Python Lessons

Oleg Kishinskii

мар. 30, 2021

Введение

Данная документация содержит описание языка програмирования Python. Надеюсь, что данный справочник поможет вам в изучении программирования и будет полезен при написании программ.

Глава 1

Что такое Python

Язык программирования Python 3 — это мощный инструмент для создания программ самого разнообразного назначения, доступный даже для новичков. С его помощью можно решать задачи различных типов.

Этот сайт призван помочь начинающим научиться программировать на python 3. Также здесь можно подробнее узнать об особенностях функционирования этого языка.

Язык Python обладает некоторыми примечательными особенностями, которые обуславливают его широкое распространение. Поэтому прежде чем изучать python, следует рассказать о его достоинствах и недостатках.

Глава 2

Python 3: преимущества и недостатки языка

Python - интерпретируемый язык программирования. С одной стороны, это позволяет значительно упростить отладку программ, с другой - обуславливает сравнительно низкую скорость выполнения.

- Динамическая типизация. В python не надо заранее объявлять тип переменной, что очень удобно при разработке.
- Хорошая поддержка модульности. Вы можете легко написать свой модуль и использовать его в других программах.
- Встроенная поддержка Unicode в строках. В Python необязательно писать всё на английском языке, в программах вполне может использоваться ваш родной язык.
- Поддержка объектно-ориентированного программирования. При этом его реализация в python является одной из самых понятных.
- Автоматическая сборка мусора, отсутствие утечек памяти.
- Интеграция с C/C++, если возможностей python недостаточно.
- Понятный и лаконичный синтаксис, способствующий ясному отображению кода. Удобная система функций позволяет при грамотном подходе создавать код, в котором будет легко разобраться другому человеку в случае необходимости. Также вы сможете научиться читать программы и модули, написанные другими людьми.
- Огромное количество модулей, как входящих в стандартную поставку Python 3, так и сторонних. В некоторых случаях для написания программы достаточно лишь найти подходящие модули и правильно их скомбинировать. Таким образом, вы можете думать о составлении программы на более высоком уровне, работая с уже готовыми элементами, выполняющими различные действия.
- Кроссплатформенность. Программа, написанная на Python, будет функционировать совершенно одинаково вне зависимости от того, в какой операционной системе она запущена. Отличия возникают лишь в редких случаях, и их легко заранее предусмотреть благодаря наличию подробной документации.

2.1 Установка

Для установки компилятора языка программирования скачайте инсталятор соответствующий вашей операционной ссистеме с сайта разработчика Python.org

Так же установить Python можно испльлзуя консольные утилиты

2.1.1 Windows

```
choco install python3
```

2.1.2 Linux: Ubuntu

```
sudo apt install python3
```

2.1.3 Linux: Fedora

```
sudo dnf install python3
```

Примечание: в Linux дистрибутивах установщик пакетов pip (он нам понадобится в будущем) необходимо устанавливать дополнительно.

```
sudo apt install python3-pip
```

2.2 Введение

2.2.1 Интерактивная среда IDLE

Для вызова интерактивной среды разработчика достаточно в консоли набрать команду:

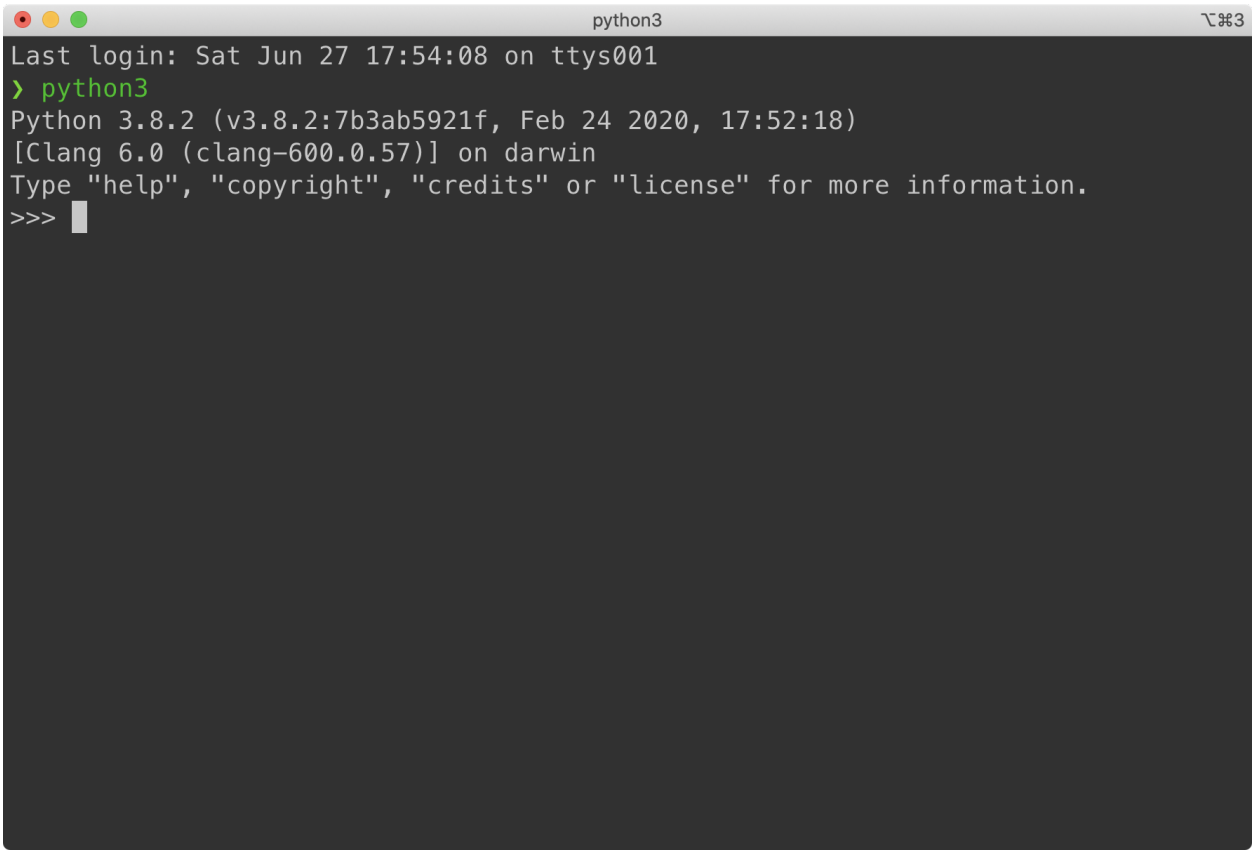
```
python3
```

В этой интерактивной среде можно выполнять команды Python и сразу же получать результат выполнение введенной команды.

2.2.2 Редакторы кода и серда разработки IDE

Для написания программ на языке Python подойдет любой текстовый редактор, но для более комфортной работы рекомендуется использовать редакторы кода с подсветкой синтаксиса и отслеживанием ошибок. один из таких редакторов, это VSCODE

Так же будет полезно использовать полноценную среду разработки IDE, одной из наиболее популярных евляется PyCharm



2.2.3 Виртуальное окружение (venv)

Для начала работы рекомендуется подготовить виртуальное окружение, оно необходимо для того что бы вслучае использования дополнительных библиотек они не конфликтовали с другими библиотеками используемыми в других проектах, например: нет необходимости держать в одном проекте библиотеки фреймворков **Django** и **Flask**

Для того что бы создать виртуальное окружение, в папке с проектом необходимо выполнить команду:

```
python -m venv venv
```

незабудьте так же активировать ваше виртуальное окружение выполнив команду:

```
source venv/bin/activate
```

это создаст внутри проекта папку venv с компилятором и всеми необходимыми библиотеками используемые по умолчанию.

