МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«УФИМСКИЙ ГОСУДАРСТВЕННЫЙ НЕФТЯНОЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Л.В. Еникеева, В.Ф. Шамшович, Н.Ю. Фаткуллин

Основы программирования на языке Python

Учебное пособие

Уфа
Издательство УГНТУ
2018

УДК 004.43(07)
ББК 22.18я7
Е63

Утверждено Редакционно-издательским советом УГНТУ
в качестве учебного пособия

Рецензенты:
доцент кафедры «Математика» УГНТУ, к.т.н. И.Н. Сулейманов
профессор кафедры математического моделирования БашГУ,
д.ф.-м.н., доцент А.С. Исмагилова

Еникеева, Л.В.
Е63 Основы программирования на языке Python : учеб. пособие / Л.В. Еникеева, В.Ф. Шамшович, Н.Ю. Фаткуллин. – Уфа: Изд-во УГНТУ, 2018. – 87 с.

ISBN ...

В учебном пособии в общедоступной форме излагаются начальные сведения о программировании на языке Python. Содержание пособия соответствует требованиям федерального государственного образовательного стандарта. Пособие может служить справочным материалом при выполнении курсовых и дипломных работ, связанных с расчетами на компьютере.

Учебное пособие предназначается студентам специальности 09.03.03 – Прикладная информатика по дисциплине «Языки программирования» и может быть рекомендовано студентам различных специальностей технических вузов, занимающихся программированием, математическим моделированием и численными методами, и призвано дополнить другие источники информации.

УДК 004.43(07)
ББК 22.18я7

ISBN ...

Оглавление

Глава 1

Знакомство с Python и средами программирования

1.1 История

История языка программирования Python началась в конце 1980-х. Гвидо ван Россум задумал Python в 1980-х годах [1], а приступил к его созданию в декабре 1989 года в центре математики и информатики в Нидерландах. Язык Python был задуман как потомок языка программирования ABC, способный к обработке исключений и взаимодействию с операционной системой Амёба. Ван Россум является основным автором Python и по сей день продолжает выполнять центральную роль в принятии решений относительно развития языка.

Версия Python 2.0 была выпущена 16 октября 2000 года и включала в себя много новых крупных функций — таких как полный сборщик мусора и поддержка Unicode. Однако наиболее важным из всех изменений было изменение самого процесса развития языка и переход на более прозрачный процесс его создания.

Первая обратно-несовместимая версия Python 3.0 была выпущена 3 декабря 2008 года после длительного периода тестирования. Многие её функции были портированы в обратно совместимые Python 2.6 и Python 2.7.

После того, как Россум разработал язык, он выложил его в Интернет, где уже целое сообщество программистов присоединилось к его улучшению. Python активно совершенствуется и в настоящее время. Часто выходят его новые версии. Официальный сайт http://python.org.

1.2 Общие сведения о Python. Достоинства и недостатки

Python – высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Синтаксис ядра Python минималистичен. В то же время стандартная библиотека включает большой объём полезных функций. Python распространяется свободно на основании лицензии, совместимой GNU General Public License [2].

Python поддерживает несколько парадигм программирования, в том числе структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентиро-ванное. Основные архитектурные черты – динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений и удобные высокоуровневые структуры данных. Код в Python организовывается в функции и классы, которые могут объединяться в модули (они в свою очередь могут быть объединены в пакеты).

1.2.1 Преимущества языка

- Интерпретатор Python реализован практически на всех платформах и операционных системах. Первым таким языком был C, однако его типы данных на разных машинах могли занимать разное количество памяти и это служило некоторым препятствием при написании действительно переносимой программы. Python же таким недостатком не обладает.

- Расширяемость языка. Язык изначально был задуман именно как расширяемый. Это означает, что имеется возможность совершенствования языка всеми заинтересованными программистами. Интерпретатор написан на C и исходный код доступен для любых манипуляций. В случае необходимости, можно вставить его в свою программу и использовать как

встроенную оболочку. Или же, написав на C свои дополнения к Python и скомпилировав программу, получить «расширенный» интерпретатор с новыми возможностями.

- Наличие большого числа подключаемых к программе модулей, обеспечивающих различные дополнительные возможности. Такие модули пишутся на C и на самом Python и могут быть разработаны всеми достаточно квалифицированными программистами. В качестве примера можно привести следующие модули:

  Numerical Python – расширенные математические возможности, такие как манипуляции с целыми векторами и матрицами;

  Tkinter – построение приложений с использованием графического пользовательского интерфейса (GUI) на основе широко распространенного на X-Windows Tk-интерфейса;

  OpenGL – использование обширной библиотеки графического моделирования двух- и трехмерных объектов Open Graphics Library фирмы Silicon Graphics Inc.

- Входит в поставку большинства дистрибутивов Linux, следовательно, есть на большинстве серверов.

- Сравнительно простой (что может являться и минусом, так как не прививает новобранцу должную культуру программирования), но в то же время строгий синтаксис.

- Python подходит для любых решений в области программирования, будь то офисные программы, web-приложения, GUI-приложения и т. д.

1.2.2 Недостатки языка

- Скорость. Одним из главных недостатков является его относительно низкая скорость выполнения. Python является языком с полной динамической типизацией, автоматическим управлением памятью. Если на первый взгляд это может казаться преимуществом, то при разработке программ с повышенным требованием к эффективности, Python может значительно проигрывать по скорости таким языкам программирования, как C/C++, Java, Go. Что касается PHP, Ruby, JavaScript, то Python в большинстве случаев выполняет код быстрее за счет предварительной компиляции в байт-код и значительной части стандартной библиотеки, написанной на Си.

- Динамическая типизация. Для начинающих программистов, язык программирования с динамической типизацией на первый взгляд может казаться довольно простым. Но с ростом кодовой базы (а это часто неизбежный процесс в успешных проектах), следить за типом передаваемых друг другу данных бывает очень сложно (а при отсутствии внятной документации практически невозможно), отсюда появляются проблемы, когда, например, у None пытаются вызвать метод или обратиться к атрибуту в процессе выполнения кода.

1.3 Возможности языка Python

- Работа с xml/html файлами
- Работа с http запросами
- GUI (графический интерфейс)
- Создание веб-сценариев

- Работа с FTP
- Работа с изображениями, аудио и видео файлами
- Робототехника
- Программирование математических и научных вычислений и т.д.

Таким образом, Python подходит для решения многих повседневных задач, будь то резервное копирование, чтение электронной почты, игра. Язык программирования Python практически ничем не ограничен, поэтому также может использоваться в крупных проектах. К примеру, Python интенсивно применяется Google и Yandex. К тому же простота и универсальность Python делают его одним из лучших языков программирования.

1.4 Упражнения и контрольные вопросы к главе 1

1. Кто является автором языка программирования Python? Когда был создан Python?
2. Перечислите преимущества и недостатки языка программирования Python.

Глава 2

Установка Python

2.1 Установка Python на Windows

Для установки Python в операционной системе Windows необходимо выполнить следующие шаги:

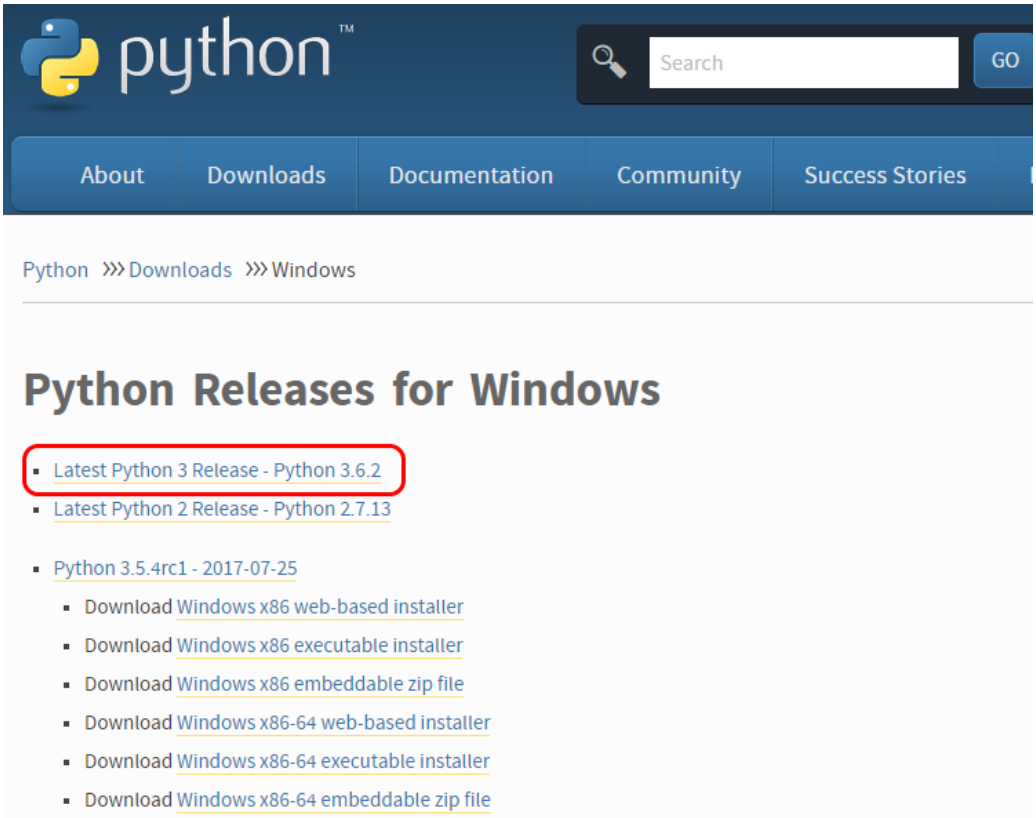


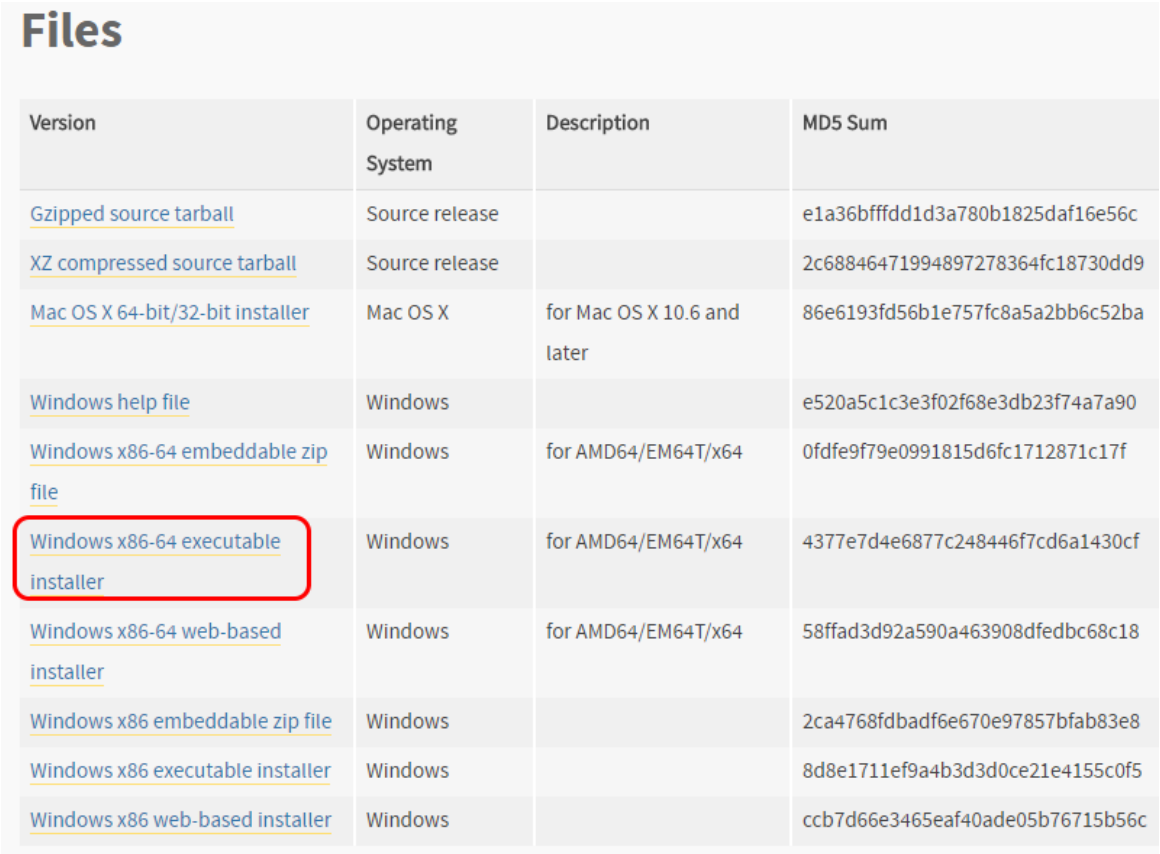
Рис. 2.1: Страница загрузки Python

1) Скачать Python с официального сайта. Не рекомендуется скачивать интерпретатор python с других сайтов или через торрент, в них могут быть вирусы. Программа бесплатная. Необходимо зайти на сайт[1], выбрать python 3. На python 2 могут не работать некоторые примеры программ, приведенные

[1] https://python.org/downloads/windows/

в данном пособии. На момент написания пособия это python 3.6.2. Выбираем «Latest Python 3 Release» (Рис. 2.1). Появляется страница с описанием данной версии на Python. В конце страницы в разделе «Files» имеется список файлов, которые можно скачать.

2) Выбрать Windows x86 (если система 32-х битная) или Windows x86-64 (если система 64-х битная) (Рис. 2.2). Ждём, пока python загрузится. Затем открываем загрузившийся файл. Файл подписан Python Software Foundation (Рис. 2.3).



Files

Version	Operating System	Description	MD5 Sum
Gzipped source tarball	Source release		e1a36bfffdd1d3a780b1825daf16e56c
XZ compressed source tarball	Source release		2c68846471994897278364fc18730dd9
Mac OS X 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	86e6193fd56b1e757fc8a5a2bb6c52ba
Windows help file	Windows		e520a5c1c3e3f02f68e3db23f74a7a90
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	0fdfe9f79e0991815d6fc1712871c17f
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	4377e7d4e6877c248446f7cd6a1430cf
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	58ffad3d92a590a463908dfedbc68c18
Windows x86 embeddable zip file	Windows		2ca4768fdbadf6e670e97857bfab83e8
Windows x86 executable installer	Windows		8d8e1711ef9a4b3d3d0ce21e4155c0f5
Windows x86 web-based installer	Windows		ccb7d66e3465eaf40ade05b76715b56c

Рис. 2.2: Страница загрузки Python

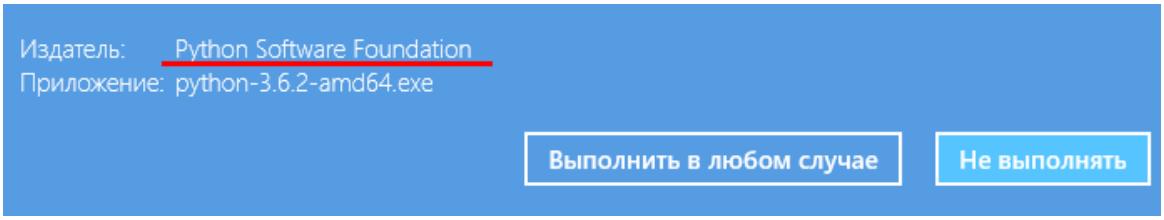


Рис. 2.3: Установка Python

3) Поставить галочку в пункте «Add Python 3.6 to Path». По желанию

ставится галочка в пункте «Install Launcher for all users». Нажать на кнопку «Customize installation» (Рис. 2.4).

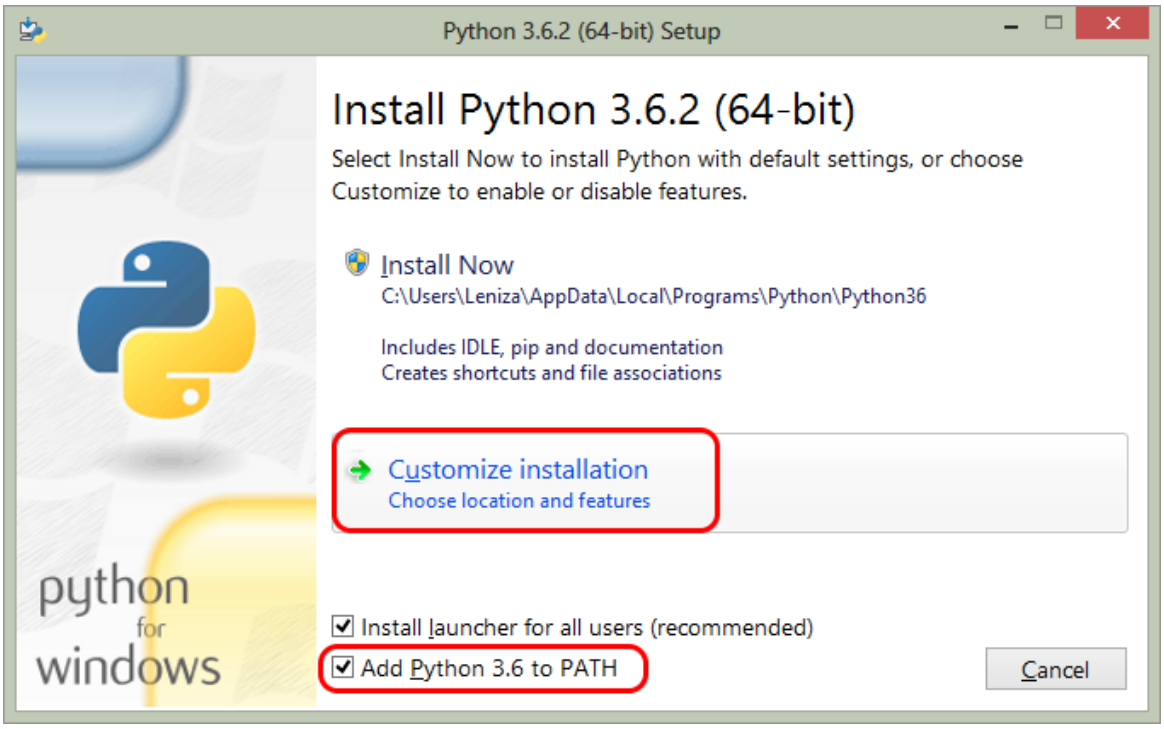


Рис. 2.4: Установка Python

4) Выбрать компоненты, которые будут установлены и нажать на кнопку «Next» (Рис. 2.5).

