Программа Python.
2021-2022 год.

Стандартные функции (функции из стандартных библиотек самого Python, но для использования, которых нужно уже импортировать библиотеки в код программы).

Эти библиотеки автоматически устанавливаются при установке языка Python на компьютер.

Стандартные библиотеки можно быстро найти на компьютере следующим образом: Найти эмблему программы самого Python - навести мышку на ярлык программы - правой клавишей мышки - расположение файла.

Мы попадаем в папку (в моем случае) Python38. Заходим в **папку Lib** и внутри нее мы видим огромное количество встроенных в язык библиотек и даже папок, в которые собраны еще библиотеки.

Можно открыть эти файлы библиотек и посмотреть коды функций. Но некоторые библиотеки написаны на языке С и поэтому там внутри привычного питоновского кода мы не увидим.

И здесь в папке Lib мы видим не только файлы библиотек, но и папки в которых есть тоже файлы библиотек и могут быть также еще папки и так далее.

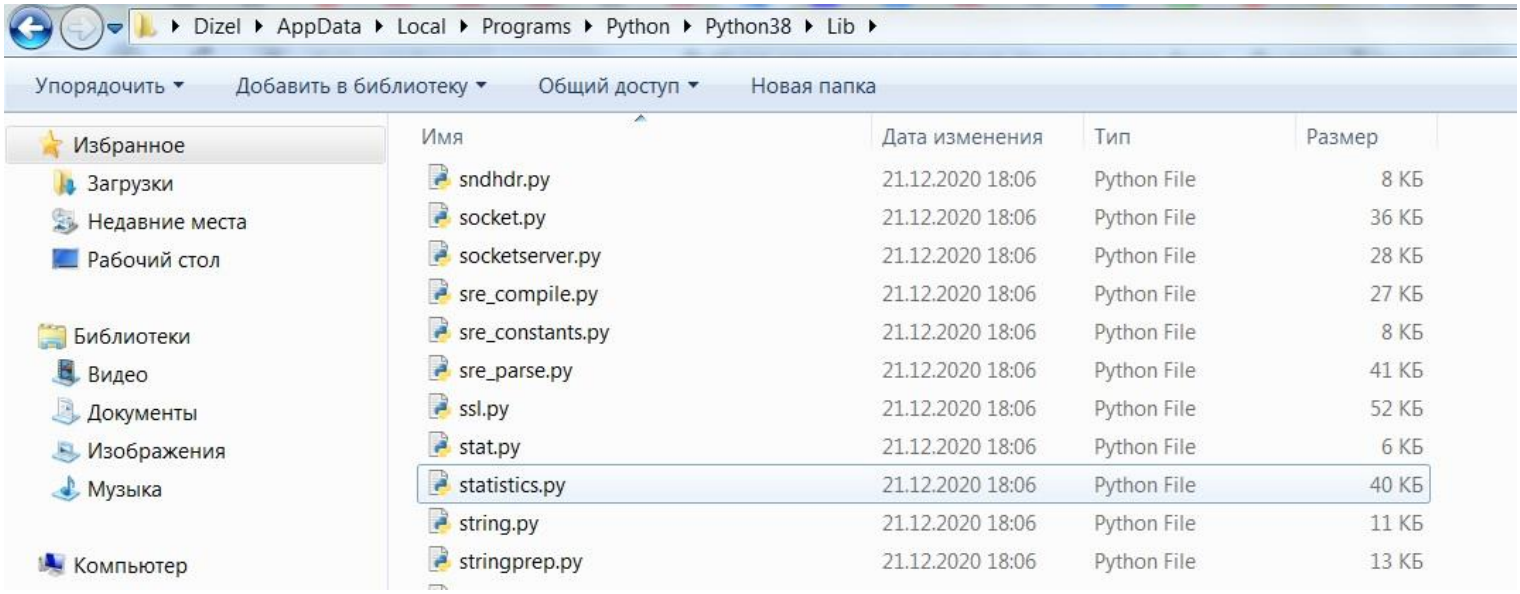
Поэтому нужно уметь правильно (согласно пути расположения файла) подключать библиотеки и импортировать функции.

Например:

Если файл библиотеки лежит сразу в папке Lib

К примеру, возьмем библиотеку (модуль) - **statistics.py**

Модуль, который содержит различные математические функции.



Этот модуль (так короче писать, чем библиотека) лежит напрямую в папке Lib.

Используем из этой библиотеки в нашей программе функцию - **mean()**

Функция mean() - выводит среднее арифметическое введенных чисел.

Можно даже открыть файл **statistics.py**, найти функцию mean() и посмотреть код этой функции.

```
290         yield x
291
292
293 # === Measures of central tendency (averages) ===
294
295 def mean(data):
296     """Return the sample arithmetic mean of data.
297
298     >>> mean([1, 2, 3, 4, 4])
299     2.8
300
301     >>> from fractions import Fraction as F
302     >>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
303     Fraction(13, 21)
304
305     >>> from decimal import Decimal as D
306     >>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
307     Decimal('0.5625')
308
309     If ``data`` is empty, StatisticsError will be raised.
310     """
311     if iter(data) is data:
312         data = list(data)
313     n = len(data)
314     if n < 1:
315         raise StatisticsError('mean requires at least one data point')
316     T, total, count = _sum(data)
317     assert count == n
318     return _convert(total/n, T)
319
320
321 def fmean(data):
322     """Convert data to floats and compute the arithmetic mean.
323
324     This runs faster than the mean() function and it always returns a float.
```

Над кодом функции не стоит долго размышлять, ведь нас интересует именно как научиться использовать готовую функцию из стандартной библиотеки Python.

Поэтому подключаем библиотеку **statistics.py**, импортируем функцию **mean()** и затем используем ее в нашей программе:

```
from statistics import mean

print(mean([1, 2, 3, 4, 4]))
```

Результат работы нашей программы: **2.8**

Как можно заметить, функция mean() в качестве аргумента получает список из чисел и на выходе выдает среднее арифметическое этих чисел.

Попробуем ввести в функцию несколько значений через запятую:

```
from statistics import mean
print(mean(1,2,3))
```

```
builtins.TypeError: mean() takes 1 positional argument but 3 were given
```

Получаем ошибку (перевод яндекса): встроенные.Ошибка типа: mean() принимает 1 позиционный аргумент, но было дано 3.

Таким образом, функция mean() принимает на вход только один аргумент и этот аргумент должен быть списком.

Очень удобно использовать готовые функции в своих программах. Мы быстро получаем ответ.

Но может возникнуть очень много вопросов по использованию функции, которую не мы создавали:

- Если не мы написали (создали) функцию, то каким образом мы можем понять какие данные в качестве аргумента, или даже, может быть, сколько аргументов ждет на входе данная функция?

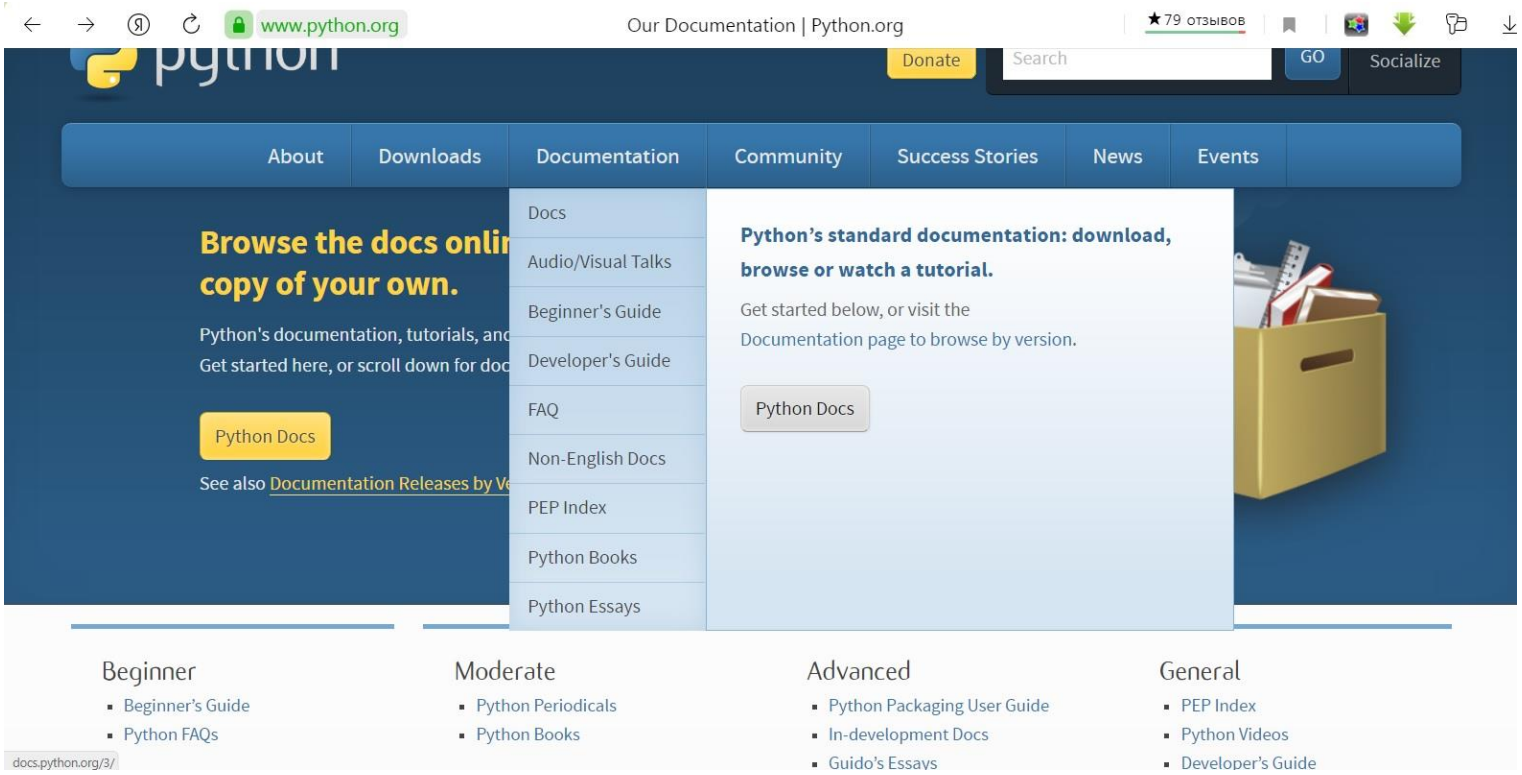
- Сколько и какой тип параметров в данной функции?

