Функция – это структура, которую вы определяете. Вам нужно решить, будут ли в ней аргументы, или нет. Вы можете добавить как аргументы ключевых слов, так и готовые по умолчанию. Функция – это блок кода, который начинается с ключевого слова def, названия функции и двоеточия, пример:

In [5]:

```
def a_function():
    print("Просто создаем функцию")
```

In [6]:

```
a_function() # Вызываем созанную функцию
# обратите внимание, что если вы измените текст сообщения и не запустите блок, то сообщ
ение не поменяется
```

```
Просто создаем функцию
```

Пустая функция (stub) Иногда, когда вы пишете какой-нибудь код, вам нужно просто ввести определения функции, которое не содержит в себе код. Я сделал небольшой набросок, который поможет вам увидеть, каким будет ваше приложение. Вот пример:см. ниже оператор pass. Это пустая операция, это означает, что когда оператор pass выполняется, не происходит ничего

In [26]:

```
def empty_function():
    pass
```

In [ ]:

```

```

Передача аргументов функции Теперь мы готовы узнать о том, как создать функцию, которая может получать доступ к аргументам, а также узнаем, как передать аргументы функции. Создадим простую функцию, которая может суммировать два числа:

Функция может и не заканчиваться инструкцией return, при этом функция вернет значение None

In [27]:

```
def func():
     pass

print(func())
```

```
None
```

In [23]:

```
def add(a, b):
    return a + b #Инструкция return говорит, что нужно вернуть значение. В нашем случае функция возвращает сумму a и b.

print( add(1, 2) )
```

```
3
```

Каждая функция выдает определенный результат. Если вы не указываете на выдачу конкретного результата, она, тем не менее, выдаст результат None (ничего). В нашем примере мы указали выдать результат a + b. Как вы видите, мы можем вызвать функцию путем передачи двух значений. Если вы передали недостаточно, или слишком много аргументов для данной функции, вы получите ошибку:

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции!): def func(a, b, c=2): # c - необязательный аргумент return a + b + c func(1, 2) # a = 1, b = 2, c = 2 (по умолчанию)

func(1, 2, 3) # a = 1, b = 2, c = 3

func(a=1, b=3) # a = 1, b = 3, c = 2

func(a=3, c=6) # a = 3, c = 6, b не определен

In [30]:

```
def newfunc(n):
     def myfunc(x):
         return x + n
     return myfunc

new = newfunc(100)  # new - это функция

new(200)
```

Out[30]:

```
300
```

In [31]:

```
add(1) # неправильная передача аргументов
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
t)
<ipython-input-31-4dfda21dfbe6> in <module>
----> 1 add(1) # неправильная передача аргументов

TypeError: add() missing 1 required positional argument: 'b'
```

In [28]:

```
#правильная передача аргументов
print( add(a = 2, b = 3) )

total = add(b = 4, a = 5)
print(total)
```

```
5
9
```

Стоит отметить, что не важно, в каком порядке вы будете передавать аргументы функции до тех пор, как они называются корректно. Во втором примере мы назначили результат функции переменной под названием total. Это стандартный путь вызова функции в случае, если вы хотите дальше использовать её результат. А что, собственно, произойдет, если мы укажем аргументы, но они названы неправильно? Это сработает? Давайте попробуем на примере ниже

In [14]:

```
add(ac=5, bc=2)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-14-1adb3b06a8fb> in <module>
----> 1 add(ac=5, bc=2)

TypeError: add() got an unexpected keyword argument 'ac'
```

Ключевые аргументы Функции также могут принимать ключевые аргументы. Более того, они могут принимать как регулярные, так и ключевые аргументы. Это значит, что вы можете указывать, какие ключевые слова будут ключевыми, и передать их функции. Это было в примере выше.

In [15]:

```
def keyword_function(a=1, b=2):
    return a+b

print( keyword_function(b=4, a=5) )
```

```
9
```

Вы также можете вызвать данную функцию без спецификации ключевых слов. Эта функция также демонстрирует концепт аргументов, используемых по умолчанию. Каким образом? Попробуйте вызвать функцию без аргументов вообще Функция вернет нам с числом 3.т.к. a и b по умолчанию имеют значение 1 и 2 соответственно.