5) Выбрать папку для установки путем нажатия на кнопку «Browse» и нажать на кнопку «Install» (Рис. 2.6).

6) Дождаться конца установки Python. После успешного завешения установки нажать на кнопку «Close» (Рис. 2.7).

С документацией по Python можно ознакомиться на сайте https://docs.python.org/3.6/index.html.

Также можно запустить «локальную» версию документации, для этого на компьютере, на котором установлен Python, в меню «Пуск» выбираем пункт «Программы» | Python 3.6 | Python 3.6 (Manuals).

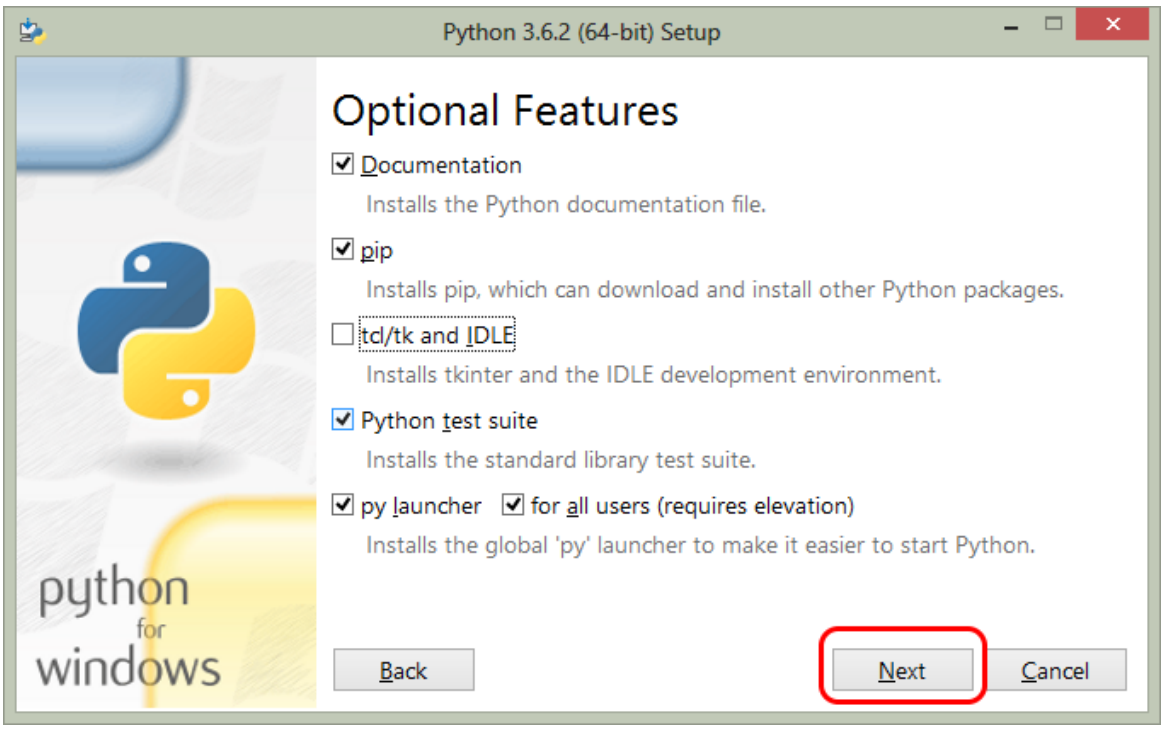


Рис. 2.5: Установка Python

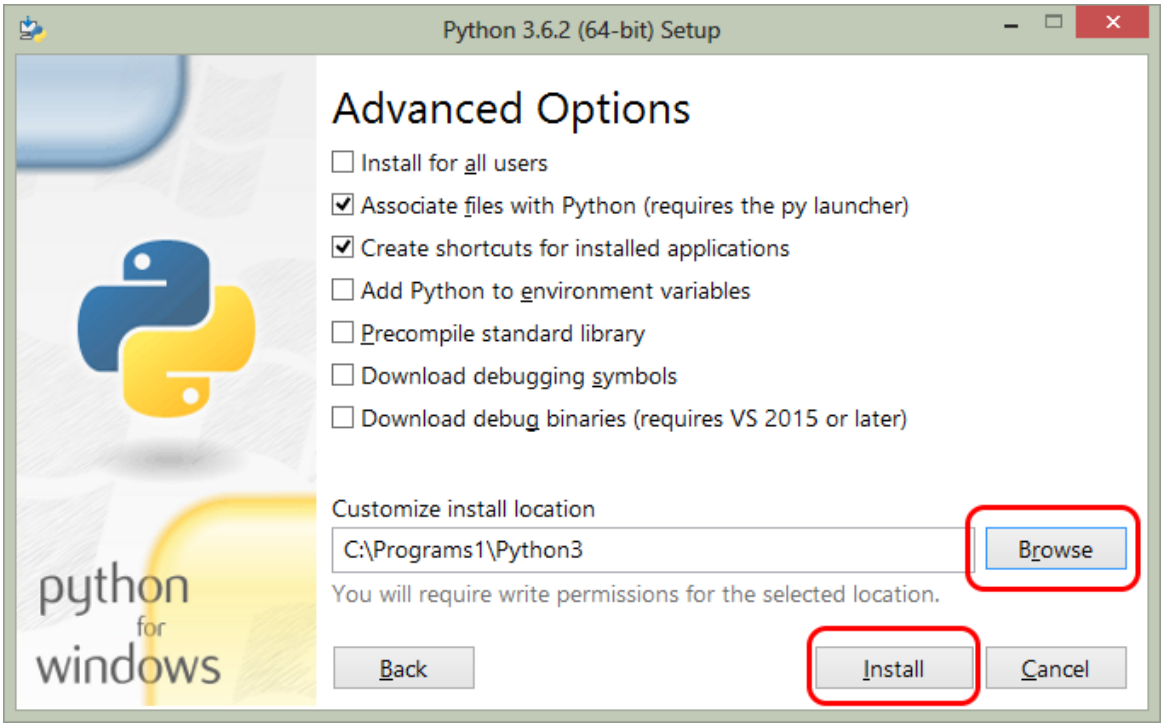


Рис. 2.6: Установка Python

2.2 Установка Python на linux системы

Для установки Python необходимо открыть консоль (обычно комбинация ctrl+alt+t). Затем в консоли нужно ввести:

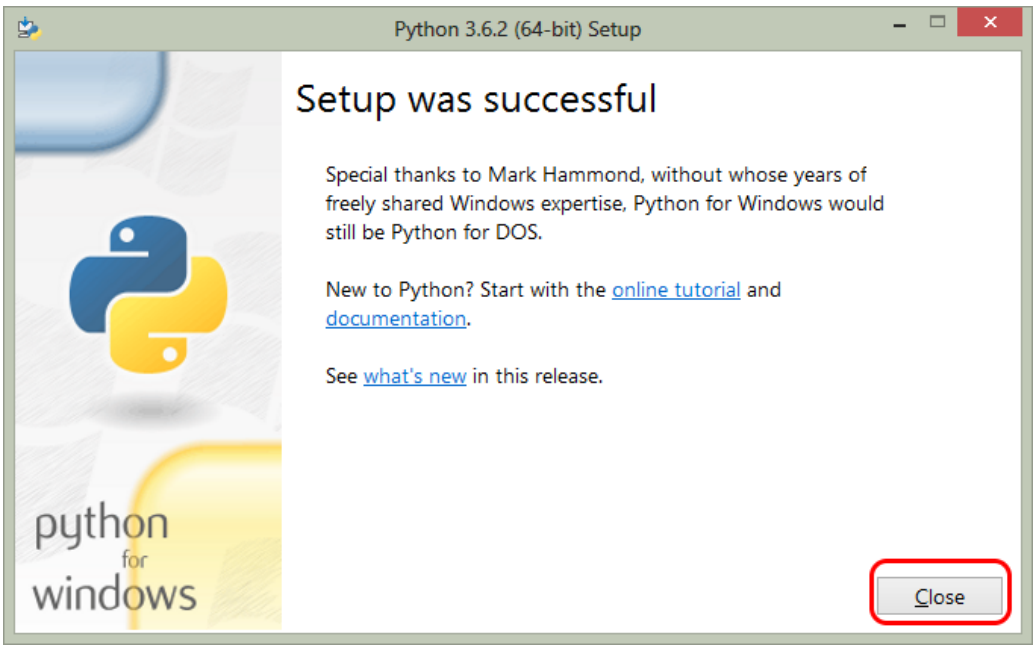


Рис. 2.7: Установка Python

```
python3
```

Скорее всего, в консоли отобразится текущая версия Python, которая установлена в системе. В противном случае нужно установить пакет **python3**:

```
sudo apt-get install python3
```

2.3 Установка IDE PyCharm

IDE с (англ. Integrated Development Environment – Интегрированная среда разработки) – система программных средств, используемая программистами для разработки программного обеспечения.

Рис. 2.8: Установка PyCharm

Для разработки прогамм на языке Python установим IDE PyCharm.

PyCharm – интегрированная среда разработки для языка програм-

мирования Python [3]. Предоставляет средства для анализа кода, графический отладчик и поддерживает веб-разработку на Django.

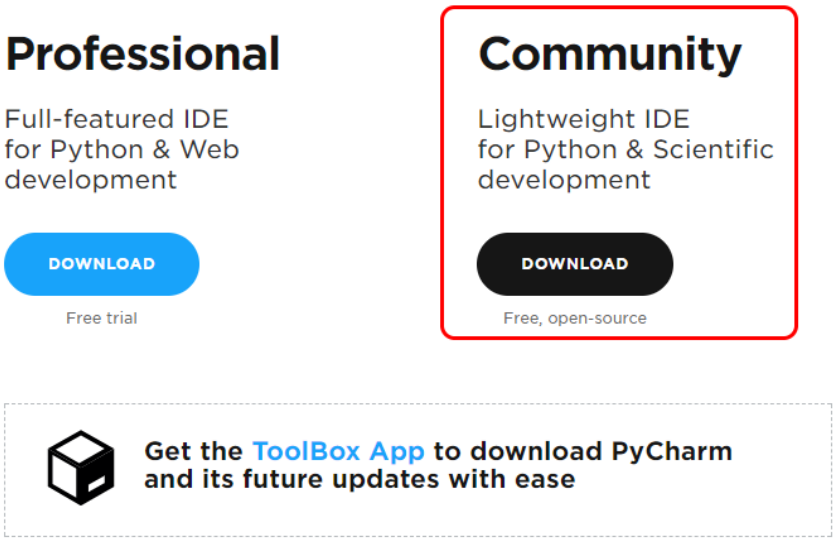


Рис. 2.9: Установка PyCharm

PyCharm разработана компанией JetBrains.

PyCharm работает под операционными системами Windows, Mac OS X и Linux.

Для установки PyCharm на компьютер, необходимо:

1) Скачать PyCharm с официального сайта https://www.jetbrains.com/pycharm/. Нажать на кнопку «Download now» (Рис. 2.8).

(Руководство по установке и системные требования приведены на странице: PyCharm Help). После чего выбрать версию «Community» (Рис. 2.9).

2) Открыть загрузившийся файл.

3)Выбрать папку для установки путем нажатия на кнопку «Browse» и нажать на кнопку «Next» (Рис. 2.10).

4) Отметить необходимые галочки и нажать на кнопку «Next» (Рис. 2.11).

5) В следующем окне нажать на кнопку «Install» (Рис. 2.12).

6) Дождаться конца установки PyCharm. После успешного завешения установки нажать на кнопку «Finish» (Рис. 2.13).

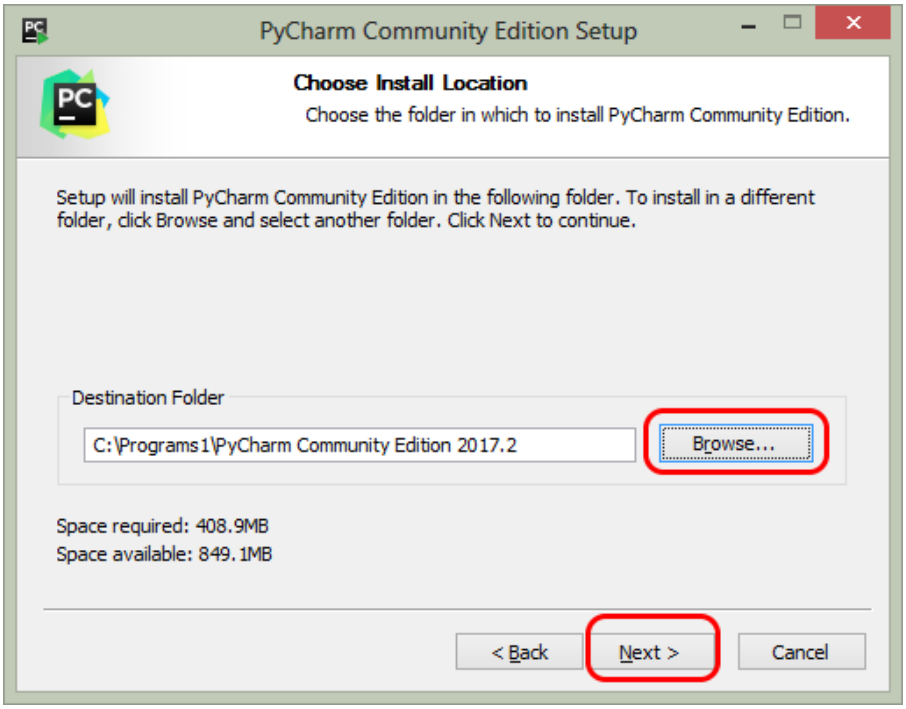


Рис. 2.10: Установка PyCharm

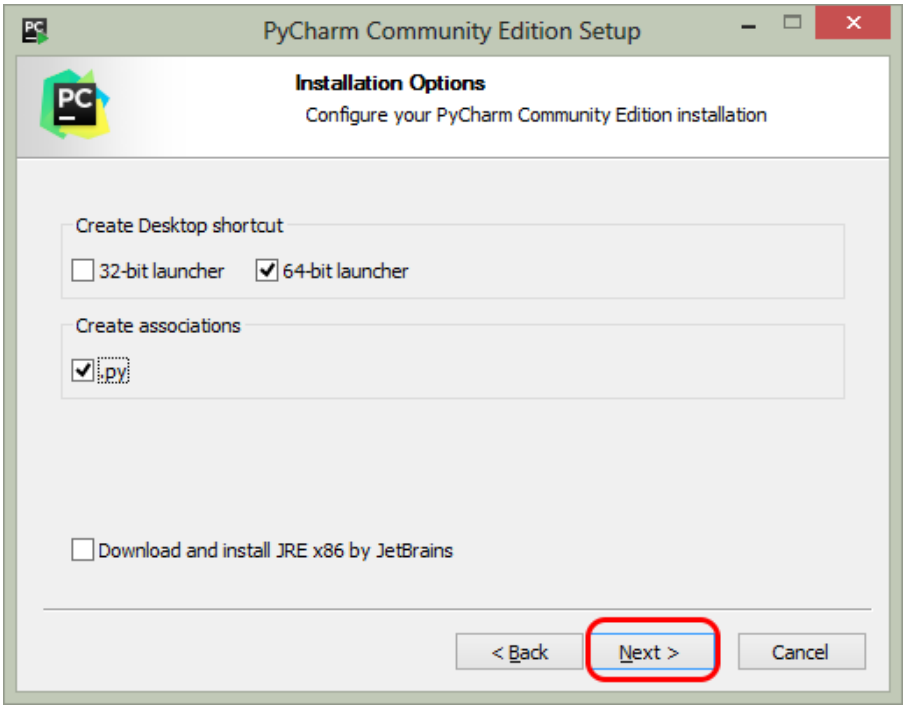


Рис. 2.11: Установка PyCharm

Теперь можно переходить к написанию первой программы на Python (Глава 3).