Так же для того что бы сохранить список используемых в нашем проекте библиотек, нужно создать файл reqirements.txt его можно создать автоматически выполнив команду:

```
pip freeze > requirements.txt
```

Для установки в наше виртуальное окружение необходимых библиотек нужно выполнить команду:

```
pip install requirements.txt
```

Попробуйте установить фреймворк **flask** используя менеджер пакетов pip и сохраните список установленных пакетов:

```
pip install flask
pip freeze > requirements.txt
```

Теперь если вы откроете файл **requirements** вы увидите список наших зависимостей используемые в нашем проекте

```
click==7.1.2
Flask==1.1.2
itsdangerous==1.1.0
Jinja2==2.11.2
MarkupSafe==1.1.1
Werkzeug==1.0.1
```

2.2.4 Первая программа

Настало время написать нашу первую программу, создайте файл **Hello_World.py** со следующим содержимым:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# Это моя первая программа
"""
Это многострочный коментарий используется для создания докуентации к ксласам или функциям
и описывает их принцип работы.
В данном уроке мы пока не будет рассматривать примеры с документацией классов
"""
a = "World"  # так помно присвоить конкретной переменной какое либо значение, в данном случее␣
↪строку.

if __name__ == '__main__':
    print("Hello", a)  # программа напечатет в консоли Hello World
```

Теперь запустите нашу программу выполгив в консоле команнду:

```
python Hello_World.py
```

в консои вы должны увидеть результат нашей программы.

Примечание: Обратите внимание, что первая строка указывает какой интерпритатор мы будет использовать, она необходима если мы хотим запускать нашу команду с интерпритатором по умолчанию, например так: .. code:: sh

    ./Hello_World.py

Если этой строки не будет, то мы получим ошибку.

Вторая строка отвечает за кодировку, тоесть если мы хотим вывести на экран Русские символы, то нам надо указать какую кодировку мы будем ипользовать.

2.3 Типы данных

Примечание: Статья в разработке...

2.3.1 Переменные в Python:

Переменная в языке программирования это название для зарезервированного места в памяти компьютера, предназначенное для хранения значений. Это означает, что когда вы создаете переменную, вы на самом деле резервируете определенное место в памяти компьютера. Основываясь на типе данных переменной, интерпретатор выделяет необходимое количество памяти и решает, что может находится в зарезервированной области памяти. Для понимания, можете думать о переменной как о коробке, в которую можно положить любую вещь, но только определенного размера. Размер в данном примере будет типом переменной. Это не совсем верное определение, но оно дает общее представление о картине в целом.

Присвоение значения переменной:

В Python вам не нужно объявлять тип переменной вручную (как, например в C++). Объявление происходит автоматически (это называется динамическая типизация), когда вы присваиваете значение переменной. Знак равенства (=) используется для присвоения значения переменной. Операнд по левую сторону от знака равно (=) это имя переменной, операнд по правую сторону - значение присвоенное этой переменной.

Таблица - Обзор встроенных типов объектов

Имя	Тип	Описание и пример
Целые Числа	int	**Целые положительные или отрицательные** -35, 0, 24, 123467890033373747428
Числа с плавающей точкой	float	Дробные числа 3.14, 2.5, -2.33333, 0.12334
Строки	str	Строки «asdf», «Hello world», «123456»
Списки	list	**последовательность элементов** [«hello», -123, 0.34, «345»]
Словарь	dict	Последовательность пар элементов содержаших ключ-значение (key-value) {«Language»: «Python», «Version»: «3.8»}
Кортеж (Tuple)	tup	Неизменяемая упорядоченная последовательность элементов («hostname», 1234, -0.45, -32)
Множество	set	**Изменяемая неупоряоченная последовател** элементов {10, «Name», -30, 4.02, 100}
Булевые значения	bool	**Тип данных принимающий одно из двух з** true - истина false - ложь

2.3.2 Числа

Числа - Это не изменяемый тип данных. Числа в Python бывают трёх типов: целые, с плавающей точкой и комплексные. * Примером целого числа может служить 2. * Примерами чисел с плавающей точкой (или «плавающих» для краткости) могут быть 3.23 и 52.3E-4. Обозначение E показывает степени числа 10. В данном случае 52.3E-4 означает $52.3 \cdot 104$. * Примерыкомплексныхчисел:(-5+4j)и(2.3-4.6j)

Примечание: Нет отдельного типа 'long int' (длинное целое). Целые числа по умолчанию могут быть произвольной длины.

2.3.3 Строки

Строки - это неизменяемая упорядоченная последовательность символов, заключенная в кавычки. Строки применяются для записи текстовой информации (кажем, вашего имени) и произвольных совокупностей байтов (наподобие содержимого файла изображения). Они являются первым примером того, что в Python называется последовательностью — позиционно упорядоченной коллекцией других объектов. Для содержащихся элементов последовательности поддерживают порядок слева направо: элементы сохраняются и извлекаются по своим относительным позициям. Строго говоря, строки представляют собой последовательности односимвольных строк.

Строки можно суммировать. Тогда они объединяются в одну строку, такая операция называется «Конкатенацией строк»:

```
firts_string = "asdfgh"
second_string = "oiuytr"
print(firts_string + second_string)
```

Примечание: В Python 3 нет ASCII-строк, потому что Unicode является надмножеством (включает в себя) ASCII. Если необходимо получить строку строго в кодировке ASCII, используйте str.encode(«ascii»). По умолчанию все строки в Unicode.

Предупреждение: Нельзя производить арифметтические операции над строкам и числами Например: «qwerty» + 3 Это вызовет ошибку, Но строки можно перемножать «#» * 10 выведет на экран строку ##########

Логические и физические строки

Физическая строка – это то, что вы видите, когда набираете программу. Логическая стро- ка – это то, что Python видит как единое предложение. Python неявно предполагает, что каждой физической строке соответствует логическая строка. Примером логической строки может служить предложение print(„Привет, Мир!“) – ес- ли оно на одной строке (как вы видите это в редакторе), то эта строка также соответствует физической строке. Python неявно стимулирует использование по одному предложению на строку, что об- легчает чтение кода. Чтобы записать более одной логической строки на одной физической строке, вам придёт- ся явно указать это при помощи точки с запятой (;), которая отмечает конец логической строки/предложения. Например:

```
i=5
print(i)
#то же самое, что
i = 5; print(i);
#и то же самое может быть записано в виде
i = 5; print(i);
```

Однако я настоятельно рекомендую вам придерживаться написания одной логической строки в каждой физической строке. Таким образом вы можете обойтись совсем без точки с запятой. Кстати, я никогда не использовал и даже не встречал точки с запятой в программах на Python. Можно использовать более одной физической строки для логической строки, но к этому следует прибегать лишь в случае очень длинных строк. Пример написания одной логиче- ской строки, занимающей несколько физических строк, приведён ниже. Это называется явным объединением строк.

```
s = 'Это строка. \
Это строка продолжается.'
print(s) #Это строка. Это строка продолжается.
```

Операции над последовательностями

Как последовательности, строки поддерживают операции, которые предполагают наличие позиционного порядка среди элементов. Например, если мы имеем четырех символьную строку, записанную в кавычках (обычно одинарных), то можем проверить ее длину с помощью встроенной функции **len()** и извлечь ее компоненты посредством выражений индексации.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

simple_string = 'Spam'
len(simple_string)
# 4
simple_string[0]
# 'S'
simple_string[1]
# 'p'
```

Нумерация всех символов в строке идет с нуля. Но, если нужно обратиться к какому-то по счету символу, начиная с конца, то можно указывать отрицательные значения (на этот раз с единицы).

```
simple_string = "StringBody"
simple_string[1]
# t

simple_string[-1]
# y
```

Кроме обращения к конкретному символу, можно делать срезы строк, указав диапазон номеров (срез выполняется по второе число, не включая его):

```
example_string = "Lorem Ipsum is simply dummy text of the printing and typesetting"
example_string[0:9]
# 'Lorem Ips'

example_string[10:22]
# 'm is simply '

# Если не указывается второе число, то срез будет до конца строки:
example_string[-3:]
# 'ing'
```

Также в срезе можно указывать шаг:

```
# Так можно получить нечетные числа
a = '0123456789'
a[1::2]
# '13579'

# А таким образом можно получить все четные числа строки a:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
a[::2]
# '02468'

# Срезы также можно использовать для получения строки в обратном порядке:
a[::-1]
# '9876543210'
```

