Тема: Самостоятельная работа по теме “Функции python”

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:

```
def a_function():
    print("You just created a function!")
```

Эта функция не делает ничего, кроме отображения текста. Чтобы вызвать функцию, вам нужно ввести название функции, за которой следует открывающаяся и закрывающаяся скобки:

```
a_function() # You just created a function!
```

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:

```
def empty_function():
    Pass
```

А вот здесь кое-что новенькое: оператор pass. Это пустая операция, это означает, что когда оператор **pass** выполняется, не происходит ничего.

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:

```
def add(a, b):
    return a + 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:

```
add(1)
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: add() takes exactly 2 arguments (1 given)
```

Вы также можете вызвать функцию, указав наименование аргументов:

```
print( add(a = 2, b = 3) ) # 5


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

print(total) # 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.

Вы, возможно, подумаете: «*А что, собственно, произойдет, если мы укажем аргументы, но они названы неправильно? Это сработает?*» Давайте попробуем на примере:

```
add(c=5, d=2)

Traceback (most recent call last):

  File "<string>", line 1, in <fragment>

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

Ошибка. Кто бы мог подумать? Это значит, что мы указали ключевой аргумент, который функция не распознала. Кстати, ключевые аргументы описана ниже.

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.

```
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!

```
keyword_function() # 3
```

Функция вернулась к нам с числом 3. *Почему?* Причина заключается в том, что a и b по умолчанию имеют значение 1 и 2 соответственно. Теперь попробуем создать функцию, которая имеет обычный аргумент, и несколько ключевых аргументов:

```
def mixed_function(a, b=2, c=3):
   return a+b+c
mixed_function(b=4, c=5)
Traceback (most recent call last):
   File "<string>", line 1, in <fragment>
TypeError: mixed_function() takes at least 1 argument (2 given)


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


print( mixed_function(1) ) # 6
```

Выше мы описали три возможных случая. Проанализируем каждый из них. В первом примере мы попробовали вызвать функцию, используя только ключевые аргументы. Это дало 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! *Удивительно, не так ли?*

***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. Обратите внимание: в дополнение к конвенциям *args и *kwargs, вы также, время от времени, будете видеть andkw. Давайте взглянем на следующий пример:

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


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

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

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:

```
def function_a():
    a = 1
    b = 2
    return a+b
def function_b():
    c = 3
    return a+c
print( function_a() )
print( function_b() )
```

Если вы запустите этот код, вы получите ошибку:

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

Это вызвано тем, что *переменная определенна только внутри первой функции*, но не во второй. Вы можете обойти этот момент, указав в Пайтоне, что переменная a – глобальная (global). Давайте взглянем на то, как это работает:

```
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() )
```

Этот код работает, так как мы указали Пайтону 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.

Копировать и вставлять один и тот же кусок кода – хороший пример **спагетти-кода.** Постарайтесь избегать этого так часто, как только получится. Вы будете сожалеть об этом в какой-то момент либо потому, что вам придется все это исправлять, либо потому, что вы столкнетесь с чужим кодом, с которым вам придется работать и исправлять вот это вот всё.

Задачи:
Найти наименьшее общее кратное (НОК) пары чисел по формуле

НОК = ab / НОД(a, b),

где a и b - это натуральные числа, НОД - наибольший общий делитель.

Из условия задачи ясно, чтобы найти НОК, надо сначала найти НОД. Последний можно вычислить, постепенно находя остаток от деления большего числа из пары на меньшее и присваивая остаток переменной, связанной с большим числом (см. алгоритм Евклида). В какой-то момент значение одной из переменных станет равным 0. Когда это произойдет, другая будет содержать НОД. Если неизвестно, какая именно переменная содержит НОД, то можно просто сложить значения обоих переменных.

В коде ниже используется функция для нахождения НОК, которая принимает два числа и возвращает найденное наименьшее общее кратное.

В основной ветке программы функция вызывается в цикле, который завершается, если то, что было введено, нельзя преобразовать к целому. В этом случае генерируется исключение и поток выполнения переходит к ветке except.

```
def lcm(a,b):
    m = a*b
    while a != 0 and b != 0:
         if a > b:
           a %= b
         Else:
            b %= a
    return m // (a+b)
while 1:
    try:
        x = int(input('a='))
        y = int(input('b='))
        print('НОК:',lcm(x,y))
except:
Break
```

Циклический сдвиг
Выполнить циклический сдвиг в списке целых чисел на указанное число шагов. Сдвиг также должен быть кольцевым, то есть элемент, вышедший за пределы списка, должен появляться с другого его конца.

Для решения этой задачи можно воспользоваться следующими методами списка:

- append() - добавляет элемент в конец списка,
- insert() - вставляет элемент по указанному индексу,
- pop() - извлекает элемент с конца списка или, если был передан индекс, по индексу.

Пусть функция shift() в качестве аргументов принимает список и количество шагов сдвига. Если шаги представлены положительным числом, то сдвиг выполняется вправо, то есть надо извлечь последний элемент и добавить его в начало.

Если шаги - отрицательное число, то будем выполнять сдвиг справа налево, то есть надо извлечь первый элемент и добавить его в конец.

Подведем итоги
Теперь вы обладаете основательными знаниями, которые необходимы для эффективной работы с функциями. Попрактикуйтесь в создании простых функций, и попробуйте обращаться к ним различными способами.

Задания:

https://younglinux.info/python/task/list-dict

https://younglinux.info/python/task/percentage-char

https://taskcode.ru/function/digits

Ответы отправлять на почту - magus-grand@mail.ru

Программа рассчитана на два урока 06.04.2020-08.04.2020