Лекция 8.

Правила написания функций в Python. Специализированные функции

Функция это блок организованного, многократно используемоего кода, который используется для выполнения конкретного задания. Функции обеспечивают лучшую модульность приложения и значительно повышают уровень повторного использования кода.

Создание функции

Существуют некоторые правила для создания **функций в Python**.

- Блок функции начинается с ключевого слова **def**, после которого следуют название функции и круглые скобки (()).
- Любые аргументы, которые принимает функция должны находиться внутри этих скобок.
- После скобок идет двоеточие (:) и с новой строки с отступом начинается тело функции.

Пример функции в Python:

```
?
1def my_function(argument):
2    print argument
```

Вызов функции

После создания функции, ее можно исполнять вызывая из другой функции или напрямую из оболочки **Python**. Для вызова функции следует ввести ее имя и добавить скобки.

Например:

```
?
1my_function("abracadabra")
```

Аргументы функции в Python

Вызывая функцию, мы можем передавать ей следующие типы аргументов:

- *Обязательные аргументы (Required arguments)*
- *Аргументы-ключевые слова (Keyword argument)*
- *Аргументы по умолчанию (Default argument)*
- *Аргументы произвольной длины (Variable-length argumens)*

Обязательные аргументы функции:

Если при создании функции мы указали количество передаваемых ей аргументов и их порядок, то и вызывать ее мы должны с тем же количеством аргументов, заданных в нужном порядке.

Например:

```
?
1 def bigger(a,b):
2     if a > b:
3         print a
4     else:
5        print b
6
7 # В описании функции указано, что она принимает 2 аргумента
8
9 # Корректное использование функции
10bigger(5,6)
11
12# Некорректное использование функции
13bigger()
14bigger(3)
15bigger(12,7,3)
```

Аргументы - ключевые слова

Аргументы - ключевые слова используются при вызове функции. Благодаря ключевым аргументам, вы можете задавать произвольный (то есть не такой каким он описа при создании функции) порядок аргументов.

Например:

```
?
1 def person(name, age):
2     print name, "is", age, "years old"
3
4 # Хотя в описании функции первым аргументом идет имя, мы можем
5 вызвать функцию вот так
6
person(age=23, name="John")
```

Аргументы, заданные по-умолчанию

Аргумент по умолчанию, это аргумент, значение для которого задано изначально, при создании функции.

Например:

```
?
1 def space(planet_name, center="Star"):
2     print planet_name, "is orbiting a", center
3
4 # Можно вызвать функцию space так:
5 space("Mars")
6 # В результате получим: Mars is orbiting a Star
7
8 # Можно вызвать функцию space иначе:
9 space("Mars", "Black Hole")
10# В результате получим: Mars is orbiting a Black Hole
```

Аргументы произвольной длины

Иногда возникает ситуация, когда вы заранее не знаете, какое количество аргументов будет необходимо принять функции. В этом случае следует использовать аргументы произвольной длины. Они задаются произвольным именем переменной, перед которой ставится звездочка (*).

Например:

```
?
1 def unknown(*args):
2     for argument in args:
3         print argument
4
5 unknown("hello","world") # напечатает оба слова, каждое с новой
6 строки
7 unknown(1,2,3,4,5) # напечатает все числа, каждое с новой строки
unknown() # ничего не выведет
```

Ключевое слово *return*

Выражение return прекращает выполнение функции и возвращает указанное после выражения значение. Выражение return без аргументов это то же самое, что и выражение **return None**. Соответственно, теперь становится возможным, например, присваивать результат выполнения функции какой либо переменной.

Например:

```
?
 def bigger(a,b):
1    if a > b:
2        return a # Если a больше чем b, то возвращаем b и
3прекращаем выполнение функции
4    return b # Незачем использовать else. Если мы дошли до этой
5строки, то b, точно не меньше чем a
6
7# присваиваем результат функции bigger переменной num
 num = bigger(23,42)
```

Область видимости

Некоторые переменные скрипта могут быть недоступны некоторым областям программы. Все зависит от того, где вы объявили эти переменные.