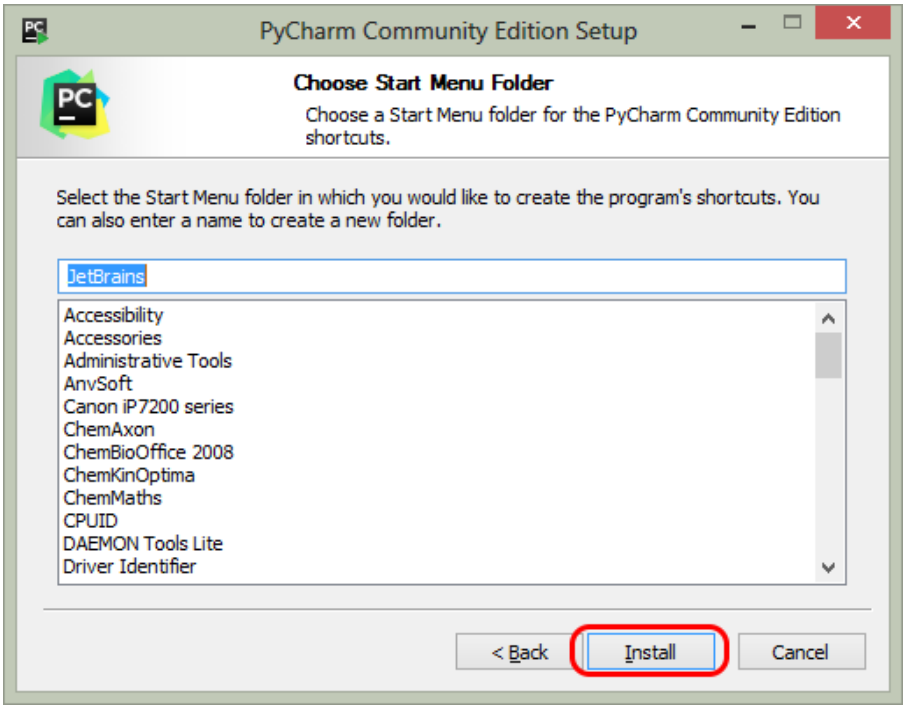


Рис. 2.12: Установка PyCharm

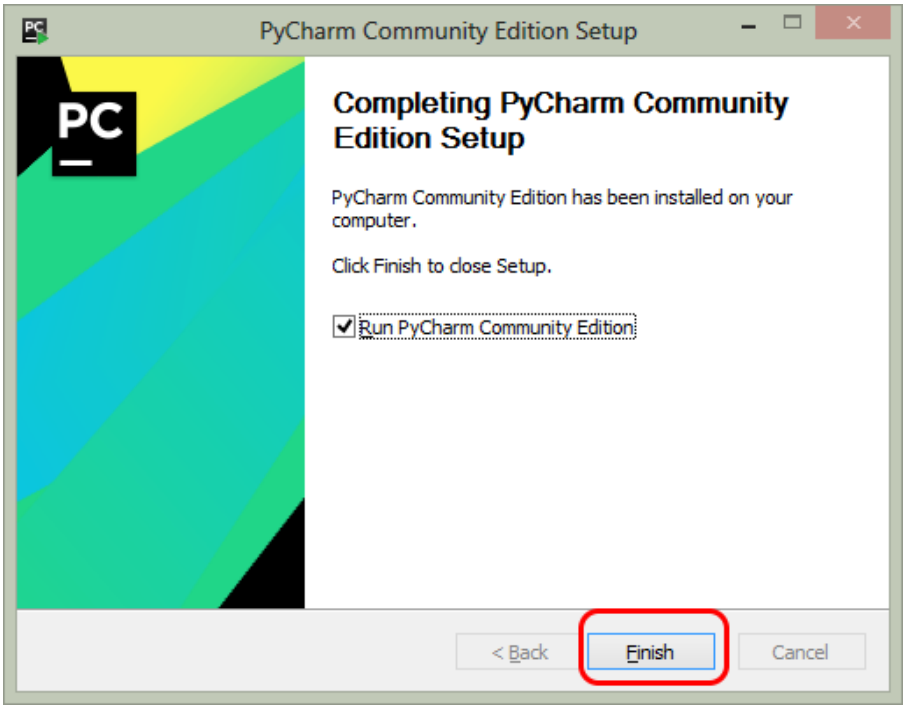


Рис. 2.13: Установка PyCharm

2.4 Упражнения и контрольные вопросы к главе 2

1. Выполните установку Python на свой компьютер.

2. Чем отличается версия Windows x86-64 от Windows x86?

3. Что такое IDE?

4. Что такое PyCharm? Под какими операционнными системами работает PyCharm?

5. Выполните установку PyCharm на свой компьютер.

Глава 3

Первые шаги

3.1 Hello, World!

Напишем первую программу на Python в интегрированной среде разработки программного обеспечения PyCharm. Установка PyCharm описана в главе 2.

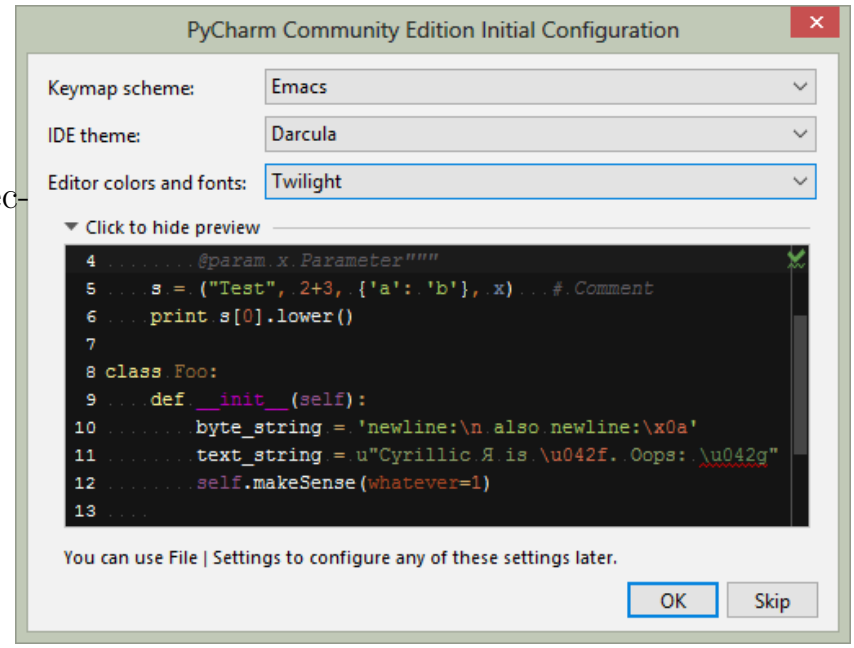


Рис. 3.1: Настройка конфигурации окна PyCharm

При первом запуске PyCharm появляется окно настройки внешнего вида среды (Рис. 3.1). Можно пропустить данный шаг и провести настройку позже.

Программирование в PyCharm подразумевает создание проекта. Поэтому первым шагом написания прораммы в PyCham является создание проекта. Чтобы создать проект, в окне, появившемся после запуска среды PyCharm, выбираем «Create new project» (Рис. 3.2):

Далее вводим название проекта, к примеру, «HelloWorld». Можно также изменить расположение проекта в графе «Location». В графе «Interpreter» следим, чтобы была выбрана версия Python 3 (в случае, если установлены

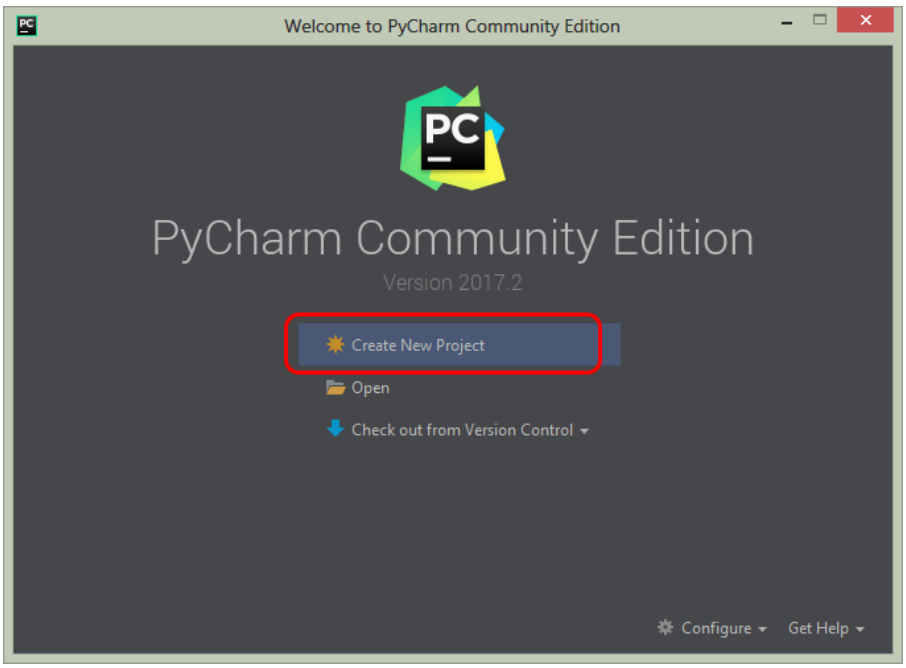


Рис. 3.2: Создание проекта в PyCharm

также другие версии Python) (Рис. 3.3). Далее нажимаем на кнопку «Create».

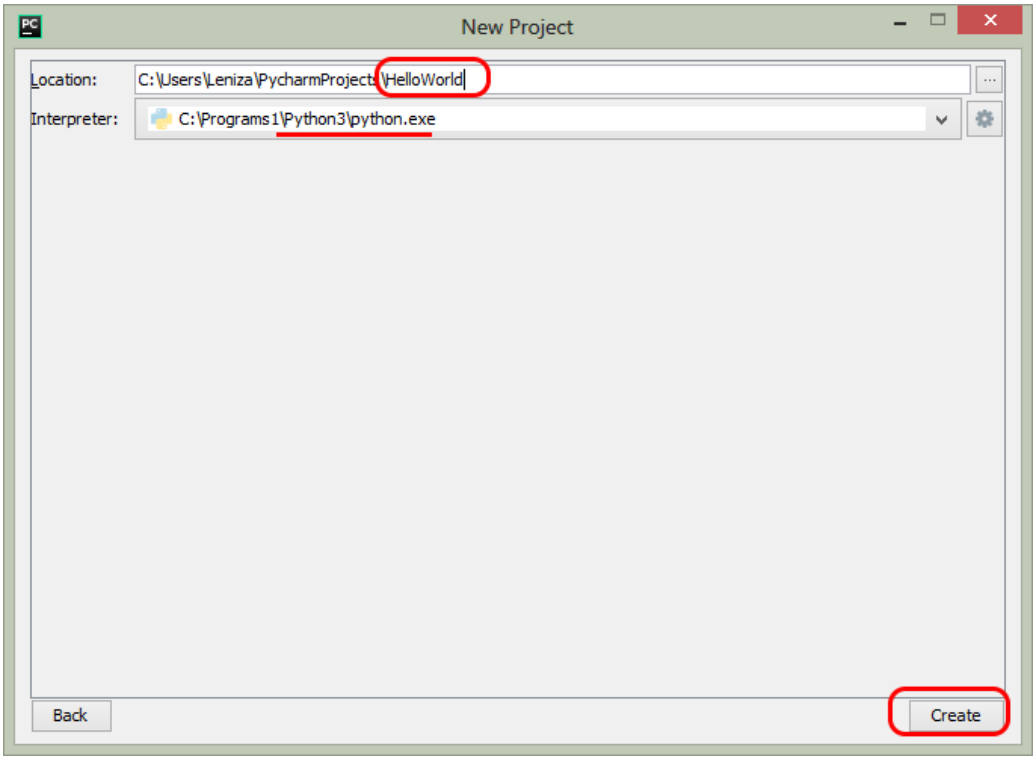


Рис. 3.3: Создание проекта в PyCharm

Открывается окно среды PyCharm. Наводим на проект в левой части окна и нажимаем на правую кнопку мышки. Далее выбираем «New», затем «Python file». Таким образом, создаем новый Python-файл (Рис. 3.4).

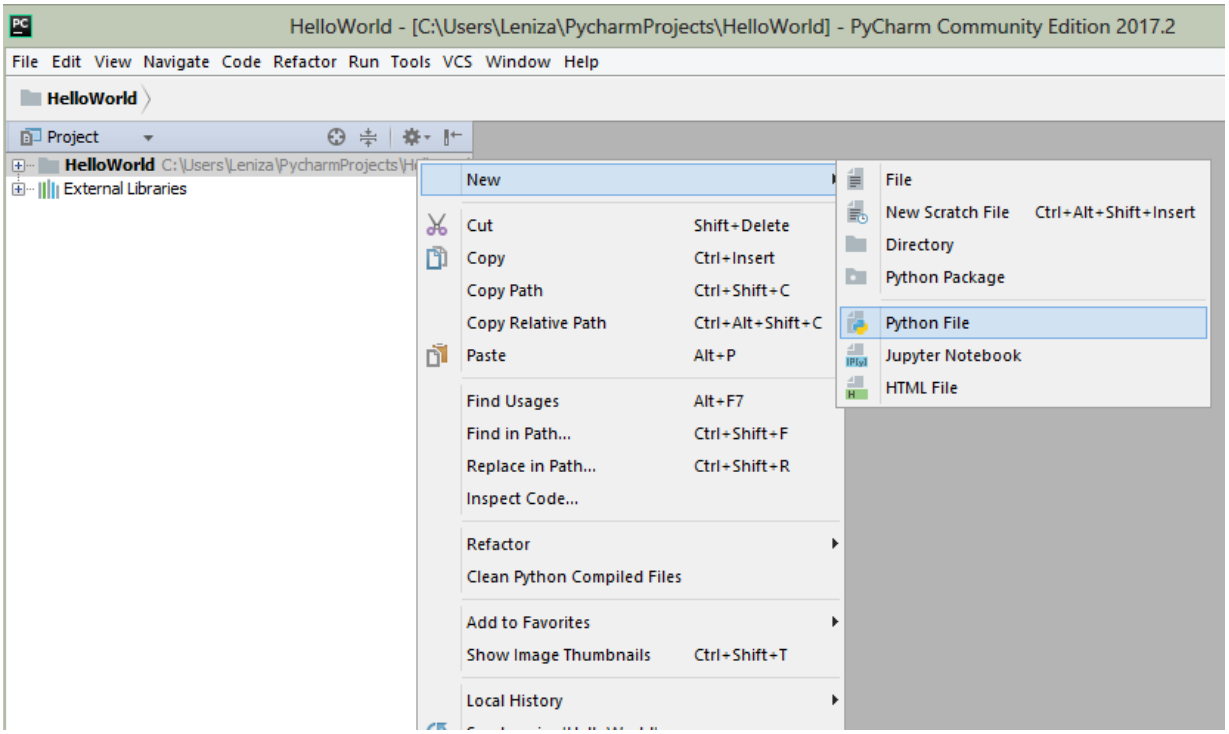


Рис. 3.4: Создание Python-файла

Вводим название файла, к примеру, «demo» (Рис. 3.5). Нажимаем на кнопку «Ok».

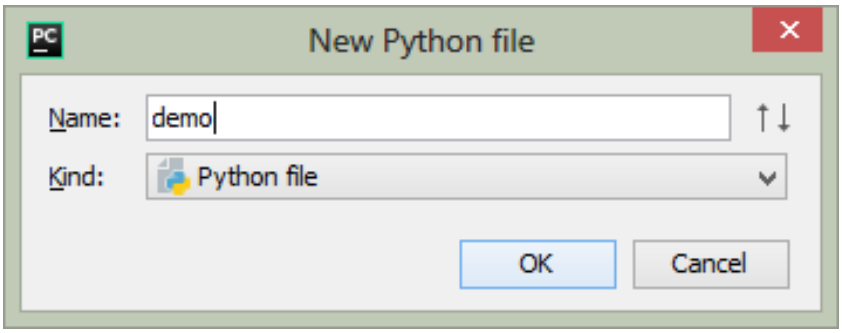


Рис. 3.5: Создание Python-файла

Набираем следующий код:

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

Для запуска программы необходимо навести курсор на строку «demo.py», нажать на правую кнопку мыши, в выпадающем меню выбрать «Run 'demo'» (Рис. 3.6).

