 8)])
        koch_snowflake(length, depth)
        Turtle.right(60)
        Turtle.color(colors[random.randint(0, 8)])
        koch_snowflake(length, depth)
        Turtle.left(120)
```

```
        Turtle.color(colors[random.randint(0, 8)])
        koch_snowflake(length, depth)
        Turtle.right(60)
        Turtle.color(colors[random.randint(0, 8)])
        koch_snowflake(length, depth)
Turtle.speed(10)
colors = ['Blue', 'Red', 'Orange',
          'Green', 'Magenta', 'Purple',
          'Cyan', 'Violet', 'Black']
for i in range(3):
    koch_snowflake(200, 3)
    Turtle.left(120)
```

В результате получится полная снежинка, как показано на рис.6-17:

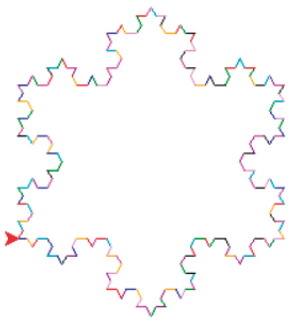


Рис.*6-17:* Снежинка Коха.

Мы можем нарисовать более сложную фигуру. В последних примерах (**prog03.py** и **prog04.py**) мы вызываем функцию рекурсии трижды. В следующем примере мы вызовем функцию рекурсивно четыре раза. Посмотрите на следующий код. Мы используем в нем много новых процедур.

prog05.py

```
import turtle as Turtle
def draw_line(pos1, pos2):
    Turtle.penup()
    Turtle.goto(pos1[0], pos1[1])
    Turtle.pendown()
    Turtle.goto(pos2[0], pos2[1])

def recursive_draw(x, y, width, height, count):
    draw_line([x + width * 0.25, height // 2 + y],
              [x + width * 0.75, height // 2 + y])
    draw_line([x + width * 0.25, (height * 0.5) // 2 + y],
              [x + width * 0.25, (height * 1.5) // 2 + y])
    draw_line([x + width * 0.75, (height * 0.5) // 2 + y],
```

```
                [x + width * 0.75, (height * 1.5) // 2 + y])
    if count <= 0:
        # The leaf node
        return 1
    else:
        recursive_draw(x, y, width // 2, height // 2, count-1)
        recursive_draw(x + width // 2, y, width // 2, height // 2, count-1)
        recursive_draw(x, y + width // 2, width // 2, height // 2, count-1)
        recursive_draw(x + width // 2, y + width // 2, width // 2, height // 2,
count-1)

height = width = 800
screen = Turtle.Screen()
screen.setup(height, width)
screen.title('H Tree Fractal')
screen.bgcolor('White')
Turtle.hideturtle()
Turtle.color('Black')
Turtle.speed(0)
recursive_draw(-height//2, -width//2, height, width, 0)
```

Как мы видим, в этой программе используется несколько новых процедур. Процедура **penup()** заставит Turtle прекратить рисование. Процедура **pendown()** снова активирует рисование. Процедура **goto()** переместит Turtle в указанную позицию. Как упоминалось ранее, функция выполняет четыре рекурсивных вызова самой себя.

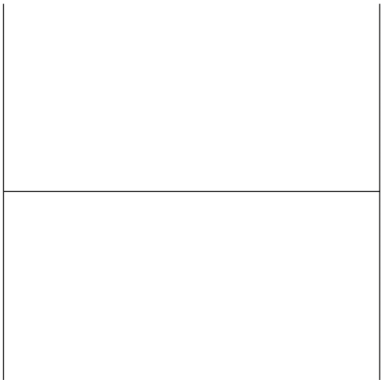
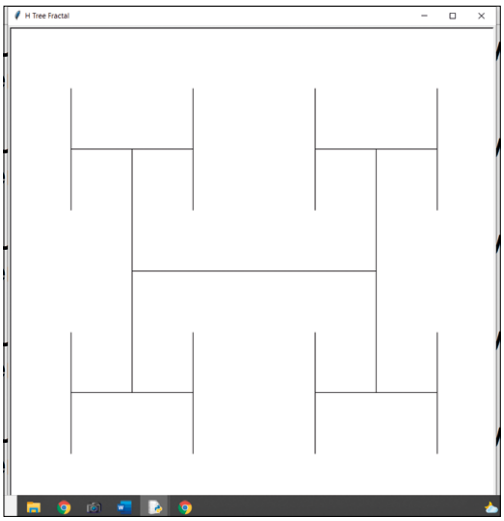


Рис.*6-18: H*-фрактал.

Он известен как фрактал H, поскольку имеет форму символа **H**. Поскольку критерии завершения соблюдены, рекурсивная часть не вызывается. Мы можем изменить уровень рекурсии, чтобы выполнялась рекурсивная часть. Это можно сделать, изменив последний аргумент вызова функции в основной секции:

```
recursive_draw(-height//2, -width//2, height, width, 1)
```

Результат следующий:

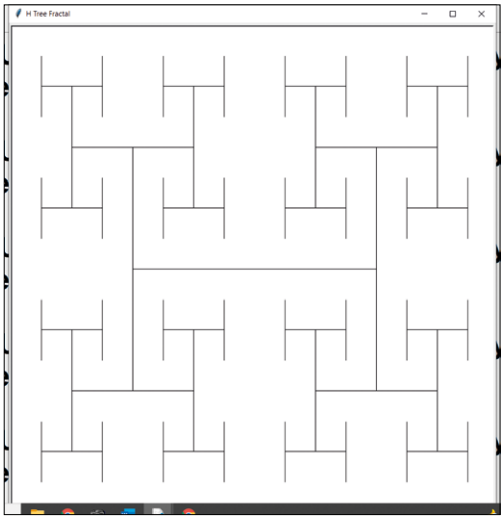


Рис/6-19: H-фрактал с рекурсивной глубиной 1.

На концах вертикальных линий мы имеем уменьшенную версию большей формы. Теперь это действительно рекурсивный вывод. Давайте сделаем рекурсивную глубину равной 2:

```
recursive_draw(-height//2, -width//2, height, width, 2)
```

Вот результат:

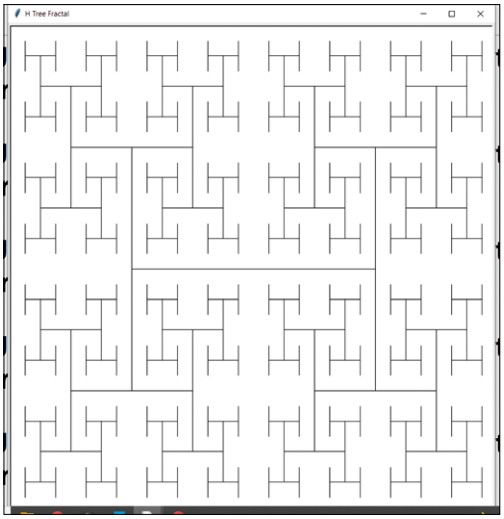


Рис/6-20: H-фрактал с рекурсивной глубиной 2.

Давайте сделаем рекурсивную глубину равной 3:

```
recursive_draw(-height//2, -width//2, height, width, 3)
```

Вот результат

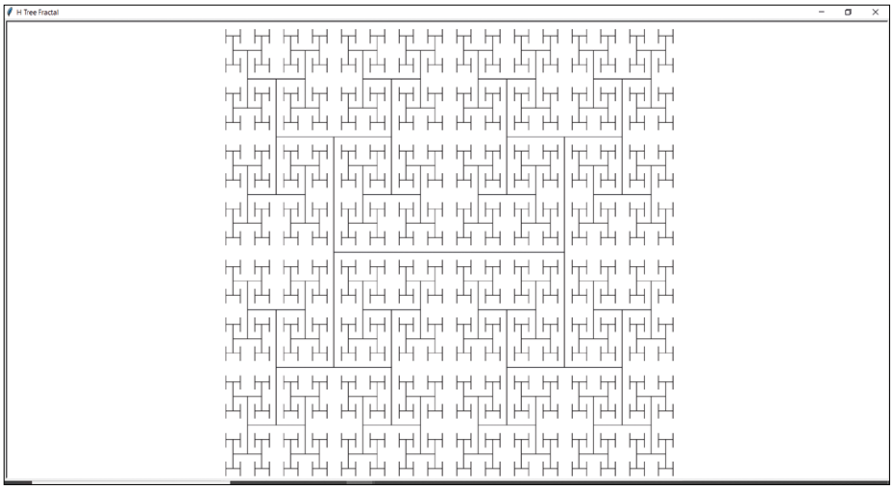


Рис/6-21: H-фрактал с рекурсивной глубиной 3.

Давайте сделаем рекурсивную глубину равной 4:

```
        recursive_draw(-height//2, -width//2, height, width, 4)
```

Вот результат:



Рис/6-22: H-фрактал с рекурсивной глубиной 4.

6.6 Несколько черепах

До сих пор мы использовали в нашем коде только одну черепаху. Мы также можем использовать несколько черепах в нашем коде. Мы должны сделать это объектно-ориентированным способом. Нам нужно создать уникальный объект для каждой черепахи. Давайте посмотрим код:

prog06.py

```
import turtle
t1 = turtle.Turtle()
t2 = turtle.Turtle()
t1.speed(10)
t2.speed(10)
t1.color('Red')
t2.color('Green')
count = 0
while(count < 360):
    t1.forward(1)
    t1.left(1)
    t2.forward(1)
    t2.right(1)
    count = count + 1
```

Мы заставляем обе черепахи рисовать небольшие сегменты за итерацию. Это создает иллюзию, что оба работают параллельно. Запустите код и посмотрите результат.

Краткое содержание

В этой главе мы исследовали библиотеку черепах. Мы нарисовали много фигур. Самое интересное — это визуализация рекурсии. Мы создали различные рекурсивные рисунки, умело используя черепаху. Наконец, мы научились рисовать нескольких черепах.

В следующей главе мы рассмотрим еще одну графическую библиотеку, известную как aspygame.

Глава 7. Программирование анимации и игр.

В предыдущей главе мы подробно изучили графическую библиотеку turtle. Мы научились делать красивые рисунки и рекурсивные фигуры.

В этой главе мы продолжим путь работы с графикой, изучив еще одну популярную библиотеку для графики и анимации. Ниже приводится список тем, которые мы обсудим в этой главе:

- Начало работы с Pygame
- Рекурсия с Pygame
- Треугольник Серпинского от Chaos Game
- Простая анимация с помощью Pygame
- Игра Snake (Змейка)

Как и в предыдущей главе, здесь будет много практических занятий. Прочитав эту - главу, мы научимся рисовать, создавать анимацию и программировать небольшие игры.

7.1 Начало работы с Pygame

Ранее мы использовали библиотеку turtle. Библиотека имеет множество ограничений из-за самой ее природы. Поэтому мы научимся использовать парочку новых библиотек Python. Первая библиотека, о которой мы узнаем, — это Pygame. Давайте сначала установим её. Запустите следующую команду в командной строке вашей операционной системы, чтобы установить Pygame:

```
C:\Users\Ashwin>pip3 install pygame
```

Библиотека будет установлена на ваш компьютер. Команда одинакова на всех платформах.Теперь давайте начнем с основ:

prog00.py

```
import pygame, sys
result = pygame.init()
if result[1] > 0:
    print('Error initializing Pygame : ' + str(result[1]))
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
screen = pygame.display.set_mode((640, 480))
pygame.quit()
sys.exit(0)
```

Расшифруем программу построчно. Первая строка импортирует все библиотеки, которые мы будем использовать в программе. Затем мы используем процедуру **init()** для инициализации библиотеки Pygame в текущей программе. Мы сохраняем возвращаемое значение в переменную и проверяем, возвращает ли оно ошибку.

Затем мы устанавливаем разрешение окна вывода с помощью подпрограммы **set_mode()**. Наконец, мы используем функцию **quit()**, чтобы закрыть сеанс pygame. Запустите программу. Результат не очень заметен. Он создает окно pygame заданных размеров. У него черный фон. Окно на мгновение мигает, прежде чем закрыть его. Поздравляем, мы начинаем работу с библиотекой Pygame! Также не забудьте проверить вывод консоли:

```
pygame 2.0.1 (SDL 2.0.14, Python 3.9.7)

Hello from the pygame community. https://www.pygame.org/
contribute.html

Pygame initialized successfully! - Pygame успешно инициализирован!
```

Давайте добавим к этому цикл событий. Цикл событий записывает все события мыши и клавиатуры и завершает работу программы при нажатии кнопки закрытия. Ниже приведен расширенный код:

prog01.py

```
import pygame, sys
result = pygame.init()
if result[1] > 0:
    print('Error initializing Pygame : ' + str(result[1]))
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
screen = pygame.display.set_mode((640, 480))
running = True
while running:
    for event in pygame.event.get():
        print(event)
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
sys.exit(0)
```

Запустите вышеуказанную программу (prog01.py) и наблюдайте за выводом в терминале. Вы заметите, что программа печатает все действия, выполняемые пользователем (события клавиатуры и мыши).

Давайте создадим небольшое приложение, которое меняет цвет фона при **НАЖИМЕ КНОПКИ МЫШИ**. Это полная программа:

prog02.py

```
import pygame, sys, random
result = pygame.init()
```

```
if result[1] > 0:
    print('Error initializing Pygame : ' + str(result[1]))
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
screen = pygame.display.set_mode((640, 480))
BLACK = (0, 0, 0)
GRAY = (127, 127, 127)
WHITE = (255, 255, 255)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
YELLOW = (255, 255, 0)
CYAN = (0, 255, 255)
MAGENTA = (255, 0, 255)
bgcolor = [BLACK, GRAY, WHITE,
           RED, GREEN, BLUE,
           YELLOW, CYAN, MAGENTA]
background = BLACK
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            background = bgcolor[random.randint(0, 8)]
        screen.fill(background)
        pygame.display.flip()
pygame.quit()
sys.exit(0)
```

Давайте посмотрим на новый код, который мы добавили в файл. Мы изучили кортежи в предыдущей главе. Мы определяем цвета с помощью кортежей, которые имеют комбинацию значений Red, Green и Blue. Эти значения могут варьироваться от 0 до 255. Мы определили 9 различных кортежей для цветов. Затем мы добавляем эти кортежи цветов в список. Если в цикле события мы обнаруживаем событие **MOUSEBUTTONDOWN**, мы случайным образом выбираем цвета из списка и присваиваем этот цвет фону. Наконец, мы устанавливаем фон с помощью процедуры fill() и обновляем изображение с помощью процедуры **flip()**. Процедура **flip()** используется для обновления содержимого всего дисплея. С этого момента мы будем часто использовать его.

Запустите программу и посмотрите, как изменится фон события **MOUSEBUTTONDOWN**.

7.2 Рекурсия с Pygame

Мы изучили концепцию рекурсии в третьей главе. В предыдущей главе мы исследовали это визуально. Давайте еще раз изучим его визуально. Но на этот раз мы будем использовать библиотеку Pygamelibrary. Начнем с чего-то простого. Мы нарисуем простое рекурсивное дерево. Визуализация будет иметь черный фон и белые ветви. Последние несколько ветвей и листьев будут зелеными. Давайте посмотрим код:

prog03.py

