

Digital Career Institute

Python Course - Functions



Goal of the Submodule

The goal of this submodule is to learn how to take advantage of functions to organize and optimize the code. By the end of this submodule, the learners will be able to understand:

- What is the purpose of functions
- How to create custom functions
- How functions relate to each other
- The different types of functions in Python
- The different types of arguments
- What is the scope of a variable
- How to create and use decorators, recursive functions and lambda functions.

Topics

- Introduction to functions
- Parts of a function
 - Header
 - Body
- Benefits of functions
- Packing and unpacking arguments
- Difference between local and global scopes
 - Lifespan of variables
 - Pass by reference, by value and by assignment
- Callables
- Recursivity
- Lambda functions
- Decorators

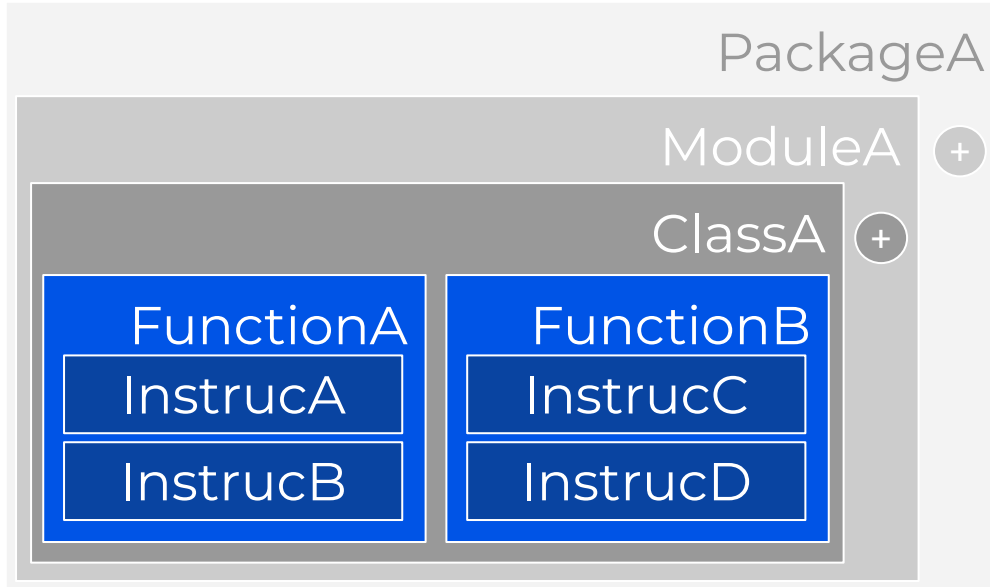
Term	Definition
Instruction	A single operation performed, often a line of code.
Call a function	To use/execute a function.

Introduction to Python Functions

What is a function?

A **named** and **encapsulated** set of related **instructions that work together** to perform a specific task.

What is a function?



It is the first and most basic way to organize and group meaningful sets of instructions under a callable name.

Why use functions

Reusability

The function will have a **name**. This means we can call it any time (it is a callable) and reuse the code, instead of repeating it.

Organizational

Instructions are **encapsulated**, giving us a more readable code, less bugs and reduced maintenance cost.

Semantics

Instructions that **work together** are clearly identified as such in the code. Improves readability.

There are many available **built-in functions**:

`print()`, `type()`, `str()`, `max()`, `min()`, ...

But we can also define our own **custom functions**.

How to create a custom function?

Parts of a function

```
>>> def hello_world():
...     """Print Hello World"""
...     print("Hello World!")
...
>>> hello_world()
Hello World!
```

Header

- **def** tells Python this is a function
- **hello_world** is the name we give it
- **()** indicates there are no arguments
- **:** indicates the end of the header

Body

A set of instructions to perform a task, in this case, printing *Hello World!*. The first line may contain a string that serves as documentation (**docstring**).

The body of the function must be indented.

Using a function

To call (use) a function we just have to write its name **with the opening and closing parenthesis**.