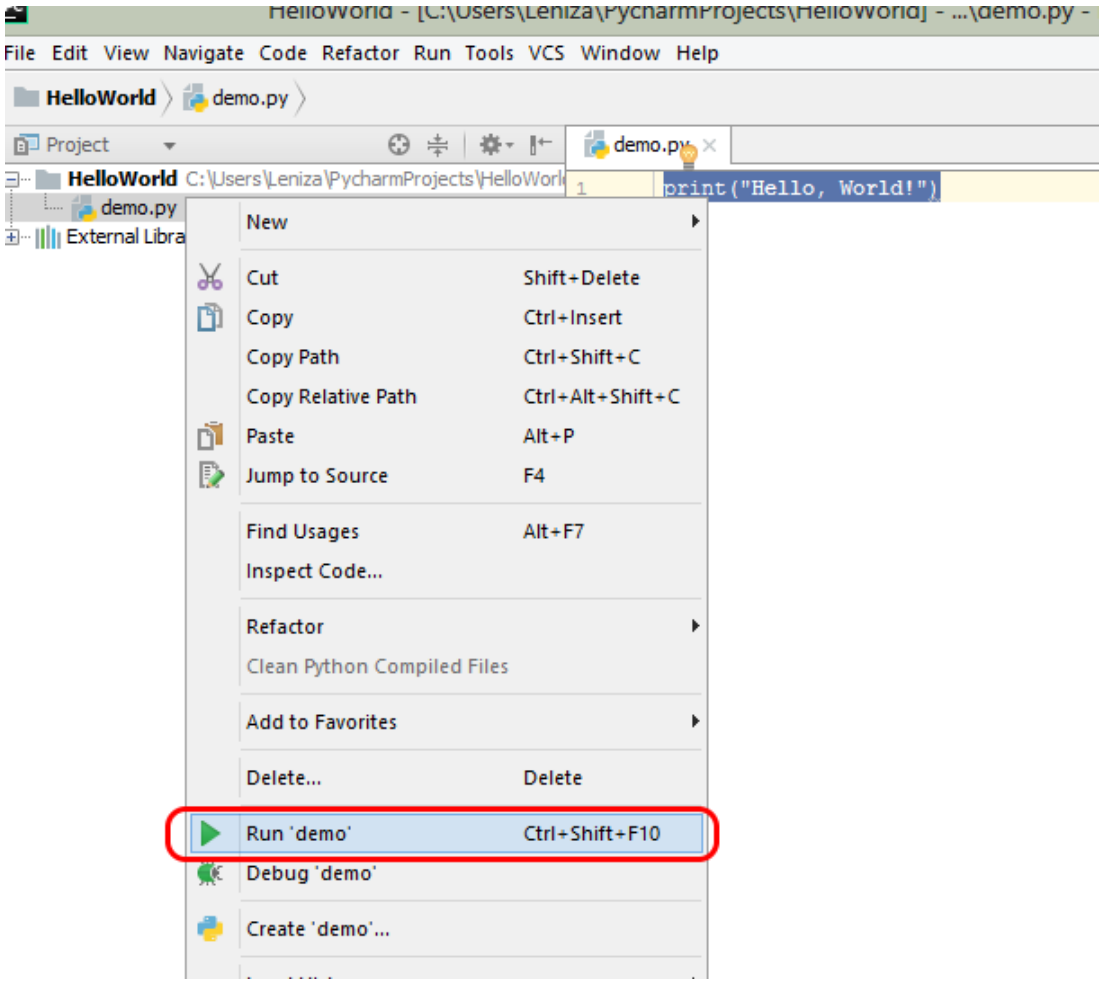


Рис. 3.6: Запуск программы

В нижней части окна отобразится результат выполнения программы (Рис. 3.7).

Теперь наберем следующий код в файле demo.py:

```
print(3 + 4)
print(3*5)
```

Запустим программу. В результате, в нижней части окна отобразятся числа 7 и 15.

3.2 Структура программы

Программа на языке Python представляет собой обычный текстовый файл с инструкциями. Каждая инструкция располагается на отдельной строке. Ес-

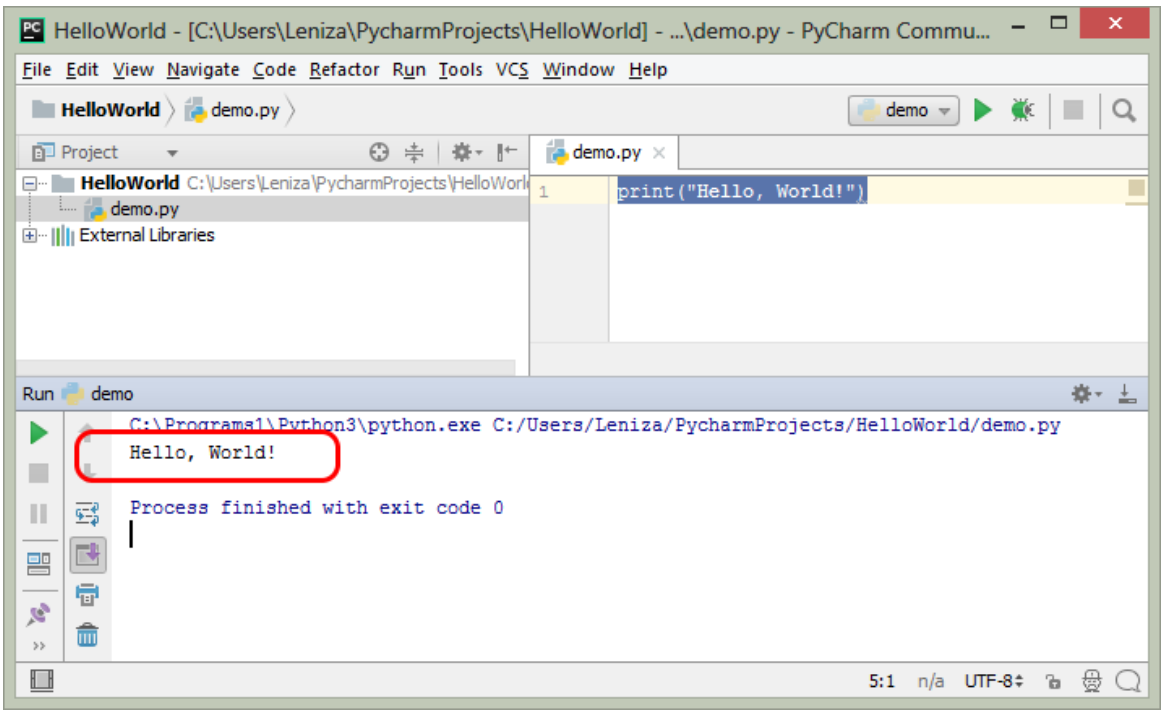


Рис. 3.7: Запуск программы

ли инструкция не является вложенной, то она должна начинаться с начала строки, иначе будет выведено сообщение об ошибке:

```
  import sys
```

```
Результат:
SyntaxError: unexpected indent
```

В данном случае перед инструкцией import расположен один лишний пробел, который привел к выводу сообщения об ошибке.

Во многих языках программирования (например, в PHP, Perl и др.) каждая инструкция должна завершаться точкой с запятой. В языке Python в конце инструкции также можно поставить точку с запятой, но это не обязательно. Более того, в отличие от языка JavaScript, где рекомендуется завершать инструкции точкой с запятой, в языке Python точку с запятой ставить не рекомендуется. Концом инструкции является конец строки. Тем не менее, если необходимо разместить несколько инструкций на одной строке, точку с

запятой следует указать:

```
x = 5; y = 10; z = x + y
print(z)

Результат:
15
```

Еще одной отличительной особенностью языка Python является отсутствие ограничительных символов для выделения инструкций внутри блока. Например, в языке PHP инструкции внутри цикла while выделяются фигурными скобками:

```
$i = 1;
while ($i < 11) {
    echo $i. "\n";
    $i++;
}
echo "Конец программы";
```

В языке Python тот же код будет выглядеть по-другому:

```
i = 1
while i < 11:
    print(i)
    i  +=  1
print ("Конец программы")
```

Перед всеми инструкциями внутри блока расположено одинаковое количество пробелов. Так в языке Python выделяются блоки. Инструкции, перед которыми расположено одинаковое количество пробелов, являются *телом*

блока. В примере выше две инструкции выполняются десять раз. Концом блока является инструкция, перед которой расположено меньшее количество пробелов. В нашем случае это функция print (), которая выводит строку "Конец программы". Если количество пробелов внутри блока окажется разным, то интерпретатор выведет сообщение об ошибке, и программа будет остановлена. Так язык Python приучает программистов писать красивый и понятный код. В языке Python принято использовать четыре пробела для выделения инструкций внутри блока.

Если блок состоит из одной инструкции, то допустимо разместить ее на одной строке с основной инструкцией. Например, код:

```
for i in range(1, 11):
    print(i)
print ("Конец  программы")
```

можно записать так:

```
for i in range(1, 11): print(i)
print("Конец  программы")
```

Если инструкция является слишком длинной, то ее можно перенести на следующую строку, например, в конце строки разместить символ \. После этого символа должен следовать символ перевода строки. Другие символы (в том числе и комментарии) недопустимы.

```
x = 15 + 20 \
+ 30
print(x)
```

Можно также поместить выражение внутри круглых скобок. Этот способ лучше, т. к. внутри круглых скобок можно разместить любое выражение. Например:

```
x = (15 + 20              #  Это комментарий
+ 30)
print(x)
```

Определение списка и словаря можно разместить на нескольких строках, т. к. при этом используются квадратные и фигурные скобки соответственно. Примеры будут приведены в последующих главах.

3.3 Комментарии

Комментарии предназначены для вставки пояснений в текст программы, интерпретатор полностью их игнорирует. Комментарии нужны программисту, а не интерпретатору Python. Вставка комментариев в код позволит через некоторое время быстро вспомнить предназначение фрагмента кода.

В языке Python присутствует только однострочный комментарий. Он начинается с символа #:

```
# Это комментарий
```

Однострочный комментарий может начинаться не только с начала строки, но и располагаться после инструкции. Например, в следующем примере комментарий расположен после инструкции, предписывающей вывести надпись "Привет, мир!":

```
print("Привет, мир!") # Выводим надпись с помощью функции print()
```

3.4 Работа в командной строке

Запуск командной строки Python происходит следующим образом: необходимо на компьютере открыть папку, в которую был установлен Python (см. главу 2) и выполнить двойной щелчок на файле python.exe. Интерактивная оболочка запустится в окне консоли (Рис. 3.8).

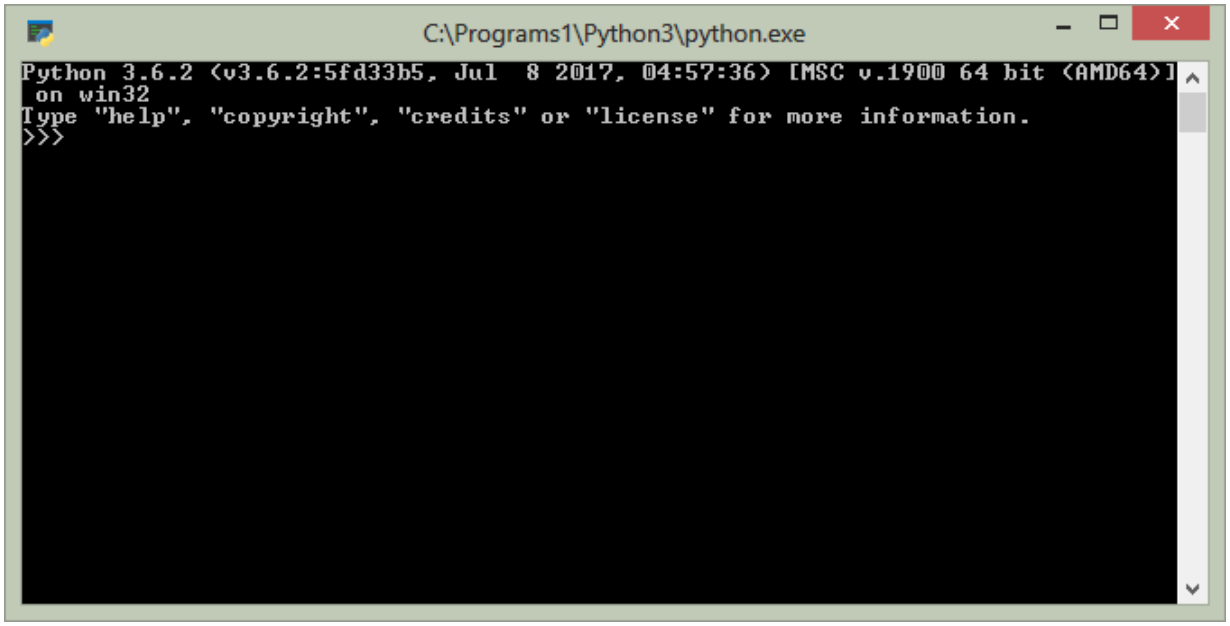


Рис. 3.8: Интерактивная оболочка

Символы >>> в этом окне означают приглашение для ввода инструкций на языке Python. Если после этих символов ввести, например, 2 + 2 и нажать клавишу <Enter>, то на следующей строке сразу будет выведен результат выполнения, а затем опять приглашение для ввода новой инструкции.

При вводе многострочной команды после нажатия клавиши <Enter> редактор автоматически вставит отступ и будет ожидать дальнейшего ввода. Чтобы сообщить редактору о конце ввода команды, необходимо дважды нажать клавишу <Enter>. Пример:

```
>>>  for n in range(1, 3):
         print(n)
1
2
>>>
```

Окно Python Shell можно использовать в качестве многофункционального калькулятора:

```
>>>  5 + 6 * 7
```

```
47
>>>
```

Результат вычисления последней инструкции сохраняется в переменной _ (одно подчеркивание). Это позволяет производить дальнейшие расчеты без ввода предыдущего результата. Вместо него достаточно ввести символ подчеркивания, например:

```
>>>  5 + 6 * 7
47
>>> _ + 50                 # Эквивалентно 47 + 50
97
>>> _ / 2                 # Эквивалентно 97 / 2
48.5
```

3.5 Вывод результатов работы программы

Вывести результаты работы программы можно с помошью функции print (). Функция print () преобразует объект в строку.

После вывода строки автоматически добавляется символ перевода строки:

```
print ("Строка 1")
print ("Строка 2")

Результат:

Строка 1
Строка 2
```

Если необходимо вывести результат на той же строке, то в функции print () данные указываются через запятую:

```
print ("Строка 1", "Строка 2")
```

```
Результат:
```

```
Строка 1 Строка  2
```

При этом между выводимыми строками автоматически вставляется пробел. С помощью параметра sep можно указать другой символ. Например, выведем строки без пробела между ними:

```
print ("Строка 1", "Строка 2", sep="")
```

```
Результат:
```

```
Строка 1Строка 2
```

После вывода объектов в конце добавляется символ перевода строки. Если необходимо произвести дальнейший вывод на той же строке, то в параметре end следует указать другой символ:

```
print ("Строка 1", "Строка 2", end=" ")
print("Строка  3")
```

```
Результат:
```

```
Строка 1 Строка 2 Строка 3
```

Если, наоборот, необходимо вставить символ перевода строки, то функция print () указывается без параметров. Пример:

```
for n in range(1, 5):
```

```
    print(n, end=" ")
print ()
print ("Этот текст на новой строке")
```

```
Результат:

1 2 3 4
Этот текст на новой строке
```

Если необходимо вывести большой блок текста, то его следует разместить между утроенными кавычками или утроенными апострофами. В этом случае текст сохраняет свое форматирование. Пример:

```
print("""Строка 1
Строка 2
Строка 3""")
```

```
Результат:

Строка 1
Строка 2
Строка 3
```

3.6 Ввод данных

Для ввода данных в Python 3 предназначена функция input ().

Запустим PyCHarm и откроем файл demo.py, который мы создавали в разделе 3.1. Наберем в файле следующий код:

```
name = input("Как Вас зовут? ")
print("Привет,", name)
```

Первая строка печатает вопрос («Как Вас зовут?»), ожидает, пока вы не напечатаете что-нибудь и не нажмёте Enter и сохраняет введённое значение в переменной name. Во второй строке мы используем функцию print для вывода текста на экран, в данном случае для вывода «Привет, » и того, что хранится в переменной “name”. Теперь запустим программу и убедимся, что то, что мы написали, работает. Результат программы следующий (Рис. 3.9).