В **Python** две базовых области видимости переменных:

- *Глобальные переменные*
- *Локальные переменные*

Переменные объявленные внутри тела функции имеют локальную область видимости, те что объявлены вне какой-либо функции имеют глобальную область видимости.

Это означает, что доступ к локальным переменным имеют только те функции, в которых они были объявлены, в то время как доступ к глобальным переменным можно получить по всей программе в любой функции.

Например:

```
?
1 # глобальная переменная age
2 age = 44
3
4 def info():
5     print age # Печатаем глобальную переменную age
6
7 def local_info():
8     age = 22 # создаем локальную переменную age
9     print age
10
11 info() # напечатает 44
12 local_info() # напечатает 22
```

Важно помнить, что для того чтобы получить доступ к глобальной переменной, достаточно лишь указать ее имя. Однако, если перед нами стоит задача *изменить* глобальную переменную внутри функции - необходимо использовать ключевое слово **global**.

Например:

```
?
1 # глобальная переменная age
2 age = 13
3
4 # функция изменяющая глобальную переменную
5 def get_older():
6     global age
7     age += 1
8
9 print age # напечатает 13
10 get_older() # увеличиваем age на 1
11 print age # напечатает 14
```

Рекурсия

Рекурсией в программировании называется ситуация, в которой функция вызывает саму себя. Классическим примером рекурсии может послужить функция вычисления факториала числа.

Напомним, что факториалом числа, например, 5 является произведение всех натуральных (целых) чисел от 1 до 5. То есть, 1 * 2 * 3 * 4 * 5

Рекурсивная функция вычисления факториала на языке Python будет выглядеть так:

```
?
 def fact(num):
1    if num == 0:
2        return 1 # По договоренности факториал нуля равен единице
3    else:
4        return num * fact(num - 1) # возвращаем результат
5произведения num и результата возвращенного функцией fact(num -
 1)
```

Однако следует помнить, что использование рекурсии часто может быть неоправданным. Дело в том, что в момент вызова функции в оперативной памяти компьютера резервируется определенное количество памяти, соответственно чем больше функций одновременно мы запускаем - тем больше памяти потребуется, что может привести к переполнению стека (stack overflow) и программа завершится аварийно, не так как предполагалось. Учитывая это, там где это возможно, вместо рекурсии лучше применять *циклы*.

Рецепт создания функции в Python

Существует следующий алгоритм - рекомендация по созданию **функции в Python**. Например, мы создаем функцию вычисления площади прямоугольника.

1. Начинать следует с примеров того, что делает функция, и подобрать подходящее название. В нашем случае это будет выглядеть так:

```
?
# На данном этапе мы еще не указываем имена переменных
def rectangle_area_finder( ):
    """
    >>> rectangle_area_finder(3, 5)
    15
    >>> rectangle_area_finder(17.2, 6)
    103.2
    """
```

2. Указать типы данных, которые принимает функция и тип данных, который она возвращает

```
?
1 # функция принимает два числа, а возвращает одно
2 def rectangle_area_finder( ):
3     """
4     (num, num) -> num
5
6     >>> rectangle_area_finder(3, 5)
7     15
8     >>> rectangle_area_finder(17.2, 6)
9     103.2
10    """
```

3. Подобрать подходящие названия для переменных

```
?

# Поскольку это математическая функция нам вполне подойдут имена a и b
def rectangle_area_finder(a, b):
    """
    (num, num) -> num

    >>> rectangle_area_finder(3, 5)
    15
    >>> rectangle_area_finder(17.2, 6)
    103.2
    """
```

4. Написать краткое, но содержательное описание функции

```
?
def rectangle_area_finder(a, b):
    """
    (num, num) -> num

    Returns an area of a rectangle with given sides a and
b.

    >>> rectangle_area_finder(3, 5)
    15
    >>> rectangle_area_finder(17.2, 6)
    103.2
    """
```