In [16]:

```
keyword_function()
```

Out[16]:

```
3
```

попробуем создать функцию, которая имеет обычный аргумент, и несколько ключевых аргументов: опишем три возможных случая. Проанализируем каждый из них. В первом примере мы попробуем вызвать функцию, используя только ключевые аргументы. Это даст нам только ошибку. Traceback указывает на то, что наша функция принимает, по крайней мере, один аргумент, но в примере было указано два аргумента. Что же произошло? Дело в том, что первый аргумент необходим, потому что он ни на что не указывает, так что, когда мы вызываем функцию только с ключевыми аргументами, это вызывает ошибку. Во втором примере мы вызвали смешанную функцию, с тремя значениями, два из которых имеют название. Это работает, и выдает нам ожидаемый результат: 1+4+5=10. Третий пример показывает, что происходит, если мы вызываем функцию, указывая только на одно значение, которое не рассматривается как значение по умолчанию. Это работает, если мы берем 1, и суммируем её к двум значениям по умолчанию: 2 и 3, чтобы получить результат 6!

In [17]:

```
def mixed_function(a, b=2, c=3):
    return a+b+c

mixed_function(b=4, c=5)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-17-0dd9609e91ff> in <module>
      2     return a+b+c
      3
----> 4 mixed_function(b=4, c=5)

TypeError: mixed_function() missing 1 required positional argument: 'a'
```

In [18]:

```
print( mixed_function(1, b=4, c=5) ) # 10

print( mixed_function(1) )
```

```
10
6
```

*args и **kwargs Вы также можете настроить функцию на прием любого количества аргументов, или ключевых аргументов, при помощи особого синтаксиса. Чтобы получить бесконечное количество аргументов, мы используем* args, а чтобы получить бесконечное количество ключевых аргументов, мы используем *kwargs. Сами слова “args” и “kwargs” не так важны. Это просто сокращение. Вы можете назвать их* lol и *omg, и они будут работать таким же образом. Главное здесь – это количество звездочек Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится **. Обратите внимание: в дополнение к конвенциям* args и *kwargs, вы также, время от времени, будете видеть andkw. Давайте взглянем на следующий пример:

In [19]:

```
def many(*args, **kwargs):
    print( args )
    print( kwargs )

many(1, 2, 3, name="Mike", job="programmer")
```

```
(1, 2, 3)
{'name': 'Mike', 'job': 'programmer'}
```

Сначала мы создали нашу функцию, при помощи нового синтаксиса, после чего мы вызвали его при помощи трех обычных аргументов, и двух ключевых аргументов. Функция показывает нам два типа аргументов. Как мы видим, параметр args превращается в кортеж, а kwargs – в словарь. Вы встретите такой тип кодинга, если взгляните на исходный код Пайтона, или в один из сторонних пакетов Пайтон

Область видимость и глобальные переменные Концепт области (scope) в Пайтон такой же, как и в большей части языков программирования. Область видимости указывает нам, когда и где переменная может быть использована. Если мы определяем переменные внутри функции, эти переменные могут быть использованы только внутри это функции. Когда функция заканчиваются, их можно больше не использовать, так как они находятся вне области видимости. Давайте взглянем на пример:

In [20]:

```
def function_a():
    a = 1
    b = 2
    return a+b


def function_b():
    c = 3
    return a+c

print( function_a() )
print( function_b() )
```

```
3

---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
<ipython-input-20-7e085c4eebd5> in <module>
     10
     11 print( function_a() )
---> 12 print( function_b() )

<ipython-input-20-7e085c4eebd5> in function_b()
      7 def function_b():
      8     c = 3
----> 9     return a+c
     10
     11 print( function_a() )

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

ОШИБКА!!! Это вызвано тем, что переменная определенна только внутри первой функции, но не во второй. Вы можете обойти этот момент, указав в Python, что переменная a – глобальная (global). Попробуйте пример ниже:

In [21]:

```
def function_a():
    global a
    a = 1
    b = 2
    return a+b


def function_b():
    c = 3
    return a+c

print( function_a() )
print( function_b() )
```

```
3
4
```

Этот код работает, так как мы указали Python сделать a – глобальной переменной, а это значит, что она работает где-либо в программе. Из этого вытекает, что это настолько же хорошая идея, насколько и плохая. Причина, по которой эта идея – плохая в том, что нам становится трудно сказать, когда и где переменная была определена. Другая проблема заключается в следующем: когда мы определяем «a» как глобальную в одном месте, мы можем случайно переопределить её значение в другом, что может вызвать логическую ошибку, которую не просто исправить. Советы в написании кода Одна из самых больших проблем для молодых программистов – это усвоить правило «не повторяй сам себя». Суть в том, что вы не должны писать один и тот же код несколько раз. Когда вы это делаете, вы знаете, что кусок кода должен идти в функцию. Одна из основных причин для этого заключается в том, что вам, вероятно, придется снова изменить этот фрагмент кода в будущем, и если он будет находиться в нескольких местах, вам нужно будет помнить, где все эти местоположения И изменить их.

Анонимные функции, инструкция lambda Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции lambda. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией def func(): lambda функции, в отличие от обычной, не требуется инструкция return, а в остальном, ведет себя точно так же:

In [33]:

```
func = lambda x, y: x + y
print (func(1, 2))
print (func('a', 'b'))
print ((lambda x, y: x + y)(1, 2))
print ((lambda x, y: x + y)('a', 'b'))
```

```
3
ab
3
ab
```

In [34]:

```
func = lambda *args: args
func(1, 2, 3, 4)
```

Out[34]:

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

Работа с файлами r Открывает файл только для чтения. Указатель стоит в начале файла. rb Открывает файл для чтения в двоичном формате. Указатель стоит в начале файла. r+ Открывает файл для чтения и записи. Указатель стоит в начале файла. rb+ Открывает файл для чтения и записи в двоичном формате. Указатель стоит в начале файла. w Открывает файл только для записи. Указатель стоит в начале файла. Создает файл с именем имя_файла, если такового не существует. wb Открывает файл для записи в двоичном формате. Указатель стоит в начале файла. Создает файл с именем имя_файла, если такового не существует. w+ Открывает файл для чтения и записи. Указатель стоит в начале файла. Создает файл с именем имя_файла, если такового не существует. wb+ Открывает файл для чтения и записи в двоичном формате. Указатель стоит в начале файла. Создает файл с именем имя_файла, если такового не существует. a Открывает файл для добавления информации в файл. Указатель стоит в конце файла. Создает файл с именем имя_файла, если такового не существует. ab Открывает файл для добавления в двоичном формате. Указатель стоит в конце файла. Создает файл с именем имя_файла, если такового не существует. a+ Открывает файл для добавления и чтения. Указатель стоит в конце файла. Создает файл с именем имя_файла, если такового не существует. ab+ Открывает файл для добавления и чтения в двоичном формате. Указатель стоит в конце файла. Создает файл с именем имя_файла, если такового не существует Атрибуты файла: file.closed Возвращает True если файл был закрыт. file.mode Возвращает режим доступа, с которым был открыт файл. file.name Возвращает имя файла. file.softspace Возвращает False если при выводе содержимого файла следует отдельно добавлять пробел.

In [37]:

```
my_file = open("test1.txt", "w")
print("Имя файла: ", my_file.name)
print("Файл закрыт: ", my_file.closed)
print("В каком режиме файл открыт: ", my_file.mode)

print("Имя файла: ", my_file.name)
print("Файл закрыт: ", my_file.closed)
my_file.close()
print("А теперь закрыт: ", my_file.closed)
```

```
Имя файла:  test1.txt
Файл закрыт:  False
В каком режиме файл открыт:  w
Имя файла:  test1.txt
Файл закрыт:  False
А теперь закрыт:  True
```

Запись в файл в Python. Метод write(). Метод write() записывает любую строку в открытый файл. Важно помнить, что строки в Python могут содержать двоичные данные, а не только текст.

Метод write() не добавляет символ переноса строки ('\n') в конец файла.

In [38]:

```
#Синтаксис метода write(). my_file.write(string)
```

In [40]:

```
my_file = open("some.txt", "w")
my_file.write("Мне нравится Python!\nЭто классный язык!\n Дописать")
my_file.close()
#Обратите внимание, что есди файл не существовал, то write его создает. поэкперемнтриру
йте с открытием
#файлов для записи и для чтения с дозаписью информации в файл
```

Чтение из файла в Python. Метод read(). Метод read() читает строку из открытого файла.

Синтаксис метода read().

? 1 my_file.read([count]) Необязательный параметр count - это количество байт, которые следует прочитать из открытого файла. Этот метод читает информацию с начала файла и, если параметр count не указан, до конца файла.

Например, прочтем созданный нами файл some.txt

In [1]:

```
my_file = open("some.txt")
my_string = my_file.read()
print("Было прочитано:")
print(my_string)
my_file.close()
```

```
Было прочитано:
Мне нравится Python!
Это классный язык!
 Дописать
```

Как узнать позицию указателя в файле в Python. После того как вы вызвали метод read() на файловом объекте, если вы повторно вызовете read(), то увидите лишь пустую строку. Это происходит потому, что после первого прочтения указатель находится в конце файла. Для того чтобы узнать позицию указателя можно использовать метод tell().

Например:

In [3]:

```
my_file = open("some.txt")
my_file.read(10)
print ("Я на позиции:", my_file.tell())
my_file.close()
```

```
Я на позиции: 10
```

Говоря проще, метод tell() сообщает в скольких байтах от начала файла мы сейчас находимся.

Чтобы перейти на нужную нам позицию, следует использовать другой метод - seek(). Синтаксис my_file.seek(offset, [from]) Аргумент offset указывает на сколько байт перейти. опциональный аргумент from означает позицию, с которой начинается движение. 0 - означает начало файла, 1 нынешняя позиция, 2 - конец файла.

The seek(offset[, from]) method changes the current file position. The offset argument indicates the number of bytes to be moved. The from argument specifies the reference position from where the bytes are to be moved.

Например:

In [4]:

```
my_file = open("some.txt", "r")
print(my_file.read(10))
print("Мы находимся на позиции: ", my_file.tell())
# Возвращаемся в начало
my_file.seek(0)
print(my_file.read(10))
my_file.close()
```

```
Мне нравит
Мы находимся на позиции:  10
Мне нравит
```

Добавление в файл. Метод write() Если вы хотите не перезаписать файл полностью (что делает метод write в случае открытия файла в режиме 'w'), а только добавить какой-либо текст, то файл следует открывать в режиме 'a' - appending. После чего использовать все тот же метод write.

Например:

In [5]:

```
# Удалит существующую информацию в some.txt и запишет "Hello".
my_file = open("some.txt", 'w')
my_file.write("Hello")
my_file.close()
# Оставит существующую информацию в some.txt и добавит "Hello".
my_file = open("some.txt", 'a')
my_file.write("Hello")
my_file.close()
```

Расширенная работа с файлами в Python. Для доступа к более широкому функционалу в работе с файлами в Python, как то удаление файлов, создание директорий и т.д. Следует подключить библиотеку os.

In [8]:

```
handle = open("some.txt", "r")
data = handle.read()
print(data)
handle.close()
```

```
HelloHello
```

Давайте обратим внимание на различные способы чтения файлов.

In [9]:

```
handle = open("some.txt", "r")
data = handle.readline() # read just one line
print(data)
handle.close()
```

```
HelloHello
```

Если вы используете данный пример, будет прочтена и распечатана только первая строка текстового файла. Это не очень полезно, так что воспользуемся методом readlines() в дескрипторе:

In [12]:

```
handle = open("some.txt", "r")
data = handle.readlines() # read ALL the lines!
print(data)
handle.close()
```

