Digital Career Institute

Python Course - Functions

Digital Career Institute

Scopes

What is a scope

The areas of the program where an item *that has an identifier name* ***is recognized****.*

An item can be a variable, constant, function, etc.

The “problem”

```
global_var = "I'm global"

def my_function():
    local_var = "I'm local"
    print("Inside global", global_var)
    print("Inside local", local_var)

my_function()
print("Outside global", global_var)
print("Outside local", local_var)
```

\>

```
Inside global I'm global
Inside local I'm local
Outside global I'm global
Traceback (most recent call last):
  File "scope_1.py", line 9, in
<module>
    print("Outside local", local_var)
NameError: name 'local_var' is not
defined
```

Variables ***defined*** *on the main script are accessible from any function.*

But variables ***defined*** *inside a function are not accessible outside of it.*

Scopes & variable lifetime

Global scope	Local scopes
Variables, constants and functions defined on the main script can be accessed from any function in our code.	Variables, constants and functions defined inside a function can only be accessed from within the function and its childs.
The global scope is defined by the file where we write the code.	A local scope is defined by a function execution lifespan.
Everything in the global scope gets destroyed once the script stops running.	Everything in the local scope gets destroyed when the function finishes executing all its instructions.

Scopes & variable lifespan

DLI

Global scope

Local scopes

```
global_var = "I'm global"

def my_function():
    local_var = "I'm local"
    print("Inside global", global_var)
    print("Inside local", local_var)

my_function()
print("Outside global", global_var)
print("Outside local", local_var)
```

The name **global_var** gets destroyed once we exit the main script.

The name **local_var** gets destroyed once we exit the function.

‼ The name **local_var** does not exist here any more.

Scopes & passing variables to functions

DLI

What if we pass the global variable as an argument to the function?

```
global_var = "I'm global"

def my_function(global_var):
    print(global_var)
    global_var = "What am I now?"
    print(global_var)

my_function(global_var)

print(global_var)
```

>

```
I'm global
What am I now?
I'm global
```

The function did not change the global variable value!

Why did it not change?

Passing variables to functions

The standard ways for languages to pass variables to functions are:

By value

The interpreter sends the value to the function. The function knows nothing about the variable that was pointing to it, so the changes in the function will not affect the variable on the outer scope.

By reference

The function receives the variable itself (a reference to the value), thus changes in the variable's value will transcend the scope of the function and affect the global scope.

Passing variables to functions

```
>>> global_var = "I'm global"
>>> def my_function(global_var):
...     global_var = "What am I now?"
...
>>> my_function(global_var)
>>> print(global_var)
I'm global
```

In most cases *Python seems to treat the arguments passed with the* ***pass-by-value*** *approach.*

Why it does not change

First we define a global name called **global_var**.

The function creates a local name called **global_var**.

This is a new variable name that just happens to have the same name.

They are two different names pointing to the same object.

Passing variables to functions

```
>>> global_var = [1]
>>> def my_function(global_var):
...     global_var.append(2)
...
>>> my_function(global_var)
>>> print(global_var)
[1, 2]
```

__In some cases__ Python seems to treat the arguments passed with the __pass-by-reference__ approach.

But ...

First we define a global name called **global_var**. This time, it is a **list** with 1 element.

In the function we change the contents of the list in the local variable name **global_var**, adding one more element.

When we check the contents of the list in the global scope, it changed!

Why?

Passing variables to functions

Python passes variables to functions:

By assignment

The functions receive neither a value nor a reference to a value, but a **reference to an object**.

So what is an **object**?

Names, objects and values

Name	Object	Value
The identifier or tag we use to refer to the variable in our code. It refers to an *object*.	A container that holds a *value* (and other things).	A literal value.

global_var

=

The object is not seen in the code.

“I’m global"

Names, objects and values

```
>>> global_var = "I'm global"
>>> print( id(global_var) )
139825206344488
>>> def my_function(global_var):
...     print( id(global_var) )
...     global_var = "What am I now?"
...     print( id(global_var) )
...
>>> my_function(global_var)
139825206344488
139825180496112
>>> print( id(global_var) )
139825206344488
```

Objects are passed

The **id()** function returns an identifier of the actual object attached to a variable name.