Методы для работы со строками

Методы upper, lower, swapcase, capitalize

Методы upper(), lower(), swapcase(), capitalize() выполняют преобразование регистра строки:

```
string1 = 'FastEthernet'

string1.upper()
#'FASTETHERNET'

string1.lower()
#'fastethernet'

string1.swapcase()
#'fASTeTHERNET'

string2 = 'tunnel 0'

string2.capitalize()
#'Tunnel 0'

#Очень важно обращать внимание на то, что часто методы возвращают преобразованную строку. И,␣
↪значит, надо не забыть присвоить ее какой-то переменной (можно той же).
string1 = string1.upper()
print(string1)
#FASTETHERNET
```

Метод count

Метод count() используется для подсчета того, сколько раз символ или подстрока встречаются в строке:

```
string1 = 'Hello, hello, hello, hello'
string1.count('hello') # 3
string1.count('ello') # 4
string1.count('l') # 8
```

Метод find

Методу find() можно передать подстроку или символ, и он покажет, на какой позиции находится первый символ подстроки (для первого совпадения):

```
string1 = 'interface FastEthernet0/1'
string1.find('Fast') # 10
string1[string1.find('Fast')::] # 'FastEthernet0/1'
```

Методы startswith, endswith

Проверка на то, начинается или заканчивается ли строка на определенные символы (методы startswith(), endswith()):

```
string1 = 'FastEthernet0/1'
string1.startswith('Fast') # True
string1.startswith('fast') # False
string1.endswith('0/1') # True
string1.endswith('0/2') # False
```

Метод replace

Замена последовательности символов в строке на другую последовательность (метод replace()):

```
string1 = 'FastEthernet0/1'
string1.replace('Fast', 'Gigabit') # 'GigabitEthernet0/1'
```

Метод strip

Часто при обработке файла файл открывается построчно. Но в конце каждой строки, как правило, есть какие-то спецсимволы (а могут быть и в начале). Например, перевод строки.

Для того, чтобы избавиться от них, очень удобно использовать метод strip():

```
string1 = '\n\tinterface FastEthernet0/1\n'
print(string1)
#
#interface FastEthernet0/1
#

string1.strip()
#'interface FastEthernet0/1'
```

Примечание: По умолчанию метод strip() убирает пробельные символы. В этот набор символов входят: tnrfv

Методу strip можно передать как аргумент любые символы. Тогда в начале и в конце строки будут удалены все символы, которые были указаны в строке:

Метод strip() убирает спецсимволы и в начале, и в конце строки. Если необходимо убрать символы только слева или только справа, можно использовать, соответственно, методы lstrip() и rstrip().

Метод split

Метод split() разбивает строку на части, используя как разделитель какой-то символ (или символы) и возвращает список строк:

```
string1 = 'switchport trunk allowed vlan 10,20,30,100-200'
commands = string1.split()
print(commands) # ['switchport', 'trunk', 'allowed', 'vlan', '10,20,30,100-200']

#По умолчанию в качестве разделителя используются пробельные символы (пробелы, табы, перевод␣
↪строки), но в скобках можно указать любой разделитель:
vlans = commands[-1].split(',')
print(vlans) #['10', '20', '30', '100-200']
```

Метод join

Метод join() позволяет объеденить список, кортеж или словарь в строку разделяя ее елементы другой строкой.

```
myTuple = ("John", "Peter", "Vicky")
x = "-".join(myTuple)
print(x) #John-Peter-Vicky
```

Метод format

Метод format() позволяет подставлять в отмеченные в строке области символами «{}» значени из списка аргументов

например:

```
price = 49
txt = "The price is {} dollars"
print(txt.format(price))
```

Так же можно указать тип подставлемых значений:

```
#Строковые занчения
'{} {}'.format('one', 'two') #one two

#Числовые значения
'{} {}'.format(1, 2) #1 2

#порядок значений можно указывать
'{1} {0}'.format('one', 'two') #two one

#Можно так же подставлять значения классов
class Data(object):

    def __str__(self):
        return 'str'

    def __repr__(self):
        return 'repr'

'{0!s} {0!r}'.format(Data()) #str repr

#Отступы и выравнивания
#по правому краю
'{:>10}'.format('test') #      test

#по левому краю
'{:_<10}'.format('test') #test______

#по центру
'{:^10}'.format('test') #   test

#срезы
'{:.5}'.format('xylophone') #xylop

#срезы и отступы
'{:10.5}'.format('xylophone') #xylop

#числа
'{:d}'.format(42) #42
'{:f}'.format(3.141592653589793) #3.141593

#числа и отступы
```

(continues on next page)

(продолжение с предыдущей страницы)

```
'{:4d}'.format(42) #  42
'{:06.2f}'.format(3.141592653589793) #003.14
'{:04d}'.format(42) # 0042

#знаковые числа
'{:+d}'.format(42) #+42
'{: d}'.format((- 23)) #-23
'{: d}'.format(42) # 42
'{:=5d}'.format((- 23)) #-  23
'{:=+5d}'.format(23) #+  23

#можно вставлять значения по именам
data = {'first': 'Hodor', 'last': 'Hodor!'}
'{first} {last}'.format(**data) #Hodor Hodor!
'{first} {last}'.format(first='Hodor', last='Hodor!') #Hodor Hodor!

#Формат даты и времени
from datetime import datetime
'{:%Y-%m-%d %H:%M}'.format(datetime(2001, 2, 3, 4, 5)) #2001-02-03 04:05
```

другие примеры форматированого вывода можно ныйти по следующим ссылкам pyformat.info w3schools.com

Пример программы

Подсчет слов

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#Программа подсчета слов в файле
import os #подуль для взаимодействия с операционной системой, например просмотр файлов


def get_words(filename):
    '''функци принимает в качестве аргумента путь до файла'''
    with open(filename, encoding="utf8") as file: #эта строка открывает файл
        text = file.read()  #читаем содержимое файла и записываем все в переменную text
        text = text.replace("\n", " ") #преобразуем наш текст в одну длинную строку заменив символ␣
↪перевода строки на пробел
        text = text.replace(",", "").replace(".", "").replace("?", "").replace("!", "") #а так же␣
↪уберем все запетые, пробелы, и прочие знаки пунктуации
        text = text.lower()  #переведем все слова в строчные, тоесть если было "Начало изучения␣
↪Языка Программирования", то будет "начало изучения языка программирования"
        words = text.split() #создадим список слов ("списки","выгледят","вот","так")
        words.sort() #метод sort отсортирует все слова по алфовиту
    return words  #эта строка вернет нам массив строк


def get_words_dict(words):
    ''''эта функция содасть словарь где ключь, это слово, а значение, то сколько раз оно␣
↪встречается в тексте'''
    words_dict = dict() #создаем простой словарь

    for word in words:  #цикл который перебирает наш список и записывает каждое слово в отдельную␣
↪переменную word
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        if word in words_dict:  #проверим условие, если слово уже есть в словаре,
            words_dict[word] = words_dict[word] + 1 # увеличим его значение на 1
        else:
            words_dict[word] = 1 # если не встречается, то просто присвоим значение 1
    return words_dict #вернем наш словарь


def main():
    '''наша главная функция в которой выполняется основная логика нашей программы
    Вызываются все созданные ранее функции, передаются значения, выводятся на экран результаты'''
    filename = input("Введите путь к файлу: ") #функция input() выводит на экран сообщение и␣
↪ожидает ввода, все что будет введенобудет записано в переменную filename
    if not os.path.exists(filename): #проверяем соществует ли наш файл на диске
        print("Указанный файл не существует")
    else:
        words = get_words(filename)  #получаем список слов
        words_dict = get_words_dict(words) #получаем словарь из слов и их количества в тексте
        print("Кол-во слов: %d" % len(words)) #печатаем количество слов в тексте
        print("Кол-во уникальных слов: %d" % len(words_dict)) #печатаем количество уникальных слов
        print("Все использованные слова:") # и все слова
        for word in words_dict:
            print(word.ljust(20), words_dict[word])


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

2.3.4 Списки

Списки – это изменяемые упорядоченные последовательности произвольных объектов. Списки создаются посредством заключения элементов списка в квадратные скобки

```
names = [ “Dave”, “Mark”, “Ann”, “Phil” ]
```

Элементы списка индексируются целыми числами, первый элемент списка имеет индекс, равный нулю. Для доступа к отдельным элементам списка используется оператор индексирования

```
a = names[2] # Вернет третий элемент списка, “Ann”
names[0] = “Jeff” # Запишет имя “Jeff” в первый элемент списка

С помощью оператора среза можно извлекать и изменять целые фрагменты списков:
```

```
b = names[0:2]     # Вернет [“Jeff”, “Mark”]
c = names[2:]      # Вернет [“Thomas”, “Ann”, “Phil”, “Paula”]
names[1] = ‘Jeff’  # Во второй элемент запишет имя ‘Jeff’
names[0:2] = [‘Dave’,’Mark’,’Jeff’]  # Заменит первые два элемента списком справа.
```

Списки могут содержать объекты любого типа, числа, строки, другие списки

```
a = [1,”Dave”,3.14, [“Mark”, 7, 9, [100,101]], 10]
```

Списки так же как и строки можно конкатинировать между собоой

```
[1,2,3] + [4,5] # Создаст список [1,2,3,4,5]
```

Меторы для работы со списками

List()

создаст пустой список, либо преобразует аргументы в список

```
l = list('1234567890')
print(l) # [1,2,3,4,5,6,7,8,9,0]
```

join()

собирает список строк в одну строку с разделителем, который указан перед join:

```
a = ['10', '20', '30']
print(','.join(a)) # 10,20,30
```

Примечание: Метод join на самом деле относится к строкам, но так как значение ему надо передавать как список, он рассматривается тут.