- Что мы получаем на выходе?

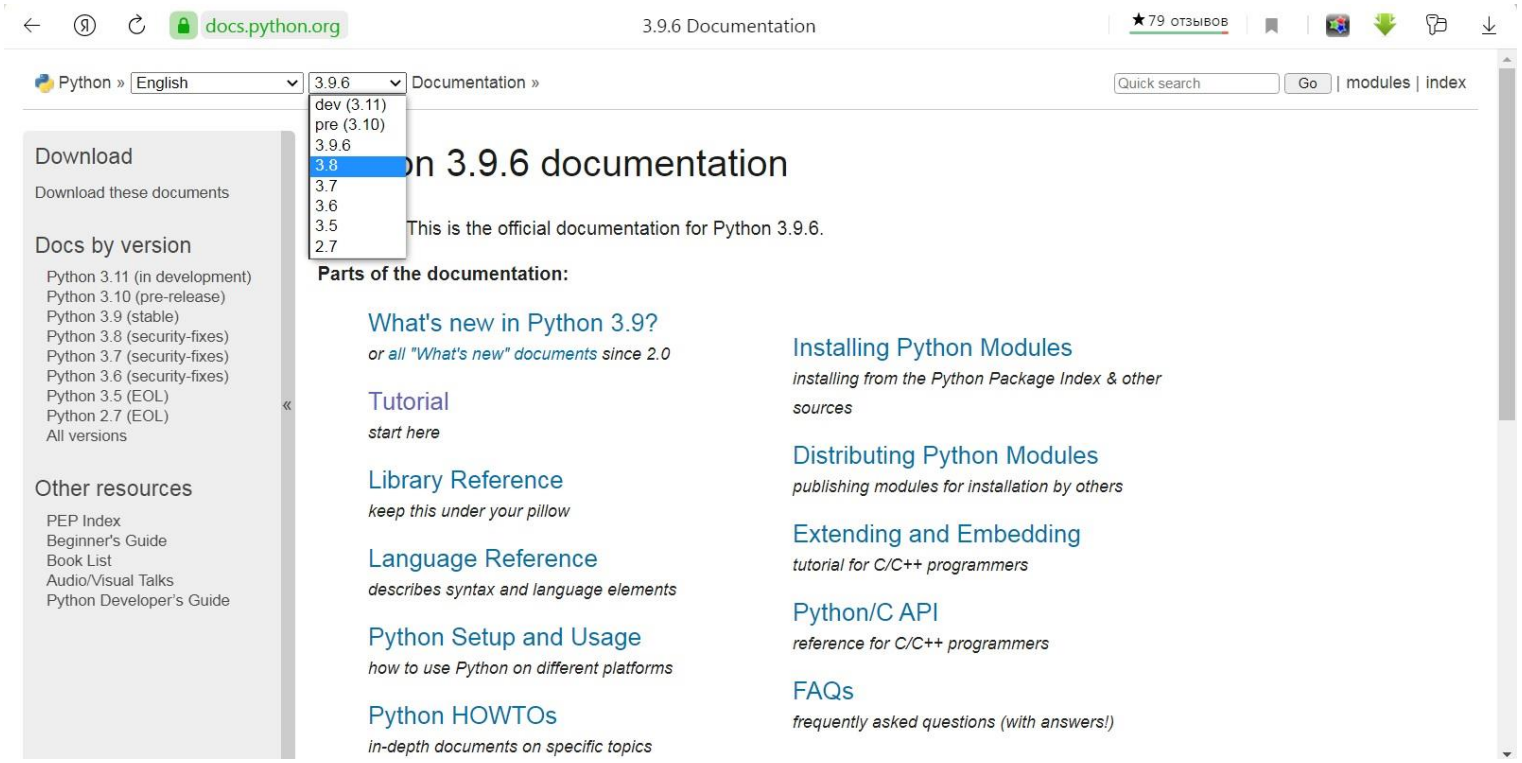
Что делает в этом случае программист?

Первое, что делает обычно программист - это посмотреть документацию по данной функции на официальном сайте https://www.python.org/

На сайте переходим в раздел документации: Меню Documentation -> Docs -> Python Docs.



Далее мы можем выбрать версию языка Python для просмотра документации именно по той версии, которая нас интересует:



Далее переходим в меню Library Reference, так как нас интересует описание функций, а функции в свою очередь лежат в библиотеках (мы уже это с вами выяснили).

docs.python.org 3.8.11 Documentation ★79 отзывов

Python » English 3.8.11 Documentation » Quick search Go | modules | index

Download
Download these documents

Docs by version
Python 3.11 (in development)
Python 3.10 (pre-release)
Python 3.9 (stable)
Python 3.8 (security-fixes)
Python 3.7 (security-fixes)
Python 3.6 (security-fixes)
Python 3.5 (EOL)
Python 2.7 (EOL)
All versions

Other resources
PEP Index
Beginner's Guide
Book List
Audio/Visual Talks
Python Developer's Guide

https://docs.python.org/3.8/library/index.html

Python 3.8.11 documentation

Welcome! This is the documentation for Python 3.8.11.

Parts of the documentation:

What's new in Python 3.8?
or all "What's new" documents since 2.0

Tutorial
start here

Library Reference
keep this under your pillow

Language Reference
describes syntax and language elements

Python Setup and Usage
how to use Python on different platforms

Python HOWTOs
in-depth documents on specific topics

Installing Python Modules
installing from the Python Package Index & other sources

Distributing Python Modules
publishing modules for installation by others

Extending and Embedding
tutorial for C/C++ programmers

Python/C API
reference for C/C++ programmers

FAQs
frequently asked questions (with answers!)

И попадаем на страницу, где перечислены все стандартные библиотеки Python по определенным категориям.