```
import pygame, math, random
import time, sys
width, height = 1366, 768
result = pygame.init()
if result[1] > 0:
    print('Error initializing Pygame : ' + str(result[1]))
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
window = pygame.display.set_mode((width, height))
pygame.display.set_caption('Fractal Tree')
screen = pygame.display.get_surface()

def Fractal_Tree(x1, y1, theta, depth):
    if depth:
        rand_length = random.randint(1, 10)
        rand_angle = random.randint(10, 20)
        x2 = x1 + int(math.cos(math.radians(theta))
                      * depth * rand_length)
        y2 = y1 + int(math.sin(math.radians(theta))
                      * depth * rand_length)
        if depth < 5:
            clr = (0, 255, 0)
        else:
            clr = ( 255, 255, 255)
        pygame.draw.line(screen, clr, (x1, y1), (x2, y2), 2)
        Fractal_Tree(x2, y2, theta - rand_angle, depth-1)
        Fractal_Tree(x2, y2, theta + rand_angle, depth-1)
Fractal_Tree( (width/2), (height-10), -90, 14)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
sys.exit(0)
```

Как мы видим, мы использовали новую процедуру **set_caption()** для установки заголовка окна. Мы также используем функцию **draw.line()** для рисования линии. Важно учитывать, что начало координат (0, 0) находится в верхнем левом углу Pygame. Мы знакомы с большей частью кода. Рекурсивная функция принимает в качестве аргументов координаты точки, угла и глубины. Мы также случайным образом генерируем длину ветки. Используя переданные координаты и случайно сгенерированные угол и длину, мы вычисляем конечную точку ветки. Если ветки находятся рядом с листьями, то окрашиваем их в зеленый цвет, иначе в белый. В конце мы рисуем сегмент линии и передаем конечную точку этого сегмента рекурсивному вызову функции. Мы также случайным образом генерируем другое значение угла, добавляем и вычитаем его из переданного угла и используем эти новые значения в качестве аргументов рекурсивного вызова. Давайте посмотрим результаты:

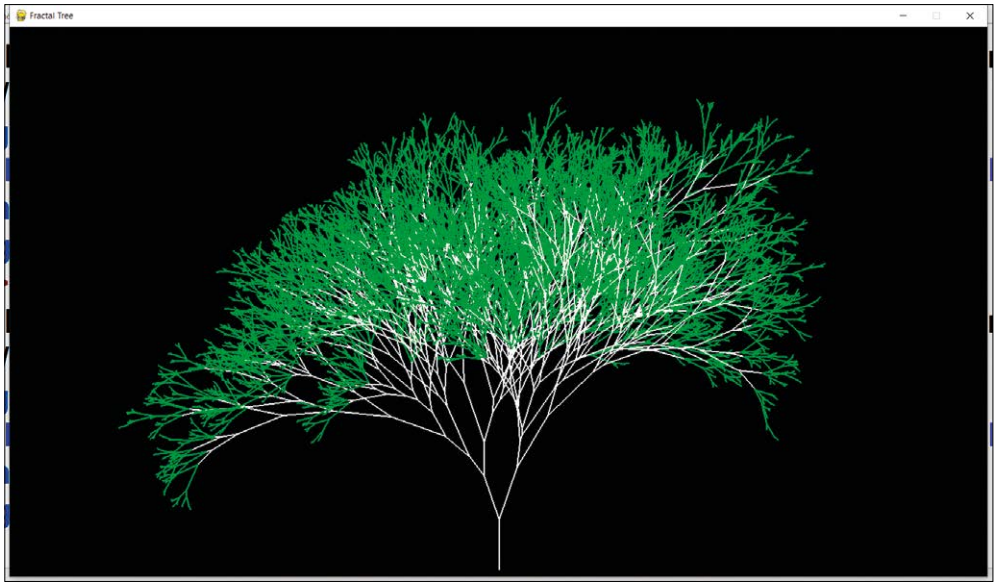


Рис.*7-1:* Рекурсивное дерево с *Pygame*.

Нарисуем треугольник Серпинского. Это немного сложнее, чем рекурсивное дерево. Нам придется вызвать функцию рекурсивно три раза. Вот код:

prog04.py

```
import pygame, math, random
import time, sys
width, height = 800, 800
result = pygame.init()
if result[1] > 0:
    print('Error initializing Pygame : ' + str(result[1]))
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
window = pygame.display.set_mode((width, height))
pygame.display.set_caption('Fibonacci Tree')
```

```
screen = pygame.display.get_surface()
shift = 20
A_x = 0 + shift
A_y = 649 + shift
B_x = 750 + shift
B_y = 649 + shift
C_x = 375 + shift
C_y = 0 + shift
RGB = [(0, 0, 255), (0, 255, 0), (255, 0, 0)]
def draw_triangle(A_x, A_y, B_x, B_y, C_x, C_y, i):
    pygame.draw.line(screen, RGB[i%3], (A_x, A_y), (B_x, B_y), 1)
    pygame.draw.line(screen, RGB[i%3], (C_x, C_y), (B_x, B_y), 1)
    pygame.draw.line(screen, RGB[i%3], (A_x, A_y), (C_x, C_y), 1)
    pygame.display.flip()
def draw_fractal(A_x, A_y, B_x, B_y, C_x, C_y, depth):
    if depth > 0:
        draw_fractal((A_x), (A_y), (A_x+B_x)/2, (A_y+B_y)/2,
                     (A_x+C_x)/2, (A_y+C_y)/2, depth-1)
        draw_fractal((B_x), (B_y), (A_x+B_x)/2, (A_y+B_y)/2,
                     (B_x+C_x)/2, (B_y+C_y)/2, depth-1)
        draw_fractal((C_x), (C_y), (C_x+B_x)/2, (C_y+B_y)/2,
                     (A_x+C_x)/2, (A_y+C_y)/2, depth-1)

        draw_triangle((A_x), (A_y), (A_x+B_x)/2,
                      (A_y+B_y)/2, (A_x+C_x)/2, (A_y+C_y)/2, depth)
        draw_triangle((B_x), (B_y), (A_x+B_x)/2,
                      (A_y+B_y)/2, (B_x+C_x)/2, (B_y+C_y)/2, depth)
        draw_triangle((C_x), (C_y), (C_x+B_x)/2,
                      (C_y+B_y)/2, (A_x+C_x)/2, (A_y+C_y)/2, depth)
draw_fractal(A_x, A_y, B_x, B_y, C_x, C_y, 1)
pygame.display.flip()
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
sys.exit(0)
```

Как мы видим, у нас есть отдельная пользовательская функция, которая рисует фактическую фигуру (**draw_triangle()**). Мы вызываем это три раза в рекурсивной функции, а затем трижды вызываем рекурсивную функцию. В основном разделе мы вызываем функцию с глубиной 1. Она выдает следующий результат:

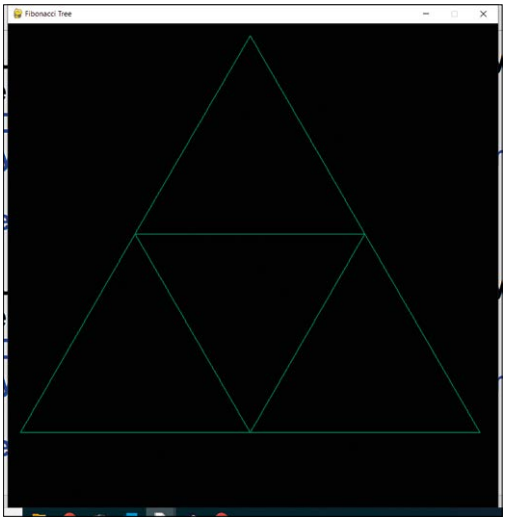


Рис.*7-2:* Треугольник Серпинского глубины *1*.

Давайте изменим вызов в основном разделе для лучшего понимания:

```
draw_fractal(A_x, A_y, B_x, B_y, C_x, C_y, 2)
```

Вывод следующий:

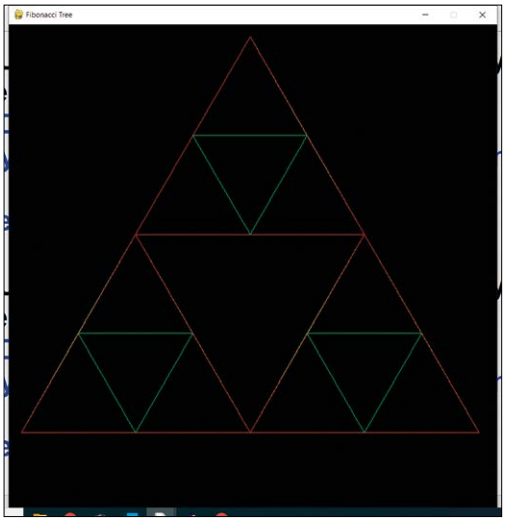


Рис.*7-3:* Треугольник Серпинского глубины *2*.

Пойдем дальше и увеличим глубину до 5 и посмотрим результат:

```
draw_fractal(A_x, A_y, B_x, B_y, C_x, C_y, 5)
```

Ниже приводится вывод:

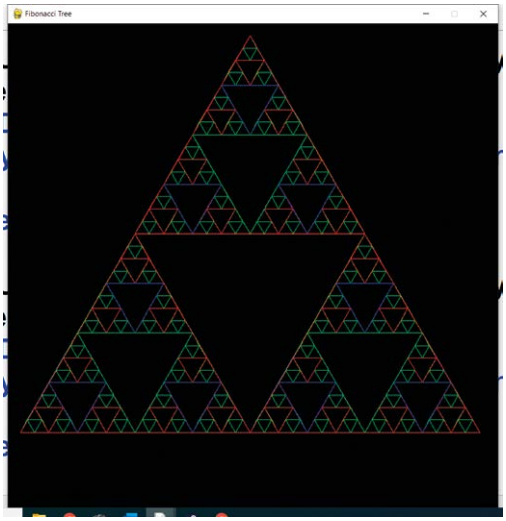


Рис.*7-4:* Треугольник Серпинского глубины *5*.

Давайте увеличим глубину до 10:

```
draw_fractal(A_x, A_y, B_x, B_y, C_x, C_y, 10)
```

Вывод следующий:

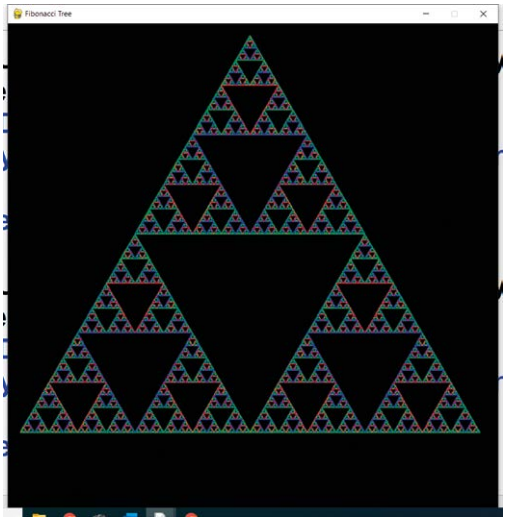


Рис.7-5: Треугольник Серпинского глубины 10.

7.3 Треугольник Серпинского от Chaos Game

Мы можем создать Треугольник Серпинского, используя Игру Хаоса. Фрактал создается путем итеративного создания последовательности точек. Мы выбираем начальную случайную точку ((400, 400) в этом примере). Каждой точке последовательности присваивается половина расстояния между предыдущей точкой и одной из вершин многоугольника, выбранной случайно на каждой итерации. Когда мы повторяем этот итерационный процесс большое количество раз, выбирая случайным образом вершину треугольника на каждой итерации, чаще всего (но не всегда) получается треугольник Серпинского. Для получения лучших результатов мы можем отказаться от построения нескольких начальных точек. Вот код:

prog05.py

```
import pygame, math, random
import time, sys
width, height = 800, 800
result = pygame.init()
if result[1] > 0:
    print('Error initializing Pygame : ' + str(result[1]))
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
surface = pygame.display.set_mode((width, height))
pygame.display.set_caption('Fibonacci Tree')
screen = pygame.display.get_surface()
def draw_pixel(x, y):
    surface.fill(pygame.Color(0, 255, 0), ((x, y), (1, 1)))
    pygame.display.flip()
shift = 20
A_x = 0 + shift
A_y = 649 + shift
B_x = 750 + shift
B_y = 649 + shift
C_x = 375 + shift
C_y = 0 + shift
x, y = 400, 400
for i in range( 1, 50000):
    choice = random.randint(1, 3)
    if choice == 1:
        x = (x+A_x)/2
        y = (y+A_y)/2
    elif choice == 2:
        x = (x+B_x)/2
        y = (y+B_y)/2
    elif choice == 3:
        x = (x+C_x)/2
        y = (y+C_y)/2
    if i < 10:
        pass
```

```
    else:
        draw_pixel(x, y)
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
pygame.quit()
sys.exit(0)
```

Теперь мы знакомы с большей частью кода. Логика выбора случайной вершины треугольника, а затем выбора новой точки на полпути между текущей точкой и выбранной вершиной находится внутри цикла **for**.
Это результат:

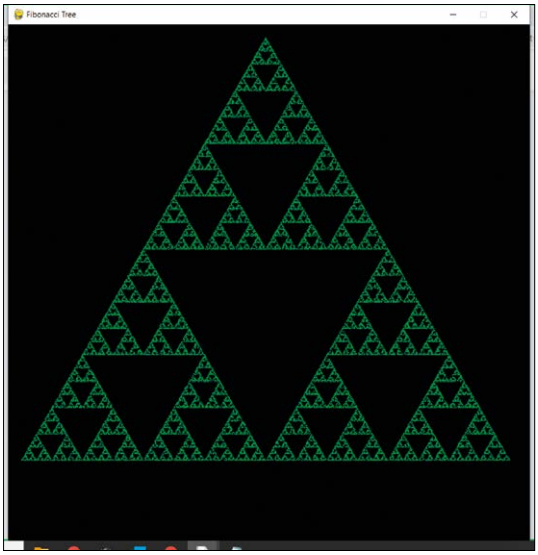


Рис.*7-6:* Треугольник Серпинского с игрой хаоса

7.4 Простая анимация с помощью Pygame

Давайте создадим простую анимацию прыгающего мяча с помощью библиотеки Pygame. На этой демонстрации мы узнаем много нового. Давайте напишем код шаг за шагом. Создайте новый файл и сохраните его как **prog06.py**. Импортируем необходимые библиотеки:

```
import pygame
from pygame.locals import *
```

Давайте инициализируем Pygame и создадим объект для экрана:

```
size = 720, 480
width, height = size
```

```
result = pygame.init()
if result[1] > 0:
    print('Error initializing Pygame : ' + str(result[1]))
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
screen = pygame.display.set_mode((size))
```

Весь этот блок кода нам уже знаком, поэтому я вам его объяснять не буду. Давайте посмотрим новый код. Добавьте в файл следующий код:

```
BLUE = (150, 150, 255)
RED = (255, 0, 0)
ball = pygame.image.load('ball_transparent.gif')
rect = ball.get_rect()
speed = [2, 2]
```

Мы определяем кортежи для синего и красного цветов. Затем мы используем подпрограмму **ruleimage.load()** для загрузки изображения в переменную. Мы можем получить прямоугольник размером с изображение с помощью подпрограммы **get_rect()**. Наконец, мы определяем список, в котором хранятся значения скорости по обеим осям. Напишем код цикла:

```
running = True
while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

    rect = rect.move(speed)
    if rect.left < 0 or rect.right > width:
        speed[0] = -speed[0]
    if rect.top < 0 or rect.bottom > height:
        speed[1] = -speed[1]

    screen.fill(BLUE)
    screen.blit(ball, rect)
    pygame.time.Clock().tick(240)
    pygame.display.flip()
```

Мы знакомы с циклом событий. Итак, давайте обсудим следующую часть. Мы используем процедуру **move()** для перемещения объекта. Нам нужно передать ему переменную, в которой хранятся значения скорости. В операторах **if** мы проверяем, касается ли прямоугольник, охватывающий мяч, границ. Если это так, мы обращаем скорость мяча на обратную.

Затем мы заполняем экран синим цветом. Затем мы используем функцию **routerblit()**, чтобы показать мяч. Функция **errortime.Clock().tick(240)** используется для определения частоты кадров анимации.

Наконец, мы используем функцию **Flip()**, чтобы показать все на экране. Мы завершаем все с помощью процедуры **quit()** следующим образом:

```
pygame.quit()
```

Весь код выглядит следующим образом:

prog06.py

```
import pygame
from pygame.locals import *

size = 720, 480
width, height = size
result = pygame.init()
if result[1] > 0:
    print('Error initializing Pygame : ' + str(result[1]))
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
screen = pygame.display.set_mode((size))


BLUE = (150, 150, 255)
RED = (255, 0, 0)
ball = pygame.image.load('ball_transparent.gif')
rect = ball.get_rect()
speed = [2, 2]

running = True
while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

    rect = rect.move(speed)
    if rect.left < 0 or rect.right > width:
        speed[0] = -speed[0]
    if rect.top < 0 or rect.bottom > height:
        speed[1] = -speed[1]

    screen.fill(BLUE)
    screen.blit(ball, rect)
    pygame.time.Clock().tick(240)
    pygame.display.flip()

pygame.quit()
```

Он создает замечательную анимацию прыгающего мяча. Ниже приведен скриншот анимации:

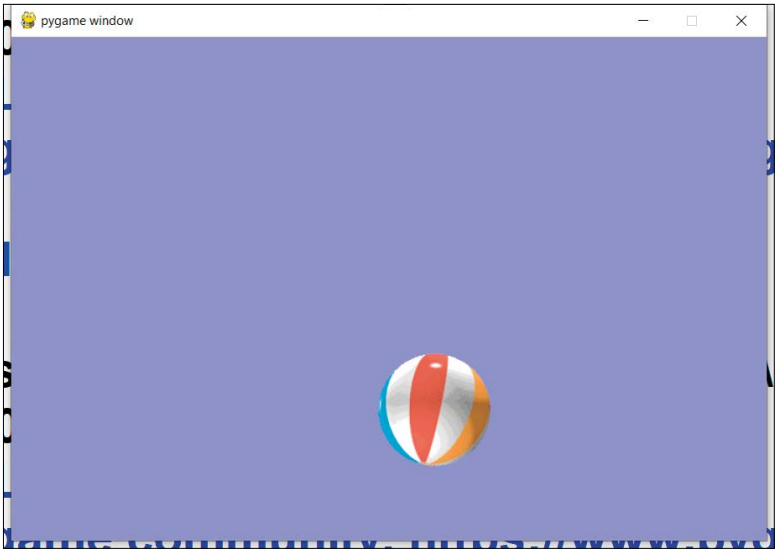


Рис.*7-7:* Прыгающий мяч.

Мы можем написать сложную программу, используя ту же концепцию. У нас может быть несколько прыгающих мячей с разной скоростью. На этот раз мы создадим объект для мяча. Давайте посмотрим код шаг за шагом:

```
import pygame
import random
```

Он импортирует все библиотеки. Давайте определим цвета:

```
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
colors = [WHITE, RED, BLUE, GREEN]
```

Определим размер мяча и разрешение экрана,

```
size = 720, 480
width, height = size
BALL_SIZE = 25
```

Давайте определим класс для шаров,

```
class Ball:

    def __init__(self):
        self.x = 0
        self.y = 0
        self.change_x = 0
        self.change_y = 0
        self.color = colors[random.randint(0, 3)]
```

Мы определяем координаты, скорость в обоих измерениях и цвет (который является случайным). Давайте определим функцию для создания мяча и назначим случайные значения положения и скорости:

```
def make_ball():
    ball = Ball()
    ball.x = random.randrange(BALL_SIZE, width - BALL_SIZE)
    ball.y = random.randrange(BALL_SIZE, height - BALL_SIZE)
    ball.change_x = random.randint(1, 3)
    ball.change_y = random.randint(1, 3)
    return ball
```

Давайте инициализируем pygame:

```
result = pygame.init()
if result[1] > 0:
    print('Error initializing Pygame : ' + str(result[1]))
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
screen = pygame.display.set_mode((size))
pygame.display.set_caption("Bouncing Balls")
```

Определим fps (кадров в секунду):

```
fps = 30
```

Давайте определим список для хранения объектов-шаров:

```
ball_list = []
```

Давайте создадим шар и добавим его в список:

```
ball = make_ball()
ball_list.append(ball)
```

Напишем основной цикл:

```
running = False
while not running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = True
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_SPACE:
                if len(ball_list) < 5:
                    ball = make_ball()
                    ball_list.append(ball)
                else:
                    print("Screen already has five balls!")
            elif event.key == pygame.K_BACKSPACE:
                if len(ball_list) == 0:
                    print("Ball list is empty!")
                else:
                    ball_list.pop()
            elif event.key == pygame.K_q:
                if fps == 30:
                    print("Minimum FPS")
                else:
                    fps = fps - 30
                    print("Current FPS = " + str(fps))
            elif event.key == pygame.K_e:
                if fps == 300:
                    print("Maximum FPS")
                else:
                    fps = fps + 30
                    print("Current FPS = " + str(fps))
            elif event.key == pygame.K_r:
                for ball in ball_list:
                    ball.change_x = random.randint(-2, 3)
                    ball.change_y = random.randint(-2, 3)
                    ball.color = colors[random.randint(0, 3)]
```

Это большой блок и выглядит устрашающе. Однако это просто. Если мы нажмем пробел, он создаст новый шар и добавит его в список. Если шаров уже пять, новый шар не создается. Если мы нажмем клавишу Backspace, если список шаров не пуст, он удалит последний созданный шар. Если мы нажмем клавишу **E**, это увеличит количество кадров в секунду на 30 (что, в свою очередь, увеличит скорость анимации). Если скорость = 300, то она не увеличивается. Аналогично, нажатие **q** уменьшает скорость на 30. Если скорость уже равна 30, она не уменьшается дальше. Нажатие клавиши **R** случайным образом меняет скорость и цвет шаров. Мы еще не закончили с этим блоком кода:

```
    for ball in ball_list:
        ball.x = ball.x + ball.change_x
        ball.y = ball.y + ball.change_y

        if ball.y > height - BALL_SIZE or ball.y < BALL_SIZE:
            ball.change_y = -ball.change_y
        if ball.x > width - BALL_SIZE or ball.x < BALL_SIZE:
            ball.change_x = -ball.change_x
```

Здесь, в этом блоке кода, мы меняем положение шаров и проверяем, не сталкиваются ли они с краями экрана. Если да, то меняем направление. Наконец, мы рисуем каждый кадр:

```
    screen.fill(BLACK)
    for ball in ball_list:
        pygame.draw.circle(screen, ball.color,
                           [ball.x, ball.y], BALL_SIZE)
    pygame.time.Clock().tick(fps)
    pygame.display.flip()
```

Мы заполняем весь экран черным. Затем мы рисуем текущие позиции шаров. Наконец, мы отображаем все с помощью подпрограммы **Flip()**.

```
pygame.quit()
```

В конце мы завершаем процедуру **quit()**. Запустите программу. Мы можем использовать клавиши **Q**, **E** и **R**. Мы также можем использовать пробел и клавишу **Backspace**. Соберем всю программу воедино:

prog07.py

```
import pygame
import random

BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
colors = [WHITE, RED, BLUE, GREEN]

size = 720, 480
width, height = size
BALL_SIZE = 25

class Ball:

    def __init__(self):
```

```
        self.x = 0
        self.y = 0
        self.change_x = 0
        self.change_y = 0
        self.color = colors[random.randint(0, 3)]


def make_ball():
    ball = Ball()
    ball.x = random.randrange(BALL_SIZE, width - BALL_SIZE)
    ball.y = random.randrange(BALL_SIZE, height - BALL_SIZE)
    ball.change_x = random.randint(1, 3)
    ball.change_y = random.randint(1, 3)
    return ball

result = pygame.init()
if result[1] > 0:
    print('Error initializing Pygame : ' + str(result[1]))
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
screen = pygame.display.set_mode((size))
pygame.display.set_caption("Bouncing Balls")

ball_list = []
ball = make_ball()
ball_list.append(ball)
fps = 30

running = False
while not running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = True
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_SPACE:
                if len(ball_list) < 5:
                    ball = make_ball()
                    ball_list.append(ball)
                else:
                    print("Screen already has five balls!")
            elif event.key == pygame.K_BACKSPACE:
                if len(ball_list) == 0:
                    print("Ball list is empty!")
                else:
                    ball_list.pop()
```

```
            elif event.key == pygame.K_q:
                if fps == 30:
                    print("Minimum FPS")
                else:
                    fps = fps - 30
                    print("Current FPS = " + str(fps))
            elif event.key == pygame.K_e:
                if fps == 300:
                    print("Maximum FPS")
                else:
                    fps = fps + 30
                    print("Current FPS = " + str(fps))
            elif event.key == pygame.K_r:
                for ball in ball_list:
                    ball.change_x = random.randint(-2, 3)
                    ball.change_y = random.randint(-2, 3)
                    ball.color = colors[random.randint(0, 3)]

    for ball in ball_list:
        ball.x = ball.x + ball.change_x
        ball.y = ball.y + ball.change_y

        if ball.y > height - BALL_SIZE or ball.y < BALL_SIZE:
            ball.change_y = -ball.change_y
        if ball.x > width - BALL_SIZE or ball.x < BALL_SIZE:
            ball.change_x = -ball.change_x

    screen.fill(BLACK)
    for ball in ball_list:
        pygame.draw.circle(screen, ball.color,
                           [ball.x, ball.y], BALL_SIZE)
    pygame.time.Clock().tick(fps)
    pygame.display.flip()

pygame.quit()
```

Ниже приводится вывод:

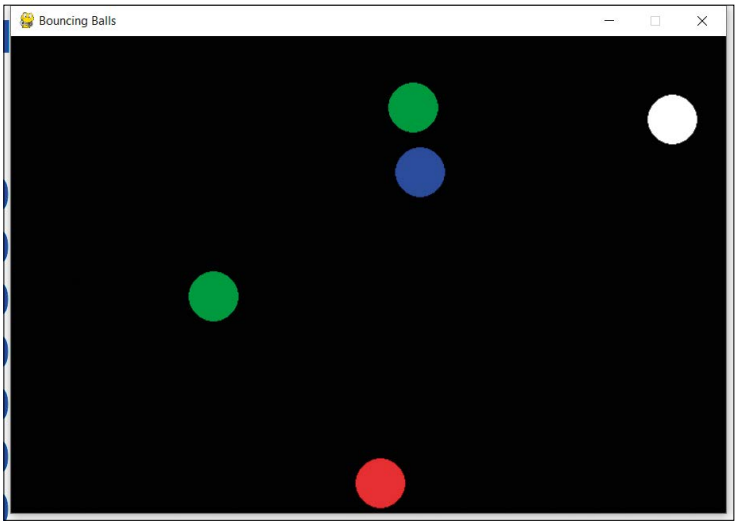


Рис.*7-8:* Множество прыгающих мячей.

Мы можем изменить эту программу, чтобы вместо создания кругов отображалось изображение. Попробуйте это в качестве упражнения.

7.5 Игра Snake (Змейка)

Изучив хитрости профессии, начнем работу со сложным игровым проектом. В этом проекте мы напишем код классической игры в змейку. Вначале есть змея, состоящая из двух квадратов размером 10x10 пикселей. В левом верхнем углу мы видим сложность и количество очков. Порождается случайный квадрат 10x10, и когда змея его съедает, он поглощается змеей, и змея увеличивается в размерах. Как только квадрат потребляется, в другом случайном месте появляется другой квадрат. Змея находится в состоянии вечного движения. Мы можем переместить её с помощью клавиш WSAD, но не можем напрямую изменить его направление. Давайте посмотрим код поблочно:

```
import pygame, sys, time, random
```

Он импортирует все необходимые библиотеки. Зададим начальную сложность:

```
difficulty = 5
```

Давайте инициализируем pygame и создадим окно:

```
window_size_x = 720
window_size_y = 480
result = pygame.init()
if result[1] > 0:
    print('Error initializing Pygame : ' + str(result[1]))
```

```
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
pygame.display.set_caption('Snake')
game_window = pygame.display.set_mode((window_size_x,
                                       window_size_y))
```

Давайте определим несколько цветов:

```
black = pygame.Color(0, 0, 0)
white = pygame.Color(255, 255, 255)
green = pygame.Color(0, 255, 0)
```

Определим начальные сегменты змеи и направление:

```
snake_pos = [100, 100]
snake_body = [[100, 100],
              [100-10, 100]]
direction = 'DOWN'
change_to = direction
```

и определим случайное положение для квадрата с едой:

```
food_pos = [random.randrange(1, (window_size_x//10)) * 10,
            random.randrange(1, (window_size_y//10)) * 10]
food_spawn = True
```

Давайте определим несколько переменных, связанных с игрой:

```
score = 0
difficulty_counter = 0
difficulty = 5
```

Давайте определим подпрограмму (функцию) для отображения счета:

```
def show_score(choice, color, font, size):
    score_font = pygame.font.SysFont(font, size)
    score_surface = score_font.render('Score : ' + str(score) +
                                      ' Difficulty : ' + str(difficulty),
                                      True, color)
    score_rect = score_surface.get_rect()
    if choice == 1:
        score_rect.midtop = (window_size_x/10 + 30, 15)
    else:
        score_rect.midtop = (window_size_x/2, window_size_y/1.25)
    game_window.blit(score_surface, score_rect)
```

В нем предусмотрена возможность отображения счета либо в левом верхнем углу мелким шрифтом, либо посередине крупным шрифтом. Мы отображаем счет слева вверху при обычном выполнении игры и посередине крупным шрифтом, когда игра окончена.

Давайте определим функцию, которая будет вызываться по окончании игры:

```
def game_over():
    game_over_font = pygame.font.SysFont('Times New Roman', 90)
    game_over_surface = game_over_font.render('Game Over',
                                              True, green)
    game_over_rect = game_over_surface.get_rect()
    game_over_rect.midtop = (window_size_x/2, window_size_y/4)
    game_window.fill(black)
    game_window.blit(game_over_surface, game_over_rect)
    show_score(0, green, 'Times New Roman', 20)
    pygame.display.flip()
    time.sleep(3)
    pygame.quit()
    sys.exit()
```

Эта функция вызывает функцию **show_score()**, которую мы определили ранее. Давайте определим цикл основной игры:

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit(0)
```

Мы уже знакомы с этим кодом. Все приведенные ниже сегменты кода являются частью этого основного игрового цикла, поэтому будьте осторожны с отступами, если вы печатаете вручную блок за блоком. Для вашего удобства я перечислю весь код после объяснения того, как работают блоки.

Напишем логику захвата нажатия клавиш:

```
        elif event.type == pygame.KEYDOWN:

            if event.key == pygame.K_UP or event.key == ord('w'):
                change_to = 'UP'
            if event.key == pygame.K_DOWN or event.key == ord('s'):
                change_to = 'DOWN'
            if event.key == pygame.K_LEFT or event.key == ord('a'):
                change_to = 'LEFT'
            if event.key == pygame.K_RIGHT or event.key == ord('d'):
                change_to = 'RIGHT'
```

```
            if event.key == pygame.K_ESCAPE:
                pygame.event.post(pygame.event.Event(pygame.QUIT))
```

В зависимости от нажатия клавиши мы устанавливаем переменную, которая сообщает нам, в каком направлении нам следует изменить змею. Теперь давайте посмотрим блок кода, который устанавливает переменную для направления:

```
    if change_to == 'UP' and direction != 'DOWN':
        direction = 'UP'
    if change_to == 'DOWN' and direction != 'UP':
        direction = 'DOWN'
    if change_to == 'LEFT' and direction != 'RIGHT':
        direction = 'LEFT'
    if change_to == 'RIGHT' and direction != 'LEFT':
        direction = 'RIGHT'
```

Мы можем видеть, движется ли змея в направлении UP - ВВЕРХ, но мы не можем напрямую установить ее в направлении DOWN - ВНИЗ. Это справедливо и для всех остальных направлений. Таким образом мы убедимся, что змея не меняет направления.