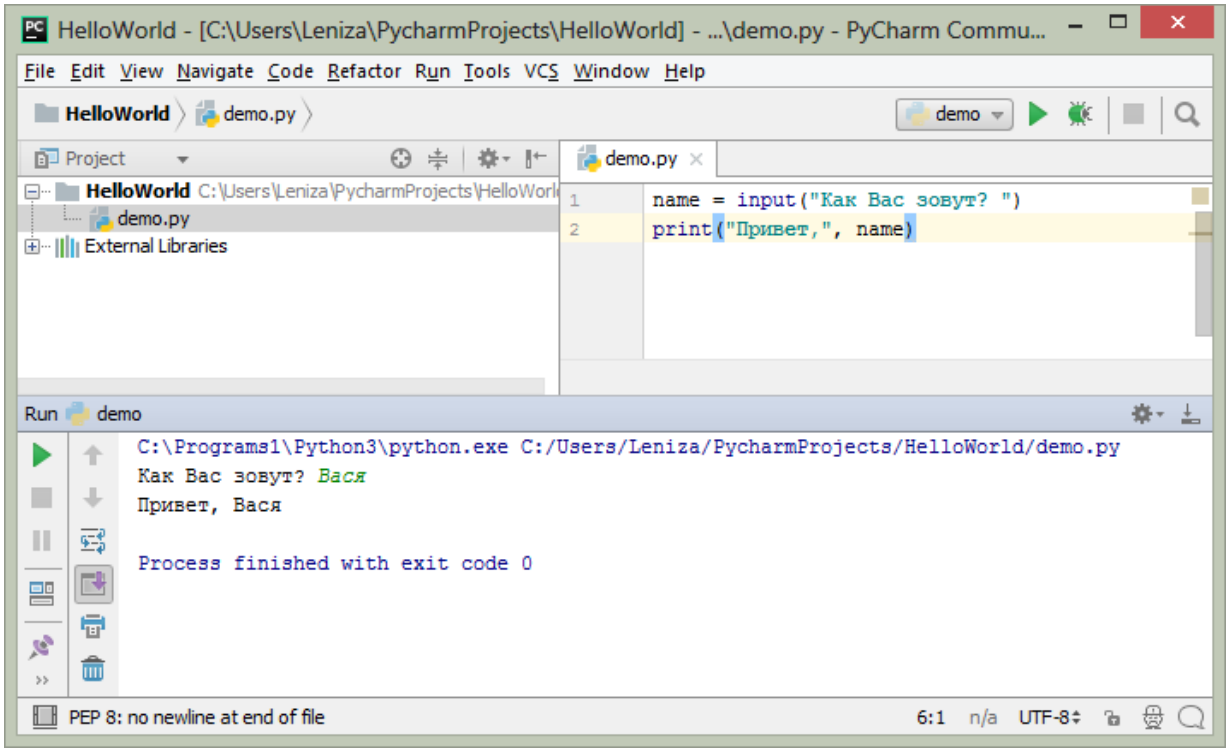


Рис. 3.9: Запуск программы

3.7 Упражнения и контрольные вопросы к главе 3

1. Выполните создание программы «Hello, World» в PyCharm.

2. Какой результат получится, если выполнить следующий код?

```
print(3**2)
```

3. Что представляет из себя программа на языке Python?

4. В результате выполнения программы появляется ошибка:

   ```
   SyntaxError: unexpected indent
   ```

   Как устранить данную ошибку?

5. Какой символ является концом инструкции на языке Python?

6. Можно ли размещать несколько инструкций на одной строке в Python?

7. Как можно запустить командную строку в Python?

8. Какой командой можно вывести строку "Hello, world"в командной строке?

9. Каков будет конечный результат после выполнения следующих двух инструкций?

   ```
   >>>  45 + 48 * 345
   >>> _ / 5
   ```

Глава 4

Переменные

4.1 Типы данных

В Python 3 объекты могут иметь следующие типы данных:

- `bool` – логический тип данных. Может содержать значения `True` или `False`, которые ведут себя как числа 1 и 0 соответственно

```
>>> type (True), type(False)
(<class  'bool'>, <class  'bool'>)
>>> int(True), int(False)
(1, 0)
```

- `NoneType` – объект со значением `None` (обозначает отсутствие значения):

```
>>> type(None)
<class  'NoneType'>
```

В логическом контексте значение `None` интерпретируется как `False`:

```
>>> bool(None)
False
```

- `int` – целые числа. Размер числа ограничен лишь объемом оперативной памяти:

```
>>> type(2147483647), type(99999999999999999999999)
(<class 'int'>, <class 'int'>)
```

- `float` – вещественные числа:

```
>>> type(5.1), type(8.5e-3)
(<class 'float'>, <class 'float'>)
```

- `complex` – комплексные числа:

```
>>> type (2 + 2j)
<class  'complex'>
```

- `str` – Unicode-строки:

```
>>> type ("Строка")
<class 'str'>
```

- `bytes` – неизменяемая последовательность байтов:

```
>>> type (bytes ("Строка", "utf-8"))
<class 'bytes'>
```

- `bytearray` – изменяемая последовательность байтов:

```
>>> type (bytearray ("Строка", "utf-8"))
<class 'bytearray'>
```

- `list` – списки. Тип данных `list` аналогичен массивам в других языках программирования:

```
>>> type ([1, 2, 3])
<class  'list'>
```

- `tuple` – кортежи:

```
>>> type ((1, 2, 3))
<class 'tuple'>
```

- `range` – диапазоны:

```
>>> type (range(1, 10))
<class 'range'>
```

- `dict` – словари. Тип данных `dict` аналогичен ассоциативным массивам в других языках программирования:

```
>>>  type ({"x": 5, "y": 20})
<class 'dict'>
```

- `set` – множества (коллекции уникальных объектов):

```
>>> type ({"a", "b", "c"})
<class 'set'>
```

- `frozenset` – неизменяемые множества:

```
>>> type (frozenset (["a", "b", "c"]))
<class 'frozenset'>
```

- `function` – функции:

```
>>> def func(): pass
>>>  type(func)
<class 'function'>
```

- `module` – модули:

```
>>> import sys
>>> type(sys)
<class 'module'>
```

- `type` – классы и типы данных. Все данные в языке Python являются объектами, даже сами типы данных!

```
>>> class C: pass
>>> type(C)
<class 'type'>
>>> type (type (""))
<class 'type'>
```

Основные типы данных делятся на *изменяемые* и *неизменяемые*. К изменяемым относятся списки, словари и тип `bytearray`. Пример изменения элементся списка:

```
>>> arr = [1, 2, 3]
>>> arr [0] = 0      # Изменяем первый элемент списка
>>> arr
[0, 2, 3]
```

К неизменяемым типам относятся числа, строки, кортежи, диапазоны и тип `bytes`. Например, чтобы получить строку из двух других строк, необходимо использовать операцию *конкатенации*, а ссылку на новый объект присвоить переменной:

```
>>> str1 = "авто"
>>> str2 = "транспорт"
>>> str3 = str1 + str2     # Конкатенация
>>> print(str3)
автотранспорт
```

Кроме того, типы данных делятся на последовательности и отображения. К последовательностям относятся строки, списки, кортежи, диапазоны, типы `bytes` и `bytearray`, а к отображениям – словари. Последовательности и отображения поддерживают механизм итераторов, позволяющий произвести обход всех элементов с помощью метода `__next__()` или функции `next()`.

Подробнее с типами данных можно ознакомиться в книге [4].

4.2 Присваивание значения переменной

В языке Python используется *динамическая типизация*. Это означает, что при присваивании переменной значения интерпретатор автоматически относит переменную к одному из типов данных. Значение переменной присваивается с помощью оператора = следующим образом:

```
>>> x = 7    # Тип int
>>> y = 7.8   # Тип float
>>> s1 = "Строка"    # Переменной s1 присвоено значение Строка
>>> s2 = 'Строка'    # Переменной s2  также  присвоено значение Строка
>>> b = True    # Переменной b присвоено логическое значение True
```

В одной строке можно присвоить значение сразу нескольким переменным:

```
>>> x = y = 10
>>> x, y
(10, 10)
```

После присваивания значения в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при групповом присваивании. Групповое присваивание можно использовать для чисел, строк и кортежей, но для изменяемых объектов этого делать нельзя. Пример:

```
>>> x = y = [1, 2]    # Якобы создали два объекта
```

```
>>> x, y
([1, 2], [1, 2])
```

В этом примере мы создали список из двух элементов и присвоили значение переменным `x` и `y`. Теперь попробуем изменить значение в переменной `y`:

```
>>> y[1] = 100    # Изменяем второй элемент
>>> x, y
([1, 100], [1, 100])
```

Как видно из примера, изменение значения в переменной `y` привело также к изменению значения в переменной `x`. Таким образом, обе переменные ссылаются на один и тот же объект, а не на два разных объекта. Чтобы получить два объекта, необходимо производить раздельное присваивание:

```
>>> x = [1, 2]
>>> y = [1, 2]
>>> y[1] = 100
>>> x, y
([1, 2], [1, 100])
```

Язык Python также поддерживает позиционное присваивание. В этом случае переменные указываются через запятую слева от оператора `=`, а значения – через запятую справа. Пример позиционного присваивания:

```
>>> x, y, z = 1, 2, 3
>>> x, y, z
(1, 2, 3)
```

С помощью позиционного присваивания можно поменять значения переменных местами. Пример:

```
>>> x, y = 1, 2; x, y
(1, 2)
>>> x, y = y, x; x, y
(2, 1)
```

По обе стороны оператора = могут быть указаны последовательности. Пример:

```
>>> x, y, z = "123"   # Строка
>>> x, y, z
('1', '2', '3')
>>> x, y, z = [1, 2, 3]    # Список
>>> x, y, z
(1, 2, 3)
>>> x, y, z = (1, 2, 3)    # Кортеж
>>> x, y, z
(1, 2, 3)
>>>[x, y, z] = (1, 2, 3)    # Список слева, кортеж справа
>>> x, y, z
(1, 2, 3)
```

4.3 Проверка типа данных

Python изменяет тип переменной в соответствии с данными, хранящимися в ней. Пример:

```
>>> a = "Строка"   # Тип str
>>> a = 7          # Теперь переменная имеет тип int
```

Определить, на какой тип данных ссылается переменная, позволяет функция

`type(<Имя переменной>)`:

```
>>> type(a)
<class 'int'>
```

Проверить тип данных, хранящихся в переменной, можно следующими способами:

- сравнить значение, возвращаемое функцией `type()`, с названием типа данных:

```
>>> x = 10
>>> if type(x) == int:
        print("Это тип int")
```

- проверить тип с помощью функции `isinstance()`:

```
>>> s = "Строка"
>>> if isinstance(s, str):
        print("Это тип str")
```

4.4 Преобразование типов данных

Для преобразования типов данных предназначены следующие функции:

- `bool([<Объект>])` – преобразует объект в логический тип данных. Примеры:

```
>>> bool(0), bool(1), bool("")
(False, True, False)
>>> bool("Строка"), bool([1, 2]),  bool([])
(True, True, False)
```

- `int([<Объект> [,<Система счисления>]])` – преобразует объект в число. Во втором параметре можно указать систему счисления (значение по умолчанию – 10). Примеры:

```
>>> int(7.5), int("71")
(7, 71)
>>> int("71", 10), int("71", 8), int("0o71", 8), int("A", 16)
(71, 57, 57, 10)
```

  Если преобразование невозможно, то генерируется исключение.

- `float([<Число или строка>])` – преобразует целое число или строку в вещественное число. Примеры:

```
>>> float(7), float("7.1")
(7.0,  7.1)
>>> float("Infinity"), float("-inf")
(inf, -inf)
>>> float("Infinity") + float("-inf")
nan
```

- `str([<Объект>])` – преобразует объект в строку. Примеры:

```
>>> str(125), str([1, 2, 3])
('125', '[1, 2, 3]')
>>> str((1, 2, 3)), str({"x": 5, "y": 10})
('(1, 2, 3)',  "{'x': 5, 'y': 10}")
>>> str (bytes ("строка", "utf-8"))
"b'\xd1\x81\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'"
>>> str (bytearray("строка", "utf-8"))
"bytearray(b'c\xd1\x82po\xd0\xbaa')"
```

- `str (<Объект>[,<Кодировка>[,<Обработка ошибок>]])` – преобразует объект типа `bytes` или `bytearray` в строку. В третьем параметре можно задать значение `"strict"` (при ошибке возбуждается исключение `UnicodeDecodeError` – значение по умолчанию), `"replace"` (неизвестный символ заменяется символом, имеющим код `\uFFFD`) или `"ignore"` (неизвестные символы игнорируются). Примеры:

```
>>> obj1 = bytes("строка1", "utf-8")
>>> obj2 = bytearray("строка2", "utf-8")
>>> str(obj1, "utf-8"), str(obj2, "utf-8")
('строка1', 'строка2')
>>> str(obj1, "ascii", "strict")
Traceback (most recent call last):
UnicodeDecodeError: 'ascii' codec can't decode byte
>>> str(obj1, "ascii", "ignore")
'1'
```

- `bytes (<Строка>, <Кодировка> [,<Обработка ошибок>])` – преобразует строку в объект типа `bytes`. В третьем параметре могут быть указаны значения `"strict"` (значение по умолчанию), `"replace"` или `"ignore"`. Примеры:

```
>>> bytes ("строка", "cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
>>> bytes("строка123", "ascii", "ignore")
b'123'
```

- `bytes (<Последовательность>)` – преобразует последовательность целых чисел от 0 до 255 в объект типа `bytes`. Если число не попадает в диапазон, то возбуждается исключение `ValueError`. Пример:

```
>>> b = bytes ([225, 226, 224, 174, 170, 160])
>>> b
b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
'строка'
```

- `list (<Последовательность>)` – преобразует элементы последовательности в список. Примеры:

```
>>> list ("12345")     # преобразование строки
['1', '2', '3', '4', '5']
>>> list ((1, 2, 3, 4, 5))    # преобразование кортежа
[1, 2, 3, 4, 5]
```

- `tuple (<Последовательность>)` – преобразует элементы последовательности в кортеж:

```
>>> tuple ("123456")     #  Преобразование строки
('1', '2', '3', '4', '5', '6')
>>> tuple([1,  2,  3,  4,  5])    # Преобразование списка
(1, 2, 3, 4, 5)
```

Рассмотрим сложение двух чисел, введенных пользователем. Вводить данные позволяет функция `input ()`. Воспользуемся этой функцией для получения чисел от пользователя.

```
x = input("x = ")    # Вводим 5
y = input("y = ")    # Вводим 12
print (x + y)
```

Результатом выполнения этого скрипта будет не число, а строка 512. Таким образом, функция input () возвращает результат в виде строки. Чтобы просумми ровать два числа, необходимо преобразовать строку в число:

```
x = int(input("x = "))    # Вводим 5
y = int(input("y = "))    # Вводим 12
print (x + y)
```

В этом случае мы получим число 17, как и должно быть.

4.5 Удаление переменной

Удалить переменную можно с помощью инструкции del:

```
del <Переменная1>[, ..., <ПеременнаяN>]
```

Пример удаления одной переменной:

```
>>> x = 10; x
10
>>> del x;
```

Пример удаления нескольких переменных:

```
>>> x, y = 10, 20
>>> del x, y
```

4.6 Упражнения и контрольные вопросы к главе 4

1. Назовите базовые типы данных в языке Python?

2. Как в логическом контексте интерпретируется значение None?

3. Какой результат получим после выполнения следующей команды?

```
type(5.9e+4)
```

4. Какая команда применяется для удаления переменной в Python? Как выполняется удаление нескольких переменных одной инструкцией?

Глава 5

Операторы, условные операторы и циклы

Операторы позволяют произвести с данными определенные действия. Рассмотрим некоторые операторы, доступные в Python 3.

5.1 Операторы присваивания

Операторы присваивания предназначены для сохранения значения в переменной.

- `=` – присваивает переменной значение:

```
>>> x = 5; x
5
```

- `+=` – увеличивает значение переменной на указанную величину:

```
>>> x = 5; x += 10   # Эквивалентно x = x + 10
15
```

  Для последовательностей оператор `+=` производит конкатенацию.

- `-=` – уменьшает значение переменной на указанную величину; `*=` – умножает значение переменной на указанную величину; `/=` – делит значение переменной на указанную величину; `//=` – делит с округлением вниз на указанную величину; `%=` – делит по модулю на указанную величину; `**=` – возводит в степень на указанную величину.

5.2 Приоритет выполнения операторов

Перечислим операторы в порядке убывания приоритета:

1. `-x, +x, ~x, **` – унарный минус, унарный плюс, двоичная инверсия, возведение в степень. Если унарные операторы расположены слева от оператора `**`, то возведение в степень имеет больший приоритет, а если справа – то меньший. Например, выражение: `-10 ** -2` эквивалентно следующей расстановке скобок: `-(10 ** (-2))`
2. `*, %, /, //` – умножение (повторение), остаток от деления, деление, деление с округлением вниз.
3. `+, -` – сложение (конкатенация), вычитание.
4. `<<, >>` – двоичные сдвиги.
5. `&` – двоичное И.
6. `^` – двоичное исключающее ИЛИ.
7. `|` – двоичное ИЛИ.
8. `+, +=, -=, *=, /=, //=, %=, **=` – присваивание.

Подробнее про двоичные операторы в книге [5].

5.3 Математические операторы

- $+$, $-$, $*$, $/$ – оператор сложения, вычитания, умножения и деления соответственно. Результатом деления всегда является вещественное число, даже если производится деление целых чисел.
- `//` – деление с округлением вниз. Вне зависимости от типа чисел остаток отбрасывается. Примеры:

```
>>> 10 // 3
3
```

- % – остаток от деления:

```
>>> 10 % 3
1
```

- ** – возведение в степень:

```
>>> 10**2, 10.0**2
(100,100.0)
```

- унарный минус (−) и унарный плюс (+):

```
>>> +10, +10.0, -10, -(-10)
(10, 10.0, -10, 10)
```

5.4 Операторы для работы с последовательностями

- + – конкатенация:

```
>>> print ("Строка1" + "Строка2")   # конкатенация строк
Строка1Строка2
>>> [1, 2, 3] + [4, 5, 6]   # списки
[1, 2, 3, 4, 5, 6]
>>> (1, 2, 3) + (4, 5, 6)   # кортежи
(1, 2, 3, 4, 5, 6)
```

- * – повторение:

```
>>> "s" * 10        # строки
'ssssssssss'
>>> [1, 2] * 2   # списки
[1, 2, 1, 2]
>>> (1, 2) * 2   # кортежи
(1, 2, 1, 2)
```

- `in` – проверка на вхождение. Если элемент входит в последовательность, то возвращается логическое значение `True`:

```
>>> "Солнце" in "Солнце светит ярко"        # строки
True
>>> "Луна" in "Солнце светит ярко"          # строки
False
>>> 2 in [1, 2, 3]   # списки
True
>>> 6 in (1, 2)      # кортежи
False
```

- `not in` – проверка на невхождение. Если элемент не входит в последовательность, то возвращается логическое значение `True`.

