

Введение в программирование на Python

Введение в язык Python

Язык **Python** – это высокоуровневый интерпретируемый язык программирования, нацеленный на повышение продуктивности разработки и читаемости кода. Синтаксис Python минималистичен, и в то же время язык обладает богатой стандартной библиотекой с широким набором функций.

Ввиду своей простоты и наглядности и большого количества библиотек по работе с данными Python на сегодня является одним из самых популярных языков для решения самых разных задач. В рамках курса ММАД Python будет использоваться для извлечения данных с сайтов, из баз данных и файлов различных форматов, преобразования и визуализации данных, построения математических моделей и оценки их точности.

Установка интерпретатора Python и необходимых модулей

Наиболее простой способ установить Python с необходимыми библиотеками – скачать дистрибутив **Anaconda**, который уже включает всё необходимое: Python, интерактивную оболочку **IPython**, набор библиотек для расчётов и моделирования и т.д.

Скачать дистрибутив Anaconda для своей операционной системы можно на сайте <http://continuum.io>.

Необходимо запустить загруженный инсталлятор. При установке все настройки можно оставить по умолчанию.

Можно ставить Python и не из дистрибутива Anaconda. Тогда для установки некоторых библиотек необходимо будет предварительно установить Build Tools, так как некоторые библиотеки представлены в виде исходного кода, который должен быть скомпилирован на целевой платформе.

Работа с интерпретатором Python

Дистрибутив Anaconda включает в себя среду **Spyder**. Среда позволяет редактировать скрипты Python и выполнять их, выполнять команды в консоли Python и IPython, отображает справку по функциям и параметрам функций, обладает автодополнением кода, визуальными подсказками, ведёт журнал выполненных команд, которые затем могут быть сохранены в скрипт и т.д.

При первом запуске Spyder может возникнуть ошибка:

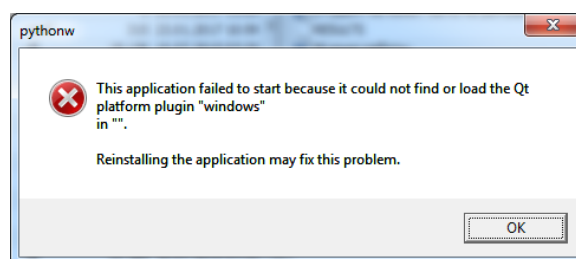


Рисунок 1 Сообщение об ошибке при запуске Spyder

Проблема решается следующим образом. Пусть Anaconda установлена в каталог **ANACONDA_DIR** (по умолчанию это **c:\Users\<user>\AppData\Local\Continuum\Anaconda3**). Тогда необходимо скопировать каталог **ANACONDA_DIR \Library\plugins\platforms** в каталог **ANACONDA_DIR**. После этого Spyder должен запускаться без проблем.

Вот как выглядит главное окно Spyder:

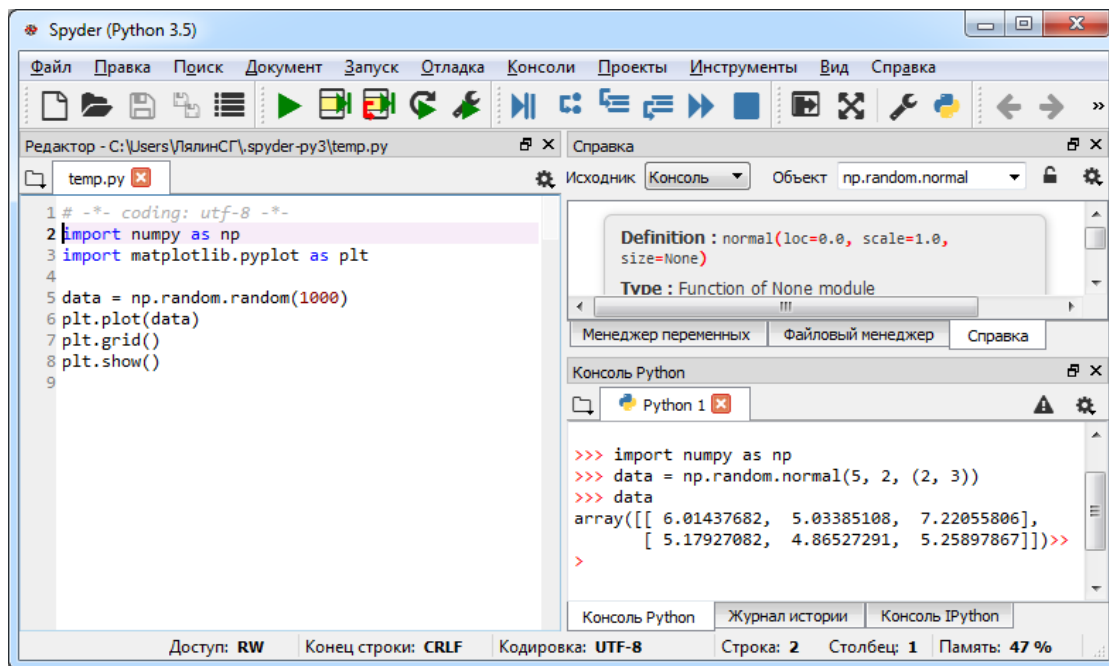


Рисунок 2 Главное окно Spyder

В левой части расположен редактор скриптов с подсветкой синтаксиса и автодополнением. Писать код можно в нём. В правой части экрана сверху можно посмотреть информацию по переменным, расположенным в памяти, их типу и размерности.

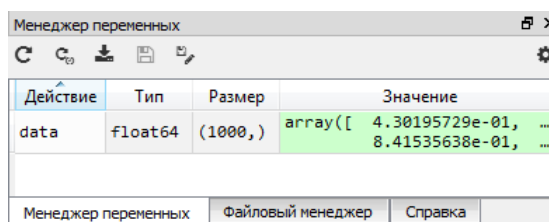


Рисунок 3 Менеджер переменных

Следующая вкладка позволяет просматривать структуру каталогов, выполнять операции с файлами, запускать Python для указанного каталога.

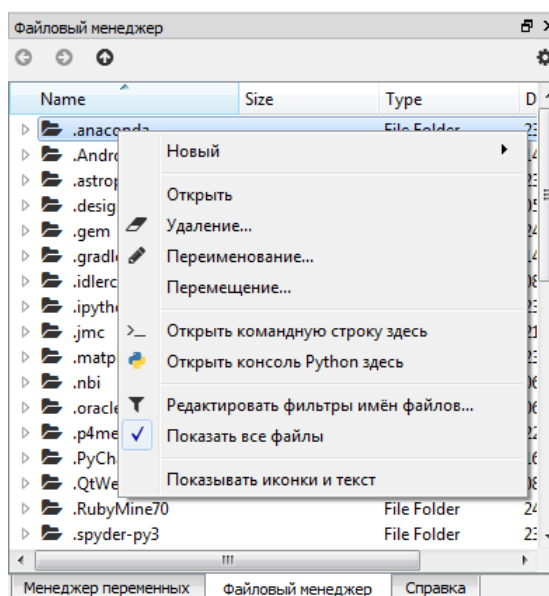


Рисунок 4 Файловый менеджер

Следующая вкладка отображает справку по указанному модулю или функции. Для отображения справки необходимо поставить курсор на интересующую функцию и нажать сочетание клавиш **Ctrl+I**.

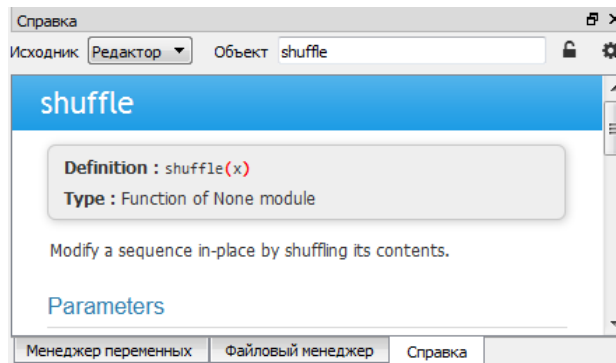


Рисунок 5 Отображение справки по функциям и модулям

Справку по заданному модулю или функции можно получить и из консоли Python (см. про консоль ниже), выполнив команду `help` с параметром, например

```
help('numpy.random.shuffle')
```

Справа внизу располагается консоль Python, журнал выполненных команд и консоль IPython. Здесь можно вводить свои команды и экспериментировать с возможностями языка.