Parts of a function - Arguments

```
>>> def greet(name):
...     return f"Hello {name}!"
...
>>> greeting = greet("World")
>>> print(greeting)
Hello World!
```

Input argument

Our function requires an input argument containing a string with the name of the person to greet. *Inside the function we will have a variable called **name** with the content of the text sent to the function.*

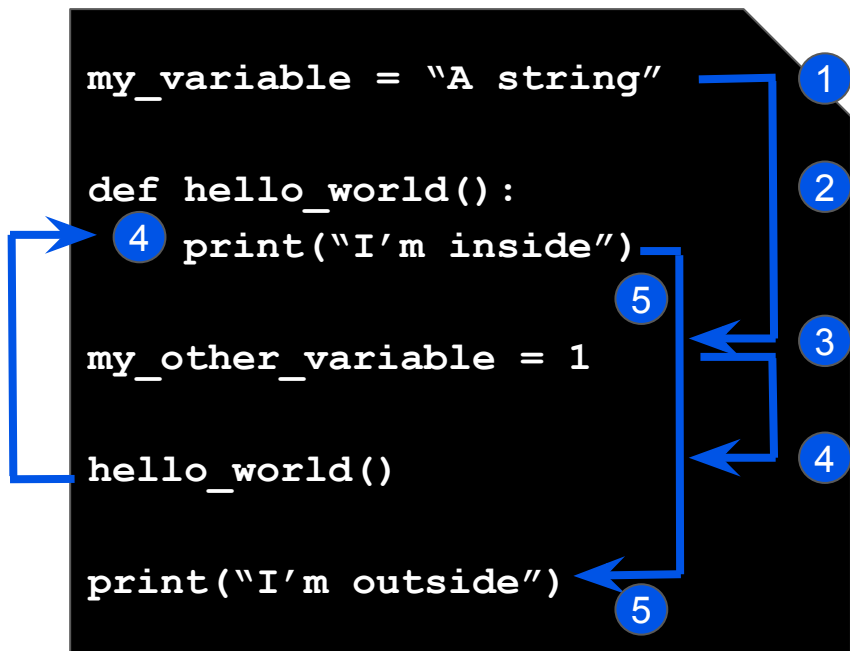
Using a function

To call (use) a function with arguments we just have to include them between the parentheses. *The output of the function can be stored in a variable name, in this case **greeting**.*

Output argument

Our function outputs a string containing the name of the person sent to the function. *To tell the interpreter what should the function return we use the reserved word **return**.*

Runtime execution of functions



What does the interpreter do?

1. Executes any instructions before the function definition.
2. Reads the function and stores it, but does not execute its instructions yet.
3. Executes any instructions after the function definition.
4. When we call a function, moves the pointer to the function and executes its contents.
5. When it finishes, goes back to the place it was before and keeps executing the remaining instructions.

Input arguments

Input arguments

```
>>> def full_name(first, last):
...     return f"{first} {last}!"
...
>>> friend = full_name("James", "Brown")
>>> print(friend)
James Brown
```

Positional arguments

This function defines two parameters.

When we call it, we define two values for those arguments, in the same order.

If we change the position of the arguments in the call, the output changes.

```
>>> friend = full_name("Brown", "James")
>>> print(friend)
Brown James
```

*These are called **positional arguments** because the variable name they are assigned to depends on their **position** in the call.*

Input arguments

```
>>> def full_name(first, last):
...     return f"{first} {last}!"
... 
```

```
>>> print(full_name(
...     first="James",
...     last="Brown"
... ))
James Brown
```

```
>>> print(full_name(
...     last="Brown",
...     first="James"
... ))
James Brown
```

Keyword arguments

This is the same function.

But now the arguments are named in the parentheses when calling the function.

If we change the position of the arguments in the call, the output is still the same.