5.5 Операторы сравнения

- `==` – равно:

```
>>> 2 == 2, 6 == 5
(True, False)
```

- `!=` – не равно, `<` – меньше, `>` – больше, `<=` – меньше или равно, `>=` – больше или равно:

```
>>> 1 >= 1, 1 > 5
(True, False)
```

- `is` – проверяет, ссылаются ли две переменные на один и тот же объект. Если переменные ссылаются на один и тот же объект, то оператор `is` возвращает значение `True`:

```
>>> x = y = [1, 2]
>>> x is y
True
>>> x = [1, 2]; y = [1, 2]
>>> x is y
False
```

- `is not` – проверяет, ссылаются ли две переменные на разные объекты. Если это так, то оператор возвращает значение `True`.

- `and` – логическое И:

```
1 < 5 and 2 < 5    # True and True = True
True
```

- `or` – логическое ИЛИ:

```
1 < 5 and 2 > 5    # True or False = True
True
```

Значение логического выражения можно инвертировать с помощью оператора `not`:

```
>>> x = 1; y = 1
>>> not (x == y), not x == y
(False, False)
```

В логическом выражении можно указывать сразу несколько условий:

```
>>> x = 10
>>> 1 < x < 20, 11 < x < 20
(True, False)
```

Перечислим операторы сравнения в порядке убывания приоритета:

1. `<`, `>`, `<=`, `>=`, `==`, `!=`, `<>`, `is`, `is not`, `in`, `not in`.
2. `not` – логическое отрицание.
3. `and` – логическое И.
4. `or` – логическое ИЛИ.

5.6 Условная инструкция if-elif-else

Приведем пример оператора ветвления `if-else`:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

Блоки внутри составной инструкции выделяются одинаковым количеством пробелов (обычно четырьмя пробелами). Концом блока является инструкция, перед которой расположено меньшее количество пробелов.

Больше примеров на условную инструкцию `if` и синтаксические правила в 12-ой главе книги [6].

5.7 Цикл For

Этот цикл проходится по любому итерируемому объекту (например строке или списку), и во время каждого прохода выполняет тело цикла.

```
for i in 'hello world':
    print(i * 2, end='')


hheellllloo wwoorrlldd
```

5.8 Цикл While

Выполнение инструкций в цикле `while` продолжается до тех пор, пока логическое выражение истинно. Пример цикла:

```
i = 5
while i < 10:
    print(i)
    i = i + 2


5
7
9
```

5.9 Оператор continue

Оператор `continue` начинает следующий проход цикла, минуя оставшееся тело цикла (`for` или `while`):

```
for i in 'hello world':
    if i == 'o':
```

```
        continue
    print(i * 2, end = '')
```

```
hheellll  wwrrlldd
```

5.10 Оператор break

Оператор `break` досрочно прерывает цикл [7]:

```
for i in 'hello world':
    if i == 'o':
        break
print(i * 2, end = '')
```

```
hheellll
```

5.11 Модуль math. Математические функции

Модуль `math`предоставляет дополнительные функции для работы с числами, а также стандартные константы. Прежде чем использовать модуль, необходимо подулючить его с помощью инструкции [5, 8]:

```
import  math
```

Модуль `math` предоставляет следующие стандартные константы:

- `pi` – возвращает число π:

```
import math
math.pi
```

```
3.141592653589793
```

- `e` – возвращает значение константы e:

```
math.e
```

```
2.718281828459045
```

Перечислим основные математические функции:

- `sin()`, `cos()`, `tan()` – тригонометрические функции (синус, косинус, тангенс). Значение указывается в радианах;
- `asin()`, `acos()`, `atan()` – обратные тригонометрические функции (арксинус, арккосинус, арктангенс). Значение возвращается в радианах;
- `degrees()` – преобразует радианы в градусы:

```
math.degrees(math.pi)
```

```
180.0
```

- `radians()` – преобразует градусы в радианы:

```
math.radians(180.0)
```

```
3.141592653589793
```

- `exp()` – экспонента;
- `log(<Число>[, <Основание>])` – логарифм по заданному основанию. Если основание не указано, то вычисляется натуральный логарифм;
- `log10()` – десятичный логарифм;
- `log2()` – логарифм по основанию 2;

- `sqrt()` – квадратный корень;
- `ceil()` – значение, округленное до ближайшего большего целого:

```
math.ceil(5.39), math.ceil(5.51)
```

```
6, 6
```

- `floor()` – значение, округленное до ближайшего меньшего целого;
- `pow(<Число>, <Степень>)` – возводит число в степень;
- `fabs()` – абсолютное значение (модуль числа);
- `fmod()` – остаток от деления:

```
math.fmod(10,5), math.fmod(10,3)    # эквивалентно 10 % 5, 10 %3
```

```
0.0, 1.0
```

- `factorial()` – факториал числа.

5.12 Модуль random. Генерация глучайных чисел

Модуль `random` позволяет генерировать случайные числа. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
import random
```

Перечислим основные функции модуля:

- `random()` – возвращает псевдослучайное число от 0 до 1:

```
import random
random.random()

0.6247985122932341
```

- `seed()` – настраивает генератор случайных чисел на новую последовательность. Если параметр не указан, в качестве базы для случайных чисел будет использовано системное время. Если значение параметра будет одинаковым, то генерируется одинаковое число:

```
random.seed(2); random.random(); random.seed(2); random.random()

0.9560342718892494
0.9560342718892494
```

- `uniform(<Начало>,<Конец>)` – возвращает псевдослучайное вещественное число в диапазоне от <Начало> до <Конец>:

```
random.uniform(1,10)

1.2696044276812932
```

- `randint(<Начало>,<Конец>)` – возвращает псевдослучайное целое число в диапазоне от <Начало> до <Конец>:

```
random.randint(1,10)

5
```

- `randrange([< Начало>, ]<Конец> [, <Шаг>])` – возвращает случайный элемент из числовой последовательности:

```
random.randrange(3),random.randrange(0,5),random.randrange(1,10,3)
```

```
1,3,7
```

- `choice(<Последовательность>)` – возвращает случайный элемент из заданной последовательности (строки, списка, кортежа):

```
random.choice("hello")
```

```
e
```

- `shuffle(<Список>)` – перемешивает элементы списка случайным образом:

```
arr = [1,2,3,4,5]
random.shuffle(arr)
arr
```

```
[1, 4, 2, 3, 5]
```

- `sample(<Последовательность>, <Количество элементов>)` – возвращает список из указанного количества элементов, которые будут выбраны случайным образом из заданной последовательности:

```
random.sample("hello",3)
```

```
['h', 'o', 'l']
```

5.13 Упражнения и контрольные вопросы к главе 5

1. Для чего нужны операторы?

2. Какой результат получится после выполнения следующего кода:

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

3. Какой результат получится после выполнения следующего кода:

```
>>> bool(-20)
```

4. Какой результат получится после выполнения следующего кода:

```
>>> bool(0.1)
```

5. Какой результат получится после выполнения следующего кода:

```
>>> bool("0")
```

6. Какой результат получится после выполнения следующего кода:

```
>>> bool(0.0)
```

7. Какой результат получится после выполнения следующего кода:

```
>>> bool("")
```

8. Какой результат получится после выполнения следующего кода:

```
>>> bool([])
```

9. Какой результат получится после выполнения следующего кода:

```
>>> bool(())
```

10. Какой результат получится после выполнения следующего кода:

```
>>> bool(None)
```

11. Какой результат получится после выполнения следующего кода:

```
>>> 20 and 10
```

12. Какой результат получится после выполнения следующего кода:

```
>>> 10 and 20
```

13. Какой результат получится после выполнения следующего кода:

```
>>> 1 > 5 or 2 < 5
```

14. Какой результат получится после выполнения следующего кода:

```
>>> 10 or 20
```

15. Какой результат получится после выполнения следующего кода:

```
>>> 20 or 0
```

16. Какой результат получится после выполнения следующего кода:

```
>>> math.pow(11,2)
```

17. Какой результат получится после выполнения следующего кода:

```
>>> math.factorial(5)
```

18. Какие возможные результаты выполнения следующего кода:

```
>>> random.choice(["s","t","r"])
```

19. Какие возможные результаты выполнения следующего кода:

```
>>> random.choice(("p","r","s"))
```

Глава 6

Работа со строками в Python

Строки в Python – упорядоченные последовательности символов, используемые для хранения и представления текстовой информации. Строки относятся к неизменяемым типам данных. Поэтому практически все строковые методы в качестве значения возвращают новую строку. Язык Python 3 поддерживает следующие строковые типы: `str` – Unicode-строка, `bytes` – неизменяемая последовательность байтов, `bytearray` – изменяемая последовательность байтов. Типы `bytes` и `bytearray` следует задействоватъ для записи бинарных данных – например, изображений, а также для промежуточного хранения текстовых данных.

6.1 Литералы строк

Строки в апострофах и в кавычках – одно и то же. Причина наличия двух вариантов в том, чтобы позволить вставлять в литералы [9] строк символы кавычек или апострофов, не используя экранирование [10].

Экранированные последовательности позволяют вставить символы, которые сложно ввести с клавиатуры, некоторые из них представлены в таблице 6.1.

6.2 Операции над строками

Некоторые операции (конкатенация, повторение, проверки на вхождение и на невхождение) были приведены разделе 5.4. Познакомимся с еще несколькими:

Таблица 6.1: Экранированные последовательности

Экранированная последовательность	Назначение
\n	Перевод строки
\t	Знак табуляции
\"	Кавычка
\\	Обратный слеш
\0	Нулевой символ (не является концом строки)

- Длина строки (функция `len`):

```
len('Python')
```

```
6
```

- Доступ по индексу (нумерация начинается с нуля):

```
s = 'Python'
s[0]
```

```
'P'
```

В Python возможен и доступ по отрицательному индексу, при этом отсчет идет от конца строки:

```
s = 'Python'
s[-2]
```

```
'o'
```

- Извлечение среза – `[<Начало>:<Конец>:<Шаг>]` – все парметры являются необязательными:

```
s = 'Python'
```

```
s[:]
```

```
'Python'
```

Выведем символы в обратном порядке:

```
s = 'Python'
s[::-1]     # Указываем отрицательное значение в параметре <Шаг>
```

```
'nohtyP'
```

К строкам применяются операции форматирования, подробнее в книгах [5, 7, 6].

6.3 Функции и методы для работы со строками

В таблице 6.2 приведем описание некоторых строковых методов в версии Python 3.0.

Приведем примеры исопльзования данных методов.

- ```
  s = "strstrstrokstrstrstr"
  s.strip("tsr")
  ```

  ```
  'ok'
  ```

- ```
  s = "str1 str2 str3"
  s.split()
  ```

  ```
  ['str1','str2','str3']
  ```

- ```
  " => ".join(["str1", "str2", "str3"])
  ```
  ```

Таблица 6.2: Строковые методы

`S.capitalize()`	Переводит первый символ строки в верхний регистр, а все остальные в нижний
`S.swapcase()`	Переводит символы нижнего регистра в верхний, а верхнего – в нижний
`S.title()`	Первую букву каждого слова переводит в верхний регистр, а все остальные в нижний
`S.upper()`	Преобразование строки к верхнему регистру
`S.lower()`	Преобразование строки к нижнему регистру
`S.isupper()`	Состоит ли строка из символов в верхнем регистре
`S.islower()`	Состоит ли строка из символов в нижнем регистре
`S.lstrip()`	Удаление пробельных символов в начале строки
`S.rstrip()`	Удаление пробельных символов в конце строки
`S.strip()`	Удаление пробельных символов в начале и в конце строки
`S.strip(<Символы>)`	Удаление указанных символов в начале и в конце строки
`S.find(str,[start],[end])`	Поиск подстроки в строке. Возвращает номер первого вхождения или -1. Метод зависит от регистра
`S.rfind(str,[start],[end])`	Поиск подстроки в строке. Возвращает номер последнего вхождения или -1
`S.count(str,[start],[end])`	Возвращает количество непересекающихся вхождений подстроки в диапазоне [начало, конец] (0 и длина строки по умолчанию)
`S.replace(шаблон,замена)`	Замена шаблона
`S.split(символ)`	Разбиение строки по разделителю. Если разделитель не указан, то используется символ пробела
`S.isdigit()`	Состоит ли строка из цифр
`S.isalpha()`	Состоит ли строка из букв
`S.isalnum()`	Состоит ли строка из цифр или букв
`S.join(<Список>)`	Сборка строки из списка с разделителем S

```
'str1 => str2 => str3'
```

- ```
  s = "string1 sttring2 String3"
  s.find("str"), s.find("Str"), s.find("123")
  ```

  ```
  (0,16,-1)
  ```

- ```
  s = "Hello, Tom"
  s.replace("Tom","Bob")
  ```

```
'Hello, Bob'
```

6.4 Упражнения и контрольные вопросы к главе 6

1. Что представляют собой строки в Python?

2. Какая последовательность символ предназначена для знака табуляции?

3. Какая последовательность символ предназначена для перевода строки?

4. Какой результат получим после выполнения следующего кода:

```
s = 'String'
s[:-1]
```

5. Какой результат получим после выполнения следующего кода:

```
s = 'String'
J + s[1:]
```

6. Какой результат получим после выполнения следующего кода:

```
s = 'String'
s[0:1]
```

7. Какой результат получим после выполнения следующего кода:

```
s = 'String'
s[-1:]
```

8. Какой результат получим после выполнения следующего кода:

```
s = 'String'
s[2:5]
```

9. Какой результат получим после выполнения следующего кода:

```
len("\n\t")
```

10. Какой результат получим после выполнения следующего кода:

```
print("string1" "string2")
```

11. Какой результат получим после выполнения следующего кода:

```
s = "str1 str2 str3"
s.title()
```

12. Какой результат получим после выполнения следующего кода:

```
s = "str1 str2 Str3 Str4"
s.count("str")
```

Глава 7

Списки, кортежи

Списки в Python – упорядоченные изменяемые наборы объектов произвольных типов (почти как массив, но типы могут отличаться).

7.1 Создание списков

Создать список можно несколькими способами. Например, можно обработать любой итерируемый объект (например, строку) встроенной функцией list:

```
list("list")
```

```
['l','i','s','t']
```

Список может содержать любое количество любых объектов (в том числе и вложенные списки), или не содержать ничего:

```
list("list")
```

```
['l','i','s','t']
```

Еще один способ создать список – это генераторы списков. *Генератор списков* – способ построить новый список, применяя выражение к каждому элементу последовательности. Генераторы списков очень похожи на цикл `for`:

```
c = [c * 3 for c in 'list']
c
```

```
['lll', 'iii', 'sss', 'ttt']
```

7.2 Функции и методы списков

Методы списков, в отличие от строковых методов (см. табл. 6.2), изменяют сам список, а потому результат выполнения не нужно записывать в переменную. Основные методы списков представлены в таблице 7.1.

Таблица 7.1: Строковые методы

`list.append(x)`	Добавляет элемент в конец списка
`list.extend(L)`	Расширяет список `list`, добавляя в конец все элементы списка `L`
`list.insert(i,x)`	Вставляет на `i`-ый элемент значение `x`
`list.remove(x)`	Удаляет первый элемент в списке, имеющий значение `x`
`list.pop([i])`	Удаляет `i`-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
`list.index(x[,start[,end]])`	Возвращает положение первого элемента со значением `x` (при этом поиск ведется от `start` до `end`)
`list.count(x)`	Возвращает количество элементов со значением `x`
`list.reverse()`	Разворачивает список
`list.copy()`	Поверхностная копия списка
`list.clear()`	Очищает список
`max(list), min(list)`	Максимальное и минимальное значение списка
`list.sort([key=None][,reverse = False])`	Сортировка списка

Приведем примеры использования некоторых функций:

- ```
  arr =[1,2,3]
  arr.append(4); arr
  ```

  ```
  [1,2,3,4]
  ```
  ```

- arr =[1,2,3]

  arr.extend([4,5,6]); arr

  [1,2,3,4,5,6]

- arr =[0,1,2,3]

  arr.insert(2, 100); arr

  [0,1,100,2,3]

- arr =[1,2,3,4,5]

  arr.pop(0)

  1

- arr =[1,2,3,1,1]

  arr.remove(1); arr

  [2, 3, 1, 1]

- arr =[1,2,3,1,1]

  arr.clear(); arr

  []

- arr =[1,2,1,2,1]

  arr.index(1),arr.index(2)

  (0,1)

- arr =[1,2,1,2,1]

  arr.count(1),arr.count(2),arr.count(3)

```
(3,2,0)
```

- ```
  arr =[1,2,3,4,5]
  max(arr), min(arr)
  ```

  ```
  (5,1)
  ```

- ```
  arr =[1,2,3,4,5]
  arr.reverse(); arr
  ```

  ```
  [5,4,3,2,1]
  ```

- ```
  arr =[4,3,7,4,1,6,9,10]
  arr.sort(reverse = True); arr     #  Сортировка по убыванию
  ```

  ```
  [10,9,7,6,4,4,3,1]
  ```

7.3 Заполнение списка числами

Создать список, содержащий диапазон чисел, можно с помощью функции `range()`. Эта функция возвращает диапазон, который преобразуется в список вызовом функции `list()`. Функция `range()` имеет следующий формат:

```
range([<Начало>,] <Конец>[, <Шаг>])
```

Первый параметр задает начальное значение – если он не указан, используется значение `0`. Во втором параметре указывается конечное значение, причем это значение не входит в возвращаемый диапазон. Если параметр `<Шаг>` не указан, то используется значение `1`. Примеры:

```
list(range(11))
```
```

```
[0,1,2,3,4,5,6,7,8,9,10]
```

```
list(range(7,0,-1))
```

```
[7,6,5,4,3,2,1]
```

7.4 Преобразование списка в строку

Преобразовать список в строку позволяет метод `join()`. Элементы добавляются через указанный разделитель. Формат метода:

```
<Строка> = <Разделитель>.join(<Последовательность>)
```

Пример:

```
list = ["item1", "item2", "item3"]
" - ".join(list)
```

```
'item1 - item2 - item3'
```

Элементы списка должны быть строками, иначе возвращается исключение.