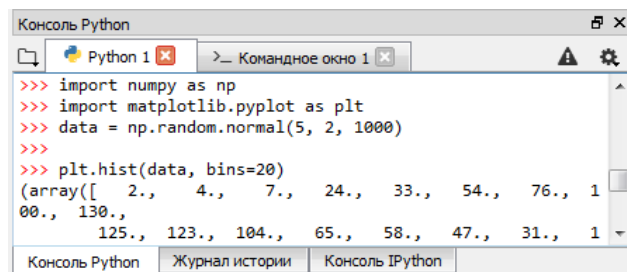


Рисунок 6 Стандартная консоль Python

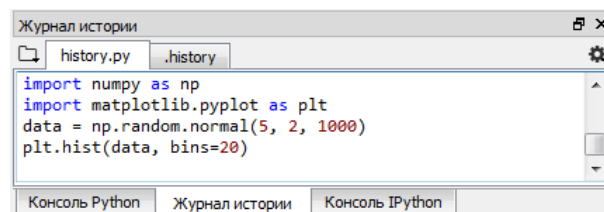


Рисунок 7 Журнал выполненных команд

Скрипты можно создавать не только в Spyder, но и в любом текстовом редакторе. Удобно для этого использовать редактор с подсветкой синтаксиса, например, **Sublime Text** или **Notepad++**. Кроме того, существуют полноценные **IDE** (интегрированные среды разработки с редактором кода, автодополнением, отладчиком, рефакторингом и т.д.). Одной из лучших бесплатных IDE на сегодня является **PyCharm** от JetBrains.