And the object that the function receives has the exact same id as the one in the global scope.

But when we change its value, it is not affecting the global variable.

Why?

Mutability

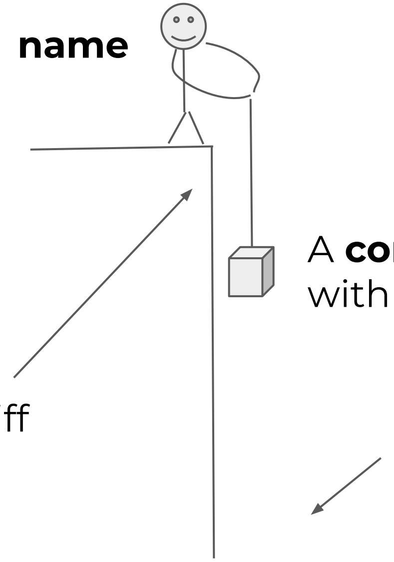
Python objects are immutable

- **Most objects** cannot change or mutate their value, they are **immutable**. Only a few are mutable: lists, dictionaries, sets and byte arrays.
- Therefore, **Python cannot change the value of an object**. Instead, **it creates a new object** and assigns it to the name.

Names, objects and mutability

Graphic explanation

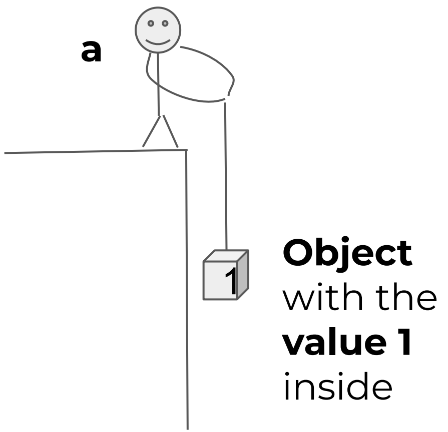
Consider this drawing as a variable **name** holding an **object** with a **value**...



Names, objects and mutability

Assignment

```
a = 1
```



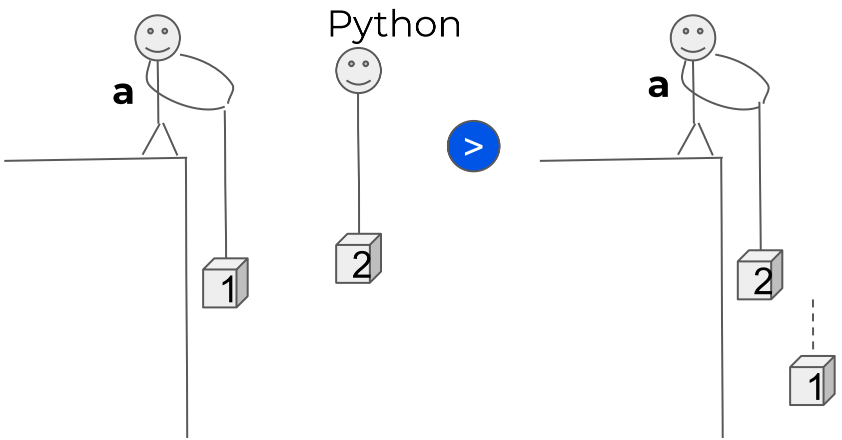
*The name **a** holds an object that has a value of **1***

Names, objects and mutability

DLI

Reassignment

```
a = a + 1
```



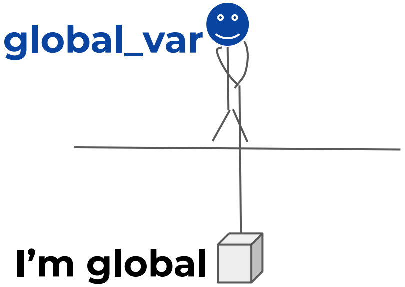
*The name **a** holds a brand new object that has a value of **2**.*