*These are called **keyword arguments** because the variable name they are assigned to depends on the **keyword** in the call.*

Input arguments

```
>>> def full_name(first="John", last="Doe"):
...     return f"{first} {last}"
...
>>> print(full_name(first="James"))
James Doe
>>> print(full_name(last="Brown"))
John Brown
>>> print(full_name("James", "Brown"))
James Brown
>>> print(full_name("James"))
James Doe
```

Argument default values

We can define a default value in the header using the equal sign =.

If we don't specify either argument, it will use the default value.

We can still use positional arguments, but we can't get "John Brown" this way.

Input arguments

```
>>> def full_name(first, last="Doe"):
...     return f"{first} {last}"

>>> def full_name(first="John", last):
...     return f"{first} {last}"
...
File "<stdin>", line 1

SyntaxError: non-default argument follows
default argument
```

Combining with non-default

If we have parameters with and without default values, first we need to define the non-default arguments, and leave the default ones at the end.

If we define the default arguments first, the interpreter will produce an error.

Default arguments must be defined at the end in the function header.

List

- A list is a built-in data structure that represents an **ordered collection of items**.
- Lists are created using **square brackets** and items are **separated by commas**.

Example:

```
my_list = [1, 2, 3, "four", 5.0]
```

This creates a list with **5 items**, including integers, a string, and a floating-point number.

Accessing List Items

- We can access individual items in the list using their **index**, which **starts at 0**.
For example:

```
my_list = [1, 2, 3, "four", 5.0]
```

```
print(my_list[0])
```

Output: 1

```
print(my_list[3])
```

Output: "four"

Packing and unpacking arguments

```
>>> def full_name(*args):  
...     return f"{args[0]} {args[1]}!"  
...  
>>> friend = full_name("James", "Brown")  
>>> print(friend)  
James Brown
```

```
>>> def full_name(*args):  
...     return f"{args[0]} {args[1]}!"  
...  
>>> data = ["James", "Brown"]  
>>> friend = full_name(*data)  
>>> print(friend)  
James Brown
```

Positional arguments

We can automatically **pack** all incoming **positional arguments** into a **list** with the ***** operator.

This converts a series of elements into a list with those elements.

With the same operator ***** we can also do the opposite and **unpack** all elements in the list **data** to pass them on to the function.

- A dictionary is a built-in data structure that represents a **collection of key-value pairs**.
- Each **key** in a dictionary **maps to a corresponding value**, and you can use the key to look up the value.
- Dictionaries are created using **curly braces**, with key-value pairs **separated by colons**, and each key-value pair separated by commas.

Example:

```
my_dict = {"name": "Alice", "age": 30, "city": "New York"}
```

Accessing Dictionary Values

```
my_dict = {"name": "Alice", "age": 30, "city": "New York"}
```

- This creates a dictionary with **three key-value pairs**, where "name", "age", and "city" are the keys, and "Alice", 30, and "New York" are the corresponding values.
- We can access the value associated with a key using **square brackets**.

```
print(my_dict["name"])
```

Output: "Alice"

```
print(my_dict["age"])
```

Output: 30

Packing and unpacking arguments

```
>>> def full_name(**kwargs):  
...     first = kwargs["first"]  
...     last = kwargs["last"]  
...     return f"{first} {last}!"  
...  
>>> data = {  
...     "first": "James",  
...     "last": "Brown"  
... }  
>>> friend = full_name(**data)  
>>> print(friend)  
James Brown
```

Keyword arguments

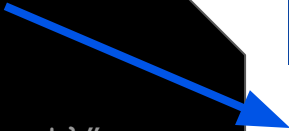
We can automatically **pack** all incoming **keyword arguments** into a **dictionary** with the ****** operator.

This converts a series of named elements into a dictionary with those elements.

With the same operator ****** we can also do the opposite and **unpack** all elements in the dictionary **data** to pass them on to the function as keyword arguments.

