Python Functions

https://www.w3schools.com/python/python_functions.asp

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

Creating a Function:
In Python a function is defined using the def keyword:

```
def my_function():
  print("Hello from a function")
```

Calling a Function:
To call a function, use the function name followed by parenthesis:

```
def my_function():
  print("Hello from a function")
my_function()
```

Arguments:
- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.
- The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):
  print(fname + " Refsnes")
my_function("Emil")
my_function("Tobias")
my_function("Linus")

Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

Параметры или аргументы?

Параметр и *аргумент* терминов могут использоваться для одного и того же: информации, которая передается в функцию.

С точки зрения функции:

- **Параметр** — это переменная, указанная в скобках в определении функции.
- **Аргумент** — это значение, которое передается функции при ее вызове.

Количество аргументов:

По умолчанию функция должна вызываться с правильным количеством аргументов. Это означает, что если ваша функция ожидает 2 аргумента, вы должны вызывать функцию с 2 аргументами, не больше и не меньше.

Эта функция ожидает 2 аргумента и получает 2 аргумента:

```
def my_function(fname, lname):
  print(fname + " " + lname)
my_function("Emil", "Refsnes")
```

Если вы попытаетесь вызвать функцию с 1 или 3 аргументами, вы получите ошибку:

```
def my_function(fname, lname):
  print(fname + " " + lname)
my_function("Emil")
```

Arbitrary Arguments, *args:

Если вы не знаете, сколько аргументов будет передано в вашу функцию, добавьте `*` перед именем параметра в определении функции.

Таким образом, функция получит *кортеж* аргументов и сможет получить соответствующий доступ к элементам:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")
```

**kwargs

https://book.pythontips.com/en/latest/args_and_kwargs.html

kwargs позволяет вам передавать произвольное число **именованных аргументов в функцию. Таким образом, вам необходимо использовать **kwargs там, где вы хотите работать с **именованными** аргументами. Очередной пример:

```
def greet_me(**kwargs):
    for key, value in kwargs.items():
        print("{0} = {1}".format(key, value))

>>> greet_me(name="yasoob")
name = yasoob
```

В результате, мы оперируем произвольным числом именованных аргументов в нашей функции. Это были основы использования **kwargs и можете сами отметить насколько это может быть удобно в определенных ситуациях.
Когда их использовать?
Все зависит от ваших потребностей. Наиболее часто *args и **kwargs используются при написании декораторов (подробнее о декораторах в другой главе).

Keyword Arguments:

Вы также можете отправлять аргументы с синтаксисом *ключ = значение* .

Таким образом, порядок аргументов не имеет значения.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)
my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

The youngest child is Linus
```

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: `**` before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

```
def my_function(**kid):
  print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes")

His last name is Refsnes
```

Default Parameter Value:

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

```
def my_function(country = "Norway"):
  print("I am from " + country)
my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")

I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

Passing a List as an Argument:

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

```
def my_function(food):
  for x in food:
    print(x)
fruits = ["apple", "banana", "cherry"]
my_function(fruits)

apple
banana
cherry
```

Return Values:

To let a function return a value, use the `return` statement:

```
def my_function(x):
  return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))

15
25
45
```

The pass Statement:

`function` definitions cannot be empty, but if you for some reason have a `function` definition with no content, put in the `pass` statement to avoid getting an error.

```
def myfunction():
  pass
# having an empty function definition like this, would raise an error without the pass statement
```

Рекурсия

Python также допускает рекурсию функций, что означает, что определенная функция может вызывать сама себя.

Рекурсия является общей математической и программной концепцией. Это означает, что функция вызывает сама себя. Преимущество этого заключается в том, что вы можете перебирать данные для достижения результата.

Разработчик должен быть очень осторожен с рекурсией, так как может быть довольно легко написать функцию, которая никогда не завершится, или функцию, которая использует избыточное количество памяти или мощности процессора. Однако при правильном написании рекурсия может быть очень эффективным и математически элегантным подходом к программированию.

В этом примере tri_recursion() — это функция, которую мы определили для вызова самой себя («рекурсия»). Мы используем переменную k в качестве данных, которые уменьшаются (-1) каждый раз, когда мы рекурсивно. Рекурсия заканчивается, когда условие не больше 0 (т.е. когда оно равно 0).

Новому разработчику может потребоваться некоторое время, чтобы понять, как именно это работает, лучший способ узнать это - протестировать и изменить его.

```
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
    result = 0
  return result
print("\n\nRecursion Example Results")
tri_recursion(7)
```

```
Recursion Example Results
1
3
6
10
15
21
28
```