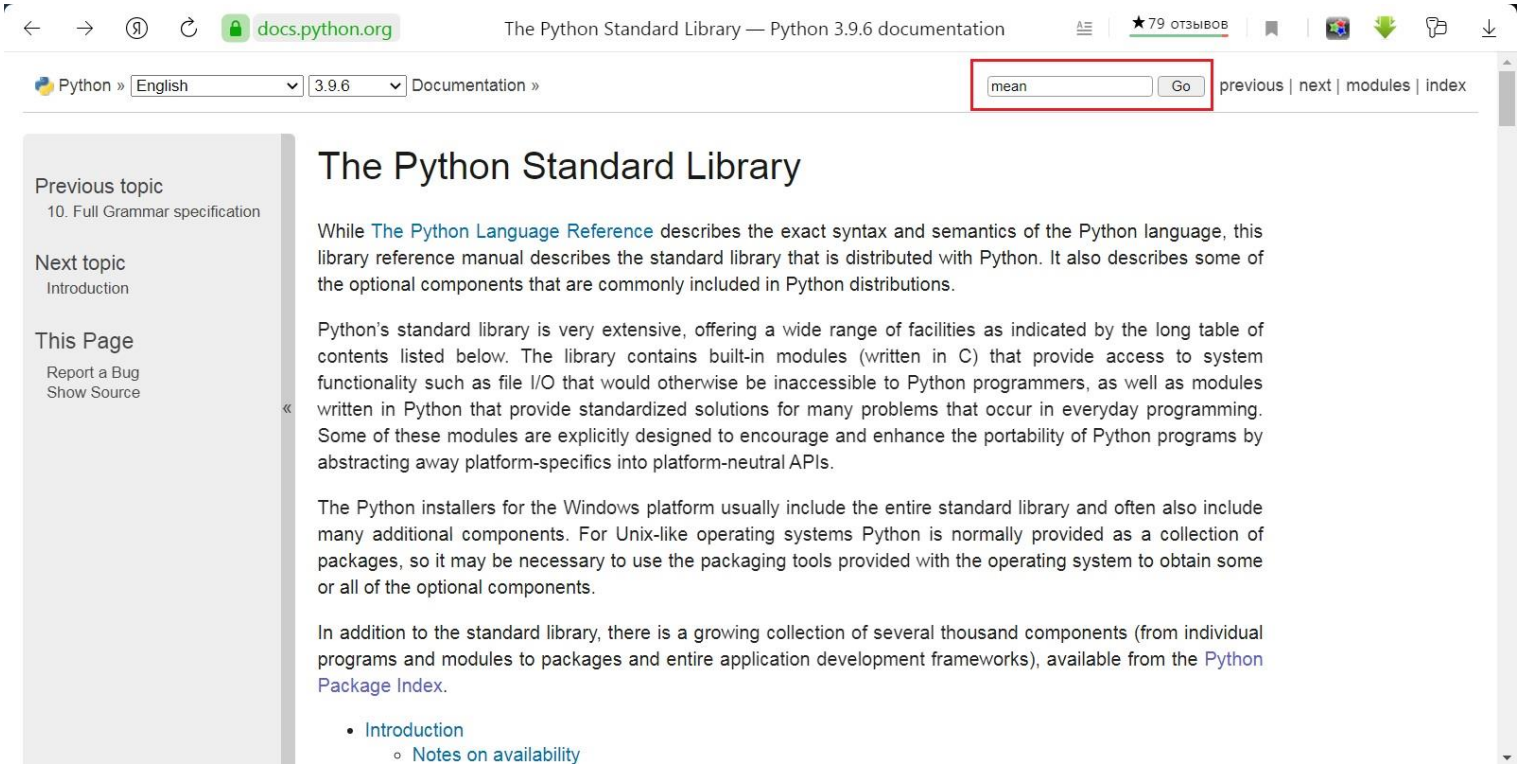
docs.python.org The Python Standard Library — Python 3.8.11 documentation ★79 отзывов

- Concrete exceptions
- Warnings
- Exception hierarchy

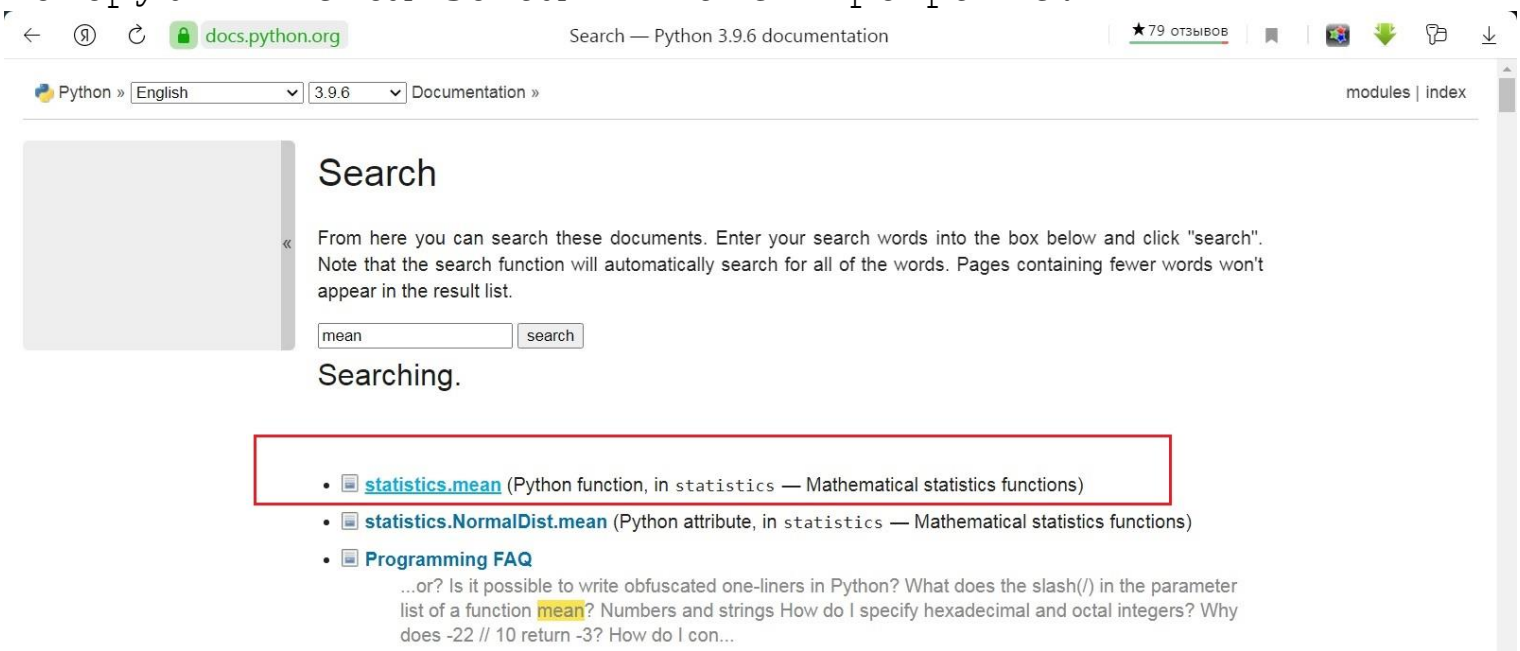
- Text Processing Services
  - `string` — Common string operations
  - `re` — Regular expression operations
  - `difflib` — Helpers for computing deltas
  - `textwrap` — Text wrapping and filling
  - `unicodedata` — Unicode Database
  - `stringprep` — Internet String Preparation
  - `readline` — GNU readline interface
  - `rlcompleter` — Completion function for GNU readline
- Binary Data Services
  - `struct` — Interpret bytes as packed binary data
  - `codecs` — Codec registry and base classes
- Data Types
  - `datetime` — Basic date and time types
  - `calendar` — General calendar-related functions
  - `collections` — Container datatypes
  - `collections.abc` — Abstract Base Classes for Containers
  - `heapq` — Heap queue algorithm
  - `bisect` — Array bisection algorithm
  - `array` — Efficient arrays of numeric values
  - `weakref` — Weak references
  - `types` — Dynamic type creation and names for built-in types
  - `copy` — Shallow and deep copy operations
  - `pprint` — Data pretty printer
  - `reprlib` — Alternate `repr()` implementation

Мы видим, что перед нами именно имена библиотек. Функции, как мы выяснили, лежат в самой библиотеке. Поэтому, чтобы найти функцию по прямой ссылке на библиотеку, нам необходимо наизусть знать в какой библиотеке находится интересующая нас функция. Но так как этих функций создали

не мало, то мы для удобства воспользуемся формой поиска на этой странице и введем название нашей функции mean():



В качестве примера, находим ссылку на нашу функцию mean(), которую мы использовали в нашей программе:



По этой ссылке на функцию mean() даже видно к какой библиотеке она относится, и что это функция математической статистики - Mathematical statistics functions.

Проходим дальше по этой ссылке:

Теперь мы попадаем на страницу библиотеки statistics. Где одной из первых в списке многочисленных функций библиотеки значится наша функция mean().

docs.python.org statistics — Mathematical statistics functions — Python 3.9.6 documentati...

Python » English 3.9.6 Documentation » The Python Standard Library » Numeric and Mathematical Modules » Quick search Go | previous | next | modules | index

Table of Contents

statistics — Mathematical statistics functions
- Averages and measures of central location
- Measures of spread
- Function details
- Exceptions
- NormalDist objects
  - NormalDist Examples and Recipes

Previous topic
random — Generate pseudo-random numbers

Next topic
Functional Programming Modules

This Page
Report a Bug
Show Source

statistics — Mathematical statistics functions

New in version 3.4.