---

append()

добавляет в конец списка указанный элемент:

```
a = ['10', '20', '30', '100-200']
a.append('300')
print(a) # ['10', '20', '30', '100-200', '300']
```

extend()

Если нужно объединить два списка, то можно использовать два способа: метод extend() и операцию сложения.

У этих способов есть важное отличие - extend меняет список, к которому применен метод, а суммирование возвращает новый список, который состоит из двух.

```
a = ['10', '20', '30', '100-200']
b = ['300', '400', '500']
a.extend(b)
print(a) # ['10', '20', '30', '100-200', '300', '400', '500']
```

pop()

удаляет элемент, который соответствует указанному номеру. Но, что важно, при этом метод возвращает этот элемент:

```
a = ['10', '20', '30', '100-200']
a.pop(-1) # '100-200'
print(a) #['10', '20', '30']
```

Примечание: Без указания номера удаляется последний элемент списка.

---

remove()

удаляет указанный элемент и не возвращает удаленный элемент:

```
a = ['10', '20', '30', '100-200']
a.remove('20')
print(a) # ['10', '30', '100-200']
```

Примечание: В методе remove надо указывать сам элемент, который надо удалить, а не его номер в списке. Если указать номер элемента, возникнет ошибка:

insert()

позволяет вставить элемент на определенное место в списке:

```
a = ['10', '20', '30', '100-200']
a.insert(1, '15')
print(a) # ['10', '15', '20', '30', '100-200']
```

sort()

сортирует список на месте:

```
a = [1, 50, 10, 15]
a.sort()
print(a) # [1, 10, 15, 50]
```

2.3.5 Словари

Словари Python — нечто совершенно иное; они вообще не являются последовательностями и взамен известны как отображения. Отображения также представляют собой коллекции других объектов, но они хранят объекты по ключам, а не по относительным позициям. В действительности отображения не поддерживают какой-либо надежный порядок слева направо; они просто отображают ключи на связанные значения. Словари — единственный тип отображения в наборе основных объектов Python — являются изменяемыми,' как и списки, их можно модифицировать на месте и они способны увеличиваться и уменьшаться по требованию. Наконец, подобно спискам словари — это гибкий инструмент для представления коллекций, но их мнемонические ключи лучше подходят, когда элементы коллекции именованы или помечены, скажем, как поля в записи базы данных.

Словари - это изменяемый упорядоченный тип данных:

- данные в словаре - это пары ключ: значение
- доступ к значениям осуществляется по ключу, а не по номеру, как в списках
- данные в словаре упорядочены по порядку добавления элементов
- так как словари изменяемы, то элементы словаря можно менять, добавлять, удалять
- ключ должен быть объектом неизменяемого типа: число, строка, кортеж
- значение может быть данными любого типа

для удобства словарь можно записать так:

```
london = {
'id': 1,
'name':'London',
'it_vlan':320,
'user_vlan':1010,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
'mngmt_vlan':99,
'to_name': None,
'to_id': None,
'port':'G1/0/11'
}
```

Для того, чтобы получить значение из словаря, надо обратиться по ключу, таким же образом, как это было в списках, только вместо номера будет использоваться ключ:

```
london = {'name': 'London1', 'location': 'London Str'}
print(london['name'], london['location'])
#'London1' 'London Str'
```

При написании в виде литералов словари указываются в фигурных скобках и состо ят из ряда пар "ключ: значение". Словари удобны всегда, когда нам необходимо ассо циировать набор значений с ключами — например, для описания свойств чего-нибудь. Рассмотрим следующий словарь их трех элементов (с ключами ' food ', ' quantity ' и ' color ', возможно представляющих детали позиции гипотетического меню):

```
D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```

Мы можем индексировать этот словарь по ключу, чтобы извлекать и изменять зна чения, связанные с ключами. Операция индексации словаря имеет такой же синтак сис, как для последовательностей, но элементом в квадратных скобках будет ключ, а не относительная позиция:

```
D['food'] # Извлечь значение, связанное с ключом 'food' 'Spam'
D['quantity'] += 1 # Добавить 1 к значению, связанному с ключом 'quantity'
print(D)
#{'color': 'pink', 'food': 'Spam', 'quantity': 5}
```

Хотя форма литерала в фигурных скобках встречается, пожалуй, чаще приходится видеть словари, построенные другими способами (все данные программы редко из вестны до ее запуска). Скажем, следующий код начинает с пустого словаря и заполняет его по одному ключу за раз. В отличие от присваивания элементу в списке, находящемуся вне установленных границ, которое запрещено, присваивание новому ключу словаря приводит к созданию этого ключа:

```
D = {}
D['name'] = 'Bob'
D['job'] = 'dev'
D['age'] =40
print (D)
#{'*:age 40, 'job': 'dev', 'name': 'Bob'}
print(D['name'])
#Bob
```

В словаре в качестве значения можно использовать словарь:

```
london_co = {
'r1': {
    'hostname': 'london_r1',
    'location': '21 New Globe Walk',
    'vendor': 'Cisco',
    'model': '4451',
    'ios': '15.4',
    'ip': '10.255.0.1'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
},
'r2': {
    'hostname': 'london_r2',
    'location': '21 New Globe Walk',
    'vendor': 'Cisco',
    'model': '4451',
    'ios': '15.4',
    'ip': '10.255.0.2'
},
'sw1': {
    'hostname': 'london_sw1',
    'location': '21 New Globe Walk',
    'vendor': 'Cisco',
    'model': '3850',
    'ios': '3.6.XE',
    'ip': '10.255.0.101'
}
```

}

Получить значения из вложенного словаря можно так:

```
london_co['r1']['ios'] #'15.4'
london_co['r1']['model'] #'4451'
london_co['sw1']['ip'] #'10.255.0.101'
```

2.3.6 Методы для работы со словарями

clear()

- позволяет очистить словарь:

copy()

- создает полную копию словаря

```
london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}
london2 = london.copy()
id(london) #25524512
id(london2) #25563296
london['vendor'] = 'Juniper'
london2['vendor'] #'Cisco'
```

Примечание: Если указать, что один словарь равен другому, то london2 будет ссылкой на словарь. И при изменениях словаря london меняется и словарь london2, так как это ссылки на один и тот же объект.

---

get()

- запрашивает ключ, и если его нет, вместо ошибки возвращает None.

```
london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}
print(london.get('ios')) #None
```

(continues on next page)

(продолжение с предыдущей страницы)

```
#Метод get() позволяет также указывать другое значение вместо None
print(london.get('ios', 'Ooops')) #Ooops
```

setdefault()

- ищет ключ, и если его нет, вместо ошибки создает ключ со значением None, если ключ есть, setdefault возвращает значение, которое ему соответствует:

```
london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}
ios = london.setdefault('ios')
print(ios) #None
london #{'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco', 'ios': None}
london.setdefault('name') #'London1'

#Второй аргумент позволяет указать, какое значение должно соответствовать ключу
model = london.setdefault('model', 'Cisco3580')
print(model) #Cisco3580
london
{'name': 'London1',
 'location': 'London Str',
 'vendor': 'Cisco',
 'ios': None,
 'model': 'Cisco3580'}

# Метод setdefault заменяет такую конструкцию:
if key in london:
     value = london[key]
else:
     london[key] = 'somevalue'
     value = london[key]
```

keys(), values(), items()

Все три метода возвращают специальные объекты view, которые отображают ключи, значения и пары ключ-значение словаря соответственно.

Очень важная особенность view заключается в том, что они меняются вместе с изменением словаря. И фактически они лишь дают способ посмотреть на соответствующие объекты, но не создают их копию.