Теперь давайте установим положение змеи:

```
    if direction == 'UP':
        snake_pos[1] -= 10
    if direction == 'DOWN':
        snake_pos[1] += 10
    if direction == 'LEFT':
        snake_pos[0] -= 10
    if direction == 'RIGHT':
        snake_pos[0] += 10
```

Напишем блок, проверяющий, съела ли змея еду:

```
    snake_body.insert(0, list(snake_pos))
    if snake_pos[0] == food_pos[0] and snake_pos[1] == food_pos[1]:
        score = score + 1
        difficulty_counter = difficulty_counter + 1
        print(difficulty_counter)
        if difficulty_counter == 10:
            difficulty_counter = 0
            difficulty = difficulty + 5
```

Увеличиваем счет и длину тела змеи. Также мы увеличиваем сложность (это FPS, просто игра ускоряется) каждый раз на 5 после увеличения счета на 10. Логика появления еды следующая:

```
        food_spawn = False
    else:
        snake_body.pop()

    if not food_spawn:
        food_pos = [random.randrange(1, (window_size_x//10)) * 10,
                    random.randrange(1, (window_size_y//10)) * 10]
    food_spawn = True
```

Нарисуем еду и тело змеи,

```
    game_window.fill(black)
    for pos in snake_body:
        pygame.draw.rect(game_window, green, pygame.Rect(pos[0],
                                                         pos[1], 10, 10))

    pygame.draw.rect(game_window, white, pygame.Rect(food_pos[0],
                                                     food_pos[1], 10, 10))
```

Давайте вызовем функцию `game_over()`, если змея касается границ или собственного тела.

```
    if snake_pos[0] < 0 or snake_pos[0] > window_size_x-10:
        game_over()
    if snake_pos[1] < 0 or snake_pos[1] > window_size_y-10:
        game_over()

    for block in snake_body[1:]:
        if snake_pos[0] == block[0] and snake_pos[1] == block[1]:
            game_over()
```

Давайте покажем счет в левом верхнем углу и обновим отображение и FPS в игре.

```
    show_score(1, white, 'Times New Roman', 20)
    pygame.display.update()
    pygame.time.Clock().tick(difficulty)
```

Давайте объединим это в один файл следующим образом:

Snake_Game.py

```
import pygame, sys, time, random

difficulty = 5
window_size_x = 720
window_size_y = 480
result = pygame.init()
if result[1] > 0:
```

```
    print('Error initializing Pygame : ' + str(result[1]))
    sys.exit(1)
else:
    print('Pygame initialized successfully!')
pygame.display.set_caption('Snake')
game_window = pygame.display.set_mode((window_size_x,
                                       window_size_y))

black = pygame.Color(0, 0, 0)
white = pygame.Color(255, 255, 255)
green = pygame.Color(0, 255, 0)

snake_pos = [100, 100]
snake_body = [[100, 100],
              [100-10, 100]]
direction = 'DOWN'
change_to = direction

food_pos = [random.randrange(1, (window_size_x//10)) * 10,
            random.randrange(1, (window_size_y//10)) * 10]
food_spawn = True

score = 0
difficulty_counter = 0
difficulty = 5

def game_over():
    game_over_font = pygame.font.SysFont('Times New Roman', 90)
    game_over_surface = game_over_font.render('Game Over',
                                              True, green)
    game_over_rect = game_over_surface.get_rect()
    game_over_rect.midtop = (window_size_x/2, window_size_y/4)
    game_window.fill(black)
    game_window.blit(game_over_surface, game_over_rect)
    show_score(0, green, 'Times New Roman', 20)
    pygame.display.flip()
    time.sleep(3)
    pygame.quit()
    sys.exit()

def show_score(choice, color, font, size):
    score_font = pygame.font.SysFont(font, size)
    score_surface = score_font.render('Score : ' + str(score) +
                                      ' Difficulty : ' + str(difficulty),
                                      True, color)
    score_rect = score_surface.get_rect()
```

```
    if choice == 1:
        score_rect.midtop = (window_size_x/10 + 30, 15)
    else:
        score_rect.midtop = (window_size_x/2, window_size_y/1.25)
    game_window.blit(score_surface, score_rect)


while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit(0)

        elif event.type == pygame.KEYDOWN:

            if event.key == pygame.K_UP or event.key == ord('w'):
                change_to = 'UP'
            if event.key == pygame.K_DOWN or event.key == ord('s'):
                change_to = 'DOWN'
            if event.key == pygame.K_LEFT or event.key == ord('a'):
                change_to = 'LEFT'
            if event.key == pygame.K_RIGHT or event.key == ord('d'):
                change_to = 'RIGHT'

            if event.key == pygame.K_ESCAPE:
                pygame.event.post(pygame.event.Event(pygame.QUIT))

    if change_to == 'UP' and direction != 'DOWN':
        direction = 'UP'
    if change_to == 'DOWN' and direction != 'UP':
        direction = 'DOWN'
    if change_to == 'LEFT' and direction != 'RIGHT':
        direction = 'LEFT'
    if change_to == 'RIGHT' and direction != 'LEFT':
        direction = 'RIGHT'

    if direction == 'UP':
        snake_pos[1] -= 10
    if direction == 'DOWN':
        snake_pos[1] += 10
    if direction == 'LEFT':
        snake_pos[0] -= 10
    if direction == 'RIGHT':
        snake_pos[0] += 10

    snake_body.insert(0, list(snake_pos))
```

```
    if snake_pos[0] == food_pos[0] and snake_pos[1] == food_pos[1]:
        score = score + 1
        difficulty_counter = difficulty_counter + 1
        print(difficulty_counter)
        if difficulty_counter == 10:
            difficulty_counter = 0
            difficulty = difficulty + 5

        food_spawn = False
    else:
        snake_body.pop()

    if not food_spawn:
        food_pos = [random.randrange(1, (window_size_x//10)) * 10,
                    random.randrange(1, (window_size_y//10)) * 10]
    food_spawn = True

    game_window.fill(black)
    for pos in snake_body:
        pygame.draw.rect(game_window, green, pygame.Rect(pos[0],
                                                         pos[1], 10, 10))

    pygame.draw.rect(game_window, white, pygame.Rect(food_pos[0],
                                                     food_pos[1], 10, 10))

    if snake_pos[0] < 0 or snake_pos[0] > window_size_x-10:
        game_over()
    if snake_pos[1] < 0 or snake_pos[1] > window_size_y-10:
        game_over()

    for block in snake_body[1:]:
        if snake_pos[0] == block[0] and snake_pos[1] == block[1]:
            game_over()

    show_score(1, white, 'Times New Roman', 20)
    pygame.display.update()
    pygame.time.Clock().tick(difficulty)
```

Давайте выполним этот код. Ниже приведен скриншот игры:

Рис/7-9: Игра «Змейка» в действии

Краткое содержание

В этой главе мы подробно изучили библиотеку Pygame для графики, игр и анимации. Нам удобно создавать небольшую графику, анимацию и игры с помощью Pygame.

В следующей главе мы научимся работать с файлами различных форматов. Мы научимся читать и изменять файлы программным способом.

Глава 8 • Работа с файлами

В предыдущей главе мы узнали, как работать с библиотекой Pygame. Мы подготовили демонстрации рекурсии, игр, анимации и симуляций.

В этой главе мы рассмотрим важную тему работы с файлами,где мы храним данные в файлах различных форматов. Мы научимся работать с файлами различных форматов и подробно изучим следующие темы:

- Обработка открытого текстового файла
- CSV-файлы
- Работа с электронными таблицами

Это подробная глава, в которой использовано значительное количество практических и продвинутых концепций. После прочтения этой главы нам будет комфортно с MISSING

8.1 Обработка текстового файла

В Python предусмотрена возможность чтения данных непосредственно из текстового файла. Для этого нам не нужно импортировать библиотеку. Давайте посмотрим, как используется функция **open()** для открытия файла и присвоения его файловому объекту. Создайте новый блокнот Jupyter для демонстраций в этой главе. Напишите следующий код:

```
file1 = open('test.txt', mode='rt', encoding='utf-8')
```

Когда мы выполняем этот код, он возвращает ошибку, поскольку файл не существует. Давайте изменим код, чтобы справиться с этим:

```
try:
    file1 = open('test.txt', mode='rt', encoding='utf-8')
except Exception as e:
    print(e)
It prints the following message on execution:
[Errno 2] No such file or directory: 'test.txt'
```

Давайте подробно разберемся в использовании процедуры **open()**. Сначала давайте создадим пустой файл с именем **test.txt** в каталоге, в котором мы сохранили блокнот. Запустите код еще раз. Он будет работать без каких-либо проблем. Давайте добавим больше кода.

```
try:
    file1 = open('test.txt', mode='rt', encoding='utf-8')
except Exception as e:
    print(e)
finally:
    file1.close()
```

Давайте разберемся в значении аргументов, передаваемых в параметр функции **open()**. Первым аргументом, очевидно, является имя файла, с которым нам нужно работать. Второй — это режим открытия файла. Третий — кодирование. Существуют различные режимы, в которых мы можем открыть файл, и ниже приводится их список:

Режим	Значение
r	Открывает файл для чтения. (по умолчанию)
w	Открывает файл для записи. Создает новый файл, если он не существует, или усекает файл, если он существует.
x	Открывает файл для эксклюзивного создания.
a	Открывает файл для добавления в конец файла без его усечения. Создает новый файл, если он не существует.
t	Открывается в текстовом режиме. (по умолчанию)
b	Открывается в двоичном режиме.
+	Открывает файл для обновления (чтение и запись)

Как мы видим, по умолчанию файл открывается в режиме чтения и в текстовом режиме. То, что мы писали ранее,эквивалентно следующему коду:

```
try:
    file1 = open('test.txt')
except Exception as e:
    print(e)
finally:
    file1.close()
```

Мы также можем иметь комбинации нескольких режимов. В следующей таблице приведены значениявсе комбинации, с которыми я работал до сих пор:

Режим	Значение
rb	Открывает файл только для чтения в двоичном формате. Указатель файла помещается в начало -создание файла. Это режим "по умолчанию".
r+	Открывает файл как для чтения, так и для записи. Указатель файла, помещенный в начало файла.
rb+	Открывает файл для чтения и записи в двоичном формате. Указатель файла, помещенный в начало файла.
wb	Открывает файл для записи только в двоичном формате. Перезаписывает файл, если файл существует. Если файл не существует, создается новый файл.
w+	Открывает файл как для записи, так и для чтения. Перезаписывает существующий файл, если файл существует.Если файл не существует, создается новый файл для чтения и записи.

wb+	Открывает файл для записи и чтения в двоичном формате. Перезаписывает существующий файл, если файл существует. Если файл не существует, создается новый файл для чтения и записи.
ab	Открывает файл для добавления в двоичном формате. Указатель файла находится в конце файла, если файл существует. То есть файл находится в режиме добавления. Если файл не существует, создается новый файл для записи.
a+	Открывает файл как для добавления, так и для чтения. Указатель файла находится в конце файла, если файл существует. Файл открывается в режиме добавления. Если файл не существует, создается новый файл для чтения и записи.
ab+	Открывает файл для добавления и чтения в двоичном формате. Указатель файла находится в конце файла, если файл существует. Файл открывается в режиме добавления. Если файл не существует, создается новый файл для чтения и записи.

Давайте добавим текст (вручную) в созданный нами файл, а затем изменим код:

```
try:
    file1 = open('test.txt', mode='rt', encoding='utf-8')
    for each in file1:
        print (each)
except Exception as e:
    print(e)
finally:
    file1.close()
```

Он будет читать содержимое файла построчно и распечатывать его. Мы также можем прочитать содержимое файла в одной строке кода, как показано ниже:

```
try:
    file1 = open('test.txt', mode='rt', encoding='utf-8')
    print(file1.read())
except Exception as e:
    print(e)
finally:
    file1.close()
```

Мы можем прочитать определенное количество символов, передав это число в качестве аргумента функции **read()**.

```
try:
    file1 = open('test.txt', mode='rt', encoding='utf-8')
    print(file1.read(20))
except Exception as e:
    print(e)
finally:
    file1.close()
```

Давайте посмотрим, как открыть файл в режиме записи и записать в него данные.

```
try:
    file1 = open('test1.txt', mode='w', encoding='utf-8')
    file1.write('That fought with us upon Saint Crispin's day.')
except Exception as e:
    print(e)
finally:
    file1.close()
```

Поскольку мы знаем, что режим записи создает новый файл или усекает существующий файл с тем же именем, приведенная выше программа создает указанный файл и добавляет текст, упомянутый в подпрограмме **write()**. Если мы запустим приведенный выше код несколько раз, он просто обрежет старыйфайл и создает новый файл с тем же именем и содержимым.Давайте посмотрим, как открыть файл в режиме добавления. Если мы используемэтот режим, он создает новый файл, если указанный файл не существует. Если файлсуществует, он добавит к нему заданную строку.
Взгляните на следующий код:

```
try:
    file1 = open('test2.txt', mode='a', encoding='utf-8')
    file1.write('That fought with us upon Saint Crispin's day.')
except Exception as e:
    print(e)
finally:
    file1.close()
```

Запустите этот код несколько раз. Вы найдете указанный файл с несколькими строками одной и той же строки, указанной в коде.

Мы можем прочитать определенное количество символов, передав это число в качестве аргумента функции **read()**.

```
try:
    with open("test.txt", "w") as f:
        f.write("Hello, World!")
except Exception as e:
    print(e)
finally:
    file1.close()
We can delete a file using the following code:
import os
if os.path.exists("test.txt"):
  os.remove("test.txt")
else:
  print("The file does not exist")
```

8.2 CSV-файлы

Давайте разберемся, как работать с файлами CSV. Мы можем начать с понимания того, что такое файлы CSV. CSV означает значение, разделенное запятыми. Это означает текстовый файл с разделителями, в котором в качестве разделителя значений используется запятая. Каждая строка файла CSV представляет собой запись данных. До создания современных реляционных баз данных этот формат использовался (и до сих пор используется) для хранения записей в табличном формате. Во многих файлах CSV первая строка является строкой заголовка, в которой хранятся имена полей. Файлы CSV тесно связаны с форматом файлов DSV (файлы, разделенные разделителями), где мы используем разделители, такие как двоеточие и пробел, для разделения полей. CSV — это подгруппа DSV.

Теперь давайте вручную создадим CSV-файл в том же каталоге, где мы запускаем нашу программу. Давайте сохраним его с именем `test.csv`. Добавьте следующие или аналогичные данные в CSV-файл:

```
Name,Salary
Ashwin,100000
Thor,200000
Jane,300000
Cpt America,30000
Iron Man,4000000
```

Как мы видим, это данные, связанные с расчетом заработной платы. Вы можете иметь любые данные по вашему выбору. Я предпочитаю делать это простым для новичков.

В этом разделе мы научимся читать данные из этого и других файлов CSV. Для этого нам нужно импортировать встроенную библиотеку CSV. Эта библиотека поставляется с Python как часть философии включения батарей, и нам не нужно устанавливать ее отдельно. Начнем с программной части:

```
import csv
```

Откроем файл в текстовом режиме и в режиме чтения:

```
file = open('test.csv')
```

Мы должны рассматривать этот файл как CSV (поскольку, хотя это текстовый файл, мы знаем, что в нем есть данные CSV). Давай сделаем это:

```
csvreader = csv.reader(file)
```

Теперь у нас есть объект, который обрабатывает файл как CSV. Прочитаем первую строку. Первая строка — это строка заголовка, содержащая имена столбцов:

```
header = []
header = next(csvreader)
header
```

It prints the following in the output area of the Jupyter notebook,

```
['Name', 'Salary']
```

Now, we have the column names, let's extract and print the data. Let's define an empty list:

```
rows = []
```

Let's extract the data and print it. After every row, we are printing a visual marker to separate the rows. Simultaneously, we are upending the list variable **rows** with a row from the CSV. Let' see:

```
for row in csvreader:
    for data in row:
        print(data)
    print('---')
    rows.append(row)
```

Это результат:

```
Ashwin
100000
---
Thor
200000
---
Jane
300000
---
Cpt America
30000
---
Iron Man
4000000
---
```

We can see the data of the list:

```
rows
```

Это результат:

```
[['Ashwin', '100000'],
 ['Thor', '200000'],
 ['Jane', '300000'],
 ['Cpt America', '30000'],
```

['Iron Man', '4000000']]

This is how we extract and process the data from a CSV.