*It lets go the object with value **1**. And it falls.*

Names, objects and mutability

Global and local scopes

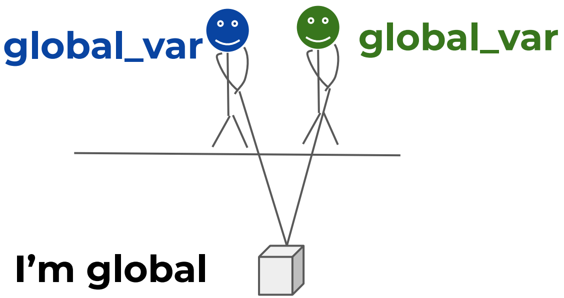
```
>>> global_var = "I'm global"
>>> def my_function(global_var):
...     global_var = "What am I now?"
...
>>> my_function(global_var)
>>> print(global_var)
I'm global
```



Names, objects and mutability

Global and local scopes

```
>>> global_var = "I’m global"
>>> def my_function(global_var):
...     global_var = "What am I now?"
...
>>> my_function(global_var)
>>> print(global_var)
I’m global
```

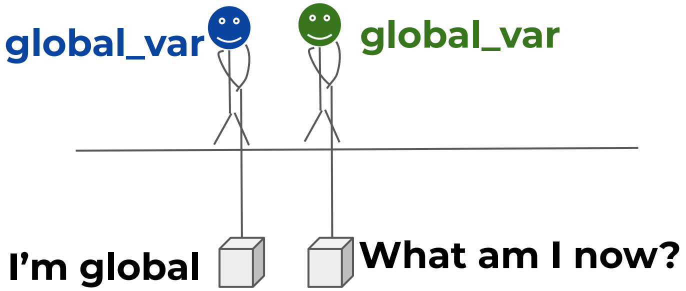


Both names are currently holding the same object, therefore the outcome of `id()` is the same.

Names, objects and mutability

Global and local scopes

```
>>> global_var = "I'm global"
>>> def my_function(global_var):
...     global_var = "What am I now?"
...
>>> my_function(global_var)
>>> print(global_var)
I'm global
```

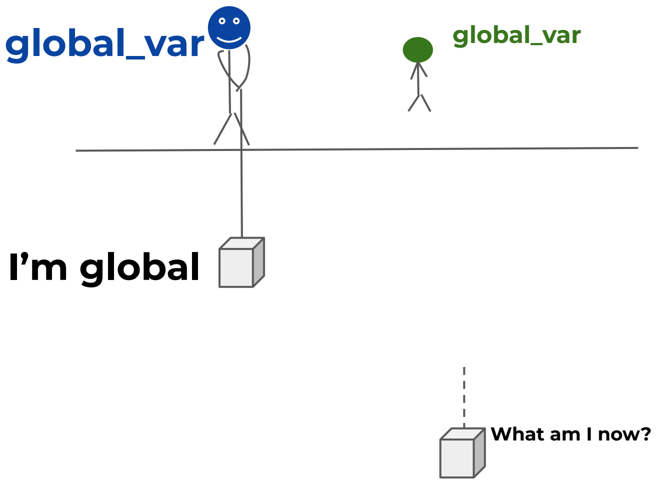


When a new value is assigned to **global_var**, it lets go the previous object and holds to the new one.

Names, objects and mutability

Global and local scopes

```
>>> global_var = "I'm global"
>>> def my_function(global_var):
...     global_var = "What am I now?"
...
>>> my_function(global_var)
>>> print(global_var)
I'm global
```



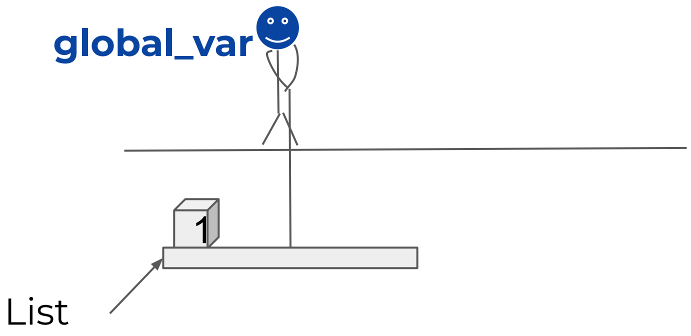
When the function exits, both the local name and object disappear and **global_var** is still holding to the original object.