```
london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}
keys = london.keys()
print(keys)
#dict_keys(['name', 'location', 'vendor'])

#Сейчас переменной keys соответствует view dict_keys, в котором три ключа: name, location и vendor.
#Но, если мы добавим в словарь еще одну пару ключ-значение, объект keys тоже поменяется:
london['ip'] = '10.1.1.1'
keys
#dict_keys(['name', 'location', 'vendor', 'ip'])

#Если нужно получить обычный список ключей, который не будет меняться с изменениями словаря,␣
↪достаточно конвертировать view в список:
list_keys = list(london.keys())
list_keys
#['name', 'location', 'vendor', 'ip']
```

del()

Удаляет ключ и значение

```
london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}
del london['name']
london
#{'location': 'London Str', 'vendor': 'Cisco'}
```

update()

Позволяет добавлять в словарь содержимое другого словаря:

```
r1 = {'name': 'London1', 'location': 'London Str'}
r1.update({'vendor': 'Cisco', 'ios':'15.2'})
r1
# {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco', 'ios': '15.2'}

#Аналогичным образом можно обновить значения:
r1.update({'name': 'london-r1', 'ios':'15.4'})
r1
'''
{'name': 'london-r1',
 'location': 'London Str',
 'vendor': 'Cisco',
 'ios': '15.4'}
'''
```

2.3.7 Кортежи

Объект кортежа примерно похож на список, который нельзя изменять — кортежи являются последовательностями подобно спискам, но они неизменяемые подобно строка ми. Функционально они используются для представления фиксированных коллекций элементов: скажем, компонентов специфической даты в календаре. Синтаксически они записываются в круглых, а не квадратных скобках и поддерживают произвольные типы, произвольное вложение и обычные операции над последовательностями:

```
T = (1, 2, 3, 4) # Кортеж из 4 элементов
len(T)           # Длина 4
T + (5, 6)       # Конкатенация (1, 2, 3, 4, 5, 6)
T[0]             # Индексация, нарезание и т.д.
```

Примечание: Главное отличие кортежей заключается в том, что после создания их нельзя из менять, т.е. они являются неизменяемыми последовательностями (одноэлементные кортежи вроде приведенного ниже требуют хвостовой запятой): tuple2 = („password“,)

Итак, зачем нам тип, который похож на список, но поддерживает меньше операций? Откровенно говоря, на практике кортежи применяются в целом не так часто, как списки, но весь смысл в их неизменяемости. Если вы передаете коллекцию объектов внутри своей программы в виде списка, тогда он может быть модифицирован где угодно; если вы используете кортеж, то изменить его не удастся. То есть кортежи обеспечивают своего рода ограничение целостности, что удобно в программах, крупнее тех, которые мы будем писать здесь. Позже в книге мы еще обсудим кортежи, включая расширение, которое построено поверх них и называется именованными кортежами.

2.3.8 Множество

Множество - это изменяемый неупорядоченный тип данных. В множестве всегда содержатся только уникальные элементы.

Множество в Python - это последовательность элементов, которые разделены между собой запятой и заключены в фигурные скобки.

С помощью множества можно легко убрать повторяющиеся элементы:

```
cities = ['Санкт-Петербург', 'Хабаровск', 'Казань', 'Санкт-Петербург', 'Казань']
un_cities = set(cities)
for city in un_cities:
    print("Один мой друг живёт в городе " + city)

'''
Один мой друг живёт в городе Хабаровск
Один мой друг живёт в городе Санкт-Петербург
Один мой друг живёт в городе Казань
'''
```

Множества полезны тем, что с ними можно делать различные операции и находить объединение множеств, пересечение и так далее.

Объединение множеств можно получить с помощью метода union() или оператора |:

```
vlans1 = {10,20,30,50,100}
vlans2 = {100,101,102,102,200}
vlans1.union(vlans2) #{10, 20, 30, 50, 100, 101, 102, 200}
vlans1 | vlans2 #{10, 20, 30, 50, 100, 101, 102, 200}

#Пересечение множеств можно получить с помощью метода intersection() или оператора &
vlans1 = {10,20,30,50,100}
vlans2 = {100,101,102,102,200}
vlans1.intersection(vlans2) #{100}
vlans1 & vlans2 #{100}
```

Предупреждение: Нельзя создать пустое множество с помощью литерала (так как в таком случае это будет не множество, а словарь): set1 = {} type(set1) #dict

Но пустое множество можно создать таким образом set2 = set() type(set2) #set

2.3.9 Методы для работы с множествами

add()

добавляет элемент во множество:

```
set1 = {10,20,30,40}
set1.add(50)
set1 #{10, 20, 30, 40, 50}
```

discard()

позволяет удалять элементы, не выдавая ошибку, если элемента в множестве нет

```
set1 #{10, 20, 30, 40, 50}
set1.discard(55)
set1 #{10, 20, 30, 40, 50}
set1.discard(50)
set1 #{10, 20, 30, 40}
```

clear()

очищает множество

```
set1 = {10,20,30,40}
set1.clear()
set1 #set()
```

2.3.10 Булевы значения

Булевы значения в Python это две константы True и False.

В Python истинными и ложными значениями считаются не только True и False.

- истинное значение:
- любое ненулевое число
- любая непустая строка
- любой непустой объект
- ложное значение:
- 0
- None
- пустая строка
- пустой объект

Остальные истинные и ложные значения, как правило, логически следуют из условия.

Для проверки булевого значения объекта, можно воспользоваться bool:

```
items = [1, 2, 3]
empty_list = []
bool(empty_list) #False
bool(items) #True
bool(0) #False
bool(1) #True
```

2.3.11 Преобразование типов

В Python есть несколько полезных встроенных функций, которые позволяют преобразовать данные из одного типа в другой.

int()

преобразует строку в int:

```
int("10") #10
int("11111111", 2) #255  С помощью функции int можно преобразовать и число в двоичной записи в
↪десятичную (двоичная запись должна быть в виде строки)
```

bin()

Преобразовать десятичное число в двоичный формат можно с помощью bin():

```
bin(10) #'0b1010'
bin(255) #'0b11111111'
```

hex()

Аналогичная функция есть и для преобразования в шестнадцатеричный формат:

```
hex(10) #'0xa'
hex(255) #'0xff'
```

list()

Функция list() преобразует аргумент в список:

```
list("string") #['s', 't', 'r', 'i', 'n', 'g']
list({1,2,3})  #[1, 2, 3]
list((1,2,3,4)) #[1, 2, 3, 4
```

set()

Функция set() преобразует аргумент в множество:

```
set([1,2,3,3,4,4,4,4]) #{1, 2, 3, 4}
set((1,2,3,3,4,4,4,4)) #{1, 2, 3, 4}
set("string string") #{' ', 'g', 'i', 'n', 'r', 's', 't'}
```

tuple()

Функция tuple() преобразует аргумент в кортеж:

```
tuple([1,2,3,4]) #(1, 2, 3, 4)
tuple({1,2,3,4}) #(1, 2, 3, 4)
tuple("string")  #('s', 't', 'r', 'i', 'n', 'g')
```

str()

Функция str() преобразует аргумент в строку:

```
str(10) #'10'
```

2.3.12 Проверка типов

isdigit()

Проверяет, сосоит ли строка из одних только цифр

```
"a".isdigit() #False
"a10".isdigit() #False
"10".isdigit() #True
```

isalpha()

Проверяет, состоит ли строка из одних букв:

```
"a".isalpha() #True
"a100".isalpha() #False
"a--  ".isalpha() #False
"a ".isalpha() #False
```

isalnum()

позволяет проверить, состоит ли строка из букв или цифр:

```
"a".isalnum() #True
"a10".isalnum() #True
```

type()

Иногда, в зависимости от результата, библиотека или функция может выводить разные типы объектов. Например, если объект один, возвращается строка, если несколько, то возвращается кортеж.

Нам же надо построить ход программы по-разному, в зависимости от того, была ли возвращена строка или кортеж.

В этом может помочь функция type():

```
type("string") #str
type("string") is str #True

#Аналогично с кортежем (и другими типами данных):
type((1,2,3)) #tuple
type((1,2,3)) is tuple #True
type((1,2,3)) is list #False
```

2.3.13 Файлы

Объекты файлов являются главным интерфейсом к внешним файлам на компью тере. Они могут применяться для чтения и записи текстовых заметок, аудиоклипов, документов Excel, сохраненных сообщений электронной почты и всего того, что вы в итоге сохранили на своем компьютере. Файлы относятся к основным типам, но они кое в чем своеобразны — специфический литеральный синтаксис для их создания от сутствует, Взамен, чтобы создать объект файла, необходимо вызвать встроенную фун кцию open, передав ей в виде строк имя внешнего файла и необязательный режим обработки.

Например, для создания выходного текстового файла понадобится передать его имя и строку режима обработки ' w ', чтобы записывать данные:

```
f = open('data.txt' , 'w') # Создать новый файл в режиме записи ('w')
f.write('Hello\n') # Записать в него строки символов
f.write ('world\n') # Возвратить количество записанных элементов
f.close()   # Закрыть для сбрасывания буферов вывода на диск
```

Код создает файл в текущем каталоге и записывает в него текст (имя файла мо жет содержать полный путь к каталогу, если нужно получить доступ к файлу где-то в другом месте на компьютере). Чтобы прочитать то, что было записано, необходимо повторно открыть файл в режиме обработки ' r ' для чтения текстового ввода (он вы бирается по умолчанию, если в вызове строка режима не указана). Затем следует про читать содержимое файла в строку и отобразить ее. В сценарии содержимое файла всегда будет строкой независимо от типа находящихся в нем данных:

```
f = open('data.txt')     # 'r' (чтение) - стандартный режим обработки
text = f.read()          # Прочитать все содержимое файла в строку
text                     #'Hello\nworld\n'
print(text)              # print интерпретирует управляющие символы
#Hello
#world
text.split()             # Содержимое файла - всегда строка
#['Hello', 'world']
```

Примечание: Обратите внимание, что для того что бы прочесь все строки из файла, нам нужно обернуть функцию чтения в цикл for line in open('data.txt'): print(line)

так же не забывайте закрывать файл после операции чтения или записи f.close

Примечание: Ранее в примере с подсчетом слов мы использовали друую констукцию

with open(filename, encoding=»utf8») as file: #эта строка открывает файл text = file.read() #читаем содержимое файла и записываем все в переменную text

text = text.replace(«n», » «) #преобразуем наш текст в одну длинную строку заменив символ перевода строки на пробел

text = text.replace(«,», «»).replace(«.», «»).replace(«?», «»).replace(«!», «») #а так же уберем все запетые, пробелы, и прочие знаки пунктуации

text = text.lower() #перведем все слова в строчные, тоесть если было «Начало изучения Языка Программирования», то будет «начало изучения языка программирования»

words = text.split() #создадим список слов («списки»,»выгледят»,»вот»,»так»)

words.sort()

Такой подход позволяет не закрывать файл в ручную.

2.4 Операторы языка Python

Большинство предложений (логических строк) в программах содержат выражения. Простой пример выражения: 2 + 3. Выражение можно разделить на операторы и операнды.

Операторы – это некий функционал, производящий какие-либо действия, который может быть представлен в виде символов, как например +, или специальных зарезервированных слов. Операторы могут производить некоторые действия над данными, и эти данные называются операндами. В нашем случае 2 и 3 – это операнды.

2.4.1 Базовые операторы

далее стоит привести таблицу операторов:

Оператор	Название	Объяснение	Примеры
„+“	Сложение	Суммирует два объекта	3 + 5 даст 8; „a“ + „b“ даст „ab“
„-„	Вычитание	**Даёт разность двух чисел; если первый операнд** отсутствует, он считается равным нулю	-5.2 даст отрицательное число, а 50 - 24 даст 26.
„*“	Умножение	**Даёт произведение двух чисел или возвращает** строку, повторённую заданное число раз.	2 * 3 даст 6. „la“ * 3 даст „lalala“.
„**“	Возведение в степень	Возвращает число x, возведённое в степень y	3** 4 даст 81 (т.е. 3 * 3 * 3 * 3)
/	Деление	Возвращает частное от деления x на y	4 / 3 даст 1.3333333333333333.
//	Целочисленное деление	Возвращает неполное частное от деления	4 // 3 даст 1. -4 // 3 даст -2.
%	Деление по модулю	Возвращает остаток от деления	8 % 3 даст 2. -25.5 % 2.25 даст 1.5.
<<	Сдвиг влево	Сдвигает биты числа влево на заданное количество позиций. (Любое число в памяти компьютера представлено в виде битов - или двоичных чисел, т.е. 0 и 1)	2 << 2 даст 8. В двоичном виде 2 представляет собой 10. Сдвиг влево на 2 бита даёт 1000, что в десятичном виде означает 8.
>>	Сдвиг вправо	Сдвигает биты числа вправо на заданное число позиций.	11 >> 1 даст 5. В двоичном виде 11 представляется как 1011, что будучи смещённым на 1 бит вправо, даёт 101, а это, в свою очередь, не что иное как десятичное 5
&	Побитовое И	Побитовая операция И над числами	5 & 3 даёт 1.
„|“	Побитовое ИЛИ	Побитовая операция ИЛИ над числами	5 | 3 даёт 7
^	Побитовое ИСКЛЮЧИТЕЛЬНО ИЛИ	Побитовая операция ИСКЛЮЧИТЕЛЬНО ИЛИ	5 ^ 3 даёт 6
~	Побитовое НЕ	Побитовая операция НЕ для числа x соответствует -(x+1)	~5 даёт -6.
<	Меньше	Определяет, верно ли, что x меньше y. Все операторы сравнения возвращают True или False [1]. Обратите внимание на заглавные буквы в этих словах.	5 < 3 даст False, а 3 < 5 даст True. Можно составлять произвольные цепочки сравнений: 3 < 5 < 7 даёт True.
>	Больше	Определяет, верно ли, что x больше y	5 > 3 даёт True. Если оба операнда - чис-

2.4.2 Управляющие операторы

Условные операторы (if/else)

Оператор if используется для проверки условий: если условие верно, выполняется блок выражений (называемый “if-блок”), иначе выполняется другой блок выражений (называемый “else-блок”). Блок “else” является необязательным.

Предупреждение: В языке Python блоки разедяются табами или пробелами

Запомните эмпирическое правило: хотя вы можете использовать для отступов пробелы или табуляции, их смешивание внутри блока обычно не будет удачной идеей применяйте либо то, либо другое. Формально табуляция считается достаточным количеством пробелов, чтобы сместить текущую строку на расстояние, кратное 8, и код будет работать в случае согласованного смешивания табуляций и пробелов. Тем не менее, такой код может быть сложнее изменять. Хуже того, смешивание табуляций и пробелов затрудняет чтение кода целиком, не говоря уже о правилах синтаксиса Python — табуляции в редакторе сменившего вас программиста могут выглядеть сов сем не так, как в вашем редакторе.

Пример использования оператора if