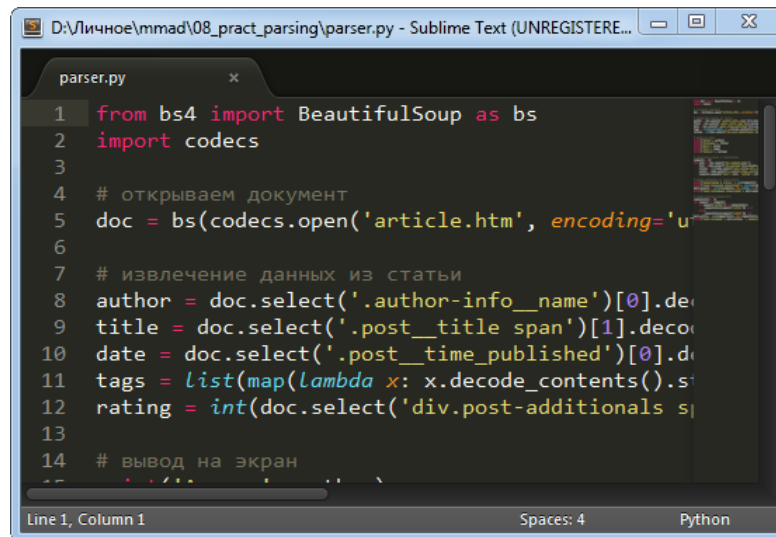


Рисунок 8 Окно редактора Sublime Text

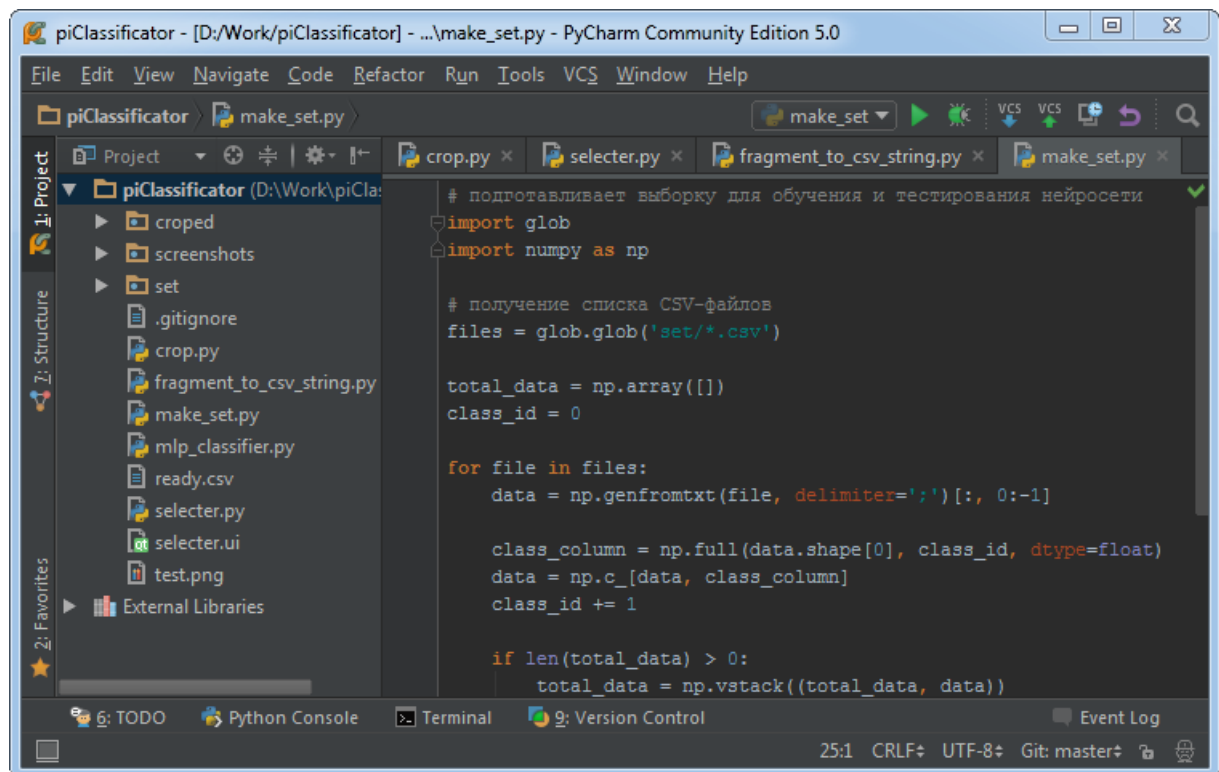


Рисунок 9 Окно IDE PyCharm

Запустить созданный скрипт (текстовый файл с расширением **.py**) можно через командную строку, выполнив в ней команду:

```
> python script.py
```

где **script.py** – запускаемый скрипт.

Вот пример запуска скрипта Python:

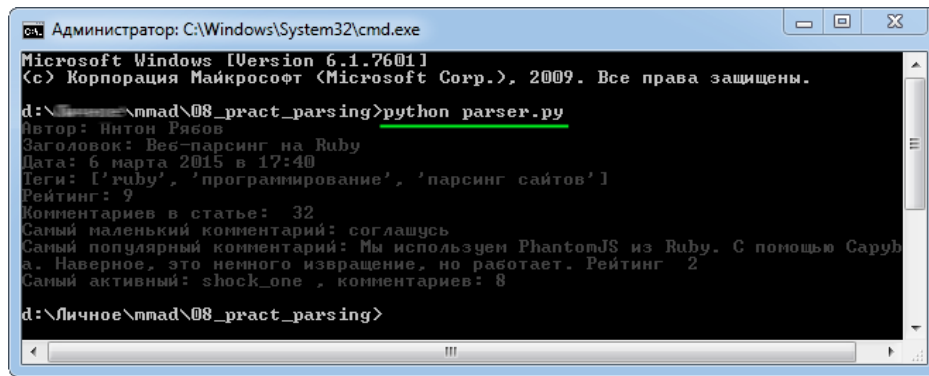


Рисунок 10 Запуск скрипта Python через командную строку

Справка по базовым возможностям языка Python

Переменные и типы данных

В Python, как и в любом другом языке программирования, можно объявлять переменные, массивы, словари. Например

```
a = 10 # переменная
b = [1, 2, 3] # массив
c = {'name': 'lemon', 'color': 'yellow'} # словарь
```