Source code: Lib/statistics.py

This module provides functions for calculating mathematical statistics of numeric (Real-valued) data.

The module is not intended to be a competitor to third-party libraries such as NumPy, SciPy, or proprietary full-featured statistics packages aimed at professional statisticians such as Minitab, SAS and Matlab. It is aimed at the level of graphing and scientific calculators.

Unless explicitly noted, these functions support int, float, Decimal and Fraction. Behaviour with other types (whether in the numeric tower or not) is currently unsupported. Collections with a mix of types are also undefined and implementation-dependent. If your input data consists of mixed types, you may be able to use map() to ensure a consistent result, for example: map(float, input_data).

Averages and measures of central location

These functions calculate an average or typical value from a population or sample.

mean()	Arithmetic mean ("average") of data.
fmean()	Fast, floating point arithmetic mean.
geometric_mean()	Geometric mean of data.

Проходим по ссылке в описание функции mean() Опускаемся немного ниже по странице и видим раздел - Function details - сведения о функциях. В этом разделе можно найти пример использования нашей функции mean().

docs.python.org statistics — Mathematical statistics functions — Python 3.9.6 documentati...

Function details

Note: The functions do not require the data given to them to be sorted. However, for reading convenience, most of the examples show sorted sequences.

statistics.mean(*data*)

Return the sample arithmetic mean of *data* which can be a sequence or iterable.

The arithmetic mean is the sum of the data divided by the number of data points. It is commonly called "the average", although it is only one of many different mathematical averages. It is a measure of the central location of the data.

If *data* is empty, StatisticsError will be raised.

Some examples of use:

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

Note: The mean is strongly affected by outliers and is not a robust estimator for central location: the mean is not necessarily a typical example of the data points. For more robust measures of central location, see median() and mode().

В примерах использования функции mean() мы отчетливо видим, что в аргументе функции mean() передается именно тип данных - список: **mean([1, 2, 3, 4, 4]).**

А результат функции это число - среднее арифметическое чисел из передаваемого в функцию списка: **2.8**

Мы видим, что вся документация на официальном сайте Python на английском языке - это принятый международный язык для общения программистов из разных стран.

Всегда эту станицу или часть текста можно перевести, воспользовавшись сервисом перевода самого браузера или сторонних переводчиков.

При переводе на русский язык может измениться значение некоторых слов, но общий смысл будет понятен.

Также, если информации на официальном сайте в документации по Python будет не достаточно, идем искать на просторы интернета.

Вас абсолютно не должен смущать такой способ получения информации. Потому, что программист, как и инженер, не обязан знать наизусть и держать в голове всю информацию, он должен уметь найти эту информацию в технической документации и использовать ее для решения поставленных перед ним задач.

При этом, конечно же, нужно понимать, чтобы воспользоваться технической документацией правильно специалист должен быть знаком с правилами написания этой самой документации. Должен знать все обозначения и термины в документации.

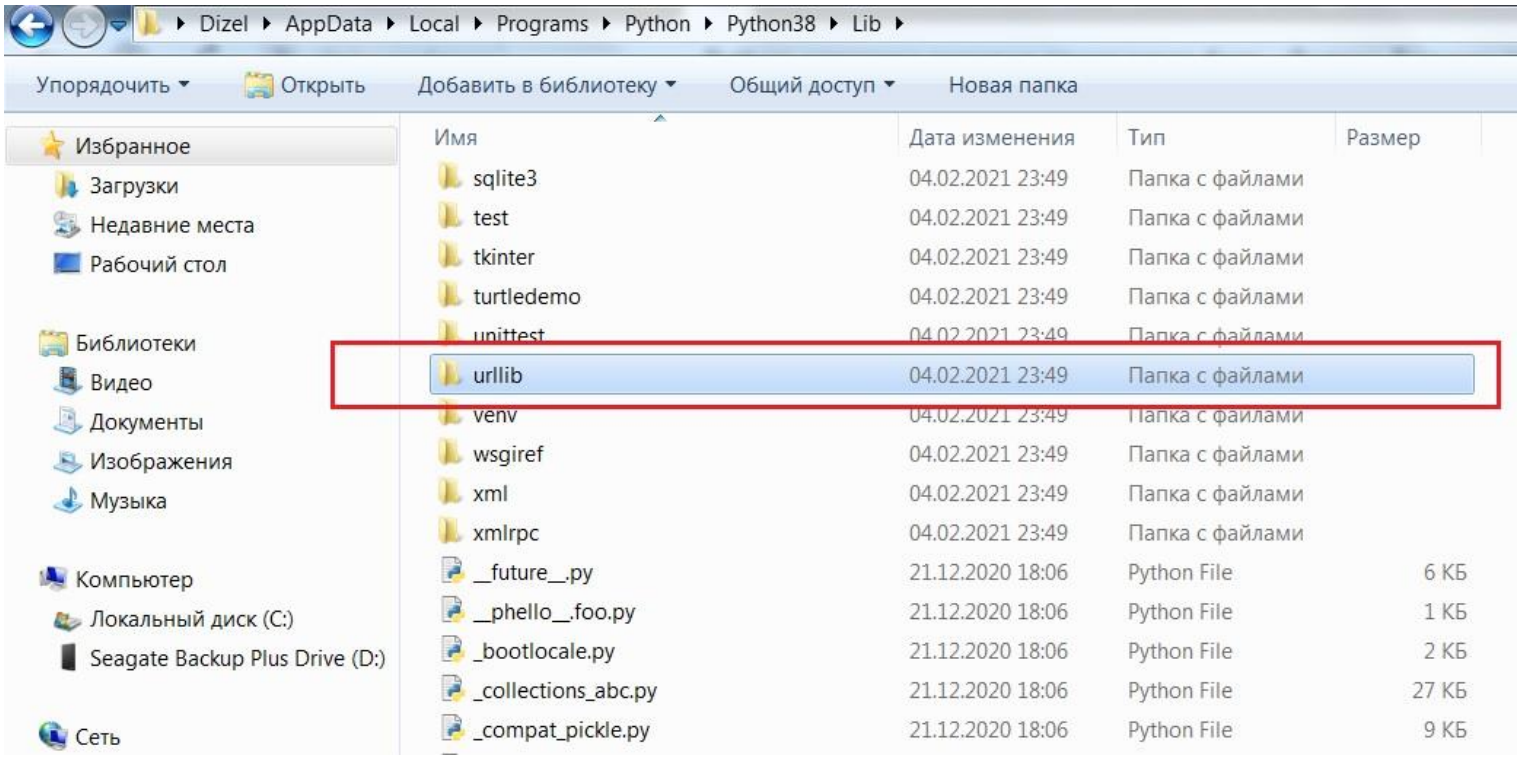
Другими словами: программист не обязан помнить наизусть имя и параметры функции mean(), но он обязательно должен знать: что такое функции в программировании, что такое аргументы в функциях и параметры. Должен знать и понимать сам принцип создания функций в языке программирования.

Теперь давайте рассмотрим другой пример использования стандартных библиотек и функций Python.

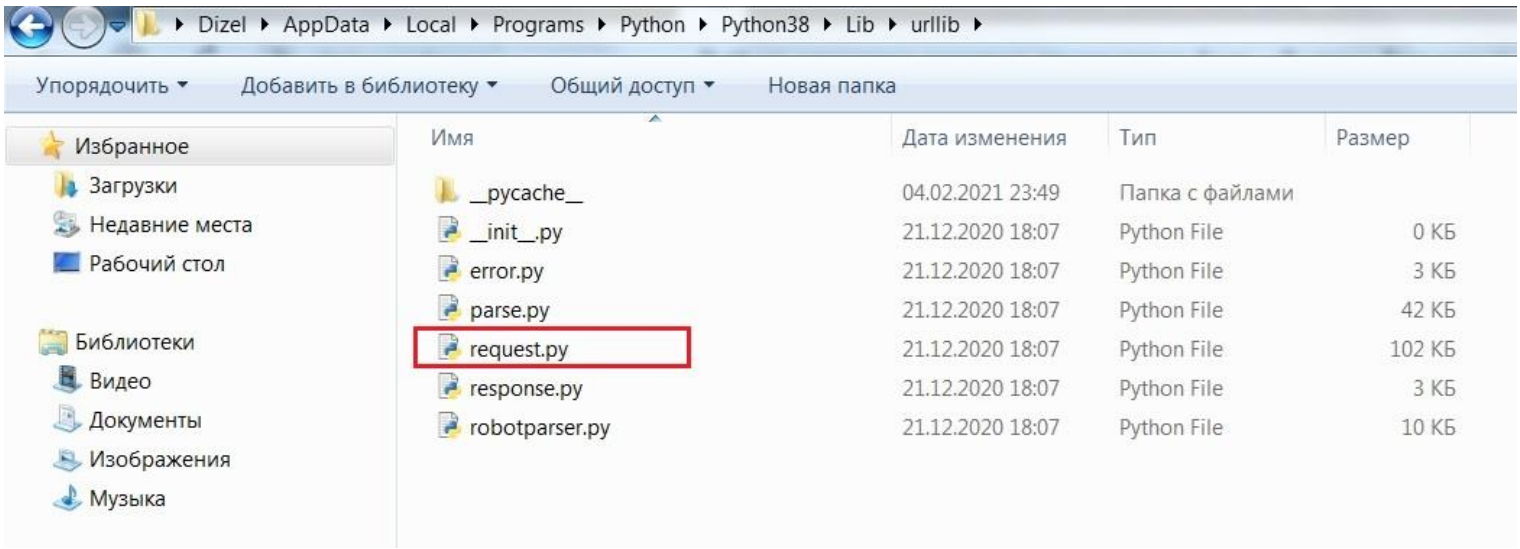
Например:

Если файл библиотеки лежит не сразу в папке Lib, а находится глубже, в другой папке.