```
['HelloHello\n', 'Мне нравился Питон\n', 'Пока не узнал java']
```

Как читать файл по частям Самый простой способ для выполнения этой задачи – использовать цикл. Сначала мы научимся читать файл строку за строкой, после этого мы будем читать по килобайту за раз. В нашем первом примере мы применим цикл:

In [13]:

```
handle = open("some.txt", "r")

for line in handle:
    print(line)

handle.close()
```

```
HelloHello

Мне нравился Питон

Пока не узнал java
```

Таким образом мы открываем файл в дескрипторе в режиме «только чтение», после чего используем цикл для его повторения. Стоит обратить внимание на то, что цикл можно применять к любым объектам Python (строки, списки, запятые, ключи в словаре, и другие). Весьма просто, не так ли? Попробуем прочесть файл по частям:

In [16]:

```
handle = open("some.txt", "r")

while True:
    data = handle.read(1024)
    print(data)

    if not data:
        break
```

```
HelloHello
Мне нравился Питон
Пока не узнал java
```

Как читать бинарные (двоичные) файлы Это очень просто. Все что вам нужно, это изменить способ доступа к файлу Мы изменили способ доступа к файлу на rb, что значит read-binaryy. Стоит отметить то, что вам может понадобиться читать бинарные файлы, когда вы качаете PDF файлы из интернете, или обмениваетесь ими между компьютерами:

In [17]:

```
handle = open("some.pdf", "rb")
```

Самлстоятельно рассмотрите режимы Мы изменили способ доступа к файлу на rb, что значит read-binaryy. Стоит отметить то, что вам может понадобиться читать бинарные файлы, когда вы качаете PDF файлы из интернете, или обмениваетесь ими между компьютерами.

Использование оператора «with» В Python имеется аккуратно встроенный инструмент, применяя который вы можете заметно упростить чтение и редактирование файлов. Оператор with создает диспетчер контекста в Пайтоне, который автоматически закрывает файл для вас, по окончанию работы в нем. Посмотрим, как это работает:

In [18]:

```
with open("some.txt") as file_handler:
    for line in file_handler:
        print(line)
```

```
HelloHello

Мне нравился Питон

Пока не узнал java
```

Выявление ошибок Иногда, в ходе работы, ошибки случаются. Файл может быть закрыт, потому что какой-то другой процесс пользуется им в данный момент или из-за наличия той или иной ошибки разрешения. Когда это происходит, может появиться IOError. В данном разделе мы попробуем выявить эти ошибки обычным способом, и с применением оператора with. Подсказка: данная идея применима к обоим способам.

In [20]:

```
try:
    file_handler = open("some.txt")
    for line in file_handler:
        print(line)
except IOError:
    print("An IOError has occurred!")
finally:
    file_handler.close()
```

```
HelloHello

Мне нравился Питон

Пока не узнал java
```

В описанном выше примере, мы помещаем обычный код в конструкции try/except. Если ошибка возникнет, следует открыть сообщение на экране. Обратите внимание на то, что следует удостовериться в том, что файл закрыт при помощи оператора finally. Теперь мы готовы взглянуть на то, как мы можем сделать то же самое, пользуясь следующим методом:

In [22]:

```
try:
    with open("some.txt") as file_handler:
    for line in file_handler:
        print(line)
except IOError:
    print("An IOError has occurred!")
```

```
  File "<ipython-input-22-4526ae252407>", line 3
    for line in file_handler:
      ^
IndentationError: expected an indented block
```

Задание на практику:

1. реализовать решение квадратного уравнения через дискриминант;
2. реализовать создание, запись, чтение и удаление файла с данными о пользователе. пользователь выбирает действие самостоятельно, а так же указывает путь к размещению файла.
3. для задания 5 или 6 из предыдущей практики реализовать:

```
    3.1 применение функций (не менее 5 штук)
    3.2 расписание полетов или поездов задается файлом, свободные места в вашон
   е или салоне указываются в файле (при продаже билета изменяете файл)
    3.3 реализовать заполнение шаблонов билетов (шаблон билетов разрабатывается
   самостоятельно) данными о рейсе, ФИО пассажира, вагоне, месте, времени и дат
   е отправления на весь путь(с учетом пересадок)
```