Packing and unpacking arguments

```
>>> def full_name(*args, **kwargs):  
...     first = kwargs["first"]  
...     last = kwargs["last"]  
...     return f"{args[0]} {first} {last}"  
...  
>>> data = {  
...     "first": "James",  
...     "last": "Brown"  
... }  
>>> friend = full_name("Mr", **data)  
>>> print(friend)  
Mr James Brown
```



Positional and Keyword

We can **pack** both at the same time in the function header by combining them.

The packed variables are often named args and kwargs, but they can take any name you like.

We learned ...

- How to create our own custom functions.
- That functions let us reuse code without repeating it.
- That they also help us organizing and keeping the code more readable and, thus maintainable.
- How do functions get executed.
- That functions may (or may not) have input and output arguments.
- How to define input positional arguments and input keyword arguments.
- How to define default values for each argument.

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

Variables, constants and functions defined on the main script can be accessed from any function in our code.

The global scope is defined by the file where we write the code.

Everything in the global scope gets destroyed once the script stops running.

Local scopes

Variables, constants and functions defined inside a function can only be accessed from within the function and its childs.

A local scope is defined by a function execution lifespan.

Everything in the local scope gets destroyed when the function finishes executing all its instructions.

Scopes & variable lifespan

Global scope

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

```
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)
```

Local scopes

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

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

The identifier or tag we use to refer to the variable in our code. It refers to an *object*.

Object

A container that holds a *value* (and other things).

Value

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.

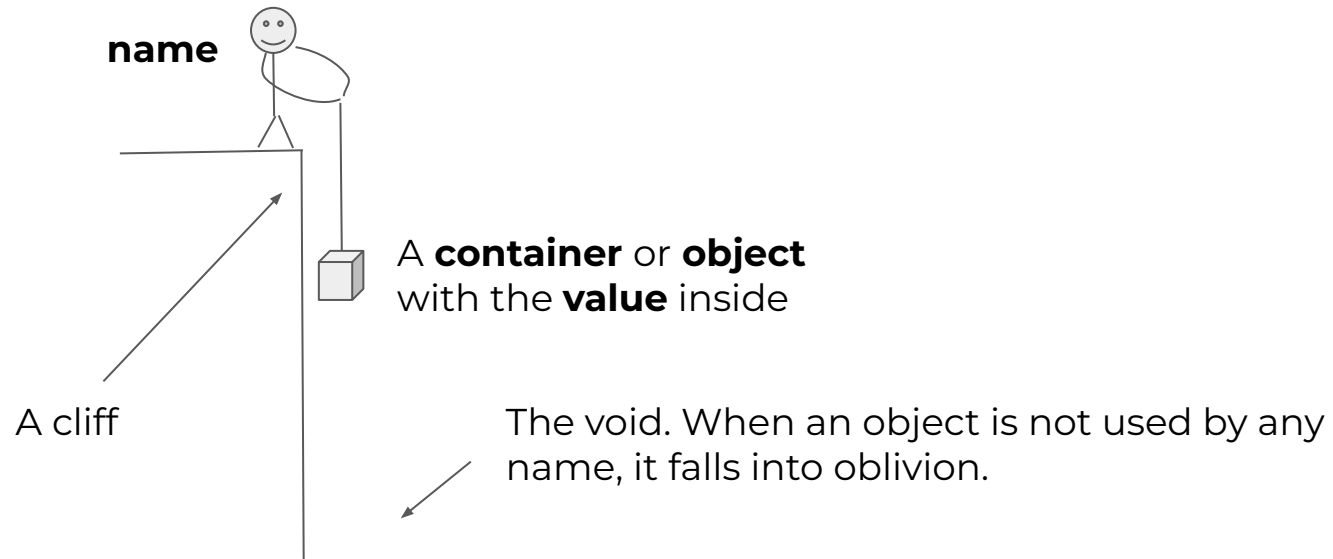
Why?

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.