Как уже было сказано выше, Python является динамически типизируемым языком, поэтому тип переменных может измениться. Узнать текущий тип переменной можно с помощью встроенной функции **type**:

```
>>> a = 1
>>> type(a)
<class 'int'>
>>> a = [1, 2, 3]
>>> type(a)
<class 'list'>
>>> a = {'a': 1, 'b': 2, 'c': 3}
>>> type(a)
<class 'dict'>
>>> a = True
>>> type(a)
<class 'bool'>
>>>
```

Ещё одной интересной особенностью Python как динамического языка является то, что элементы массива или словаря могут хранить значения разных типов, например

```
>>> player = {'rating': 180, 'scores': [35, 18, 41], 'active': True}
>>> player['scores']
[35, 18, 41]
```

Полезной на практике возможностью является использование кортежей – групп переменных. Например, описать человека с помощью значений роста и веса можно так:

```
human = (182, 86)
human[0] # 182 – рост
human[1] # 86 – вес
```

Кортежи могут выступать в качестве параметров функций, например, вот так создаётся единичная матрица размера 2×3 с помощью библиотеки **NumPy**:

```
>>> import numpy as np
```

```
>>> m = np.ones(shape=(2, 3), dtype=int)
>>> m
array([[1, 1, 1],
       [1, 1, 1]])
```

Кортежи удобны, когда нужно вернуть несколько значений из функции, например

```
def max_(arr):
    max_idx, max_val = 0, 0

    for i in range(len(arr)):
        if arr[i] > max_val:
            max_val = arr[i]
            max_idx = i
    return max_val, max_idx

res = max_([2, 5, 4, 1])
print(res[0], res[1])
```

Этот код выведет следующие значения:

```
5 1
```

Это максимальное значение в массиве и его индекс.

Приведённый выше код интересен по нескольким причинам. Во-первых, он показывает, что можно инициализировать несколько переменных сразу (хотя такой подход не рекомендуется, так как он запутывает код):

```
max_idx, max_val = 0, 0
```

Во-вторых, из примера видно, что для обозначения границ блоков кода используются не скобки или ключевые слова, а отступы. Именно эта особенность делает программы на Python наглядными. Общепринято для отступов блоков кода использовать пробелы, а не табуляцию. Настройте свой редактор соответствующим образом (Spyder по умолчанию использует пробелы).

Операции

В Python есть все базовые операции: сложение, вычитание, умножение, деление, возведение в степень, остаток от деления, круглые скобки для управления приоритетом операций. Но есть некоторые особенности, присущие именно Python.

Операция деления (/) в Python не целочисленная:

```
>>> 5 / 2
2.5
```

Чтобы выполнить именно целочисленное деление, необходимо использовать два прямых слэша:

```
>>> 5 // 2
2
```

Возведение в степень выполняется с помощью оператора **, например

```
>>> 5**2
25
```

Операция сложения (+) для чисел даст число, для строк и массивов – выполнит их конкатенацию:

```
>>> 3 + 4
7
>>> '3' + '4'
'34'
>>> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
```

Управление выполнением программы в Python

Условия

Условия в Python записываются с помощью **if-else**. После ключевого слова **if** следует условие, затем двоеточие. Код внутри блока **if** пишется со следующей строки с отступом. Например

```
height = int(input('Введите рост: '))
if height > 180:
    print('Высокий')
```

Можно использовать **if** в сочетании **else**. **else** в данном случае пишется с тем же отступом, что и **if**, после **else** следует двоеточие. Например

```
height = int(input('Введите рост: '))
if height > 180:
    print('Высокий')
else:
    print('Низкий')
```

Можно использовать сочетание условий благодаря конструкции **if-elif-else**. Например

```
height = int(input('Введите рост: '))
size = 'S'

if height < 160:
    size = 'S'
elif height < 170:
    size = 'M'
elif height < 180:
    size = 'L'
else:
    size = 'XL'
print(size)
```