8.3 Handling Spreadsheets

We can also read the data stored in spreadsheets with the extension *.xls or *.xlsx format. A spreadsheet application stores data in tabular form. It is not a plaintext format like CSV and we need specialized software to read the data stored in spreadsheets. We can also use Excel or free and open source software like LibreOffice and Apache OpenOffice. We can also write programs in python to read the data stored in the spreadsheets. Create a spreadsheet in the current directory and save it as **test.xlsx**. Add the following data:

Food Item	Color	Weight
Banana	Yellow	250
Orange	Orange	200
Grapes	Green	400
Tomoto	Red	100
Spinach	Green	40
Potatoes	Grey	400
Rice	White	300
Rice	Brown	400
Wheat	Brown	500
Barley	Yellow	500

As we can see, the data is organized in three columns. Let's read it with python. There are many libraries in Python that can read data from spreadsheets. We will use one such library, **openpyxl**. Let's install it. Let's first upgrade the **pip** utility:

```
!python -m pip install --upgrade pip
```

Install the library with the following command:

```
!pip3 install openpyxl
```

and import the library:

```
import openpyxl
```

Open the spreadsheet file with the following code:

```
wb = openpyxl.load_workbook('test.xlsx')
print(wb)
print(type(wb))
```

Он печатает следующий вывод:

<openpyxl.workbook.workbook.Workbook object at 0x02D90170>
<class 'openpyxl.workbook.workbook.Workbook'>

Давайте напечатаем имена всех листов (любая таблица организована как набор листов):

```
print(wb.sheetnames)
```

Вот результат:

['Sheet1', 'Sheet2', 'Sheet3']

Выберите лист для обработки:

```
currSheet = wb['Sheet1']
print(currSheet)
print(type(currSheet))
```

Результат следующий:

<Worksheet "Sheet1">
<class 'openpyxl.worksheet.worksheet.Worksheet'>

Мы также можем выбрать текущий лист:

```
currSheet = wb[wb.sheetnames[1]]
print(currSheet)
print(type(currSheet))
Print the title of the sheet:
currSheet = wb[wb.sheetnames[0]]
print(currSheet)
print(type(currSheet))
print(currSheet.title)
```

Вот результат:

<Worksheet "Sheet1">
<class 'openpyxl.worksheet.worksheet.Worksheet'>
Sheet1

Мы можем выбрать ячейку и напечатать ее значение:

```
var1 = currSheet['A1']
print(var1.value)
```

Вот еще один способ:

```
print(currSheet['B1'].value)
```

Еще один способ заключается в следующем:

```
var2 = currSheet.cell(row=2, column=2)
print(var2.value)
```

Мы можем напечатать максимальное количество строк и столбцов:

```
print(currSheet.max_row)
print(currSheet.max_column)
```

Используя описанную выше технику, мы можем получить и распечатать все строки и столбцы.

```
for i in range(currSheet.max_row):
    print('---Beginning of Row---')
    for j in range(currSheet.max_column):
        var = currSheet.cell(row=i+1, column=j+1)
        print(var.value)
    print('---End of Row---')
```

Вот результат:

---Beginning of Row---
Food Item
Color
Weight
---End of Row---
---Beginning of Row---
Banana
Yellow
250
---End of Row---
---Beginning of Row---
Orange
Orange
200
---End of Row---
---Beginning of Row---
Grapes
Green
400
---End of Row---
---Beginning of Row---

Tomoto
Red
100
---End of Row---
---Beginning of Row---
Spinach
Green
40
---End of Row---
---Beginning of Row---
Potatoes
Grey
400
---End of Row---
---Beginning of Row---
Rice
White
300
---End of Row---
---Beginning of Row---
Rice
Brown
400
---End of Row---
---Beginning of Row---
Wheat
Brown
500
---End of Row---
---Beginning of Row---
Barley
Yellow
500
---End of Row---

Вот как мы извлекаем и обрабатываем электронные таблицы с помощью Python

Краткое содержание

В этой главе мы научились читать и манипулировать данными, хранящимися в файлах разных форматов.

В следующей главе мы подробно рассмотрим область обработки изображений. Это будет большая и подробная глава со множеством демонстраций кода, которая покажется вам особенно интересной.

Глава 9. Обработка изображений с помощью Python

В предыдущей главе мы узнали, как работать с файлами различных типов.

В этой главе мы рассмотрим еще одну область, в которой Python обычно используется в качестве предпочтительного языка программирования: обработка изображений. Для этого мы подробно изучим библиотеку обработки изображений в Python. Имя этой библиотеки — Wand. Мы подробно рассмотрим следующие темы:

- Цифровая обработка изображений и библиотека палочек
- Начинаем
- Эффекты изображения
- Специальные эффекты
- Преобразования
- Статистические операции
- Улучшение цвета
- Квантование изображения
- Порог
- Искажения

Эта глава полна продвинутых концепций, их подробного описания и практических демонстраций. Прочитав эту главу, мы научимся обрабатывать изображения с помощью Python.

9.1 Цифровая обработка изображений и библиотека палочек

Обработка изображений — это использование алгоритмов для обработки изображений. Во времена аналоговых фильмов существовали методы улучшения качества изображений и кадров (в кинофильмах) с помощью ручных методов, таких как использование химических соединений. Это было предшественником современной идеи обработки изображений. В настоящее время большинство изображений являются цифровыми. Конечно, цифровому формату еще предстоит догнать яркие цвета и четкость аналогового изображения (изображения, полученные с помощью кинопленки). Подавляющее большинство организаций (производителей и обработчиков фильмов используют цифровые изображения при производстве изображений и видео. Современные компьютеры также достаточно быстры, чтобы их можно было использовать для обработки цифровых изображений. Обрабатывать изображения с помощью Python очень легко, поскольку для этого существует множество сторонних библиотек.

ImageMagicki — программа для манипулирования изображениями. Она поставляется с APIдля различных языков программирования. Мы можем использоватьбиблиотеку Wand, которая предоставляет питонический интерфейс дляImageMagick. Давайте настроим программу для начала. Сначала нам нужноустановить ImageMagick для нашей операционной системы.

Мы можем установить ImageMagick на macOS с помощью следующих команд:

```
brew install ghostscript
brew install imagemagick
```

Эти две команды должны установить ImageMagick на вашу macOS. Если нет, то мы должны установить его вручную. Это легко. Загрузите zip-файл, расположенный по адресу https://download.imagemagick.org/ImageMagick/download/binaries/ImageMagick-x86_64-apple-darwin20.1.0.tar.gz. Скопируйте его в домашний каталог вашего пользователя на macOS. Извлеките его с помощью следующей команды:

```
tar xvzf ImageMagick-x86_64-apple-darwin20.1.0.tar.gz
```

Теперь нам нужно добавить несколько записей в файл **.bash_profile**, расположенный в домашнем каталоге вашего пользователя в macOS.

```
# Settings for ImageMagik
export MAGICK_HOME="$HOME/ImageMagick-7.0.10"
export PATH="$MAGICK_HOME/bin:$PATH"
export DYLD_LIBRARY_PATH="$MAGICK_HOME/lib/"
```

Выйдите и перезапустите командную строку и выполните одну за другой следующие команды:

```
magick logo: logo.gif
identify logo.gif
display logo.gif
```

Он отобразит логотип проекта ImageMagick.

Установка в Windows проста. Существуют двоичные исполняемые установочные файлы для всех версий настольной Windows (32/64 бит). Из всех вариантов нам нужно выбрать тот, который имеет описание Win64/Win32 Dynamic с компонентом 16 бит на пиксель и включенным режимом обработки изображений в высоком динамическом диапазоне. Для 64-битных систем используйте

https://download.imagemagick.org/ImageMagick/download/binaries/ImageMagick-7.1.0-10-Q16-HDRI-x64-dll.exe

Для использования 32-битной версии Windows:

https://download.imagemagick.org/ImageMagick/download/binaries/ImageMagick-7.1.0-10-Q16-HDRI-x86-dll.exe

Используйте эти файлы для установки ImageMagick в Windows.

Давайте установим его на Linux. Загрузите исходный код с помощью следующей команды:

```
wget https://www.imagemagick.org/download/ImageMagick.tar.gz
```

Давайте проверим, куда он извлек все файлы:

```
ls ImageMagick*
```

Он показывает нам имя каталога.

```
ImageMagick-7.1.0-10
```

Перейдите в каталог.

```
cd ImageMagick-7.1.0-10
```

Выполните одну за другой следующие команды (если вы знакомы с Linux, вы поймете, что это стандартный набор команд для установки любой новой программы в дистрибутивах Linux):

```
./configure
make
sudo make install
sudo ldconfig /usr/local/lib
```

После успешной установки программы ImageMagick мы можем установить библиотеку wand на любую платформу с помощью следующей команды:

```
pip3 install wand
```

Вот как мы можем установить ImageMagick и Wand в любую операционную систему.

9.2 Начало работы

Создайте новый блокнот Jupyter для всех демонстраций в этой главе. Весь код с этого момента должен быть сохранен и выполнен в блокноте. Импортируем необходимые библиотеки.

```
from __future__ import print_function
from wand.image import Image
```

Эти операторы импортируют необходимые модули. Давайте прочитаем изображение и распечатаем его размеры следующим образом:

```
img = Image(filename='D:/Dataset/4.2.03.tiff')
print('width =', img.width)
print('height =', img.height)
```

Результат следующий:

```
width = 512
height = 512
```

Мы также можем увидеть тип изображения:

```
type(img)
```

Это дает следующий результат:

```
wand.image.Image
```

Мы можем показать изображение в блокноте как выходные данные, просто введя имя переменной, в которой хранится изображение.

```
img
```

Таким образом создается следующий вывод:

Рис.*9-1:* Изображение, отображаемое в блокноте *Jupyter*.

Я использую набор данных изображений, предоставленный

http://www.imageprocessingplace.com/root_files_V3/image_databases.htm.

Все изображения представляют собой стандартные тестовые изображения, часто используемые при обработке изображений.

Мы также можем клонировать изображение, изменить формат его файла и сохранить его на диск:

```
img1 = img.clone()
img1.format = 'png'
img1.save(filename='D:/Dataset/output.png')
```

Если вы еще не заметили, для этой демонстрации я использую компьютер с Windows. Если вы используете какую-либо Unix-подобную ОС, вам необходимо соответствующим образом изменить местоположение. Например, я использую следующий код для сохранения выходного файла на компьютере с ОС Raspberry Pi (вариант Debian Linux):

```
img1.save(filename='/home/pi/Dataset/output.png')
```

Мы также можем создать индивидуальное изображение с однородным цветом.

```
from wand.color import Color
bg = Color('black')
img = Image(width=256, height=256, background=bg)
img.save(filename='D:/Dataset/output.png')
```

Второй способ заключается в следующем:

```
img = Image(filename='D:/Dataset/4.2.03.tiff')
img1 = img.clone()
img1.resize(60, 60)
img1.size
```

Давайте посмотрим, как изменить размер изображения. Есть два способа:

```
img1 = img.clone()
img1.sample(60, 60)
img1.size
```

Подпрограммы **resize()** и **sample()** изменяют размер изображения до заданных размеров. Мы также можем обрезать часть изображения.

```
img1 = img.clone()
img1.crop(10, 10, 60, 60)
img1.size
```

9.3 Эффекты изображения

Мы можем создавать различные графические эффекты. Начнем с размытия изображения.

```
img1 = img.clone()
img1.blur(radius=6, sigma=3)
img1
```

Результат будет таким, как показано на рис.9-2.

Рис.*9-2:* Размытое изображение

Давайте применим адаптивное размытие:

```
img1 = img.clone()
img1.adaptive_blur(radius=12, sigma=6)
img1
```

Мы также можем применить размытие по Гауссу:

```
img1 = img.clone()
img1.gaussian_blur(sigma=8)
img1
```

Вот результат:

Рис.*9-3:* Адаптивное размытие

У нас может быть размытие в движении:

```
img1 = img.clone()
img1.motion_blur(radius=20, sigma=10, angle=-30)
img1
```

Мы упоминаем угол движения при вызове процедуры. Вот результат:

Рис.*9-4:* Размытие в движении под углом *30* градусов.

Мы также можем использовать вращательное размытие:

```
img1 = img.clone()
img1.rotational_blur(angle=25)
img1
```

Вот результат:

Рис.9-5: Размытие при вращении

Мы также можем использовать выборочное размытие:

```
img1 = img.clone()
img1.selective_blur(radius=10, sigma=5,
                    threshold=0.50 * img.quantum_range)
img1
```

Вот результат:

Рис.*9-6:* Выборочное размытие

Мы также можем очистить (уменьшить шум) изображение.

```
img1 = img.clone()
img1.despeckle()
img1
```

и обнаружить края:

```
img = Image(filename='D:/Dataset/4.1.07.tiff')
img1 = img.clone()
img1.edge(radius=1)
img1
```

Вот результат:

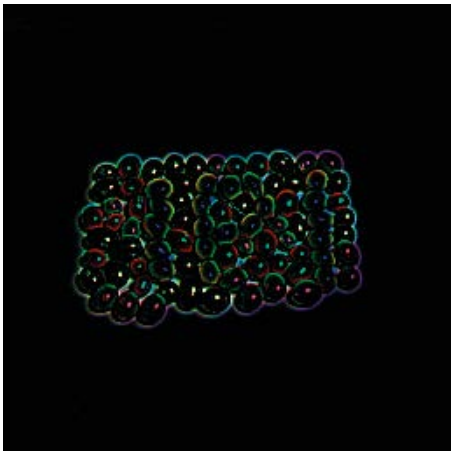


Рис.*9-7:* Обнаружение края

Мы можем создать эффект 3D-тиснения:

```
img1 = img.clone()
img1.emboss(radius=4.5, sigma=3)
img1
```

Вот результат:

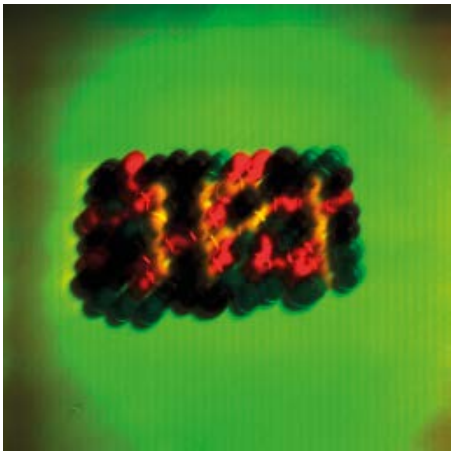


Рис.*9-8:* Тиснение

Мы также можем изменить изображение на оттенки серого и применить эффект изображения:

```
img1 = img.clone()
img1.transform_colorspace('gray')
img1.emboss(radius=4.5, sigma=3)
img1
```

Вот результат:

Рис.*9-9:* Тиснение на изображении в оттенках серого.