С помощью функции `str()` можно сразу получить строковое представление списка:

```
list = ["item1", "item2", "item3", 3]
str(list)
```

```
"['item1','item2','item3',3]"
```

7.5 Создание кортежей

Кортежи, как и списки, являются упорядоченными последовательностями элементов. Они во многом аналогичны спискам, но имеют одно очень важное отличие – изменить кортеж нельзя. Можно сказать, что кортеж – это список, доступный только для чтения. Создать кортеж можно следущими способами:

- С помощью функции `tuple([Последовательность])`. Функция преобразует последовательность к кортеж. Если параметр не указан, то создается пустой кортеж:

```
tuple()                 # Создание пустого кортежа
()


tuple("String")         # Преобразование строки в кортеж
('S','t','r','i','n','g')


tuple([1,2,3])         # Преобразование списка в кортеж
(1,2,3)
```

- указав все элементы через запятую внутри круглых скобок (или без скобок):

```
t1 = ()                 # Создание пустого кортежа
t2 = (5,)               # Создание кортежа, содержащего 1 элемент
t3 = 1, "str", (3,4)    # Кортеж из 3-х элементов
```

Особого внимания требует вторая строка примера. Чтобы создать кортеж из одного элемента, необходимо в конце указать запятую. Именно запятые формируют кортеж, а не круглые скобки. Пример:

```
t = (5); type(t)                # Получили число, а не кортеж!
<class 'int'>
t = ("str"); type(t)            # Получили строку, а не кортеж!
<class 'str'>
```

Позиция элемента в кортеже задается *индексом*. Нумерация элементов кортежа (как и списка) начинается с 0. Как и все последовательности, кортежи поддерживают обращение к элементу по индексу, получение среза, конкатенацию, повтор, проверку на вхождение и невхождение. Пример:

```
t = (1,2,3,4,5)
t[0]

1

t[::-1]

(5,4,3,2,1)

t[2:4]       # Срез

(3, 4)

4 in t, 0 in t       #  Проверка на вхождение

(True, False)

(1,2,3) * 3            # Повторение
```

```
(1,2,3,1,2,3,1,2,3)
```

```
(1, 2, 3) + (4, 5, 6)     # Конкатенация
```

```
(1, 2, 3, 4, 5, 6)
```

Кортежи поддерживают функции списков `len()`, `min()`, `max()`, методы `index()` и `count()`.

7.6 Упражнения и контрольные вопросы к главе 7

1. Что представляют собой списки в Python?
2. Что такое генератор списков?
3. Какой метод применяется для добавления элемента в конец списка?
4. Какой метод применяется для добавления последовательности в конец списка?
5. Какой результат получим после выполнения следующего кода:

```
arr =[56,34,78]
arr.insert(-1, 12); arr
```

6. Какой результат получим после выполнения следующего кода:

```
arr =[2,3,4,5]
arr.pop(); arr
```

7. Какой результат получим после выполнения следующего кода:

```
arr =[33,78,33,12]
arr.remove(33); arr
```

8. Что такое кортеж?

9. Какими способами можно создать кортеж?

Глава 8

Словари

Словари – это наборы объектов, доступ к которым осуществляется не по индексу, а по ключу. В качестве ключа можно указать неизменяемый объект, например: число, строку или кортеж. Элементы словаря могут содержать объекты произвольного типа данных и иметь неограниченную степень вложенности. Элементы в словарях располагаются в произвольном порядке. Чтобы получить элемент, необходимо указать ключ, который использовался при сохранении значения. Функции, предназначенные для работы с последовательностями, а также операции извлечения среза, конкатенации, повторения и др., к словарям не применимы. Словари относятся к изменяемым типам данных, т.е. можно не только получить значение по ключу, но и изменить его.

8.1 Создание словаря

Способы создания словарей следующие:

- с помощью функции `dict()`:

```
d = dict(); d    # создание пустого словаря
```

```
{}
```

```
d = dict(a=1, b=2); d
```

```
{'a': 1, 'b': 2}
```

```
d = dict([("a", 1), ("b", 2)]); d   # список кортежей
```

```
{'a': 1, 'b': 2}
```

```
d = dict([["a", 1], ["b", 2]]); d   # список списков
```

```
{'a': 1, 'b': 2}
```

Объединить два списка в список кортежей позволяет функция `zip()`:

```
k = ["a", "b"]      # список с ключами
v = [1, 2]          # список со значениями
list(zip(k,v))      # создание списка кортежей
```

```
[('a',1), ('b',1)]
```

```
d = dict(zip(k,v)); d  # создание словаря
```

```
{'a': 1, 'b': 2}
```

- указав все элементы словаря внутри фигурных скобок:

```
d = {}; d       #  создание пустого словаря
```

```
{}
```

```
d = {'a': 1, 'b': 2}; d
```

```
{'a': 1, 'b': 2}
```

- заполнив словарь поэлементно, при этом ключ указывается в квадратных скобках:

```
d = {};       # создаем пустой словарь
d["a"] = 1
d["b"] = 2
d
```

```
{'a': 1, 'b': 2}
```

- с помощью метода `dict.fromkeys[<Последовательность>[, <Значение>]]`:

```
d = dict.fromkeys(["a", "b", "c"])
d
```

```
{'a': None, 'c': None, 'b': None}
```

```
d = dict.fromkeys(["a", "b", "c"], 0)
d
```

```
{'a': 0, 'c': 0, 'b': 0}
```

8.2 Операции над словарями

Обращение к элементам словаря осуществляется с помощью квадратных скобок, в которых указывается ключ:

```
d = {1: "int", "a": "str", (1,2): "tuple"}
d[1], d["a"], d[(1,2)]
```

```
('int', 'str', 'tuple')
```

Проверить существование ключа можно с помощью оператора `in`. Проверить, отсутствует ли какой-либо ключ в словаре, позволит оператор `not in`.

```
d = {"a": 1, "b": 2}
"a" in d              # ключ существует
```

```
True
```

```
"c" in d              # ключ не существует
```

```
False
```

```
"c" not in d              # ключ не существует
```

```
True
```

Метод `get ()` позволяет избежать возбуждения исключения `KeyError` при отсутствии в словаре указанного ключа. Если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то возвращается `None` или значение, указанное во втором параметре:

```
d = {"a": 1, "b": 2}
d.get("a"), d.get("c"), d.get("c", 800),
```

```
(1, None, 800)
```

Кроме того, можно воспользоваться методом `setdefault()`. Если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то в словаре создается новый элемент со

значением, указанным во втором параметре. Если второй параметр не указан, значением нового элемента будет `None`:

```
d = {"a": 1, "b": 2}
d.setdefault("a"), d.setdefault("c"), d.setdefault("d", 0),
```

```
(1, None, 0)
```

```
d
{'a': 1, 'c': None, 'b': 2, 'd': 0}
```

Словари относятся к изменяемым типам данных, изменить элемент можно по ключу. Если элемент с указанным ключом отсутствует в словаре, то он будет добавлен в словарь:

```
d = {"a": 1, "b": 2}
d["a"] = 800      # изменение элемента по ключу
d["c"] = "str"    # будет добавлен новый элемент
d
```

```
{'a': 800, 'b': 2, 'c': 'str'}
```

Функция `len()` позволяет получить количество ключей в словаре:

```
d = {"a": 1, "b": 2}
len(d)
```

```
2
```

Удалить элемент из словаря можно с помощью оператора `del`:

```
d = {"a": 1, "b": 2}
del d["b"]; d
```

```
{'a': 1}
```

8.3 Перебор элементов словаря

Перебрать все элементы словаря можно с помощью цикла `for`:

```
d = {"x": 1, "y": 2, "z": 3}
for key in d.keys():
    print("({0} => {1})".format(key, d[key]), end = " ")
```

```
(x => 1) (y => 2) (x => 3)
```

```
for key in d:
    print("({0} => {1})".format(key, d[key]), end = " ")
```

```
(x => 1) (y => 2) (x => 3)
```

8.4 Методы для работы со словарями

Для работы со словарями предназначены следующие методы:

- `keys()` – возвращает объект `dict_keys`, содержащий все ключи словаря:

```
d1, d2 = {"a": 1, "b": 2}, {"a": 3, "c": 4, "d": 5}
d1.keys(), d2.keys()     # получаем объект dict_keys
```

```
(dict_keys(['a', 'b']), dict_keys(['a', 'c', 'd']))
```

```
list(d1.keys())   # список ключей

['a', 'b']

for k in d1.keys(): print(k, end = " ")

a b

d1.keys() | d2.keys()

{'b', 'a', 'd', 'c'}    # объединение

d1.keys() - d2.keys()

{'b'}    # разница

d2.keys() - d1.keys()

{'d', 'c'}    # разница

d2.keys() & d1.keys()

{'a'}    # одинаковые ключи

d1.keys() ^ d2.keys()

{'b', 'd', 'c'}    # уникальные ключи
```

- `values()` – возвращает объект `dict_values`, содержащий все значения словаря:

```
d = {"a": 1, "b": 2}
d.values()          # получаем объект dict_values

dict_values([1, 2])

list(d.values())   # получаем список значений

[1, 2]

[v for v in d.values() ]

[1, 2]
```

- `items()` – возвращает объект `dict_items`, содержащий все ключи и значения в виде кортежей:

```
d = {"a": 1, "b": 2}
d.items()          # получаем объект dict_items

dict_items([('a', 1), ('b', 2)])

list(d.items())   # получаем список кортежей

[('a', 1), ('b', 2)]
```

- `in`, `not in`, `get()`, `setdefault()` – описаны в предыдущем разделе.

- `pop()` – удаляет элемент с указанным ключом и возвращает его значение. Если ключ отсутсвует, то возвращается значение из второго параметра:

```
d = {"a": 1, "b": 2}
d.pop("a"), d.pop("n", 0)
```

```
(1, 0)
```

- `popitem()` – удаляет произвольный элемент и возвращает кортеж из ключа и значения:

```
d = {"a": 1, "b": 2}
d.popitem()
```

```
('b', 2)
```

- `clear()` – удаляет все элементы словаря:

```
d = {"a": 1, "b": 2}
d.clear(); d
```

```
{}
```

- `update()` – добавляет элементы в словарь. Метод изменяет текущий словарь и ничего не возвращает:

```
d = {"a": 1, "b": 2}
d.update(c = 3); d
```

```
{'a': 1, 'b': 2, 'c': 3}
```

```
d.update({"c": 10}); d
```

```
{'a': 1, 'b': 2, 'c': 10}
```

- `copy()` – создает поверхностную копию словаря:

```
d1 = {"a": 1, "b": 2}
d2 = d1.copy()
d1 is d2
```

```
False
```

```
d2["a"] = 800
d1, d2
```

```
({'a': 1, 'b': 2}, {'a': 800, 'b': 2})
```

8.5 Упражнения и контрольные вопросы к главе 8

1. Какие существуют способы создания словарей?
2. Что позволяет делать метод `get()`?
3. Как работает метод `setdefault()`?
4. Какая функция позволяет получить количество ключей в словаре?
5. Какими способами можно осуществить перебор элементов словаря?

Глава 9

Функции и обработка исключений

9.1 Определение и вызов функций

Функция в Python – объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции `def`. Определим простейшие функции:

```
def print_ok():
    """ Пример функции без параметров"""    # Строка документирования
    print("сообщение")


def echo(m):
    """ Пример функции с параметром"""
    print(m)


def summa (x, y)
    return x + y
```

Имя функции должно быть уникальным идентификатором, состоящим из латинских букв, цифр и знаков подчеркивания, причем имя функции не может начинаться с цифры. В качестве имени нельзя использовать ключевые слова, кроме того, следует избегать совпадений с названиями встроенных идентификаторов. Регистр символов в названии функции также имеет значение.

При вызове функции значения передаются внутри круглых скобок через запятую. Если функция не принимает параметров, то указываются только круглые скобки:

```
print_ok()

сообщение

echo("hello")

hello

a, b = 10, 50
y = summa(a,b); y

60
```

В Python можно сохранить ссылку на функцию в другой переменной:

```
def summa (x, y):
    return x + y

f = summa
f (10, 20)

30
```

Можно также передать ссылку на функцию в качестве параметра другой функции. Функции, передаваемые по ссылке, обычно называются *функциями обратного вызова*:

```
def summa (x, y):
    return x + y

def func (f, a, b):
    return f (a, b)

# передаем ссылку на функцию в качестве параметра
v = func (summa, 10, 20); v
```

```
30
```

Все инструкции в программе выполняются последовательно сверху вних. Поэтому определение функции должно быть расположено перед вызовом функции. Если определение одной функции встречается в программе несколько раз, то будет использоваться функция, которая была определена последней:

```
def echo():
    print ("hello1")

def echo():
    print ("hello2")

echo()
```

```
hello2
```

Чтобы сделать некоторые параметры необязательными, следует в определении функции присвоить этому параметру начальное значение. Необяза-

тельные параметры должны следовать после обязательных параметров. Пример:

```
def summa (x, y = 2):
    return x + y

summa (5)

7

summa (5, 8)

13
```

Язык Python позволяет также передавать значения в функцию, используя сопоставление по ключам. Для этого при вызове функции параметрам присваиваются значения, причем последовательность указания параметров в этом случае может быть произвольной:

```
def summa (x, y):
    return x + y

summa (y = 10, x = 30)

40
```

Если значения параметров, которые планируется передать в функцию, содержатся в кортеже или списке, то перед объектом следует указать символ *:

```
def summa (a, b, c):
    return a + b + c

t, arr = (1, 2, 3), [1, 2, 3]
summa (*t)      # распаковка кортежа

6

summa(*arr)      # распаковка списка

6
```

Если значения параметров содержатся в словаре, то распаковать словать можно, указав перед ним две звездочки **:

```
def summa (a, b, c):
    return a + b + c

d = {"a": 1, "b": 2, "c": 3}
summa(**d)      # распаковка словаря

6
```

9.2 Переменное число параметров в функции

Если перед параметром в определении функции указать символ *, то функции можно будет передать произвольное количество параметров. Все переданные параметры сохраняются в кортеже. Пример:

```
def summa(*t):
```

```
    res = 0
    for i in t:
        res += i
    return res

print (summa(10, 20))
```

```
30
```

```
print (summa(10, 20, 30, 40))
```

```
100
```

Если перед параметром в определении функции указать две звездочки **, то все именованные параметры будут сохранены в словаре:

```
def func (**d):
    for i in d:
        print ("{0} => {1}".format(i, d[i], end = " "))

print (a = 1, b = 2, c = 3)
```

```
a => 1 b=> 2 c => 3
```

При комбинировании параметров параметр с двумя звездочками указывается самым последним.

9.3 Функции-генераторы

Функцией-генератором называется функция, которая может возвращать одно значение из нескольких значений на каждой итерации. Приостановить

выполнение функции и превратить функцию в генератор позволяет ключевое слово `yield`. Рассмотрим пример:

```
def func (x,y):
    for i in range (1, x+1):
        yield i ** y

for n in func(10, 2):
    print (n, end = " ")

1 4 9 16 25 36 49 64 81 100

for n in func(10, 3):
    print (n, end = " ")

1 8 27 64 125 216 343 512 729 1000
```

Функции-генераторы поддерживают метод `__next__()`, который позволяет получить следующее значение:

```
def func (x,y):
    for i in range (1, x+1):
        yield i ** y

i = func(3, 3)
print(i.__next__())

1           # 1 ** 3

print(i.__next__())
```

```
8           # 2 ** 3
```

9.4 Обработка исключений

Исключения – это извещения интерпретатора, возбуждаемые в случае возникновения ошибки в программном коде или при наступлении какого-либо события. Если в коде не предусмотрена обработка исключения, то выполнение программы прерывается, и выводится сообщение об ошибке.