Можно записывать сложные условия, используя ключевые слова **not**, **and**, **or**, проверки на равенство **==** и неравенство **!=**.

```
if 165 < height < 175:
    print('Средний рост')

sun = True
rain = False
if sun and not rain:
    print('Можно выйти на прогулку')

if 2 * 2 == 4 and 2 * 2 != 5:
    print('Прописные истины')
```

Циклы

В Python есть 2 типа циклов: **while** и **for**. Цикл **while** выполняет выражение до тех пор, пока условие не станет истинным:

```
while input('q для выхода: ') != 'q':
    print('Выполнение программы')
```

Цикл **for** является аналогом цикла **for** в **C++**. Он позволяет выполнять заданное количество итераций, проходить по элементам коллекции. Например, вот так можно обработать все элементы массива:

```
for item in [1, 2, 3]:
    print(item)
```

Вместо массива могут быть и другие коллекции и даже строки:

```
for letter in 'String':  
    print(letter)
```

Часто при решении практических задач требуется указать начальное, конечное значение счётчика и шаг изменения. Для этого необходимо воспользоваться функцией **range**. Если в функцию передан один параметр, то это будет конечное значение счётчика, начинающегося с 0 с шагом 1. Если 2 параметра, то это будут начальное и конечное значения счётчика. Если в функцию передан третий параметр, то он будет отвечать за шаг изменения счётчика. Вот как может выглядеть код для вывода нечётных цифр от 1 до 10:

```
for i in range(1, 10, 2):  
    print(i)
```

У цикла **for** есть ещё альтернативная однострочная форма записи. Вот как можно переписать код из примера выше в одну строку:

```
[print(i) for i in range(1, 10, 2)]
```

Но такой подход чаще используют для генерации значений массивов. Пусть, например, требуется создать массив из степеней двойки. Вот как это может быть реализовано:

```
>>> powers_of_two = [2**i for i in range(9)]  
>>> powers_of_two  
[1, 2, 4, 8, 16, 32, 64, 128, 256]
```

В Python есть ключевые слова **break** и **continue**, смысл которых такой же, как в C++. **break** прерывает цикл, **continue** позволяет пропустить текущую итерацию и перейти к следующей.

```
>>> for letter in 'привет':  
...     if letter == 'p':  
...         continue  
...     print(letter, end='')  
...  
пивет
```

Но в отличие от C++, в Python есть особенность. После цикла **for** можно указать блок **else**, который выполнится в том случае, если в цикле была выполнена инструкция **continue** или **break**:

```
for letter in 'привет':  
    if letter == 'e':  
        break  
else:  
    print('Встретилась буква e')
```

Обработка исключений

При работе программы могут возникнуть исключительные ситуации, например, деление на ноль или переполнение стека. Такие ситуации необходимо отлавливать и корректно обрабатывать. Для этого создан механизм исключений. Обработка исключений в Python очень похожа на обработку исключений в C++.

В простейшем случае перехват исключений может быть реализован с использованием инструкций **try-except**:

```
try:  
    a = int(input('Первый множитель: '))  
    b = int(input('Второй множитель: '))  
    print('Результат: %d' %(a * b))  
except Exception:
```



```
print('Некорректные аргументы')
```

Перехватывать можно разные исключения. В идеале, должен быть свой обработчик под каждый возможный тип исключений. Вот некоторые возможные исключения: **ZeroDivisionError** (деление на ноль), **OverflowError** (результат арифметической операции слишком велик для представления числа), **ImportError** (не удалось импортировать модуль) и т.д.

Кроме ключевых слов **try** и **except** при обработке исключений могут использоваться инструкции **finally** и **else**. **finally** выполняет блок инструкций в любом случае, было ли исключение или нет (например, когда необходимо закрыть файл). Инструкция **else** выполняется в том случае, если исключения не было.