Мы можем применить сглаживающий фильтр для уменьшения шума:

```
img1 = img.clone()
img1.kuwahara(radius=4, sigma=2)
img1
```

Края сохраняются на выходе:

Рис.*9-10:* Сглаживающий фильтр

Мы также можем создать эффект тени:

```
img1 = img.clone()
img1.shade(gray=True,
           azimuth=30.0,
           elevation=30.0)
img1
```

Вот результат:

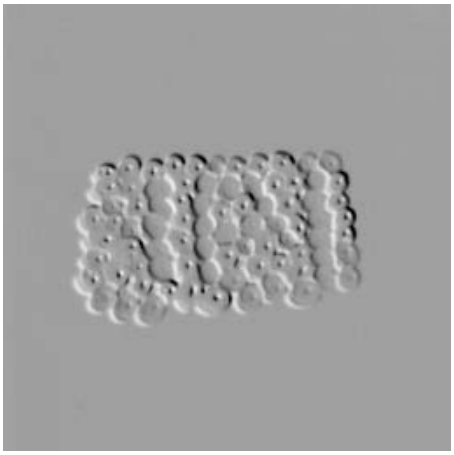


Рис.*9-11:* Фильтр тени

Мы можем повысить резкость изображения:

```
img1 = img.clone()
img1.sharpen(radius=12, sigma=4)
img1
```

Мы можем применить адаптивный алгоритм повышения резкости:

```
img1 = img.clone()
img1.adaptive_sharpen(radius=12, sigma=6)
img1
```

Мы можем использовать нерезкую маску:

```
img1 = img.clone()
img1.unsharp_mask(radius=20, sigma=5,
                  amount=2, threshold=0)
img1
```

Мы также можем случайным образом распределить пиксели по указанному радиусу:

```
img1 = img.clone()
img1.spread(radius=15.0)
img1
```

Вот результат:

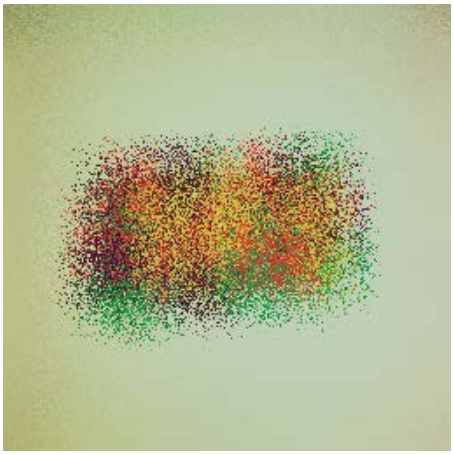


Рис.*9-12:* Распространение

9.4 Спецэффекты

Давайте изучим, как применять к изображению специальные эффекты. Первый эффект — Шум. Есть различные виды шума. Давайте посмотрим, как ввести гауссов шум.

```
img1 = img.clone()
img1.noise("gaussian", attenuate=1.0)
img1
```

Вот результат:

Рис.*9-13:* Гауссов шум

Ниже приведен список всех допустимых строк символов, которые можно использовать в качестве названий шумов:

```
'gaussian'
'impulse'
'laplacian'
'multiplicative_gaussian'
'poisson'
'random'
'uniform'
```

Мы можем сместить изображение в синий цвет следующим образом:

```
img1 = img.clone()
img1.blue_shift(factor=0.5)
img1
```

Вывод следующий:

Рис.*9-14:* Синее смещение

Мы также можем создать эффект рисунка углем:

```
img1 = img.clone()
img1.charcoal(radius=2, sigma=1)
img1
```

Вот результат:

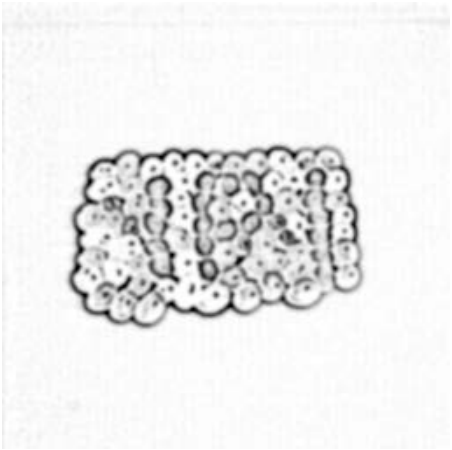


Рис.*9-15:* Эффект угля

Мы также можем применить цветовую матрицу:

```
img1 = img.clone()
matrix = [[0, 0, 1],
          [0, 1, 0],
          [1, 0, 0]]
img1.color_matrix(matrix)
img1
```

Цветовая матрица может иметь максимальный размер 6x6. В цветовой матрице каждый столбец соответствует цветовому каналу, на который можно ссылаться, а каждая строка представляет цветовой канал, на который нужно воздействовать. Для изображений RGB это красный, зеленый, синий, н/д, альфа и константа (смещение). Для изображений CMYK это голубой, желтый, пурпурный, черный, альфа и константа. В этом примере мы создали матрицу 3x3. Вот результат:

Рис.*9-16:* Цветовая матрица

Мы можем смешать изображение с постоянным цветом:

```
img1 = img.clone()
img1.colorize(color="green", alpha="rgb(10%, 0%, 20%)")
img1
```

Вот результат:

Рис.*9-17*. Смешивание с постоянным цветом.

Мы можем сжать изображение:

```
img1 = img.clone()
img1.implode(amount=0.5)
img1
```

Вывод следующий:

Рис.*9-18:* Сжатие изображения

Мы также можем иметь эффект Полароида:

```
img1 = img.clone()
img1.polaroid()
img1
```

Вот результат:

Рис.*9-19:* Эффект Полароида

Давайте применим к изображению тон сепии (на основе порога):

```
img1 = img.clone()
img1.sepia_tone(threshold=0.3)
img1
```

Вот результат:

Рис.*9-20:* Тон сепии

Мы можем преобразовать изображение в скетч:

```
img1 = img.clone()
img1.sketch(0.5, 0.0, 98.0)
img1
```

Вывод следующий:

Рис.*9-21:* Скетч

Создадим эффект выжженного:

```
img1 = img.clone()
img1.solarize(threshold=0.2 * img.quantum_range)
img1
```

Вот результат:

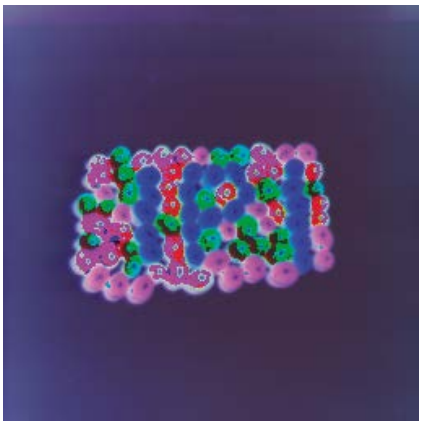


Рис.*9-22:* Соляризация

Мы можем вращать изображение:

```
img1 = img.clone()
img1.swirl(degree=90)
img1
```

Вот результат:

Рис.*9-23:* Вихрь

Мы можем тонировать изображение:

```
img1 = img.clone()
img1.tint(color="green",
          alpha="rgb(40%, 60%, 80%)")
img1
```

Вывод следующий:

Рис.*9-24:* Тонированное изображение.

Мы также можем создать эффект виньетки:

```
img1 = img.clone()
img1.vignette(sigma=3, x=10, y=10)
img1
```

Вот результат:

Рис.*9-25:* Эффект виньетки

Мы можем добавить эффект волны:

```
img1 = img.clone()
img1.wave(amplitude=img.height / 32,
          wave_length=img.width / 4)
img1
```

Вот результат:

Рис.9-26: Эффект волны

Мы также можем использовать вейвлет-шумоподавление изображения:

```
img1 = img.clone()
img1.wavelet_denoise(threshold=0.05 * img.quantum_range,
                     softness=0.0)
img1
```

Вывод следующий:

Рис.*9-27:* Эффект шумоподавления вейвлета

9.5 Преобразования

Мы можем применять преобразования к изображениям. Прочитайте новое изображение следующим образом:

```
img = Image(filename='D:/Dataset/4.1.04.tiff')
img
```

Мы можем перевернуть изображение:

```
img1 = img.clone()
img1.flip()
img1
```

Вот результат:

Рис.9-28: Эффект переворота

Мы также можем получить эффект флопа:

```
img1 = img.clone()
img1.flop()
img1
```

Это результат:

Рис.*9-29:* Эффект флопа

Мы также можем повернуть изображение:

```
img1 = img.clone()
img1.rotate(45, background=Color('rgb(127, 127, 127)'))
img1
```

Вывод следующий:

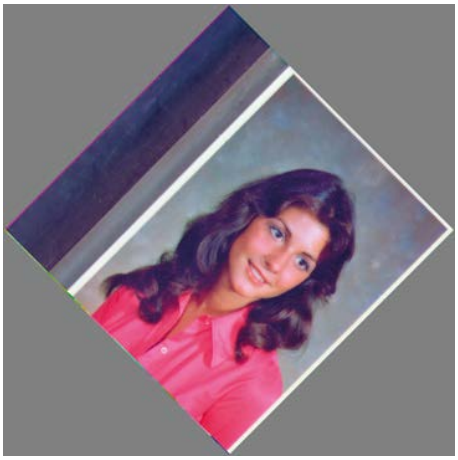


Рис.*9-30:* Повернутое изображение

9.6 Статистические операции

Мы можем выполнять статистические операции с изображениями. Их можно выполнить с помощью процедуры

```
img = Image(filename='D:/Dataset/4.1.01.tiff')
img
```

Теперь посчитаем медиану:

```
img1 = img.clone()
img1.statistic("median",
               width=8,
               height=8)
img1
```

Вывод следующий:

Рис.*9-31:* Медиана

В этом примере мы вычислили медиану каждого пикселя на основе окрестности 8x8. (передается как аргументы). Мы также можем выполнять другие статистические операции. Давайте вычислим градиент:

```
img1 = img.clone()
img1.statistic("gradient", width=8, height=8)
img1
```

Результат

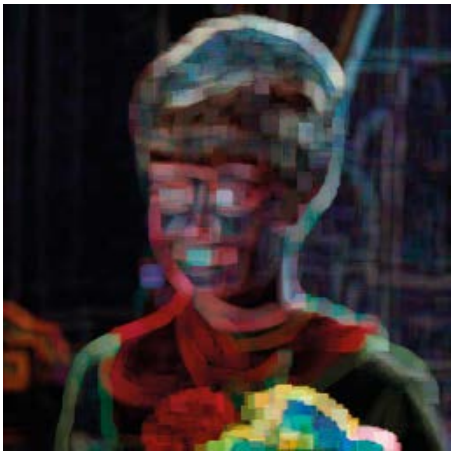


Рис.*9-32:* Градиент

Мы можем вычислить максимум:

```
img1 = img.clone()
img1.statistic("maximum", width=8, height=8)
img1
```

Вот результат:

Рис.*9-33:* Максимум

Мы можем вычислить среднее значение:

```
img1 = img.clone()
img1.statistic("mean", width=8, height=8)
img1
```

Вывод следующий:

Рис.*9-34:* Среднее значение

Мы можем вычислить минимум:

```
img1 = img.clone()
img1.statistic("minimum", width=8, height=8)
img1
```

Результат

Рис.*9-35:* Минимум

Мы можем вычислить режим:

```
img1 = img.clone()
img1.statistic("mode", width=8, height=8)
img1
```

Вот результат:

Рис.*9-36:* Режим

Мы можем вычислить непиковые значения:

```
img1 = img.clone()
img1.statistic("nonpeak", width=8, height=8)
img1
```

Вывод следующий:

Рис.*9-37:* Непиковый период

Мы также можем вычислить **Root Mean Square** - среднеквадратическое значение.

```
img1 = img.clone()
img1.statistic("root_mean_square", width=8, height=8)
img1
```

Это результат:

Рис.*9-38:* Среднеквадратическое значение

Давайте посчитаем **standard deviation** - стандартное отклонение

```
img1 = img.clone()
img1.statistic("standard_deviation", width=8, height=8)
img1
```

Вот результат:

Рис.*9-39:* Стандартное отклонение

9.7 Улучшение цвета

Давайте изучим несколько приемов улучшения цветов изображения. Мы должны использовать рутину **Assessment()** для применения различных операций. Список операций можно найти по адресу

https://docs.wand-py.org/en/0.6.7/wand/image.html#wand.image.EVALUATE_OPS.

Давайте прочитаем и отобразим новое изображение:

```
img = Image(filename='D:/Dataset/4.1.03.tiff')
img
```

Мы можем сдвинуть значения канала вправо на определенные биты:

```
img1 = img.clone()
img1.evaluate(operator='rightshift', value=2, channel='green')
img1
```

Вывод следующий:

Рис.*9-40:* Сдвиг вправо

Мы также можем применить операцию сдвига влево:

```
img1 = img.clone()
img1.evaluate(operator='leftshift', value=2, channel='blue')
img1
```

Вот результат:

Рис.*9-41:* Сдвиг влево

Еще больше операций перечислено на https://docs.wand-py.org/en/0.6.7/wand/image.html#wand.image.FUNCTION_TYPES. Мы можем применить их к изображению, используя **function()**:

```
img1 = img.clone()
frequency = 3
phase_shift = -90
amplitude = 0.2
bias = 0.7
img1.function('sinusoid', [frequency, phase_shift, amplitude, bias])
img1
```

Вот результат:

Рис.*9-42:* Синусоидальный

Давайте прочитаем новое изображение для следующих двух демонстраций:

```
img = Image(filename='D:/Dataset/4.1.06.tiff')
img
```

Мы можем настроить гамму изображения

```
img1 = img.clone()
img1.gamma(1.66)
img1
```

Вывод следующий:

Рис.*9-43:* Гамма

Мы можем настроить границы черного и белого:

```
img1 = img.clone()
img1.level(black=0.2, white=0.9, gamma=1.66)
img1
```

Расширенный вывод выглядит следующим образом:

Рис.*9-44:* Уровень

9.8 Квантование изображения

Квантование изображения — это метод сжатия с потерями. Мы можем добиться этого, сжимая диапазон значений цвета до значения одного цвета. Степень потери информации зависит от общего количества цветов в конечном выводе. Больше цветов обычно соответствует сохранение большего количества информации. Этот метод позволяет уменьшить количество байтов, необходимых для хранения и передачи изображений. Это очень полезный метод обработки изображений. Многие алгоритмы могут выполнять квантование изображения. Давайте посмотрим на некоторые из них. Для начала выберем изображение:

```
img = Image(filename='D:/Dataset/4.1.05.tiff')
img
```

Давайте применим алгоритм кластеризации K-Means:

```
img1 = img.clone()
img1.kmeans(number_colors=4,
            max_iterations=100,
            tolerance=0.01)
img1
```

Вот результат:

Рис.*9-45:* Кластеризация *KMeans*

Мы также можем использовать функцию **posterize()** для квантования изображения. Мы можем передавать разные аргументы для метода сглаживания. Давайте посмотрим на них один за другим. Давайте воспользуемся методом **Флойда Стейнберга**:

```
img1 = img.clone()
img1.posterize(levels=4,
               dither='floyd_steinberg')
img1
```

Вот еще один метод дизеринга:

```
img1 = img.clone()
img1.posterize(levels=4, dither='riemersma')
img1
```

Мы также можем избежать использования дизеринга:

```
img1 = img.clone()
img1.posterize(levels=4, dither='no')
img1
```

Мы также можем использовать процедуру **quantize()** для той же цели:

```
img1 = img.clone()
img1.quantize(number_colors=8,
              colorspace_type='srgb',
              treedepth=1,
              dither=True,
              measure_error=False)
img1
```

Вот результат:

Рис.*9-46:* Квантование с *8* цветами

9.9 Порог

При пороговом определении на основе значения каналов для пикселя мы принимаем некоторые решения. Допустим, мы определяем функцию, которая принимает аргумент и возвращает 0, если переданное значение меньше 127, тогда функция является функцией определения порога, а 127 — пороговым значением. Мы можем вручную определить такую функцию в Python. В библиотеке Wand имеется множество таких функций.
Давайте рассмотрим их один за другим.