Итак, в папке Lib помимо открытых библиотек мы, к примеру, увидели папку **urllib.**



Заходим в эту папку.



Нас будет интересовать библиотека - **request.py** и функция из этой библиотеки - **urlopen()**.

Можно открыть библиотеку, увидеть функцию urlopen(), которая принимает адрес сайта (url) на вход, и еще дополнительные не обязательные параметры.
Нас будет интересовать именно параметр - url. Он самый первый параметр, который ждет на входе функция.

```
127         'UnknownHandler', 'HTTPErrorProcessor',
128         # Functions
129         'urlopen', 'install_opener', 'build_opener',
130         'pathname2url', 'url2pathname', 'getproxies',
131         # Legacy interface
132         'urlretrieve', 'urlcleanup', 'URLopener', 'FancyURLopener',
133     ]
134
135     # used in User-Agent header sent
136     __version__ = '%d.%d' % sys.version_info[:2]
137
138     _opener = None
139     def urlopen(url, data=None, timeout=socket._GLOBAL_DEFAULT_TIMEOUT,
140                 *, cafile=None, capath=None, cadefault=False, context=None):
141         '''Open the URL url, which can be either a string or a Request object.
142
143         *data* must be an object specifying additional data to be sent to
```

Пишем небольшую программу:

from urllib.request import urlopen # подключение библиотеки и импорт функции: из папки urllib подключаем библиотеку request, импортируем функцию urlopen().

Здесь мы должны понять, что когда мы в программе мы подключаем стандартную библиотеку (не встроенную), Python по умолчанию сразу идет в папку **Lib** (об этом ему не надо напоминать). Но если библиотека лежит глубже в другой папке, то нужно указать в какой папке лежит библиотека (указать путь до библиотеки).

Важно. В указании пути, в отличие от принятого знака **'/'** - слэш (slash - это косая черта), используется знак - точка!

myConnect = urlopen('https://python.org')

Объявляем переменную myConnect. Присваиваем переменной значение, которое вернет функция urlopen(), в которую мы поместили url интересующего сайта (url сайта с которым мы хотим соединиться - другими словами - открыть сайт).

if myConnect: # И создаем простое условие: Если мы соединимся с сайтом с помощью функции urlopen(), то в переменную myConnect обязательно попадут определенные данные (какие данные мы с вами разберем немного позднее - сейчас это не важно). Если в переменной myConnect появятся данные, то значит в конструкции if эта переменная

вычислится как логическое значение True, и программа напечатает сообщение об успешном соединении с сайтом.

```
    print('Соединение с сайтом установлено!')
else: # На случай ошибки соединения, программа напечатает сообщение о неудачном соединении с сайтом.
    print('Нет соединения с сайтом')
```

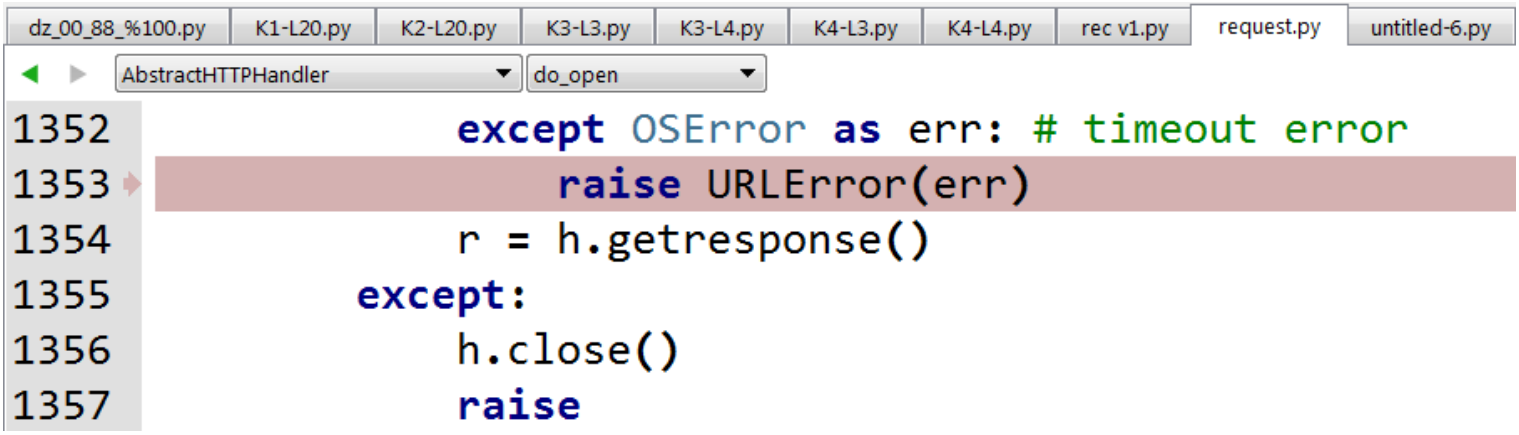
Внимание!

Если сейчас в программе указан реальный существующий адрес (url) сайта, то мы всегда получаем сообщение - **'Соединение с сайтом установлено!'**

Но, если в адресе указать не существующий сайт.

К примеру: **'https://www.pythonss.org'**

Получаем ошибку в программе, и автоматически открывается файл request.py. В этом файле нам Python указывает на ошибку - URLError.



Чтобы наша программа не уходила в ошибку, а продолжила свою работу до конца, если указан неправильный адрес (url) сайта, требуется доработка!

```
from urllib.request import Request, urlopen # дополнительно импортируем метод Request
from urllib.error import URLError, HTTPError # дополнительно подключаем проверки ошибок (методы URLError, HTTPError из отдельной библиотеки error.py)
my_url = Request('https://www.pythonwww.org') # получаем объект подключения в переменной my_url (в отличие от объекта, который мы получали в программе выше с помощью urlopen, мы получили расширенный объект)
try: # обработка исключений с помощью конструкции - try - except
    myConnect = urlopen(my_url)
except HTTPError as e:
    print('Код ошибки соединения: ', e.code)
except URLError as e:
```

```
    print('Причина ошибки соединения: ', e.reason)
else:
    print('Соединение установлено!')
```

Результат:

Причина ошибки соединения: [Errno 11004] getaddrinfo failed

Видим, что сработало исключение - URLError

Если установить действующий адрес сайта: 'https://www.python.org'

Результат:

Соединение установлено!

Мы с вами сейчас рассмотрели пример, как подключить библиотеку Python, если файл библиотеки лежит не сразу в папке Lib, а находится глубже, в другой папке.

Идея заключается в том, что все библиотеки Python находятся в главной папке Lib.

Для подключения нужной библиотеки необходимо указать правильный путь к библиотеке. Вместо косой черты (/), как принято при указании полного пути файла на компьютере, необходимо использовать точку (.).

А дальше можно написать **import *** – тогда мы можем использовать все функции из библиотеки, или написать **import urlopen -** для использования только, к примеру, функции urlopen() из большого списка функций, которые могут лежать в библиотеке.

Немного практики:

Есть интересная библиотека turtle (черепашка).

Знакомство с библиотекой Черепашка (Turtle)

Это библиотека для работы с графикой в Python.

Представьте себе объект (Turtle), который появляется в центре координатной плоскости XY x = 0, y = 0.

Если этот объект двигается по экрану, то оставляет след за собой от траектории своего передвижения.

