# Scope, *args , **kargs

## Scope

You must have heared the term “scope” mentioned in most beginning programming classes. But it’s a topic that can cause lead to some errors if you don’t understand how it works. **A scope (of a variable) tells the interpreter when a name (or variable) is accessible**. In other words, the scope defines when and where you can use your variables, functions, etc. When you try to use something that isn’t in your current scope, you will usually get a `NameError`.

**Python has three different types of scopes:**

1. Local scope
2. Global scope
3. Non-local scope (which was added in Python 3)

Let’s learn how each of these scopes work.

### Local Scope

Local scope is the scope you will use the most in Python. All assignments are done in local scope by default. **When you create a variable in a code block, its “local” to that code block**. If you want something different, then you’ll need to set your variable to “global” or “nonlocal” (we will be looking at this later).

Let’s create a simple example that demonstrates local scope assignment.

```
def my_func(a, b):
    i = 2 # `i` will be locally accesible to function my_func

x = 10
my_func(1, 2)
print(i) # lets try to access "i" out of its scope
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-25-4382f850ecbf> in <module>
      4 x = 10
      5 my_func(1, 2)
----> 6 print(i) # lets try to access "i" out of its scope


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

The variable, `i`, is only defined inside the function (local to function `my_func`. When we try to access `i` out of its local scope, we will get a `NameError`. **Whenever you define a variable within a function, its scope lies ONLY within the function**. It is accessible from the point at which it is defined until the end of the function and exists for as long as the function is executing (Source). Which means its value cannot be changed or even accessed from outside the function.

## Global Scope

This is perhaps the easiest scope to understand. **Whenever a variable is defined outside any function, it becomes a global variable, and its scope is anywhere within the program**. Which means it can be used by any function. Notice, we have declared another variable `x` in the program's scope. Let's try accessing `x` inside the function `my_func`. Ideally, it should be available inside the function too. Lets try.

```
def my_func(a, b):
    print(x)

x = 10
my_func(1, 2)



10
```

Yes, we can access `x`.

### How does this work in background?

It would first try look for variable `x` in the function's local scope. If not found, it will look in the outer scope (here program's scope)

Thus, `print(x)` prints `10` as `x` is "globally" available across the whole program.

**What if we try to change its value "inside" the function? Will it make the changes to outside too?**

```
def my_func(a, b):
    x = 5
    print(x)

x = 10
my_func(1, 2)
print(x)
```

```
5
10
```

No, looks like the changes are not reflected outside.

**But why are changes not reflected outside? Aren't we changing the global variable `x` ?**

NO, while executing `x = 5` , the interpreter sees that there is no variable `x` in its local scope, and creates one. **The `x` inside of `my_fun` has a local function scope and overrides the `x` variable outside of the function.** Thus, the `print(x)` statement inside the function prints 5, but changes are not reflected to the global variable `x` which has value 10.

**But what if I want to upddate the value `x` inside the function and make it visible outside too?**

We need a way to tell Python to use the globally defined variable instead of locally creating one. For this, we simply type the keyword 'global', followed by the variable name. Let's attempt to use `global` to fix our code.

```
def my_func(a, b):
    global x
    x = 5
    print(x)

x = 10
my_func(1, 2)
print(x)
```

```
5
5
```

Yay, we can see that the changes made inside the function `x=5` are now reflected globally.

**Will the changes be reflected in the outer function of a nested function?**

Lets try to work on an interesting example of a nested function

```
def my_outer_func(a, b):
    x = 5
    print(x)
    def my_inner_func():
        x = 6
    my_inner_func()
    print(x)

x = 10
my_outer_func(1, 2)
print(x)
```

```
5
5
10
```

No, changes `x=6` made in the inner function are not reflected in the outer function

**What if we want to changes made in inner function to reflect in the outer one?**

Lets try using `global`

```
def my_outer_func(a, b):
    x = 5
    print(x)
    def my_inner_func():
        global x
        x = 6
    my_inner_func()
    print(x)

x = 10
my_outer_func(1, 2)
print(x)
```

```
5
5
6
```

Wait, this made changes to the “global” `x` . But didn’t make changes to the other `x` .

This demands need for having a keyword through which can access the variables one scope up.

This is where `nonlocal` keyword comes handy.

### Nonlocal scope

**Python 3** added a new keyword called `nonlocal`. **The `nonlocal` keyword adds a scope override to the inner scope.**

```
def my_outer_func(a, b):
    x = 5
    print(x)
    def my_inner_func():
        nonlocal x
        x = 6
    my_inner_func()
    print(x)

x = 10
my_outer_func(1, 2)
print(x)
```

```
5
6
10
```

Yay, this time it made changes to the variables `x` defined in the scope of outer variables.

### Summarizing

In this section, we learned that we can change how a variable is referenced using Python `global` and `nonlocal` keywords. We learned where you can use and why you might want to. We also learned about local scope.

## `*args` and `**kargs`

Functions are building blocks in Python. They take zero or more arguments and return a value.

**How can we create a function which can take variable number of arguments to functions?**

**Remember the `print()` function we saw in the very first lecture?**

```
print(2)

print(2, 3)

print(2, 3, "Hello")
```

```
2
2 3
2 3 Hello
```

As you can see, this in-built `print()` allows us to pass multiple number of arguments to it. We did not need to specify how many arguments can `print()` function take. There can be situations where we might not know before-hand how many arguments we need to pass in our functions. Luckily, Python is pretty flexible in terms of how arguments are passed to a function. **We can pass a variable number of arguments to a function using special symbols.**

But before we discuss how can we do it, lets first revise different kinds of arguments we can pass to a function

## There are 2 types of arguments we can pass to functions

1. **Non-Keyword (Positional) Arguments**

2. **Keyword Arguments**

**Now, What's the difference between Positional Argument and Keyword Argument?**

**Positional arguments** are declared by a **name only**. **Keyword arguments** are declared by a **name and a default value**. When a function is called, values for positional arguments must be given. Otherwise, we will get an error. If we do not specify the value for a keyword argument, it takes the default value.