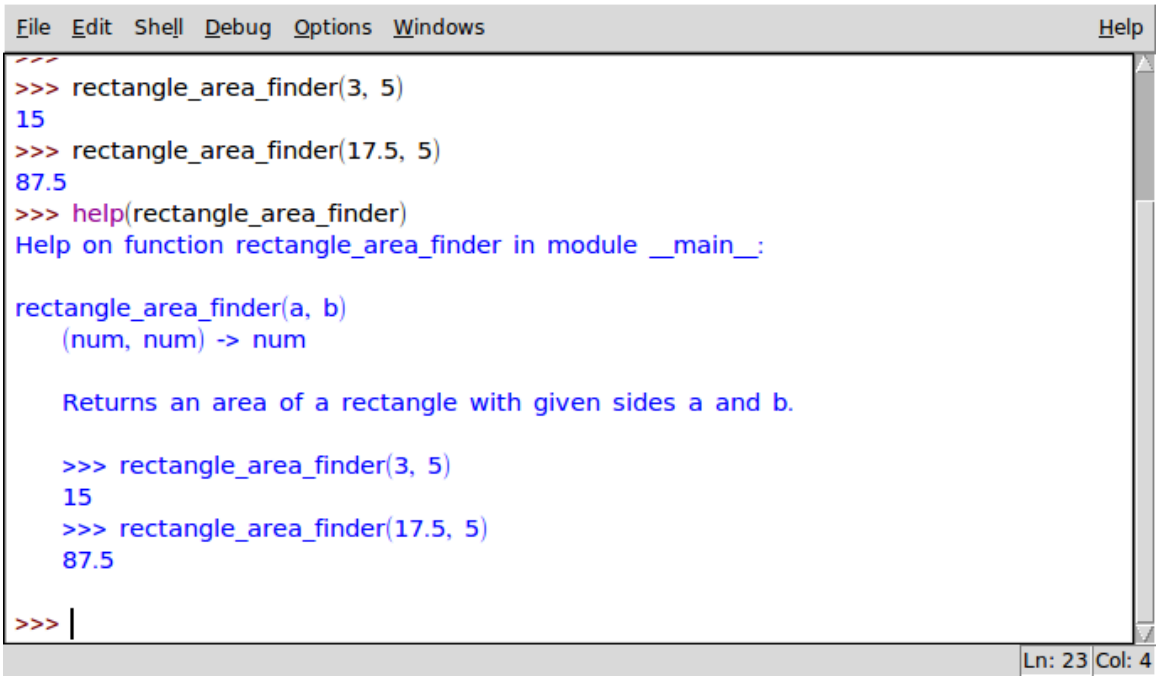
5. Написать собственно тело функции

```
?
def rectangle_area_finder(a, b):
    """
    (num, num) -> num

    Returns an area of a rectangle with given sides a and
b.

    >>> rectangle_area_finder(3, 5)
    15
    >>> rectangle_area_finder(17.2, 6)
    103.2
    """
    return a * b
```

6. Функция готова! Осталось вызвать ее с указанными в примерах аргументами



Как видно, при вызове команды help() с именем нашей функции в качестве аргумента мы получаем написанную нами документацию.

Сопровождайте ваши функции качественной документацией и программисты, которые будут работать с вашим кодом после вас будут вам благодарны.

Специализированные функции

Как правило, когда говорят о элементах функционального программировании в Python, то подразумеваются следующие функции: **lambda**, **map**, **filter**, **reduce**, **zip**.

1. Lambda выражение в Python.
2. Функция map() в Python.
3. Функция filter() в Python.
4. Функция reduce() в Python.
5. Функция zip() в Python.

Lambda выражение в Python:

lambda оператор или **lambda функция в Python** это способ создать анонимную функцию, то есть функцию без имени. Такие функции можно назвать одноразовыми, они используются только при создании. Как правило, **lambda функции** используются в комбинации с функциями filter, map, reduce.