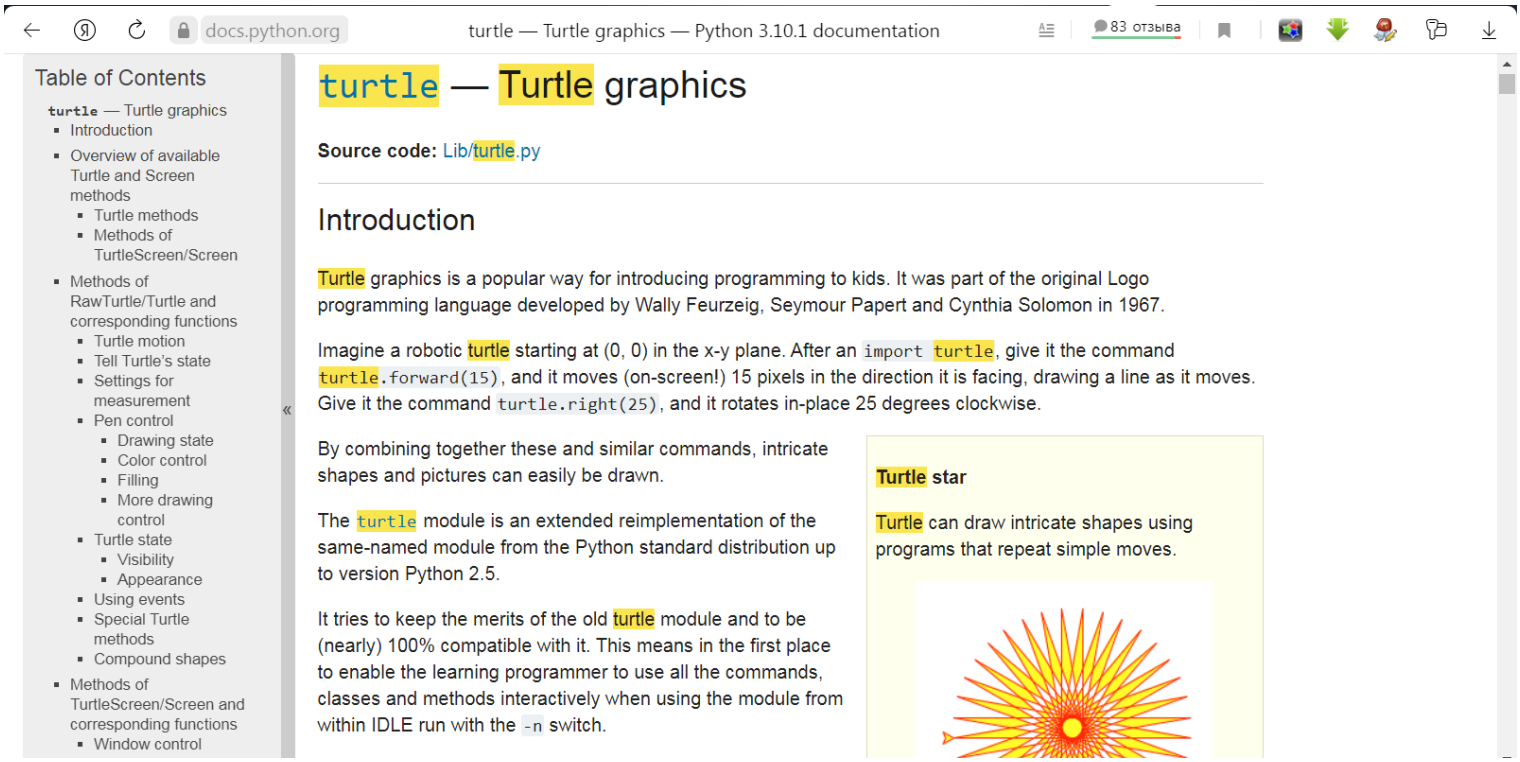
Программным способом на языке Python мы можем управлять передвижением черепашки.

На официальном сайте Python можно изучить подробно инструкцию по работе с этой библиотекой в разделе library.

https://docs.python.org/3/library

Ищем turtle в библиотеках по поиску.

https://docs.python.org/3/library/turtle.html?highlight=turtle



Эту библиотеку можно сравнить с визуальным языком программирования scratch. Потому, что, как и в scratch, так и здесь можно управлять спрайтом (черепашкой).

Если на компьютере установлен язык Python, то библиотека turtle должна уже быть установлена с языком Python.

Чтобы проверить, установлена или нет библиотека turtle, мы можем написать первую строку программы.

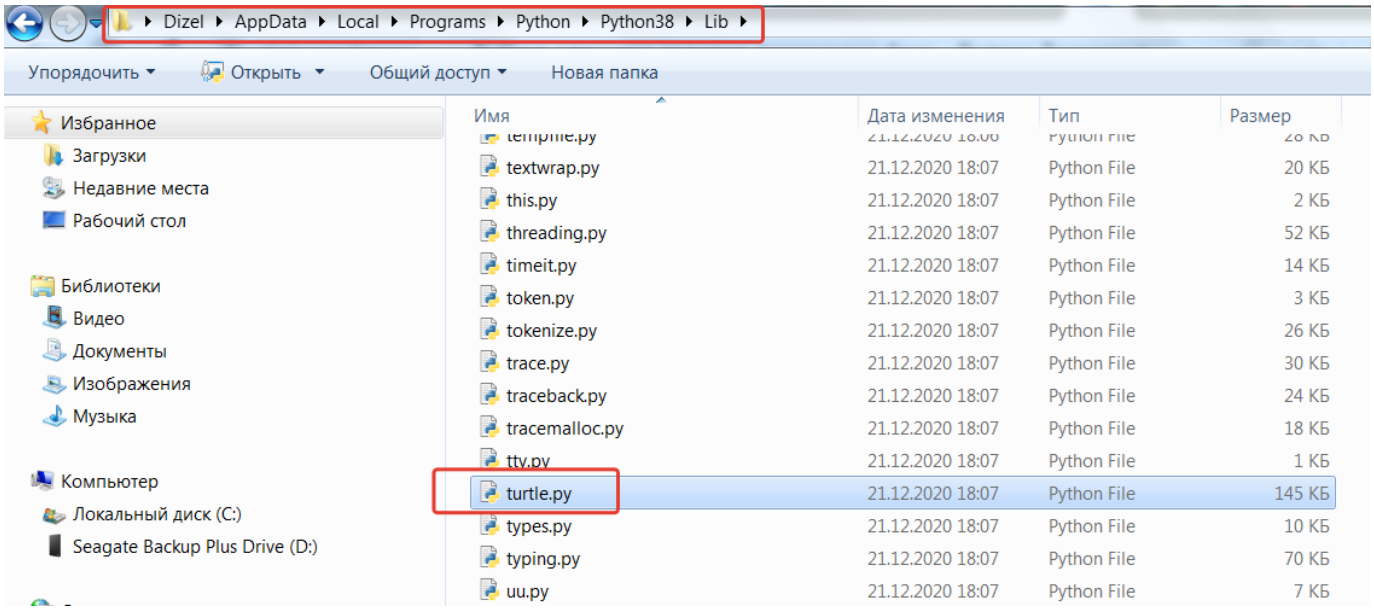
import turtle # импортируем библиотеку turtle в программу.

Запускаем программу.

Если ошибок нет, это означает, что библиотека turtle установлена на компьютере.

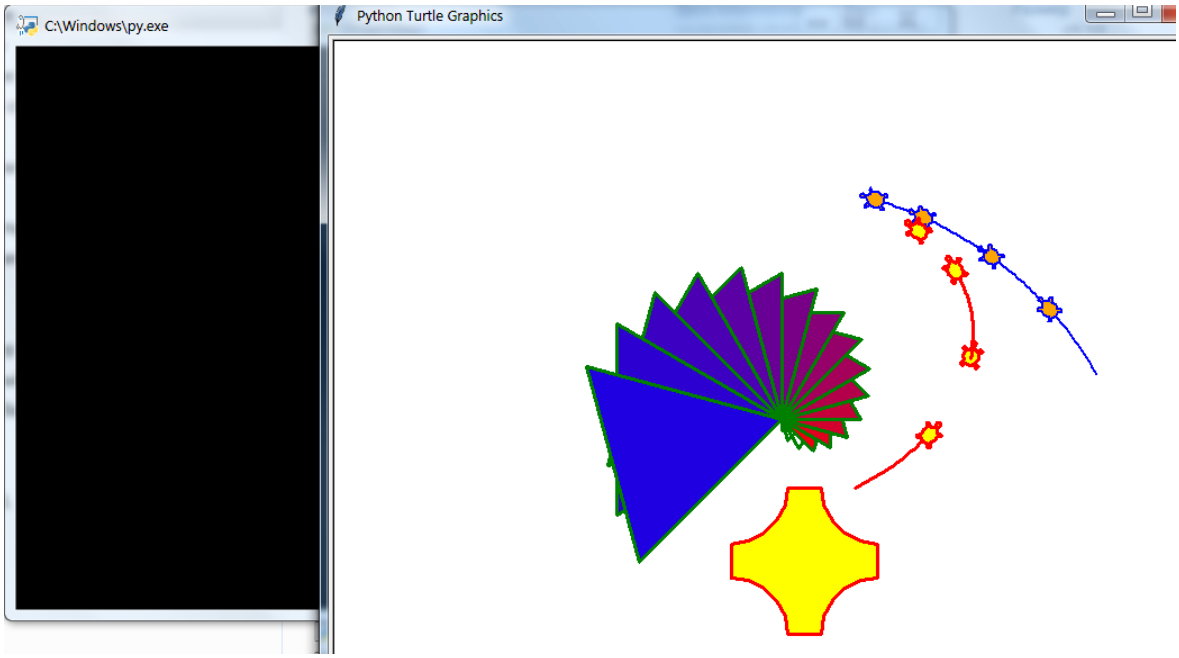
Можно продолжать писать программу и работать с данной библиотекой.