```
number = 23
guess = int(input('Введите целое число : '))

if guess == number:
    print('Поздравляю, вы угадали,') # Здесь начинается новый блок
    print('(хотя и не выиграли никакого приза!)') # Здесь заканчивается новый блок
elif guess < number:
    print('Нет, загаданное число немного больше этого.') # Ещё один блок
    # Внутри блока вы можете выполнять всё, что угодно ...
else:
    print('Нет, загаданное число немного меньше этого.')
    # чтобы попасть сюда, guess должно быть больше, чем number

print('Завершено')
# Это последнее выражение выполняется всегда после выполнения оператора if
```

Оператор while

Оператор while — самая универсальная конструкция для итераций в языке Python. Выражаясь простыми терминами, он многократно выполняет блок операторов (обыч но с отступом) до тех пор, пока проверка в заголовочной части оценивается как истин ное значение. Это называется “циклом”, потому что управление продолжает возвра щаться к началу оператора, пока проверка не даст ложное значение. Когда результат проверки становится ложным, управление переходит на оператор, следующий после блока while. Совокупный эффект в том, что тело цикла выполняется многократно, пока проверка в заголовочной части дает истинное значение. Если проверка оцени вается в ложное значение с самого начала, тогда тело цикла никогда не выполнится и оператор while пропускается.

В своей самой сложной форме оператор while состоит из строки заголовка с вы ражением проверки, тела с одним или большим количеством оператором с отступами и необязательной части else, которая выполняется, если управление покидает цикл, а оператор break не встретился. Python продолжает оценивать выражение проверки в строке заголовка и выполняет операторы, вложенные в тело цикла, пока проверка не возвратит ложное значение:

```
number = 23
running = True

while running:
    guess = int(input('Введите целое число : '))

    if guess == number:
        print('Поздравляю, вы угадали.')
        running = False # это останавливает цикл while
    elif guess < number:
        print('Нет, загаданное число немного больше этого.')
    else:
        print('Нет, загаданное число немного меньше этого.')
else:
    print('Цикл while закончен.')
    # Здесь можете выполнить всё что вам ещё нужно

print('Завершение.')
```

Цикл for

Оператор for..in также является оператором цикла, который осуществляет итерацию по последовательности объектов, т.е. проходит через каждый элемент в последовательности. Мы узнаем больше о последовательностях в дальнейших главах, а пока просто запомните, что последовательность – это упорядоченный набор элементов.

```
#Расмотрим несколько примеров

for x in ["spam", "eggs", "ham"]:
    print(x, end=' ')
#результатом этого цикла будет строка spam eggs ham
```

Примечание: оратите мнимание на параметр end=“ „, по умолчанию фукция print() завершает вывод символом конца строки «n», в случае с end=“ „ мы меняем его на пробел. Таким обазом мы выведем сообщения на экран в строку.

```
for i in range(1, 5):
    print(i)
else:
    print('Цикл for закончен')
```

Вложенные циклы for

Давайте теперь взглянем на цикл for, который сложнее тех, что мы видели до сих пор. В приведенном ниже примере иллюстрируется вложение операторов и конструк ция else цикла for. Имея список объектов (items) и список ключей (tests), код ищет каждый ключ в списке объектов и сообщает о результате поиска:

```
items = ["aaa", 111, (4, 5), 2.01]
tests = [(4, 5) , 3.14]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
for key in tests:
    for item in items:
        if item - key:
            print (key, ''was found'')
            break
        else:
            print(key, "not found!")
```

Оператор break

Оператор break служит для прерывания[7] цикла, т.е. остановки выполнения команд даже если условие выполнения цикла ещё не приняло значения False или последовательность элементов не закончилась.

Важно отметить, что если циклы for или while прервать оператором break, соответствующие им блоки else выполняться не будут.

```
while True:
    s = input('Введите что-нибудь : ')
    if s == 'выход':
        break
    print('Длина строки:', len(s))
print('Завершение')
```

Оператор continue

Оператор continue используется для указания Python, что необходимо пропустить все оставшиеся команды в текущем блоке цикла и продолжить[9] со следующей итерации цикла.

```
while True:
    s = input('Введите что-нибудь : ')
    if s == 'выход':
        break
    if len(s) < 3:
        print('Слишком мало')
        continue
    print('Введённая строка достаточной длины')
    # Разные другие действия здесь...
```

Ну и в качестве маленького примера давайте нарисуем в консоли Ёлочку :)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
'''программа рисует в консоли елочку'''
'''              *
               ***
              *****
             *******
            *********
           ***********
'''

#Пример - 1
for i in range(1, 20, 2):  #функция range() указывает начальное значение, конечное значение и шаг.
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    print('{:^20}'.format('*' * i))  # Печатаем символ выравнивая его по центру



#Пример -2
SPACE = ' '
STRAR = '*'

rows = int(input('Укажите размер ёлочки: '))
spaces = rows-1
stars = 1

for i in range(rows):
    print(
        (SPACE*spaces) +
        (STRAR*stars) +
        (SPACE*spaces)
    )
    stars += 2
    spaces -= 1
```

2.5 Функции языка Python

Функции – это многократно используемые фрагменты программы. Они позволяют дать имя определённому блоку команд с тем, чтобы впоследствии запускать этот блок по указанному имени в любом месте программы и сколь угодно много раз. Это называется вызовом функции. Мы уже использовали много встроенных функций, как то len и range.

Функция – это, пожалуй, наиболее важный строительный блок любой нетривиальной программы (на любом языке программирования), поэтому в этой главе мы рассмотрим различные аспекты функций.

Функции определяются при помощи зарезервированного слова def. После этого слова указывается имя функции, за которым следует пара скобок, в которых можно указать имена некоторых переменных, и заключительное двоеточие в конце строки. Далее следует блок команд, составляющих функцию. На примере можно видеть, что на самом деле это очень просто:

```
def sayHello():
    print('Привет, Мир!') # блок, принадлежащий функции
    # Конец функции

sayHello() # вызов функции
sayHello() # ещё один вызов функции
```

2.5.1 Параметры функций

Функции могут принимать параметры, т.е. некоторые значения, передаваемые функции для того, чтобы она что-либо сделала с ними. Эти параметры похожи на переменные, за исключением того, что значение этих переменных указывается при вызове функции, и во время работы функции им уже присвоены их значения.

Параметры указываются в скобках при объявлении функции и разделяются запятыми. Аналогично мы передаём значения, когда вызываем функцию. Обратите внимание на терминологию: имена, указанные в объявлении функции, называются параметрами, тогда как значения, которые вы передаёте в функцию при её вызове, – аргументами.

```
def printMax(a, b):
    if a > b:
        print(a, 'максимально')
    elif a == b:
        print(a, 'равно', b)
    else:
        print(b, 'максимально')

printMax(3, 4) # прямая передача значений

x = 5
y = 7

printMax(x, y) # передача переменных в качестве аргументов
```

2.5.2 Локальные переменные

При объявлении переменных внутри определения функции, они никоим образом не связаны с другими переменными с таким же именем за пределами функции – т.е. имена переменных являются локальными в функции. Это называется областью видимости переменной. Область видимости всех переменных ограничена блоком, в котором они объявлены, начиная с точки объявления имени.

```
x= 50

def func(x):
    print('x равен', x)
    x = 2
    print('Замена локального x на', x)

func(x)
print('x по-прежнему', x)
```

2.5.3 Зарезервированное слово “global”

Чтобы присвоить некоторое значение переменной, определённой на высшем уровне программы (т.е. не в какой-либо области видимости, как то функции или классы), необходимо указать Python, что её имя не локально, а глобально (global). Сделаем это при помощи зарезервированного слова global. Без применения зарезервированного слова global невозможно присвоить значение переменной, определённой за пределами функции.

Можно использовать уже существующие значения переменных, определённых за пределами функции (при условии, что внутри функции не было объявлено переменной с таким же именем). Однако, это

не приветствуется, и его следует избегать, поскольку человеку, читающему текст программы, будет непонятно, где находится объявление переменной. Использование зарезервированного слова global достаточно ясно показывает, что переменная объявлена в самом внешнем блоке.

```
x = 50

def func():
    global x

    print('x равно', x)
    x = 2
    print('Заменяем глобальное значение x на', x)

func()
print('Значение x составляет', x)
```

2.5.4 Зарезервированное слово “nonlocal”

Мы увидели, как получать доступ к переменным в локальной и глобальной области видимости. Есть ещё один тип области видимости, называемый “нелокальной” (nonlocal) областью видимости, который представляет собой нечто среднее между первыми двумя. Нелокальные области видимости встречаются, когда вы определяете функции внутри функций.

Поскольку в Python всё является выполнимым кодом, вы можете определять функции где угодно.

```
def func_outer():
    x = 2
    print('x равно', x)

    def func_inner():
        nonlocal x
        x = 5

    func_inner()
    print('Локальное x сменилось на', x)

func_outer()
```

2.5.5 Значения аргументов по умолчанию

Зачастую часть параметров функций могут быть необязательными, и для них будут использоваться некоторые заданные значения по умолчанию, если пользователь не укажет собственных. Этого можно достичь с помощью значений аргументов по умолчанию. Их можно указать, добавив к имени параметра в определении функции оператор присваивания (=) с последующим значением.