Давайте посмотрим на локальный адаптивный порог. Он также известен как локальный порог или адаптивный порог. В этом методе каждый пиксель настраивается в соответствии со значениями окружающего пикселя. Если пиксель имеет большее значение, чем среднее значение окружающих его пикселей, ему присваивается белый цвет, в противном случае — черный.

```
img1 = img.clone()
img1.transform_colorspace('gray')
img1.adaptive_threshold(width=16, height=16,
                        offset=-0.08 * img.quantum_range)
img1
```

Мы переводим изображение в оттенки серого, а затем устанавливаем пороговое значение. Таким образом, мы можем увидеть результаты. Вывод следующий:

Рис.*9-47:* Локальный адаптивный порог

Как мы видим, операция определения порога для изображения в оттенках серого приводит к созданию двоичного (черно-белого) изображения. Она известна как бинаризация, которая является простейшей формой сегментации изображения. Давайте посмотрим, как мы можем автоматически определять пороговое значение изображения, не передавая порогового значения. Есть три метода. Первый метод — метод **Капур**.

```
 img1 = img.clone()
img1.transform_colorspace('gray')
img1.auto_threshold(method='kapur')
img1
```

Второй — метод **Оцу**:

```
img1 = img.clone()
img1.transform_colorspace('gray')
img1.auto_threshold(method='otsu')
img1
```

Последний метод — **Triangle** - Треугольник:

```
img1 = img.clone()
img1.transform_colorspace('gray')
img1.auto_threshold(method='triangle')
img1
```

Мы также можем пропустить этап преобразования изображения в оттенки серого и применить алгоритм определения порога непосредственно к цветному изображению. В таких случаях алгоритм применяется ко всем каналам цветного изображения. Давайте посмотрим на порог черного цвета, где мы устанавливаем для всех пикселей ниже порогового значения черный цвет:

```
img1 = img.clone()
img1.black_threshold(threshold='#960')
img1
```

Вот результат:

Рис.9-48: Черный порог

У нас даже может быть цветовой порог, при котором значения между началом и остановкой будут белыми, а остальные — черными:

```
img1 = img.clone()
img1.color_threshold(start='#321', stop='#aaa')
img1
```

В библиотеке **Wand** есть метод применения заранее определенных карт порогов для создания размытых изображений. Ниже приведена таблица значений карты и их значений:

Карта	Описание
порог	Порог 1x1 (без сглаживания)
чеки	Шахматная доска 2x1 (дизеринг)
o2x2	Упорядоченный 2x2 (рассеянный)
o3x3	Упорядоченный 3x3 (рассеянный)
o4x4	Ordered 4x4 (рассредоточенный)
o8x8	Ordered 8x8 (рассредоточенный)
h4x4a	Полутона 4x4 (под углом)
h6x6a	Полутона 6x6 (под углом)
h8x8a	Полутона 8x8 (под углом)

h4x4o	Полутона 4x4 (ортогональные)
h6x6o	Полутона 6x6 (ортогональные)
h8x8o	Полутона 8x8 (ортогональные)
h16x16o	Полутона 16x16 (ортогональные)
c5x5b	Круги 5x5 (черные)
c5x5w	Круги 5x5 (белые)
c6x6b	Круги 6x6 (черные)
c6x6w	Круги 6x6 (белые)
c7x7b	Круги 7x7 (черные)
c7x7w	Круги 7x7 (белые)

Давайте посмотрим простой пример того же самого с последней записью в таблице:

```
img1 = img.clone()
img1.ordered_dither('c7x7w')
img1
```

Это дает следующий результат:

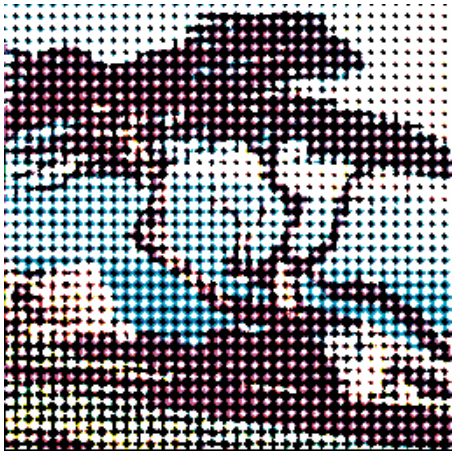


Рис.9-49: Дизеринг

Мы можем применить случайный порог между двумя заданными значениями:

```
img1 = img.clone()
img1.random_threshold(low=0.3 * img1.quantum_range,
                      high=0.6 * img1.quantum_range)
img1
```

Вот результат:

Рис.9-50: Случайный порог

У нас также может быть белый порог, который является полной противоположностью черного. Установите для всех пикселей выше заданного порога белый цвет следующим образом:

```
img1 = img.clone()
img1.white_threshold(threshold='#ace')
img1
```

Это результат:

Рис.9-51: Белый порог

Вот как мы можем пороговать изображения.

9.10 Искажения

Искажения — это геометрические преобразования, которые мы применяем к изображениям. Геометрические преобразования — это математические функции. Давайте применим к изображению простое геометрическое преобразование:

```
img1 = img.clone()
img1.distort('arc', (angle, ))
img1
```

Вот результат:

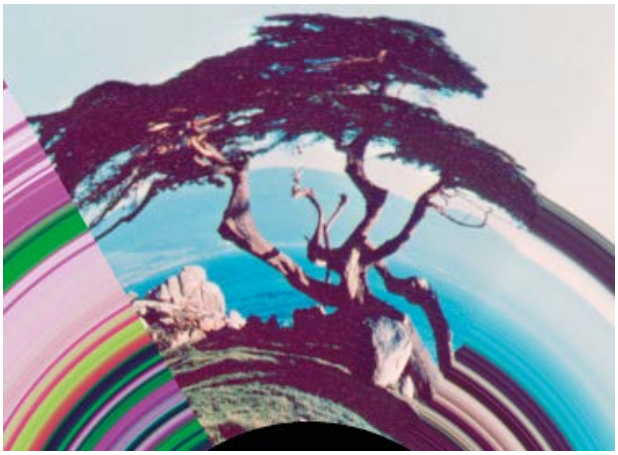


Рис.9-52: Трансформация дуги

Наблюдайте за результатом. Мы видим, что преобразование создает дополнительные пиксели, которые заполняются за счет расширения края исходного изображения. Это поведение по умолчанию. Мы можем настроить его с помощью виртуальных пикселей. Итак, прежде чем продолжить дальнейшие преобразования, мы изучим подробнее о различных методах виртуальных пикселей.

Мы можем заполнить эти дополнительные пиксели постоянным цветом:

```
img1 = img.clone()
img1.background_color = Color('rgb(127, 127, 127)')
img1.virtual_pixel = 'background'
angle = 60
img1.distort('arc', (angle, ))
img1
```

Это результат:

Рис.9-53: Серый фон

Мы можем увидеть список всех методов для виртуальных пикселей по адресу

https://docs.wand-py.org/en/0.6.7/wand/image.html#wand.image.VIRTUAL_PIXEL_METHOD.

Давайте посмотрим их демонстрации одну за другой.

У нас может быть белый фон:

```
img1 = img.clone()
img1.virtual_pixel = 'white'
angle = 60
img1.distort('arc', (angle, ))
img1
```

Мы также можем иметь черный фон:

```
img1 = img.clone()
img1.virtual_pixel = 'black'
angle = 60
img1.distort('arc', (angle, ))
img1
```

У нас может быть прозрачный фон:

```
img1 = img.clone()
img1.virtual_pixel = 'transparent'
angle = 60
img1.distort('arc', (angle, ))
```

```
img1
```

Давайте воспользуемся сглаживанием для виртуальных пикселей:

```
img1 = img.clone()
img1.virtual_pixel = 'dither'
angle = 60
img1.distort('arc', (angle, ))
img1
```

Это результат:

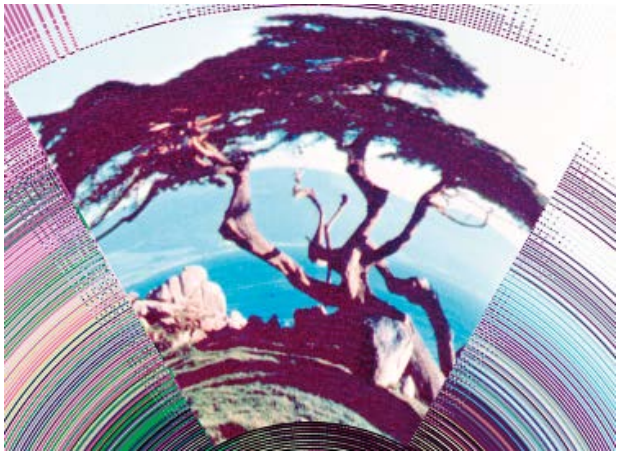


Рис.*9-54:* Дизеринг

Расширенные края — это способ по умолчанию. Мы это уже видели. Мы также можем явно указать это значение для виртуальных пикселей:

```
img1 = img.clone()
img1.virtual_pixel = 'edge'
angle = 60
img1.distort('arc', (angle, ))
img1
```

Мы также можем использовать зеркальный метод:

```
img1 = img.clone()
img1.virtual_pixel = 'mirror'
angle = 60
img1.distort('arc', (angle, ))
img1
```

Вот результат:

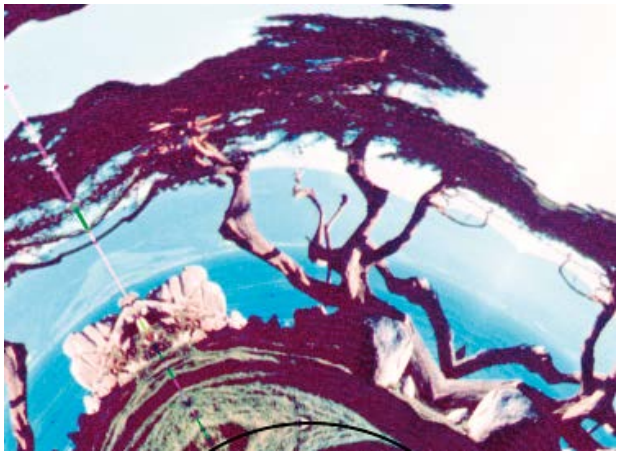


Рис.*9-55:* Зеркало

У нас могут быть случайные пиксели:

```
img1 = img.clone()
img1.virtual_pixel = 'random'
angle = 60
img1.distort('arc', (angle, ))
img1
```

Вывод следующий:

Рис.*9-56:* Случайный выбор

У нас может быть эффект **tile**:

```
img1 = img.clone()
img1.virtual_pixel = 'tile'
angle = 60
img1.distort('arc', (angle, ))
img1
```

Вот результат:

Рис.*9-57:* Плитка

9.11 Аффинные преобразования и проекции

Мы можем применить к изображению аффинное преобразование. Мы должны предоставить три точки и их отображения:

```
img1 = img.clone()
img1.resize(140, 70)
img1.background_color = Color('rgb(127, 127, 127)')
img1.virtual_pixel = 'background'
args = (10, 10, 15, 15,  # Point 1: (10, 10) => (15,  15)
        139, 0, 100, 20, # Point 2: (139, 0) => (100, 20)
        0, 70, 50, 70    # Point 3: (0,  70) => (50,  70)
)
img1.distort('affine', args)
img1
```

Результат следующий:

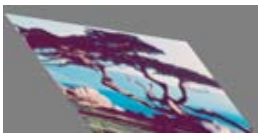


Рис.*9-58:* Аффинное преобразование

Мы также можем применять аффинные проекции, предоставляя коэффициенты масштабирования, поворота и перевода:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
img1 = img.clone()
img1.resize(140, 92)
img1.background_color = Color('skyblue')
img1.virtual_pixel = 'background'
rotate = Point(0.1, 0)
scale = Point(0.7, 0.6)
translate = Point(5, 5)
args = (scale.x, rotate.x, rotate.y,
        scale.y, translate.x, translate.y)
img1.distort('affine_projection', args)
img1
```

Это результат:

Рис.*9-59:* Аффинная проекция

9.11.1 Дуга

Мы уже видели эту трансформацию. Давайте посмотрим на это дальше подробно. Мы должны предоставить углы дуги и поворота:

```
img1 = img.clone()
img1.resize(140, 92)
img1.background_color = Color('black')
img1.virtual_pixel = 'background'
args = (270,  # ArcAngle
        45,   # RotateAngle
    )
img1.distort('arc', args)
img1
```

Вот результат:

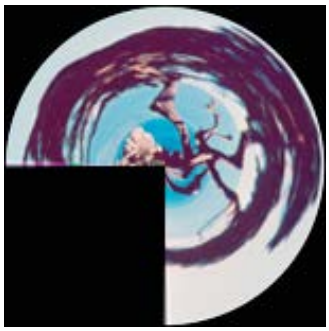


Рис.9-60: Дуга

9.11.2 Бочка и обратная бочка

У нас могут быть Бочка и обратная бочка. Нам нужно указать четыре точки данных. Математическое уравнение для барреля - бочки выглядит следующим образом:

$$R_{src} = r * (A * r^3 + B * r^2 + C * r + D)$$

r — радиус пункта назначения. Давайте посмотрим демонстрацию:

```
img1 = img.clone()
img1.resize(140, 92)
img1.background_color = Color('black')
img1.virtual_pixel = 'background'
args = (
        0.2,  # A
        0.0,  # B
        0.0,  # C
        1.0,  # D
    )
img1.distort('barrel', args)
img1
```

Это результат:

Рис.9-61: Бочка

Обратное уравнение бочки выглядит следующим образом:

$$R_{src} = r / (A * r^3 + B * r^2 + C * r + D)$$

Давайте продемонстрируем это,

```
img1 = img.clone()
img1.resize(140, 92)
img1.background_color = Color('black')
img1.virtual_pixel = 'background'
args = (
        0.0,   # A
        0.0,   # B
        -0.5,  # C
        1.5,   # D
    )
img1.distort('barrel_inverse', args)
img1
```

Вот результат:

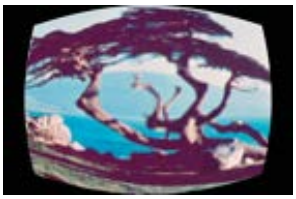


Рис.*9-62:* Инверсная бочка

9.11.3 Билинейное преобразование

При этом нам нужно указать четыре точки источника и назначения:

```
from itertools import chain
img1 = img.clone()
img1.resize(140, 92)
img1.background_color = Color('black')
img1.virtual_pixel = 'background'
source_points = (
        (0, 0),
        (140, 0),
        (0, 92),
        (140, 92))
destination_points = (
        (14, 4.6),
        (126.9, 9.2),
        (0, 92),
```

```
        (140, 92))
order = chain.from_iterable(zip(source_points, destination_points))
arguments = list(chain.from_iterable(order))
img1.distort('bilinear_forward', arguments)
img1
```

Вывод следующий:

Рис.*9-63:* Билинейный

Мы можем иметь обратную билинейную зависимость:

```
order = chain.from_iterable(zip(destination_points, source_points))
arguments = list(chain.from_iterable(order))
img1.distort('bilinear_reverse', arguments)
img1
```

Вот результат:

Рис.*9-64:* Обратный билинейный

9.11.4 Цилиндр и плоскость

Мы можем преобразовать плоское изображение в цилиндр следующим образом:

```
import math
img1 = img.clone()
img1.resize(140, 92)
img1.background_color = Color('black')
img1.virtual_pixel = 'background'
lens = 60
film = 35
args = (lens/film * 180/math.pi,)
img1.distort('plane_2_cylinder', args)
img1
```

Это результат:

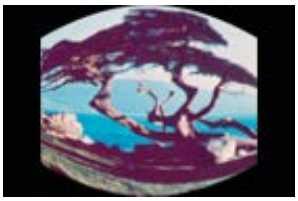


Рис.*9-65:*

Мы можем преобразовать цилиндр в плоскость:

```
img1.distort('cylinder_2_plane', args)
img1
```

Вот результат:

Рис.*9-66:* Цилиндр к плоскости

9.11.5 Полярный и деполярный

Мы можем преобразовать изображение в Polar:

```
img1 = img.clone()
img1.resize(140, 92)
img1.background_color = Color('black')
img1.virtual_pixel = 'background'
img1.distort('polar', (0,))
img1
```

Вывод следующий:

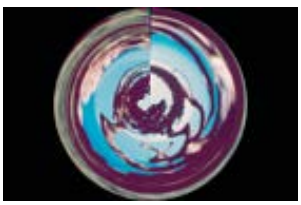


Рис.*9-67:* Полярный

Мы также можем деполяризовать изображение:

```
img1.distort('depolar', (-1,))
img1
```

Результат

Рис.*9-68:* Полярный

9.11.6 Полином

Мы можем применить полином:

```
Point = namedtuple('Point', ['x', 'y', 'i', 'j'])
img1 = img.clone()
img1.resize(140, 92)
img1.background_color = Color('black')
img1.virtual_pixel = 'background'
order = 1.5
alpha = Point(0, 0, 26, 0)
beta = Point(139, 0, 114, 23)
gamma = Point(139, 91, 139, 80)
delta = Point(0, 92, 0, 78)
args = (order,
        alpha.x, alpha.y, alpha.i, alpha.j,
        beta.x, beta.y, beta.i, beta.j,
        gamma.x, gamma.y, gamma.i, gamma.j,
        delta.x, delta.y, delta.i, delta.j)
img1.distort('polynomial', args)
img1
```

Это результат:

Рис.*9-69:* Полином

9.11.7 Искажения,преобразования

Мы можем применить преобразование **Шепарда**:

```
img1 = img.clone()
img1.resize(140, 92)
img1.background_color = Color('black')
img1.virtual_pixel = 'background'
alpha = Point(0, 0, 30, 15)
beta = Point(70, 46, 60, 70)
args = (*alpha, *beta)
img1.distort('shepards', args)
img1
```

Вот результат:

Рис.9-70: Искажение Шепарда

Краткое содержание

В этой главе мы исследовали область обработки изображений. Мы рассмотрели множество программ из библиотеки Wand для обработки цифровых изображений.

Мы продолжим наше путешествие по изучению Python в следующей главе. Мы рассмотрим подробно еще несколько полезных тем.

Глава 10 • Несколько полезных тем по Python

В предыдущей главе мы узнали, как работать с изображениями и применять различные методы обработки изображений для повышения их качества.

В этой главе рассматривается ряд тем, которые я не смог добавить в другие главы. Ниже приводится список тем, которые мы будем изучать в этой главе:

- Аргументы командной строки
- Облако слов

Надеемся, что эта глава поможет нам освоить концепции, упомянутые выше.

10.1 Аргументы командной строки

В самой первой главе мы узнали, как запустить сценарий или программу Python из командной строки. Мы также можем обрабатывать аргументы командной строки. Давайте посмотрим, как это делается. Взгляните на следующую программу:

```
prog00.py
#!/usr/bin/python3
import sys

n = len(sys.argv)
print("Total arguments passed: ", n)
print("\nArguments passed: \n")
for i in range(0, n):
    print(sys.argv[i], end = "\n")
```

Мы используем встроенный модуль **insys** для обработки аргументов командной строки. Если мы запустим эту программу из любой IDE, она выведет следующий результат:

Total arguments passed: 1

Arguments passed:

C:/Users/Ashwin/Google Drive/Elektor/Python Book Project/Code/Chaptet10/prog00.py

Первым аргументом всегда является имя скрипта.

Запустите его из командной строки (cmd / powershell в Windows или эмуляторе терминала Unix-подобной операционной системы):

```
python prog00.py 1 "test 123" test
```

Вот результат:

Arguments passed:
prog00.py
1
test 123
test

Вот как мы можем обрабатывать аргументы командной строки в Python. Это очень полезный метод, который можно использовать при программировании текстовых утилит.

10.2 Облако слов

Облака слов также известны как облака тегов. Они представляют собой визуальное представление частоты ключевых слов в исходном документе, поэтому наиболее часто встречающееся слово имеет наибольший размер. Это относится ко всем остальным ключевым словам в документе, и мы получаем визуальную форму, похожую на облако. Существует множество инструментов для создания облаков слов. Мы также можем программно генерировать их.

Начнем с программной части. Создайте новый блокнот для этого раздела. Выполните следующую команду, чтобы установить необходимые библиотеки:

```
!pip3 install matplotlib pandas wordcloud pillow
```

Matplotlib — это библиотека визуализации в экосистеме Scientific Python. **Panda** — библиотека анализа данных в Python. **Pillowis** — библиотека обработки изображений. **Wordcloud** — библиотека для создания облаков слов. Давайте запустим волшебную команду, которая включает визуализацию в блокноте:

```
%matplotlib inline
```

Импортируйте все необходимые модули и библиотеки:

```
from wordcloud import WordCloud, STOPWORDS
import matplotlib.pyplot as plt
import pandas as pd
from PIL import Image
```

Я загрузил zip-файл, содержащий файл CSV, с сайта

> https://archive.ics.uci.edu/ml/machine-learning-databases/00380/YouTube-Spam-Collection-v1.zip

Затем мы можем извлечь файл CSV из сжатого файла и использовать его в нашей программе. Мы можем прочитать CSV-файл с помощью процедуры из библиотеки pandas.

```
df = pd.read_csv(r"Youtube05-Shakira.csv", encoding = "latin-1")
```

Давайте определим две переменные: key и stop words,

```
comment_words = ' '
stopwords = set(STOPWORDS)
stopwords
```

Переменная для ключевых слов представляет собой пустую строку. Мы можем добавить ключевые слова в список следующим образом:

```
for val in df.CONTENT:
    val = str(val)
    tokens = val.split()
    for i in range(len(tokens)):
        tokens[i] = tokens[i].lower()

    for words in tokens:
        comment_words = comment_words + words + ' '
comment_words
```

Давайте сгенерируем облако слов следующим образом:

```
wordcloud = WordCloud(width= 1920, height=1080,
                      background_color='white',
                      stopwords = stopwords,
                      min_font_size = 10).generate(comment_words)
```

Давайте визуализируем это с помощью библиотеки Matplotlib.

```
plt.imshow(wordcloud)
plt.axis('off')
plt.tight_layout(pad=0)
plt.show()
```

Результат следующий:

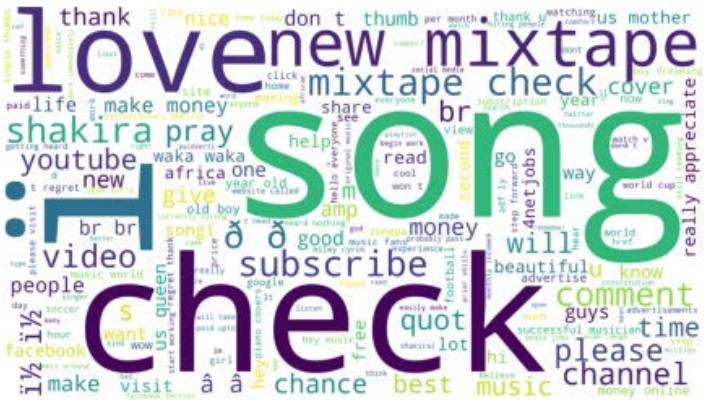


Рис.*10-1:* Облако слов с настройками по умолчанию.

Мы можем создать облако слов из текстовой строки. Давайте создадим строку:

```
text=("Python is an interpreted, high-level, general-purpose programming
language. Created by Guido van Rossum and first released in 1991, Python's design
philosophy emphasizes code readability through use of significant whitespace. Its
language constructs and object-oriented approach aim to help programmers write
clear, logical code for small and large-scale projects.")
```

Создадим облако слов:

```
# Create the wordcloud object
wordcloud = WordCloud(width= 1280, height=720,
                      margin = 0).generate(text)

# Display the generated image:
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.margins(x=0, y=0)
plt.show()
```

Вот результат:

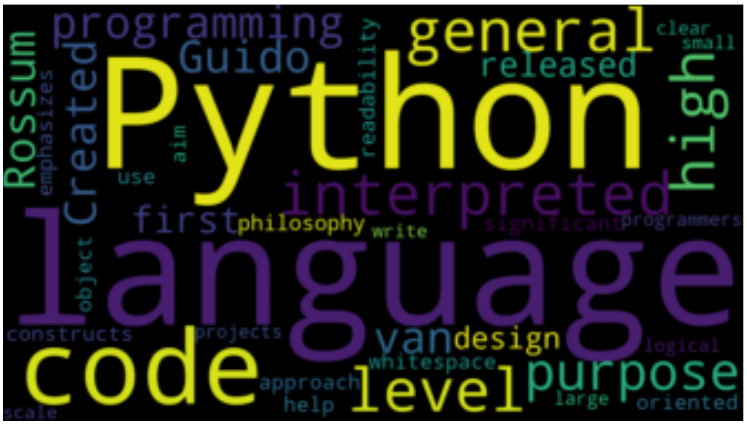


Рис.*10-2:* Облако слов текстовой строки в качестве источника

Мы также можем создать облако слов таким образом, чтобы оно отображало только заданное количество слов. Эти слова являются наиболее часто встречающимися в источнике:

```
wordcloud = WordCloud(width= 1280, height=720,
                      max_words = 3).generate(text)
plt.figure()
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.margins(x=0, y=0)
plt.show()
```

Вот результат:

Рис.10-3: Облако слов с заданным количеством слов.

Если вы еще раз запустите все ячейки, содержащие примеры кода, вы увидите, что создаются похожие, но разные изображения. Поскольку позиция и цвет случайным образом назначаются словам в облаке, мы получаем немного разные изображения для одних и тех же данных.

Мы можем удалить некоторые слова из конечного результата следующим образом:

```
wordcloud = WordCloud(width= 1280, height=720,
                      stopwords = ['Python', 'code']).generate(text)
plt.figure()
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.margins(x=0, y=0)
plt.show()
```

Вот результат:

Рис.*10-4:* Облако слов с опущенными несколькими ключевыми словами.

Мы также можем изменить цвет фона:

```
wordcloud = WordCloud(width= 1280, height=720,
                      background_color='skyblue').generate(text)
plt.figure()
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.margins(x=0, y=0)
plt.show()
```

Это результат:

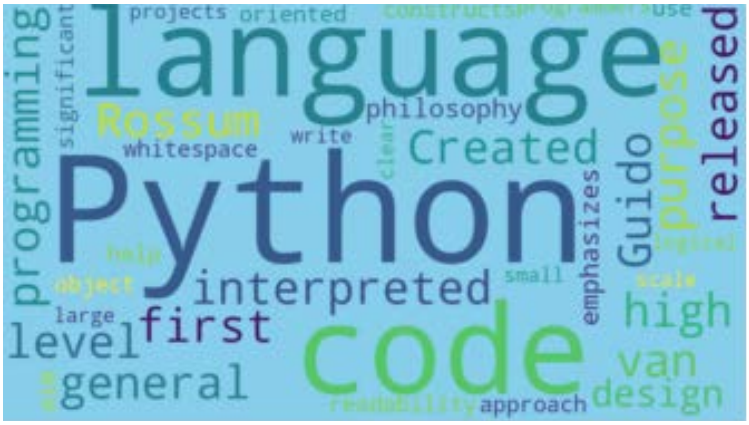


Рис.10-5: Облако слов с пользовательским цветом фона.

Мы также можем изменить цвет слов:

```
wordcloud = WordCloud(width= 1280, height=720,
                      colormap='Blues').generate(text)
plt.figure()
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.margins(x=0, y=0)
plt.show()
```

Вывод следующий:

Рис.10-6: Облако слов с настраиваемой цветовой картой слов.

Другой цвет текста можно задать с помощью этой техники:

```
wordcloud = WordCloud(width= 1280, height=720,
                      colormap='autumn').generate(text)
plt.figure()
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.margins(x=0, y=0)
plt.show()
```

Вот результат:

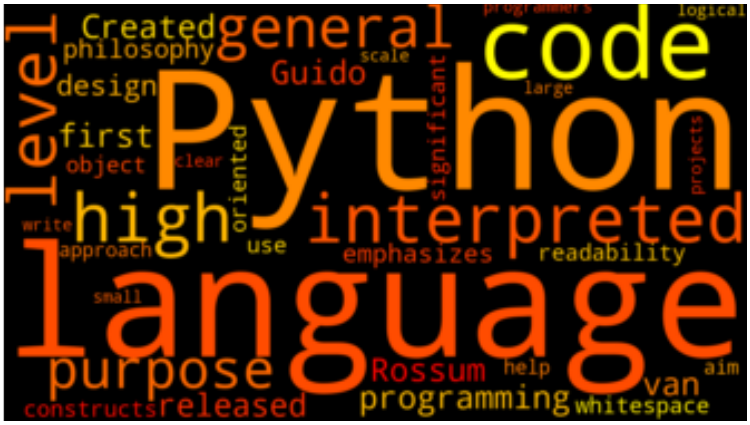


Рис.*10-7:* Облако слов с другой пользовательской цветовой картой для слов.

Краткое содержание

В этой главе мы рассмотрели некоторые важные и полезные темы Python. Мы научились принимать и обрабатывать аргументы командной строки. Мы также научились создавать графические облака слов.

Заключение

На этой главе наше путешествие по изучению Python подходит к концу. Помните, что океан языка программирования Python огромен и глубок.

В этой книге мы дали общий обзор программирования на Python. Мы можем использовать язык программирования Python в различных областях, таких как автоматизация, системное программирование, автоматизация тестов, графика, компьютерное зрение, машинное обучение и искусственный интеллект. Вооружившись базовыми знаниями Python, вы теперь можете глубже погрузиться в любую (или все, если хотите) из вышеперечисленных областей.

Кикстарт на Python 3

Курс сверхбыстрого программирования

Эта книга служит для новичков первым шагом к изучению программирования на Python. Книга разделена на десять глав. В первой главе читатели знакомятся с основами Python. Она содержит подробные инструкции по установке на различные платформы, такие как macOS, Windows, FreeBSD и Linux. Она также охватывает другие аспекты программирования на Python, такие как IDE и диспетчер пакетов. Во второй главе читатели получают возможность подробно ознакомиться с программированием на Python. Она охватывает группу встроенных структур данных, широко известных как PythonCollections. В третьей главе рассматриваются важные понятия строк, функций и рекурсии.

Четвертая глава посвящена объектно-ориентированному программированию на Python. В пятой главе обсуждаются наиболее часто используемые пользовательские структуры данных, такие как стек и очередь. Шестая глава стимулирует творчество читателей с помощью графической библиотеки Python Turtle. В седьмой главе рассматриваются анимация и разработка игр с использованием библиотеки Pygame. В восьмой главе рассматривается обработка данных, хранящихся в различных форматах файлов. В девятой главе рассматривается область обработки изображений с помощью библиотеки Wand на Python. В десятой и последней главе представлен ряд различных полезных тем, связанных с Python.

Вся книга построена поэтапно. Объяснение темы всегда сопровождается подробным примером кода. Примеры кода также объясняются достаточно подробно и сопровождаются выводом в виде текста или снимка экрана, где это возможно. Читатели освоят язык программирования Python, внимательно изучая концепции и примеры кода в этой книге. В книге также есть ссылки на внешние ресурсы, которые читатели могут изучить дальше.

Программный код и ссылки на обучающие видеоролики можно загрузить на веб-сайте Elektor.

Ашвин Паджанкар получил степень магистра технологий в IIITH Хайдарабаде и имеет более чем 25-летний опыт программирования. Свой путь в программировании и электронике он начал с языка программирования BASIC, а теперь хорошо владеет программированием на языке ассемблера, C, C++, Java, Shell Scripting и Python. Далее Технический опыт включает одноплатные компьютеры, такие как Raspberry Pi, BananaPro и Arduino.

ISBN 978-3-89576-486-8