Names, objects and mutability

Mutable types

Lists, dictionaries, sets and byte arrays.

```
>>> global_var = [1]
>>> def my_function(global_var):
...     global_var.append(2)
...
>>> my_function(global_var)
>>> print(global_var)
[1, 2]
```

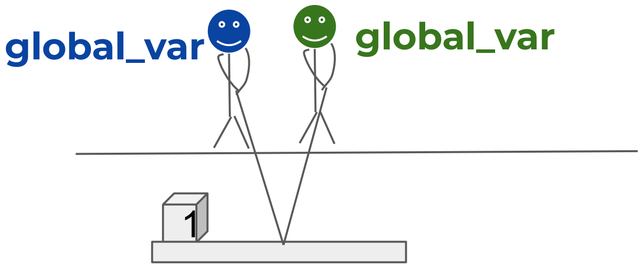


Names, objects and mutability

Mutable types

Lists, dictionaries, sets and byte arrays.

```
>>> global_var = [1]
>>> def my_function(global_var):
...     global_var.append(2)
...
>>> my_function(global_var)
>>> print(global_var)
[1, 2]
```

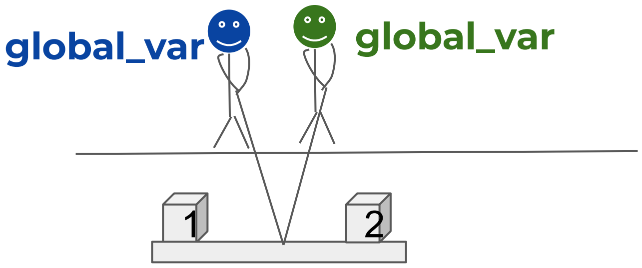


Names, objects and mutability

Mutable types

Lists, dictionaries, sets and byte arrays.

```
>>> global_var = [1]
>>> def my_function(global_var):
...     global_var.append(2)
...
>>> my_function(global_var)
>>> print(global_var)
[1, 2]
```



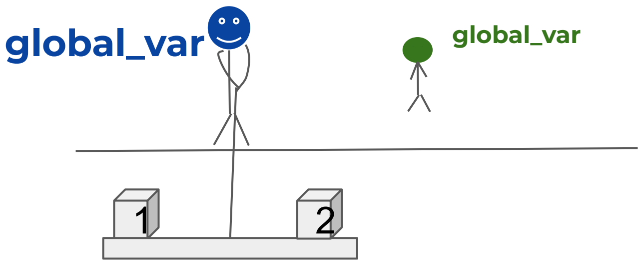
We add an object to the list, but **global_var** is still holding to the same object.

Names, objects and mutability

Mutable types

Lists, dictionaries, sets and byte arrays.

```
>>> global_var = [1]
>>> def my_function(global_var):
...     global_var.append(2)
...
>>> my_function(global_var)
>>> print(global_var)
[1, 2]
```



global_var goes away, but **global_var** is still holding to the same modified object.
Therefore, the changes remain in the global scope.

Names, objects and mutability

```
>>> def test(param1=[]):
...     print(id(param1))
...     param1.append(2)
...     print(param1)
...
>>> test()
10501056
[2]
>>> test()
10501056
[2, 2]
```

Default mutable arguments

The objects that store the default values **are created on function definition** and then are reused.

The first time, `param1` starts as an empty list.

The second time, since the object is **mutable** and is not created anew on every execution, it already contains an element from the previous call.

The default value of param1 changes when using the function.