Создание собственных функций

Объявление собственных функций в Python начинается с ключевого слова **def**, за которым через пробел идёт имя функции, а в скобках список её параметров. После закрывающей скобки ставится двоеточие. Тело функции пишется с отступом в 2 или 4 пробела (в зависимости от принятого соглашения). Если функция должна вернуть значение, его записывают после ключевого слова **return**. Функция может возвращать несколько параметров (см. пример выше – функция **max_**). Вот пример функции, определяющей минимальное и максимальное среди переданных двух чисел:

```
def minmax(a, b):  
    return (a, b) if a < b else (b, a)
```

Функции могут параметры по умолчанию. Например, если приведённую ниже функцию вызвать без параметра, она вернёт факториал 5:

```
>>> def factorial(n=5):  
...     return n * factorial(n - 1) if n > 1 else 1  
...  
>>> factorial()  
120  
>>> factorial(3)  
6
```

Импорт функций из других файлов

Часто удобно бывает поместить часто используемые функции в отдельный файл, а затем подключать их в свой скрипт и использовать в работе. Ещё более распространённый случай – использование некоторых функций из сторонней библиотеки.

Самый простой способ подключения внешнего файла – директива **import**. Например

```
import math
```

Чтобы теперь использовать функции из модуля **math**, необходимо сначала писать имя модуля, затем точку и имя функции:

```
>>> math.sqrt(4)  
2.0
```

Чтобы не приходилось каждый раз писать имя модуля, можно воспользоваться такой записью:

```
from math import *
```

Тогда можно будет вызывать функции напрямую без указания имени модуля:

```
>>> sqrt(4)  
2.0
```

Звёздочка означает, что будут импортированы все функции из модуля. Можно указать только конкретные функции:

```
>>> from math import sqrt, sin
>>> sin(0)
0.0
>>> cos(0)
NameError: name 'cos' is not defined
>>> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

Но на практике такой подход не рекомендуется из-за того, что в разных модулях могут присутствовать функции с одинаковыми именами:

```
>>> from math import *
>>> from numpy import *
>>> sqrt(4) # какая реализация используется? math или numpy?
```

Поэтому чаще для модулей задают псевдонимы, как показано в примерах ниже:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> np.sqrt(4)
2.0
```

Комментарии

Однострочные комментарии в Python начинаются с символа **#**. Например

```
# это пример однострочного комментария
```

Многострочные комментарии заключаются в тройные кавычки (одинарные или двойные). Например

```
def factorial(n):
    """
    Рекурсивная функция вычисления факториала числа n
    """
    return 1 if n == 1 else n * factorial(n - 1)
```

Этот пример показывает ряд интересных особенностей Python. Во-первых, многострочный комментарий внутри функции. Во-вторых, пример рекурсивной функции (функция **factorial** вызывает сама себя). В-третьих, интересная однострочная запись условия **if-else**, напоминающая тернарную операцию в C++.

Многострочные комментарии обычно используются для документирования функций и классов, поэтому с помощью такого кода можно вывести справку по функции на экран:

```
>>> factorial.__doc__
'\n    Рекурсивная функция вычисления факториала числа n\n    '
```

Задание на практику

Реализовать 3 функции на Python согласно варианту (см. таблицу ниже). Описания заданий приведены после таблицы.

задание вариант	1	2	3	4	5	6	7	8	9	10
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										
11										
12										
13										
14										
15										

1. Написать функцию, которая получает на вход два целочисленных массива равной длины и возвращает массив – результат модуля разностей элементов входных массивов. Используйте однострочную запись цикла **for** для перебора значений обоих массивов одновременно. Вот небольшая подсказка в реализации:

```
[код_обработки_элементов for i in range(len(m1))]
```

2. Написать функцию, которая сортирует все буквы в строке по возрастанию и возвращает получившуюся строку. Для преобразования строки в массив букв используйте встроенную функцию **list**. Для сортировки букв используйте функцию **sorted**. Для обратного преобразования строки в массив используйте конструкцию: **''.join(a)**, где **a** – это массив.