Синтаксис lambda выражения в Python

```
?
1lambda arguments: expression
```

В качестве arguments передается список аргументов, разделенных запятой, после чего над переданными аргументами выполняется expression. Если присвоить lambda-функцию переменной, то получим поведение как в обычной функции (делаем мы это исключительно в целях демонстрации)

```
?
1>>> multiply = lambda x,y: x * y
2>>> multiply(21, 2)
342
```

Но, конечно же, все преимущества lambda-выражений мы получаем, используя lambda в связке с другими функциями

Функция map() в Python:

В **Python функция map** принимает два аргумента: функцию и аргумент составного типа данных, например, список. map применяет к каждому элементу списка переданную функцию. Например, вы прочитали из файла список чисел, изначально все эти числа имеют строковый тип данных, чтобы работать с ними - нужно превратить их в целое число:

```
?
1old_list = ['1', '2', '3', '4', '5', '6', '7']
2
3new_list = []
4for item in old_list:
5    new_list.append(int(item))
6
7print (new_list)
8
9[1, 2, 3, 4, 5, 6, 7]
```

Тот же эффект мы можем получить, применив функцию map:

```
?
1old_list = ['1', '2', '3', '4', '5', '6', '7']
2new_list = list(map(int, old_list))
3print (new_list)
4
5[1, 2, 3, 4, 5, 6, 7]
```

Как видите такой способ занимает меньше строк, более читабелен и выполняется быстрее.

map также работает и с функциями созданными пользователем:

```
def miles_to_kilometers(num_miles):
    """ Converts miles to the kilometers """
    return num_miles * 1.6

mile_distances = [1.0, 6.5, 17.4, 2.4, 9]
kilometer_distances = list(map(miles_to_kilometers,
mile_distances))
print (kilometer_distances)

[1.6, 10.4, 27.84, 3.84, 14.4]
```

А теперь то же самое, только используя **lambda выражение**:

```
mile_distances = [1.0, 6.5, 17.4, 2.4, 9]
kilometer_distances = list(map(lambda x: x * 1.6, mile_distances))

print (kilometer_distances)

[1.6, 10.4, 27.84, 3.84, 14.4]
```

Функция map может быть так же применена для нескольких списков, в таком случае функция-аргумент должна принимать количество аргументов, соответствующее количеству списков:

```
?
l1 = [1,2,3]
l2 = [4,5,6]

new_list = list(map(lambda x,y: x + y, l1, l2))
print (new_list)

[5, 7, 9]
```

Если же количество элементов в списках совпадать не будет, то выполнение закончится на минимальном списке:

```
?
l1 = [1,2,3]
l2 = [4,5]

new_list = list(map(lambda x,y:  + y, l1, l2))

print (new_list)
[5,7]
```

Функция filter() в Python:

Функция filter предлагает элегантный вариант фильтрации элементов последовательности. Принимает в качестве аргументов функцию и последовательность, которую необходимо отфильтровать:

```
?
mixed = ['мак', 'просо', 'мак', 'мак', 'просо', 'мак', 'просо',
'просо', 'просо', 'мак']
zolushka = list(filter(lambda x: x == 'мак', mixed))

print(zolushka)
['мак', 'мак', 'мак', 'мак', 'мак']
```

Обратите внимание, что функция, передаваемая в **filter** должна возвращать значение True / False, чтобы элементы корректно отфильтровались.

Функция reduce() в Python:

Функция reduce принимает 2 аргумента: функцию и последовательность. reduce() последовательно применяет функцию-аргумент к элементам списка, возвращает единичное значение. Обратите внимание в Python 2.x функция reduce доступна как встроенная, в то время, как в Python 3 она была перемещена в модуль functools.

Вычисление суммы всех элементов списка при помощи reduce:

```
?
1from functools import reduce
2items = [1,2,3,4,5]
3sum_all = reduce(lambda x,y: x + y, items)
4
5print (sum_all)
6
715
```

Вычисление наибольшего элемента в списке при помощи reduce:

```
?
1from functools import reduce
2items = [1, 24, 17, 14, 9, 32, 2]
3all_max = reduce(lambda a,b: a if (a > b) else b, items)
4
5print (all_max)
632
```

Функция zip() в Python:

Функция zip объединяет в кортежи элементы из последовательностей переданных в качестве аргументов.

```
?
a = [1,2,3]
b = "xyz"
c = (None, True)

res = list(zip(a, b, c))
print (res)

[(1, 'x', None), (2, 'y', True)]
```

Обратите внимание, что **zip** прекращает выполнение, как только достигнут конец самого короткого списка.