Самый простейший пример исключения – деление на ноль:

```
100 / 0


Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

В данном случае интерпретатор сообщил нам об исключении `ZeroDivisionError`, то есть делении на ноль. Подсказки дают нам полную информацию о том, где порождено исключение, и с чем оно связано.

9.5 Инструкция try...except...else...finally

Для обработки исключений предназначена инструкция `try`. Например, обработать исключение, возникающее при делении на ноль, можно следующим образом:

```
try:
    x = 1/0
except ZeroDivisionError:       # указываем класс исключения
    print ("Обработка деления на 0")
```

```
    x = 0

print(x)
```

```
0
```

Если в обработчике присутствует блок `else`, то инструкции внутри этого блока будут выполнены только при отсутствии ошибок. При необходимости выполнить какие-либо завершающие действия вне зависимости от того, возникло исключение или нет, следует воспользоваться блоком `finally`. Пример:

```
try:
    x = 10 / 2    # нет ошибки
    #x = 10 / 0   # Ошибка деления на 0
except ZeroDivisionError:
    print("Деление на 0")
else:
    print("Блок else")
finally:
    print("Блок finally")
```

Результат выполнения при отсутствии исключения:

```
Блок else
Блок finally
```

При наличии исключения:

```
Деление на 0
Блок finally
```

9.6 Упражнения и контрольные вопросы к главе 9

1. Что такое функция в языке программирования Python?

2. Приведите примеры определения и вызова функций.

3. Как в Python сохранить ссылку на функцию в другой переменной?

4. Что такое функция обратного вызова?

5. Приведите пример передачи значения в функцию, используя сопоставление по ключам.

6. Как происходит распаковка списка, кортежа и словаря при передаче в качестве параметра функции?

7. Что такое функция-генератор?

8. Что такое исключения в программе?

Глава 10

Регулярные выражения. Модули и пакеты

10.1 Синтаксис регулярных выражений

Регулярные выражения предназначены для выполнения в строке сложного поиска или замены. В языке Python использовать регулярные выражения позволяет модуль `re`. Прежде чем задействовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
import  re
```

Создать откомпилированный шаблон регулярного выражения позволяет функция `compile()`. Функция имеет следующий формат:

```
<Шаблон> = re.compile(<Регулярное выражение>[, <Модификатор>])
```

К примеру, в параметре `<Модификатор>` может быть указан модификатор `I` – поиск без учета регистра; `M` – поиск в строке, состоящей из нескольких подстрок, разделенных символом новой строки (`"\n"`) и др. Больше примеров в книге [5].

Также используются следующие метасимволы:

- `^` – привязка к началу строки или подстроки.
- `$` – привязка к концу строки или подстроки и др.

В квадратных скобках `[]` можно указать символы, которые могут встречаться на этом месте в строке. Можно перечислить символы подряд или указать диапазон через дефис:

- `[09]` – соответствует числу 0 или 9;
- `[0-9]` – соответствует любому числу от 0 до 9;
- `[абв]` – соответствует буквам «а», «б» и «в»;
- `[а-г]` – соответствует буквам «а», «б» «в» и «г»;
- `[а-яё]` – соответствует любой букве от «а» до «я»;
- `[А-ЯЁ]` – соответствует любой букве от «А» до «Я»;
- `[0-9а-яА-ЯёЁa-zA-Z]` – любая цифра и любая буква независимо от регистра и языка.

Значение в скобках инвертируется, если после первой скобки поставить символ `^`. Тем самым можно указать символы, которых не должно быть на этом месте в строке:

- `[^09]` – не цифра 0 или 9;
- `[^0-9]` – не цифра от 0 до 9.

Метасимвол `|` позволяет сделать выбор между альтернативными значениями. Выражение `n|m` соответсвует одному из символов `n` или `m`.

10.2 Модули

Модулем в языке `Python` называется любой файл с программным кодом. Каждый модуль может импортировать другой модуль, получая, таким образом, доступ к атрибутам (переменным, функциям и классам), объявленным внутри импортированного модуля. Модуль может быть написан не только на Python, а например, на C или C++. Получить имя модуля позволяет атрибут `__name__`.

Импортировать модуль позволяет инструкция `import`. Например, чтобы подключить модуль для работы с регулярными выражениями, нужно набрать `import re`. За один раз можно импортировать сразу несколько модулей, записав из через запятую. После импортирования модуля его название становится перменной, через которую можно получить доступ к атрибутам модуля:

```
import time, math
print(math.pi)
```

```
3.141592653589793
```

Если указанный атрибут модуля не будет найден, то возбудится исключение `AttributeError`. А если не удастся найти модуль для импортирования, то `ImportError`.

10.3 Использование псевдонимов

Если название модуля слишком длинное или неудобное по каким-то причинам, то для него можно создать псевдоним с помощью ключевого слова `as`:

```
import math as m
m.e
```

```
3.141592653589793
```

Теперь доступ ко всем атрибутам модуля math осуществляется только с помощью переменной `m`, а переменной `math` в этой программе уже не будет.

10.4 Инструкция from

Подключить определенные атрибуты модуля можно с помощью инструкции `from`. Она имеет несколько форматов:

```
from <Название модуля> import <Атрибут 1>
[ as <Псевдоним 1>],
[<Атрибут 2> [as <Псеводним 2> ] ...]

from <Название модуля> import *
```

Первый формат позволяет подключить из модуля только указанные атрибуты. Для длинных имен можно также назначить псевдоним, указав его после ключевого слова `as`:

```
from math import e, ceil as c
e

2.718281828459045

c(4.6)

5
```

Импортируемые атрибуты можно разместить на нескольких строках, если их много, для лучшей читаемости кода:

```
from math import (sin, cos,
                  tan, atan)
```

Второй вариант формата инструкции `from` позволяет импортировать из модуля все идентификаторы:

```
from math import *
print (pi)
```

```
3.141592653589793
```

Следует заметить, что идентификаторы, названия которых начинаются с символа подчеркивания, импортированы не будут. Кроме того, необходимо учитывать, что импортирование всех идентификаторов из модуля может нарушить пространство имен главной программы, т. к. идентификаторы, имеющие одинаковые имена, будут перезаписаны.

10.5 Создание своего модуля на Python

Создадим файл `mymodule.py`, в которой определим какие-нибудь функции:

```
def hello():
    print('Hello, World!')
```

Теперь в этой же папке создадим другой файл, например `main.py`:

```
import mymodule

mymodule.hello()
```

```
Hello, World!
```

Модуль нельзя именовать также, как и ключевое слово. Также имена модулей нельзя начинать с цифры. И не стоит называть модуль также, как какую-либо из встроенных функций. Это создаст большие неудобства при его последующем использовании.

10.6 Пакеты

Пакетом называется каталог с модулями, в котором расположен файл инициализации `__init__.py`. Файл инициализации может быть пустым или содержать код, который будет выполнен при первом обращении к пакету. В любом случае он обязательно должен присутствовать внутри каталога с модулями.

В качетсве примера приведем следующую структуру файлов и каталогов:

```
main.py                 # Основной файл с программой
folder1\                # На одном уровне вложенности с main.py
    __init__.py         # Файл инициализации
    module1.py          # Модуль folder1\module1.py
    folder2\            # Вложенная папка
        __init__.py     # Файл инициализации
        module2.py      # Модуль folder1\folder2\module2.py
        module3.py      # Модуль folder1\folder2\module3.py
```

При использовании инструкции `import` путь к модулю должен включать не только названия каталогов, но и название модуля без расширения:

```
import folder1.folder2.module2
```

10.7 Упражнения и контрольные вопросы к главе 10

1. Какой командой можно подключить модуль для работы с регулярными выражениями?

2. Что называется модулем в Python?

3. С помощью какой инструкции можно подключить определенные атрибуты модуля?

4. Приведите пример создания пользовательского модуля в Python.

5. Что такое пакет в Python?

Глава 11

Работа с датой и временем. Работа с файлами

Для работы с датой и временем в языке Python предназначены модули:

- `time` – позволяет получить текущие дату и время, производить форматированный вывод;
- `datetime` – позволяет манипулировать датой и временем (производить арифметические операции, сравнивать даты и др.);
- `calendar` – позволяет вывести календарь в виде текста или HTML-формате;
- `timeit` – позволяет измерить время выполнения фрагментов программы.

11.1 Модуль time

Получить текущие дату и время позволяют следующие функции модуля `time`:

- `time()` – возвращает вещественное число, представляющее количество секунд, прошедшее с 1 января 1970 года.
- `localtime([Количество секунд])` – возвращает объект `struct_time`, представляющий локальное время. Если параметр указан, то время будет не текущим, а соответсвующим количеству секунд, прошедших с 1 января 1970 года.

```
time.localtime()
```

```
time.struct_time(tm_year=2017, tm_mon=9, tm_mday=5, tm_hour=0,
tm_min=35, tm_sec=35, tm_wday=1, tm_yday=248, tm_isdst=0)
```

Объект `struct_time`, возвращаемый функцией `localtime()`, содержит следующие атрибуты:

- `tm_year` – год;
- `tm_mon` – месяц;
- `tm_mday` – день месяца;
- `tm_hour` – час;
- `tm_min` – минуты;
- `tm_sec` – секунды;
- `tm_wday` – день недели;
- `tm_yday` – количество дней, прошедшее с начала года;
- `tm_isdst` – флаг коррекции летнего времени;

Форматирование даты и времени выполняют следующие функции из модуля `time`:

- `strftime(<Строка формата>[, <Объект struct_time>])` – возвращает строковое представление даты в соответствии со строкой формата:

```
import time
time.strftime("%d. %m. %Y")

'05.09.2017'
```

- strptime(<Строка с датой>[, <Строка формата>]) – разбирает строку, указанную в первом параметре, в соответствии со строкой формата:

  time.strptime("Fri Apr 03 14:01:34 2015")

  time.struct_time(tm_year=2015, tm_mon=4, tm_mday=3, tm_hour=14, tm_min=1, tm_sec=34, tm_wday=4, tm_yday=93, tm_isdst=-1)

- asctime([<Объект struct_time>]) – возвращает строку формата "%a %b %d %H:%M:%S %Y". Если параметр не указан, будут выведены текущие дата и время:

  time.asctime()

  'Tue Sep 5 01:30:52 2017'

- ctime([<Количество секунд>]) – функция аналогична asctime(), но в качестве параметра принимает не объект struct_time, а количество секунд. Если параметр не указан, то возвращается текущая дата:

  time.ctime()

  'Tue Sep 5 01:30:52 2017'

Функция sleep(<Время в секундах>) из модуля time прерывает выполнение скрипта на указанное время, по истечении которого скрипт продолжит работу:

```
import time
time.sleep(5)      # "Засыпаем" на 5 секунд
```

11.2 Модуль datetime

Модуль `datetime` позволяет манипулировать датой и временем. Например, выполнять арифметические операции, сравнивать даты и др. Подключается модуль с помощью инструкции:

```
import datetime
```

Класс `timedelta` из модуля `datetime` позволяет выполнять операции на датами – складывать, вычитать, сравнивать и др:

```
d1 = datetime.timedelta(days=2)
d1 = datetime.timedelta(days=7)
d1 + d2, d2 - d1
```

```
(datetime.timedelta(9), datetime.timedelta(5))
```

```
d1 == d2
```

```
False
```

Класс `date` из модуля `datetime` позволяет выполнять операции над датами:

```
d1 = datetime.date(2015, 4, 3)
d2 = datetime.date(2015, 1, 1)
d1 - d2
```

```
datetime.timedelta(92)
```

```
d1 < d2
```

```
False
```

Класс `time` из модуля `datetime` позволяет выполнять операции над временем:

```
t1 = datetime.time(23, 12, 38)    # часы, минуты, секунды
t2 = datetime.time(12, 28, 18)
t1 < t2
```

```
False
```

Класс `datetime` из модуля `datetime` позволяет выполнять операции над комбинацией даты и времени. Конструктор класса имеет следующий формат:

```
datetime(<Год>, <Месяц>, <День>[, hour][, mionute][, second]
[, microsecond][, tzinfo])
```

Примеры:

```
datetime.datetime.today()
```

```
datetime.datetime(2017, 9, 5, 2, 0, 7, 959)
```

```
d = datetime.datetime(2015, 4, 6, 16, 7, 22)
d.year, d.month, d.day
```

```
(2015, 4, 6)
```

11.3 Модуль Timer

Модуль `timeit` позволяет измерить время выполнения небольших фрагментов кода с целью оптимизации программы. Подключается модуль с помощью инструкции:

```
from timeit import Timer
```

11.4 Файлы

Открыть файл можно с помощью функции `open`:

```
f = open('text.txt', 'r')
```

Первый аргумент функции – это имя файла. Путь к файлу может быть относительным или абсолютным. Второй аргумент – это режим, в котором будет открываться файл (таблица 11.1).

Таблица 11.1: Режимы функции open

Режим	Обозначение
`'r'`	Открытие на чтение (значение по умолчанию)
`'w'`	Открытие на запись, содержимое файла удаляется; если файла не существует, создается новый
`'x'`	Открытие на запись
`'a'`	Открытие на дозапись, информация добавляется в конец файла
`'b'`	Открытие в двоичном режиме
`'t'`	Открытие в текстовом режиме (значение по умолчанию)
`'+'`	Открытие на чтение и запись

Режимы могут быть объеденены, например, `'rb'` – чтение в двоичном режиме. По умолчанию режим равен `'rt'`.

11.5 Чтение из файла

Одним из способов чтения из файла является метод `read`, читающий весь файл целиком, если был вызван без аргументов, и `n` символов, если был вызван с аргументом (целым числом `n`):

```
f = open('text.txt')
f.read(1)
```

```
'H'
```

```
f.read()
```

```
'ello world!\nThe end.\n\n'
```

Еще один способ прочесть файл – это прочитать файл построчно, воспользовавшись циклом `for`:

```
f = open('text.txt')
for line in f:
    line
```

```
'Hello world!\n'
'\n'
'The end.\n'
'\n'
```

11.6 Запись в файл

Запишем в файл следующий список:

```
l = [str(i) + str(i-1) for i in range(5)]
l
```

```
['0-1', '10', '21', '32', '43']
```

```
f = open('C:/Data/text.txt', 'w')
for index in l:
    f.write(index + '\n')
```

```
4
3
3
3
3
```

Метод `write` возвращает число записанных символов. После окончания работы с файлом его обязательно нужно закрыть с помощью метода `close`:

```
f.close()
```

Теперь воссоздадим этот список из получившегося файла. Откроем файл на чтение и прочитаем строки:

```
f = open('C:/Data/text.txt', 'r')
m = [line.strip() for line in f]
m
```

```
['0-1', '10', '21', '32', '43']
```

```
f.close()
```

Получили тот же список, что и был.

11.7 Упражнения и контрольные вопросы к главе 11

1. Какие модуль в Python предназначены для работы с датой и временем?

2. Для чего предназначена функция `sleep()`?

3. Какая инструкция в Python предназначена для открытия файла?

4. Какие команды служат для чтения из файла?

5. Какая команда служит для записи в файл?

Библиографический список

1. История языка программирования Python [Электронный ресурс] // Википедия: свободная энциклопедия. [Дата обращения: 28-07-2017]. – Режим доступа: https://ru.wikipedia.org/wiki/История_языка_программирования_Python.

2. GNU General Public License [Электронный ресурс] // Википедия: свободная энциклопедия. [Дата обращения: 28-07-2017]. – Режим доступа: https://ru.wikipedia.org/wiki/GNU_General_Public_License.

3. PyCharm [Электронный ресурс] // Википедия: свободная энциклопедия. [Дата обращения: 04-08-2017]. – Режим доступа: https://ru.wikipedia.org/wiki/PyCharm.

4. Буйначев, С. К. Основы программирования на языке Python: учебное пособие / С. К. Буйначев, Н. Ю. Боклаг. – Екатеринбург: Изд-во Урал. ун-та, 2014. – 91 с.

5. Прохоренок, Н.А. Python 3. Самое необходимое / Н. А. Прохоренок, В. А. Дронов. – СПб.: «БХВ-Петербург», 2016. – 461 с.

6. Лутц, М. Изучаем Python, 4-е издание. – Пер. с англ. СПб.: Символ-Плюс, 2011, 1280 с.

7. Мусин, Д. Самоучитель Python: Выпуск 0.2. [Электронный ресурс] [Дата обращения: 17-08-2017]. Режим доступа: https://pythonworld.ru/pdf.

8. Charles, R.S. Python for Everybody. Exploring Data Using Python 3. 2016, 247 с.

9. Литерал (информатика) [Электронный ресурс] // Википедия: свободная энциклопедия. [Дата обращения: 17-08-2017]. –

   Режим доступа: http://ru.wikipedia.org/?oldid=84256998.

10. Экранирование символов [Электронный ресурс] // Википедия: свободная энциклопедия. [Дата обращения: 17-08-2017]. – Режим доступа: https://ru.wikipedia.org/wiki/Экранирование_символов.

Учебное пособие

Еникеева Лениза Васимовна, Шамшович Валентина Федоровна,
Фаткуллин Николай Юрьевич

Основы программирования на языке Python

Редактор

Подписано в печать . Формат 60x84 1/16.
Усл. печ. л. . Тираж экз. Заказ .

Издательство
Уфимского государственного нефтяного
технического университета

Адрес издательства:
450062, Республика Башкортостан, г. Уфа, ул. Космонавтов, 1