Names, objects and mutability

```
>>> def test(param1=None):
...     if not param1:
...         param1 = []
...     param1.append(2)
...     print(param1)
...
>>> test()
[2]
>>> test()
[2]
```

Default mutable arguments

To assign a default value to a mutable argument we have to define it as `None` and do the assignment manually.

This way, Python creates a new object every time.

We learned ...

- What are the scopes.
- How they may affect each other.
- What is the lifespan of the variable names.
- The difference between variable names, objects and values.
- What are the standard ways of passing variables to functions.
- That Python passes arguments to the functions as objects.
- That most variable types are immutable in Python, except lists, dictionaries, sets and byte arrays.
- That Python actually creates a new object every time we change the value of an immutable type and this prevents the changes in the function from transcending the outer scope.

Calling functions

Calling functions

From within other functions

```
>>> def my_function(global_var):
...       global_var = "What am I now?"
...       my_other_function()
...       # more instructions
...
>>> def my_other_function():
...       print("I'm doing nothing")
...
>>> my_function(global_var)
I'm doing nothing
```

Only if they are defined in a scope that can be reached from there.

Calling functions

From within the same function

```
>>> def my_function(global_var):
...     my_function(global_var)
...     # more instructions
...
>>> my_function(global_var)
```

Functions can also call themselves and become **recursive**.

‼ If we don't define a way to leave the loop, this would be iterating forever and it will never reach the lines after this instruction.

Python has a recursion limit.

```
>>> from sys import getrecursionlimit
>>> getrecursionlimit()
1000
```

Recursive functions *need a* ***halting condition*** *that guarantees the function will exit when required.*

Calling functions

Recursive functions

```
>>> def sum(list):
...     if not list:   # Base case
...         return 0
...     else:   # Recursive cases
...         first = list.pop(0)
...         return first + sum(list)
...
>>> print( sum([1, 3, 5, 9]) )
18
```

*We do all the calls and when we reach the **base case** we make the calculations **in reverse order**.*

A recursive function needs a way to detect the halting condition, or **Base case**. The base case will never include a recursive call and will finish the stack of calls.

The rest of the cases will include the recursive call and will indicate the operation that will need to be performed <u>once all the calls are resolved</u>.

Calling functions

Recursive functions phases

```
>>> def sum(list):
...     if not list:   # Base case
...         return 0
...     else:   # Recursive cases
...         first = list.pop(0)
...         return first + sum(list)
...
>>> print( sum([1, 3, 5, 9]) )
18
```

1. **Winding phase**. Drill down until the base case:

```
- sum([1, 3, 5, 9])
    - return 1 + sum([3, 5, 9])
        - return 3 + sum([5, 9])
            - return 5 + sum([9])
                - return 9 + sum([])
                    - return 0
```

2. **Unwinding phase**. Process the result:

```
                    - return 0
                - return 9 + 0 = 9
            - return 5 + 9 = 14
        - return 3 + 14 = 17
    - return 1 + 17
- 18
```

Calling functions

Recursive functions

```
>>> def sum(list):
...     if not list:
...         return 0
...     else:
...         first = list.pop(0)
...         return first + sum(list)
...
>>> print( sum([1, 3, 5, 9]) )
18
```

```
>>> def sum(list):
...     total = 0
...     for number in list:
...         total = total + number
...     return total
...
>>> print( sum([1, 3, 5, 9]) )
18
```

One same task can sometimes be done recursively and iteratively.

Recursive functions are often more costly and take more time to execute.

Calling functions

Examples of recursive functions

Photo by Murad Swaleh on Unsplash

Tree data structures

- Directories & files
- Organization hierarchy
- Website pages
- DOM objects/XMLs

Network analysis

- Workflows
- Routing

In some other cases, recursive functions may be the best way to do a task.

Defining functions

Defining functions

Inside other functions

```
>>> def outer(bar):
...     def inner(foo):
...         print(“Inner”)
...     inner(bar[::-1])
...     print(“Outer”)
...
>>> outer(“hello”)
Inner
Outer
>>> inner(“hello”)
Traceback (most recent call last):
  File "using-functions-nested.py", line 8, in
<module>
    inner('hello')
NameError: name 'inner' is not defined
```

Functions can also be defined inside another function. They belong to its local scope.