3. Написать функцию, выполняющую циклический сдвиг элементов массива на указанное число элементов. Функция принимает два параметра: массив и количество элементов, на которое производить сдвиг. Подсказка: проще всего разбить исходный массив на два подмассива, а затем переставить их местами и объединить в один. Для выбора левой части массива используйте код **a[:k]**, для правой – **a[k:]**. Используйте оператор **+** для объединения массивов. Верните получившийся массив.

4. Написать функцию, которая генерирует несколько массивов случайных чисел и возвращает тот, в котором сумма элементов наибольшая. Для генерации случайных чисел используйте функцию **random** из модуля **random**. Модуль **random** необходимо предварительно подключить с помощью команды **import random**. Для вычисления суммы элементов массива используйте встроенную функцию **sum**. Функция принимает 2 параметра: количество генерируемых массивов и количество элементов в массиве, и возвращает массив с наибольшей суммой элементов.

5. Написать функцию, вычисляющую евклидово расстояние между двумя точками. Координаты точек задаются массивами. Евклидово расстояние вычисляется как корень из суммы квадратов разностей. Функция принимает два параметра – массивы с координатами точек. Возвращает расстояние. Точки могут располагаться в пространстве произвольной размерности (размеры массивов с координатами не ограничены). Сначала рекомендуется вычислить массив квадратов разностей по каждой координате. Это можно сделать таким кодом:

```
diff = [(a[i] - b[i])**2 for i in range(len(a))]
```

a и **b** – исходные массивы.

Затем следует вычислить сумму элементов с помощью встроенной функции **sum** и извлечь из неё квадратный корень с помощью функции **sqrt** из модуля **math**. Модуль **math** необходимо предварительно подключить с помощью кода: **import math**

6. Написать функцию, которая для заданной квадратной матрицы целых чисел находит сумму элементов на главной диагонали. Чтобы сформировать массив элементов на главной диагонали, используйте код:

```
[m[i][i] for i in range(len(m))]
```

Здесь **m** – исходная матрица. Она передаётся в функцию в качестве параметра. **len** – функция для получения длины массива. Для суммирования элементов массива используйте встроенную функцию **sum**.

7. Напишите рекурсивную функцию вычисления чисел Фибоначчи (последовательность, в которой каждое новое число получается как сумма двух предыдущих).

8. Написать функцию, которая получает 3-значное число и возвращает строку, содержащее это число, описанное словами. Например, для числа 195 функция должна выдать строку «сто девяносто пять». Функция должна корректно обрабатывать ситуацию, когда вводят 2-значные числа и цифры или когда в позиции десятков и единиц нули, например, 101, 220. Ситуацию, когда в числе количество десятков равно 1, например, 218, обрабатывать не нужно. Для решения этой задачи создайте 3 словаря для сотен, десятков и единиц, например

```
hund = {0: '', 1: 'сто', 2: 'двести', 3: 'триста', 4: 'четыреста', 5: 'пятсот', 6: 'шестьсот', 7: 'семьсот', 8: 'восемьсот', 9: 'девятьсот'}
```

Вычисляйте для числа количество сотен, десятков и единиц, используя целочисленное деление и операцию взятия остатка от деления. Для формирования итоговой строки используйте приведённый код:

```
'%s %s %s' % (hund[h], dec[d], one[o])
```

Здесь **hund**, **dec** и **one** словари, **h**, **d** и **o** – количество сотен, десятков и единиц в числе.

9. Написать функцию, проверяющую у пользователя знание таблицы умножения. Функция спрашивает у пользователя два целых числа, а затем спрашивает результат их перемножения. Если пользователь ответил верно, функция должна напечатать слово «Верно», иначе «Ошибка. Верный ответ <число>». Для ввода значений используйте встроенную функцию **input**, для преобразования строки к целому числу – функцию **int**.

10. Написать функцию, которая генерирует массив заданного размера из случайных чисел, а затем находит минимальное и максимальное значение в этом массиве и возвращает оба значения как кортеж. Размер массива передаётся в функцию как целочисленный параметр. Для генерации случайных чисел используйте функцию **random** из модуля **random**. Его необходимо будет предварительно подключить: **import random**. Для поиска минимального и максимального значения используйте встроенные функции **min** и **max**.