Обратите внимание, что значение по умолчанию должно быть константой. Или точнее говоря, оно должно быть неизменным[1] – это объясняется подробнее в последующих главах. А пока запомните это.

```
def say(message, times = 1):
    print(message * times)

say('Привет')
say('Мир', 5)
```

Предупреждение: Важно Значениями по умолчанию могут быть снабжены только параметры, находящиеся в конце списка параметров. Таким образом, в списке параметров функции параметр со значением по умолчанию не может предшествовать параметру без значения по умолчанию. Это связано с тем, что значения присваиваются параметрам в соответствии с их положением. Например, def func(a, b=5) допустимо, а def func(a=5, b) – не допустимо.

2.5.6 Ключевые аргументы

Если имеется некоторая функция с большим числом параметров, и при её вызове требуется указать только некоторые из них, значения этих параметров могут задаваться по их имени – это называется ключевые параметры. В этом случае для передачи аргументов функции используется имя (ключ) вместо позиции (как было до сих пор).

Есть два преимущества такого подхода: во-первых, использование функции становится легче, поскольку нет необходимости отслеживать порядок аргументов; во-вторых, можно задавать значения только некоторым избранным аргументам, при условии, что остальные параметры имеют значения аргумента по умолчанию.

```
def func(a, b=5, c=10):
    print('a равно', a, ', b равно', b, ', a с равно', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

2.5.7 Переменное число параметров

Иногда бывает нужно определить функцию, способную принимать любое число параметров. Этого можно достичь при помощи звёздочек (сохраните как function_varargs.py):

```
def total(a=5, *numbers, **phonebook):
    print('a', a)

    #проход по всем элементам кортежа
    for single_item in numbers:
        print('single_item', single_item)

    #проход по всем элементам словаря
    for first_part, second_part in phonebook.items():
        print(first_part,second_part)

print(total(10,1,2,3,Jack=1123,John=2231,Inge=1560))
```

2.5.8 Только ключевые параметры

Если некоторые ключевые параметры должны быть доступны только по ключу, а не как позиционные аргументы, их можно объявить после параметра со звёздочкой (сохраните как keyword_only.py):

```
def total(initial=5, *numbers, extra_number):
    count = initial
    for number in numbers:
        count += number
    count += extra_number
    print(count)

total(10, 1, 2, 3, extra_number=50)
total(10, 1, 2, 3)
# Вызовет ошибку, поскольку мы не указали значение
# аргумента по умолчанию для 'extra_number'.
```

2.5.9 Оператор “return”

Оператор return используется для возврата[5] из функции, т.е. для прекращения её работы и выхода из неё. При этом можно также вернуть некоторое значение из функции.

```
#!/usr/bin/python
# Filename: func_return.py

def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'Числа равны.'
    else:
        return y

print(maximum(2, 3))
```

2.5.10 Строки документации

Python имеет остроумную особенность, называемую строками документации, обычно обозначаемую сокращённо docstrings. Это очень важный инструмент, которым вы обязательно должны пользоваться, поскольку он помогает лучше документировать программу и облегчает её понимание. Поразительно, но строку документации можно получить, например, из функции, даже во время выполнения программы!

```
def printMax(x, y):
    '''Выводит максимальное из двух чисел.

    Оба значения должны быть целыми числами.'''
    x = int(x) # конвертируем в целые, если возможно
    y = int(y)

    if x > y:
        print(x, 'наибольшее')
    else:
        print(y, 'наибольшее')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
printMax(3, 5)
print(printMax.__doc__)
```

2.6 Модули

2.6.1 Файлы байткода .pyc

Импорт модуля – относительно дорогостоящее мероприятие, поэтому Python предпри- нимает некоторые трюки для ускорения этого процесса. Один из способов – создать байт-компилированные файлы (или байткод) с расширением .pyc, которые являются некой промежуточной формой, в которую Python переводит программу (помните раздел “Введение” о том, как работает Python?). Такой файл .pyc полезен при импорте модуля в следующий раз в другую программу – это произойдёт намного быстрее, поскольку зна- чительная часть обработки, требуемой при импорте модуля, будет уже проделана. Этот байткод также является платформо-независимым.

Примечание: Обычно файлы .pyc создаются в том же каталоге, где расположены и соот- ветствующие им файлы .py. Если Python не может получить доступ для записи файлов в этот каталог, файлы .pyc созданы не будут.

2.6.2 Оператор from ... import ...

Чтобы импортировать переменную argv прямо в программу и не писать всякий раз sys. при обращении к ней, можно воспользоваться выражением “from sys import argv”. Для импорта всех имён, использующихся в модуле sys, можно выполнить команду “from sys import *”. Это работает для любых модулей. В общем случае вам следует избегать использования этого оператора и использовать вместо этого оператор import, чтобы предотвратить конфликты имён и не затруднять чтение программы.

```
from math import *
n = input("Введите диапазон:- ") p = [2, 3]
count = 2
a=5
while (count < n):
    b=0
    for i in range(2,a):
        if ( i <= sqrt(a)): if (a % i == 0):
            print("a neprost",a)
            b=1
        else:
            pass
    if (b != 1):
        print("a prost",a) p = p + [a]
    count = count + 1
    a=a+2
print p
```

2.6.3 Имя модуля – __name__

У каждого модуля есть имя, и команды в модуле могут узнать имя их модуля. Это полез- но, когда нужно знать, запущен ли модуль как самостоятельная программа или импорти- рован. Как уже упоминалось выше, когда модуль импортируется впервые, содержащийся в нём код исполняется. Мы можем воспользоваться этим для того, чтобы заставить мо- дуль вести себя по-разному в зависимости от того, используется ли он сам по себе или импортируется в другую программа. Этого можно достичь с применением атрибута мо- дуля под названием __name__.

```
if __name__ == '__main__':
    print('Эта программа запущена сама по себе.')
else:
    print('Меня импортировали в другой модуль.')
```

Как это работает: В каждом модуле Python определено его имя – __name__ . Если оно равно „__main__“, это означает, что модуль запущен самостоятельно пользователем, и мы можем выполнить соответствующие действия.

2.6.4 Создание собственных модулей

Создать собственный модуль очень легко. Да вы всё время делали это! Ведь каждая про- грамма на Python также является и модулем. Необходимо лишь убедиться, что у неё уста- новлено расширение .py. Следующий пример объяснит это.

```
# mymodule.py

def sayhi():
    print('Привет! Это говорит мой модуль.')
__version__ = '0.1'
```

Выше приведён простой модуль. Как видно, в нём нет ничего особенного по сравнению с обычной программой на Python. Далее посмотрим, как использовать этот модуль в дру- гих наших программах. Помните, что модуль должен находиться либо в том же каталоге, что и программа, в ко- торую мы импортируем его, либо в одном из каталогов, указанных в sys.path.

```
# mymodule_demo.py

import mymodule

mymodule.sayhi()
print ('Версия', mymodule.__version__)
```

Примечание: Обратите внимание, что мы используем всё то же обозначение точкой для до- ступа к элементам модуля. Python повсеместно использует одно и то же обо- значение точкой, придавая ему таким образом характерный «Python-овый» вид и не вынуждая нас изучать всё новые и новые способы делать что-либо.

Вот версия, использующая синтаксис from..import

```
# mymodule_demo2.py

from mymodule import sayhi, __version__

```

(continues on next page)

(продолжение с предыдущей страницы)

```
sayhi()
print('Версия', __version__)
```

Примечание: Обратите внимание, что если в модуле, импортирующем данный модуль, уже было объ- явлено имя __version__, возникнет конфликт. Это весьма вероятно, так как объявлять версию любого модуля при помощи этого имени – общепринятая практика. Поэтому всегда рекомендуется отдавать предпочтение оператору import, хотя это и сделает вашу программу немного длиннее.

Вы могли бы также использовать from mymodule import *

Это импортирует все публичные имена, такие как sayhi, но не импортирует __version__, потому что оно начинается с двойного подчёркивания