In this case, `inner()` can only be called from within `outer()`.

The global scope has no access to `inner()`.

Defining functions

Nested functions

```
>>> def outer(bar):
...     def inner(foo):
...         print(“Inner bar”, bar)
...     inner(“Goodbye”)
...     print(“Outer foo”, foo)
...
>>> outer(“hello”)
Inner bar hello
Traceback (most recent call last):
  File "using-functions-nested.py", line 9, in
<module>
    outer('hello')
  File "using-functions-nested.py", line 7, in
outer
    print('Outer foo', foo)
NameError: name 'foo' is not defined
```

The `inner` function has access to variables in the global scope and also to those defined in the local scope of `outer()`.

The variable names defined in the local scope of `inner()` cannot be accessed by any instruction in neither the global scope or the scope of `outer()`.

Referring functions

Referring functions

Python functions are ***first-class citizens****.*

First-class citizens can be:

- assigned to variables
- stored in collections
- created and deleted dynamically
- passed as arguments

Referring functions

DLI

As variables

```
>>> def bar(bar):
...     print(bar)
...
>>> bar("hello")
hello
>>> my_alias = bar
>>> my_alias("goodbye")
goodbye
```

If we write the name of the function without the parenthesis we are not executing the function, but just **referring to it**.

`my_alias` dos not store the output of `bar`, it is just pointing to it and will behave likewise.

Referring functions

DLI

As input arguments

```
>>> def sum(a, b):
...     print(a + b)
...
>>> def operation(func, args):
...     func(*args)
...
>>> operation(sum, [2, 5])
7
```

Functions that take other functions as arguments are called ***higher order functions****.*

`sum()` adds the value of two numbers.

`operation()` takes a function and a list of arguments.

The “alias” of that function in the local scope of **`operation`** is **`func`**.

It unpacks the **`args`** list to pass it as individual arguments to the call to **`func()`**, which in this case is an alias of **`sum()`**.

Closures

As output arguments

```
>>> def make_printer(text):
...     def printer():
...         print(text)
...     return printer
...
>>> pr = make_printer("Hello World!")
>>> pr()
Hello World!
>>> pr()
Hello World!
```

A closure is a function that returns a function, who is using a variable from the first function.

- `make_printer()` defines and returns a function that is using a resource from the scope of `make_printer()`.
- It **returns a function** that we get in our global scope into a variable.
- We call the function and it prints the value of the variable named `text`.

The literal `"Hello World!"` is not stored anywhere other than `text` and `make_printer()` has finished the execution but the value of `text` did not disappear and we still can print it.

Lambda functions

Defining functions

DLI

Lambda functions

Lambda functions are special **anonymous functions** defined as **expressions**.

```
>>> add1 = lambda x: x + 1
>>> print(add1(1))
2
>>> # This is equivalent to
>>> def add1(x):
...     return x + 1
...
>>> print(add1(1))
2
```

They are anonymous, but they are still *first-class citizens* and can be assigned to a variable.

They have their own syntax.

- They use `lambda` instead of `def`.
- They do not have a name.
- They do not need parentheses.
- They do not use the `return` keyword.
- They can't use multiple lines.

Defining functions

DLI

Lambda functions

```
>>> add1 = lambda x: x + 1
```

Input parameters. There can be any number or 0.

Output parameter.

Defining functions

DLI

Lambda functions

```
>>> multiply = lambda x, y: x * y
>>> print(multiply(1, 0))
0
>>> print(multiply(3, 9))
27
>>> def printer(bar):
...     return lambda x: f"{bar}, {x}!"
...
>>> greet = printer("Hello")
>>> print(greet("John"))
Hello, John!
```

They can take any number of arguments.

They can take any number of arguments.

They help keep the code a little more concise.

Defining a lambda function and assigning it to a variable name right away is not recommended by the PEP-8 style guide.