```
def addition(a, b=2): #a is positional, b is keyword argument
   return a + b
print(addition(1))
```

```
3
```

```
def addition(a, b): #a and b are positional arguments
   return a + b
print(addition(1))



---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-30-fd0466385eed> in <module>
      1 def addition(a, b): #a and b are positional arguments
      2    return a + b
----> 3 print(addition(1))


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

**Python provides Special Symbols for passing variable number of Positional and Keyword arguments:**

1. `*args` (for non-keyword arguments)

2. `**kwargs` (for keyword Arguments)

The `*args` and `**kwargs` make it easier and cleaner to handle arguments.

We use the **"wildcard" or "*" notation** in the `*args` and `**kwargs` represent variable arguments.

**Do you know?**

The important parts are `"*"` and `"**"` .

You can use any word instead of `args` and `kwargs` . You could have also written `*var` and `**vars` .

But it is the common practice to use the words `args` and `kwargs` . Thus, there is no need for unnecessary adventures.

**Let's look at non-keyword and keyword arguments one by one now**

## *args

Consider the following function that sums up two numbers:

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

print(addition(3,4))



7
```

This function sums up only two numbers.

**What if we want a function that sums up three or four numbers?**

We may not even want to put a constraint on the number of arguments that passes to the function. In such cases, we can use `*args` as parameter.

**The special syntax `*args` in function definitions is used to pass a variable number of arguments to a function.**

It is used to pass a **non-key worded, variable-length argument list**.

What `*args` allows you to do is take in more arguments than the number of formal arguments that you previously defined. So when writing the function definition, you do not need to know how many arguments will be passed to your function.

With `*args`, any number of extra arguments can be stacked on to your current formal parameters (including zero extra arguments).

```
def addition(*args):
   result = 0
   for i in args:
      result += i
   return result
```

The parameters passed to the addition function are stored in a **tuple**. So, we can iterate over the args variable.

```
print(addition())

print(addition(1,4))

print(addition(1,7,3))
```

```
0
5
11
```

It is possible to use the `*args` and named variables together.

The following function prints the passed arguments accordingly.

```
def arg_printer(a, b, *args):
   print(f'a is {a}')
   print(f'b is {b}')
   print(f'args are {args}')

arg_printer(3, 4, 5, 8, 3)
```

```
a is 3
b is 4
args are (5, 8, 3)
```

The first two values are given to a and b. **The remaining values are stored in the args tuple**.

## **kwargs

The `**kwargs` collect all the keyword arguments that are not explicitly defined. Thus, it does the same operation as `*args` but for keyword arguments.

**The special syntax `**kwargs` in function definitions is used to pass a keyworded, variable-length argument list.**

We use the name kwargs with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them). As we saw earlier, a keyword argument is where you provide a name to the variable as you pass it into the function.

**You can think of the `kwargs` as being a dictionary that maps each keyword to the value that we pass alongside it.**

By default, `**kwargs` is an **empty dictionary**. Each undefined keyword argument is stored as a key-value pair in the `**kwargs` dictionary. That is why when we iterate over the `kwargs` there doesn't seem to be any order in which they are printed out.

**For example:**

```
def arg_printer(a, b, option=True, **kwargs):
   print(a, b)
   print(option)
   print(kwargs)

arg_printer(3, 4, param1=5, param2=6)
```

```
3 4
True
{'param1': 5, 'param2': 6}
```

**Lets take an example which uses both `*args` and `**kwargs`**

But `*args` must be put before `**kwargs`.

```
def arg_printer(a, b, *args, option=True, **kwargs):
   print(a, b)
   print(args)
   print(option)
   print(kwargs)
arg_printer(1, 4, 6, 5, param1=5, param2=6)
```

```
1 4
(6, 5)
True
{'param1': 5, 'param2': 6}
```

# Packing and Unpacking with `*args` and `**kwargs`

**We can pack and unpack variables using `*args` and `**kwargs`**

```
def arg_printer(*args):
   print(args)
```

If we pass a list to the function above, it will stored in **args tuple** as one **single element**.

```
lst = [1,4,5]
arg_printer(lst)
```

```
([1, 4, 5],)
```

However, if we put an asterisk before `lst` , the values in the list will be **unpacked and stored in args tuple separately**.

```
lst = [1,4,5]
arg_printer(*lst)
```

```
(1, 4, 5)
```

**We can pass multiple iterables to be unpacked together with single elements.**

All values will be stored in the args tuple.

```
lst = [1,4,5]
tpl = ('a','b',4)
arg_printer(*lst, *tpl, 5, 6)
```

```
(1, 4, 5, 'a', 'b', 4, 5, 6)
```

**We can do the packing and unpacking with keyword arguments as well**

```
def arg_printer(**kwargs):
   print(kwargs)
```

But the iterable that is passed as keyword arguments **must be a mapping such as a dictionary**.

```
dct = {'param1':5, 'param2':8}
arg_printer(**dct)
```

```
{'param1': 5, 'param2': 8}
```

If we also pass additional keyword arguments together with a dictionary, they will combined and stored in the kwargs dictionary.

```
dct = {'param1':5, 'param2':8}
arg_printer(param3=9, **dct)
```

```
{'param3': 9, 'param1': 5, 'param2': 8}
```