Также можно посмотреть по адресу и найти библиотеку среди других, установленных на компьютер.

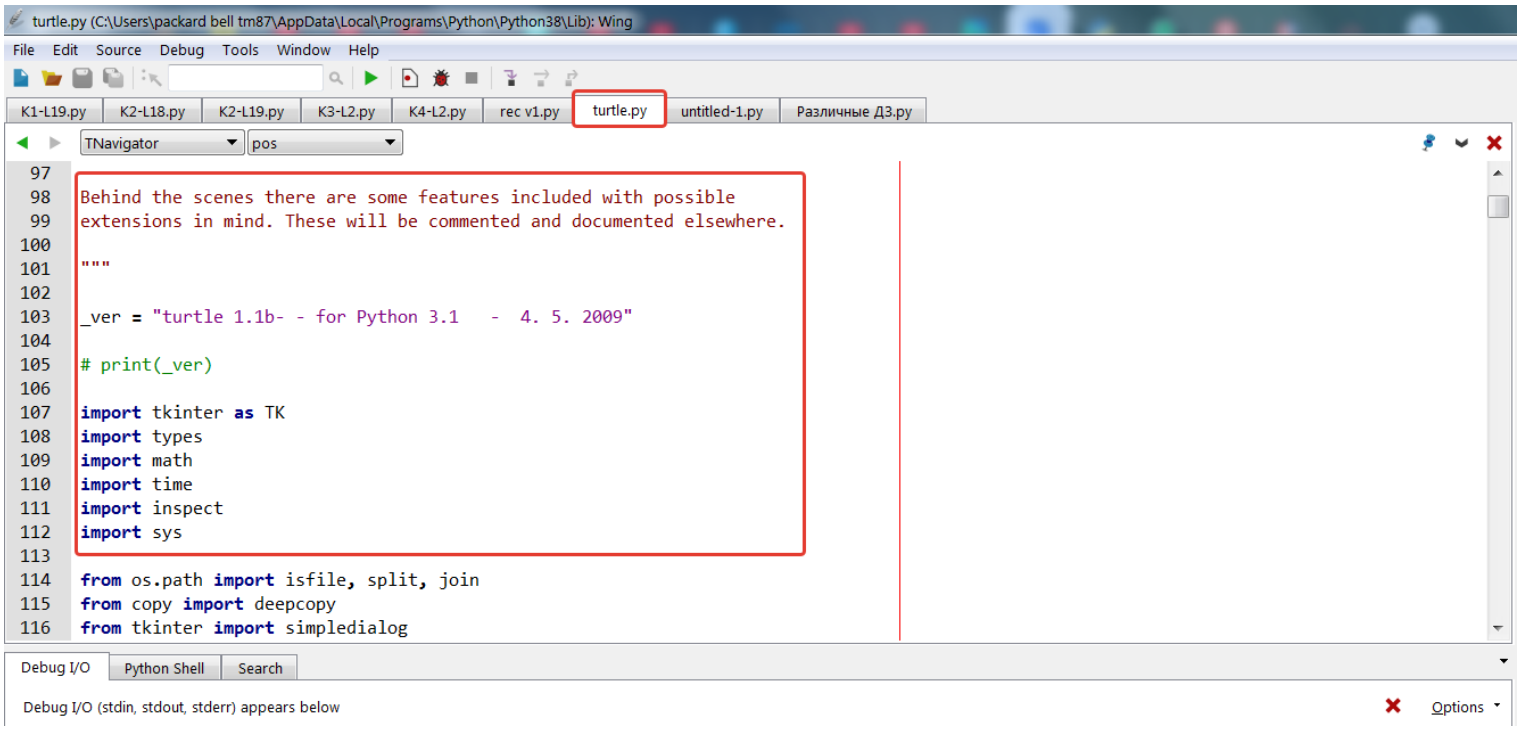


Можно запустить файл turtle.py на компьютере (два раза кликнуть на нем мышкой).

Увидим демонстрацию библиотеки:



Также можно правой клавишей мышки открыть файл библиотеки в среде разработки WING 101.



Откроется файл библиотеки. И мы увидим, как устроена библиотека на программном уровне (увидим код библиотеки).

Можно заметить, что библиотека turtle тоже использует импорт некоторых библиотек для работы.

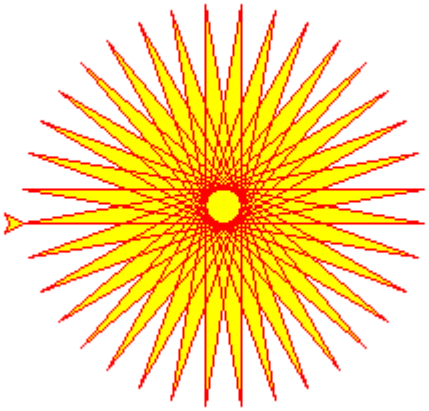
Внимание! Код не изменяем! Иначе, можно сломать библиотеку. Придется заново устанавливать исходный файл библиотеки.

Удаляем код: **import turtle**

Копируем и вставляем в нашу среду разработки WING код из первого примера документации:

```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

Видим, что наша черепашка (цветная стрелка) двигается по экрану, поворачивается и в итоге оставляет след на белом экране; в результате на экране рисуется многоконечная звезда.



Давайте добавим дополнительно метод pos() для отображения информации о стрелке в момент каждой итерации, когда стрелка уже нарисовала одну линию и меняет свое направление для продолжения цикла while.

```
from turtle import *
color('red', 'yellow')
print(pos())
begin_fill()
while True:
    forward(200)
    left(170)
    print(pos())
    if abs(pos()) < 1:
        break
end_fill()
print(abs(pos()))
done()
```

Запустим программу.

Рисуется звезда в отдельном окне.

И теперь дополнительно во вкладке Debug отображается информация о положении стрелки на концах звезды каждую итерацию.

```
Debug I/O    Python Shell

Debug I/O (stdin, stdout, stderr) appears below

(0.00,0.00)
(200.00,0.00)
(3.04,34.73)
(190.98,-33.67)
(17.77,66.33)
(170.98,-62.23)
(42.42,90.98)
(142.42,-82.23)
```

Разберем код программы.

from turtle import * # импорт всех функций и методов из библиотеки turtle (звездочка означает импорт всех функций и методов сразу)
color('red', 'yellow') # Настройка цвета и контура

print(pos()) # Метод pos() по определению это не координаты - это вектор. Этот метод используется для определения текущего местоположения черепахи (x, y) в виде вектора Vec2D.

Но, если посмотреть на данные, которые отображаются на экране во вкладке Debug, то это как раз координаты стрелки на концах звезды.

Я сказал метод (не функция) потому, что все функции, которые вызываются в этой программе, применяются к объекту черепашка.

begin_fill() # Вызываем метод перед рисованием фигуры, для заливки цветом. Метод не принимает аргументы. От английского слова fill (заполнять)
while True:
    forward(200) # Черепашка двигается вперед
    left(170) # Черепашка поворачивается влево на 170 градусов.
    print(pos()) # Этот метод используется для определения текущего местоположения черепахи (x, y)

    if abs(pos()) < 1: # Если абсолютное значение метода pos() станет меньше единицы (< 1), то цикл while останавливается. Другими словами, если черепашка вернулась

в точку 0, 0, то вектор меньше 1 и выполняется команда break для остановки цикла рисования while.

```
        break # остановка цикла while.
end_fill() # Вызываем метод после рисования фигуры, для заливки цветом. Метод не принимает аргументы.
done() # Метод, который запускает зацикливание (loop) программы, чтобы мы могли посмотреть на результат работы программы (на саму звезду). Если метод не запустить, то после выполнения (как только нарисуется звезда) сразу закроется окно рисования программы.
```

- Примечание: Чтобы понять, почему условие

if abs(pos()) < 1: срабатывает, когда стрелка возвращается в положение откуда начала свой путь, проведем эксперимент.

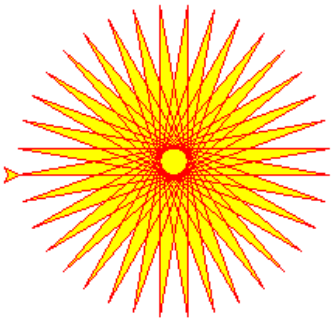
Установим в конце программы строку: **print(abs(pos()))** , чтобы вывести значение abs(pos()), при котором останавливается цикл while.

```
11  end_fill()
12  print(abs(pos()))
13  done()
```

Debug I/O | Python Shell | Search

Debug I/O (stdin, stdout, stderr) appears below

```
(91.25,-91.25)
(125.98,105.71)
(57.58,-82.23)
(157.58,90.98)
(29.02,-62.23)
(182.23,66.33)
(9.02,-33.67)
(196.96,34.73)
(0.00,-0.00)
3.1166837572216805e-13
```



Переведем в интернет-калькуляторе экспоненциальный вид числа в обычный десятичный вид.

Данный онлайн калькулятор позволяет перевести число из экспоненциальной записи в обычный вид.

Экспоненциальное число

3.116683757221680

Преобразовать

3.1166837572216805e-13 = 3.1166837572216805 · 10^{-13} = 3.1166837572216805 · 0.0000000000001 = 0.000000000000312

Видим, что это число очень маленькое, но все равно оно не равно = 0!

Вот поэтому мы используем условие: if abs(pos()) < 1:

Можно добавить дополнительно в код вывод угла черепашки в конкретный момент.

```
from turtle import *
color('red', 'yellow')
print(pos(), heading())
begin_fill()
while True:
    forward(200)
    left(170)
    print(pos(), heading())
    if abs(pos()) < 1:
        break
end_fill()
done()
```

Debug I/O | Python Shell

Debug I/O (stdin, stdout, stderr) appears below

```
(0.00,0.00) 0.0
(200.00,0.00) 170.0
(3.04,34.73) 340.0
(190.98,-33.67) 150.0
(17.77,66.33) 320.0
(170.98,-62.23) 130.0
(42.42,90.98) 300.0
```

Оба метода pos() и heading() - это методы самой библиотеки turtle.

Но как видно из возможностей библиотеки turtle, а именно, из списка функций библиотеки, мы теперь в Python можем рисовать так же, как мы рисовали любыми спрайтами на уроках по Scratch. Вспоминаем понятие из Scratch - Перо.

Сократим немного наш код программы черепашка.

```
from turtle import *
forward(200)

done() # функция, которая не позволяет закрыться окну.
```

Поговорим о том, как в Python можно несколькими способами подключать библиотеки и функции.

Первый способ - хороший способ:

```
from turtle import forward, done

forward(200)
done()
```

В этом первом способе мы конкретно импортируем из библиотеки turtle функции (или методы) forward(), done(), которые используем в своей программе.

Второй способ - хороший способ:

```
import turtle

turtle.forward(200)
turtle.done()
```

В этом втором способе мы импортируем не конкретные функции, а целую библиотеку. И далее уже ссылаемся на конкретные функции (методы) из этой библиотеки через точку (как бы показываем путь к конкретной функции (метода)).

Другими словами мы применяем методы forward() и done() к объекту turtle.

Третий способ - **не правильный способ** - он рабочий способ, но никогда так не делайте в своих программах.

```
from turtle import *

forward(200)
done()
```

Почему третий способ не совсем правильный в плане синтаксиса языка Python?

.....

Потому, что мы можем в своих программах подключать не одну библиотеку, а несколько. И если использовать вот такой способ подключения, как в третьем примере, то нам, или другому программисту, совсем невозможно будет понять, к какой библиотеке относится конкретная функция (метод) **forward(200)** или **done()**.

Пример:

```
from turtle import *
from csv import *

forward(200)
done()
```

Из примера мы видим, что понять к какой библиотеке относятся функции forward() и done() невозможно.

Конечно, если речь идет о стандартных функциях, часто используемых программистами, то может быть разработчик программ на Python и узнает по памяти, что, к примеру, функция forward() находится в библиотеке turtle. Но лучше такой синтаксис подключения библиотек не использовать.

Ведь дальше мы будем создавать собственные библиотеки. И может произойти такой момент, когда в собственной нашей библиотеке, и в сторонней библиотеке, которую мы подключим в нашу программу, будут использоваться одинаковые названия функций. И тогда точно будет в программном коде не понятно, название, чьей функции мы сейчас видим перед глазами и хотим ее использовать в программе.

Пример:

```
from myLib import *
from notMyLib import *
```

myFunction(200) # Точно не понятно, из какой библиотеки эта функция.

Напоминаю: в библиотеках **myLib** и **notMyLib** сейчас есть функция **myFunction()**.

Ну и еще один момент, который показывает минус третьего способа в код добавлять библиотеки и функции.

Когда мы таким образом (с помощью *) добавляем в программу сразу все функции, то программе для выполнения конкретной функции, когда программа встретит строку вызова функции, дольше придется перебирать загруженный список функций и искать нужную функцию для запуска.

Поэтому никогда не используем третий вид синтаксиса подключения библиотек и импорта функций в своих программах с помощью *.

Практика: Пример использования библиотеки turtle (рисуем персонажа).

Создаем новый файл.

import turtle # импортируем библиотеку

t = turtle.Turtle() # создаем объект класса turtle

turtle.done() # Внизу программы вызываем метод done() - он позволит не закрывать окно программы по её завершению.

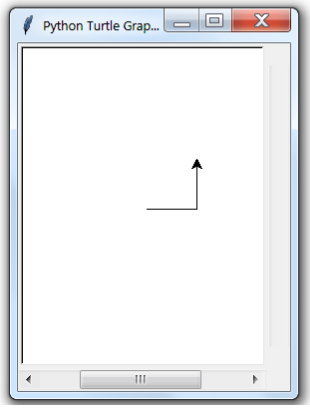
Теперь выше строки **turtle.done()** пишем весь код программы. Данная строка всегда должна быть внизу.

Временно напишем пример 1: две линии

```
# смещение стрелки вправо
t.forward(50)
# вращение стрелки на месте
```

```
t.left(90)
# смещение стрелки вверх
t.forward(50)
```

```
1  import turtle
2
3  t = turtle.Turtle()
4
5  # смещение стрелки вправо
6  t.forward(50)
7  # вращение стрелки на месте
8  t.left(90)
9  # смещение стрелки вверх
10 t.forward(50)
11
```



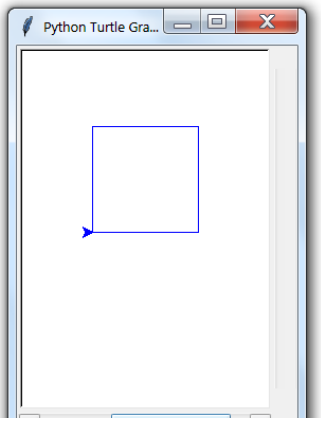
Стираем код рисования нашей стрелки.

Пишем новый пример кода (стираем прошлый код).

t.color('blue') # задаем всем линиям цвет (можно задать цвет в формате #fff)

```
# Рисуем квадрат
for i in range(4):
    t.forward(100)
    t.left(90)
```

```
1  import turtle
2  t = turtle.Turtle()
3  t.color('blue')
4
5  for i in range(4):
6      t.forward(100)
7      t.left(90)
8
9
10 turtle.done()
```



Можно закрасить квадрат цветом.

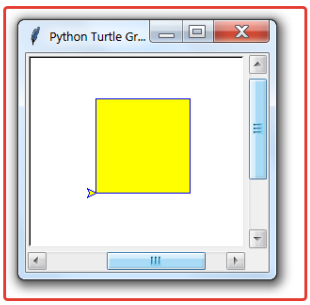
```
import turtle
```

```
t = turtle.Turtle()
t.color('blue')
t.fillcolor('yellow')

t.begin_fill()
for i in range(4):
    t.forward(100)
    t.left(90)
t.end_fill()

turtle.done()
```

```
1  import turtle
2  t = turtle.Turtle()
3  t.color('blue')
4  t.fillcolor('yellow')
5
6  t.begin_fill()
7  for i in range(4):
8      t.forward(100)
9      t.left(90)
10 t.end_fill()
11
12 turtle.done()
```



Можно увеличить толщину линии.