Graphic explanation

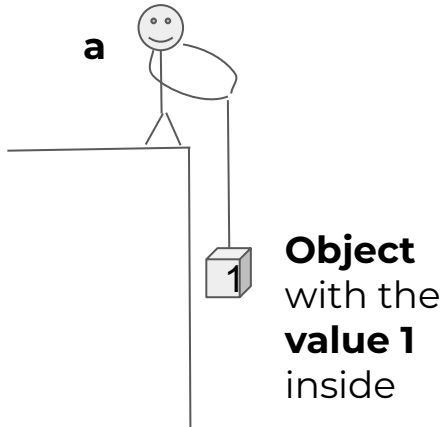
Consider this drawing as a variable **name** holding an **object** with a **value**...



Names, objects and mutability

Assignment

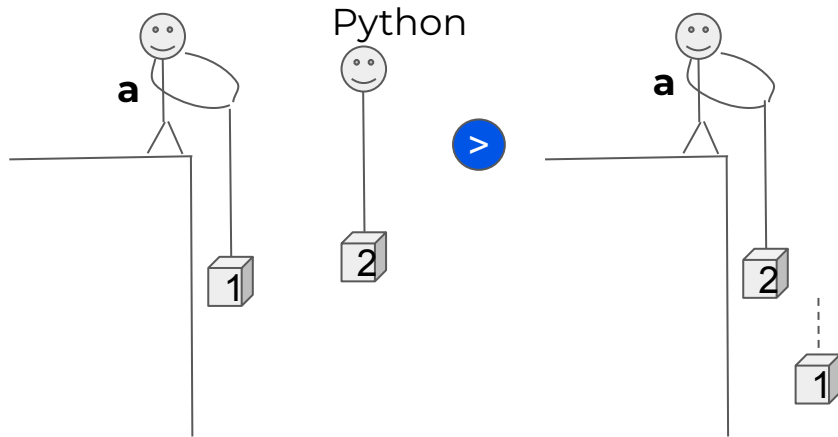
`a = 1`



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

Reassignment

`a = a + 1`



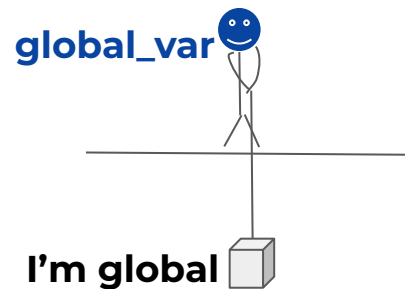
*The name **a** holds a brand new object that has a value of **2**.*

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

Names, objects and mutability

Global and local scopes

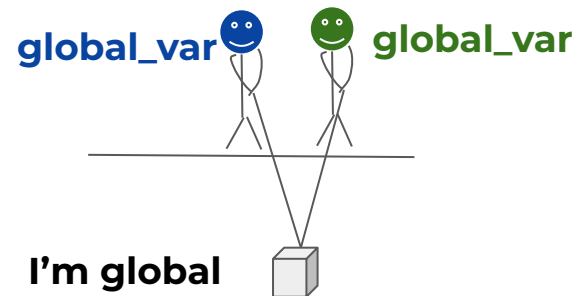
```
>>> global_var = "I'm global"
>>> def my_function(global_var):
...     global_var = "What am I now?"
...
>>> my_function(global_var)
>>> print(global_var)
I'm global
```



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
```

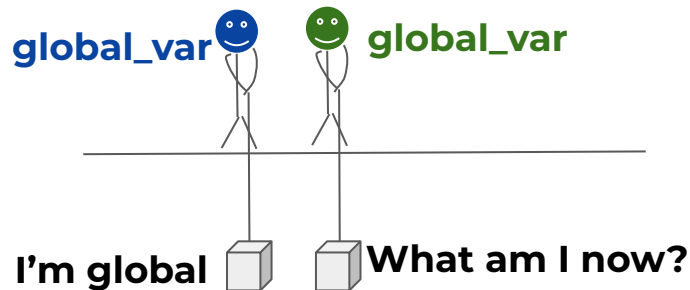


Both names are currently holding the same object, therefore the outcome of `id()` is the same.

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
```



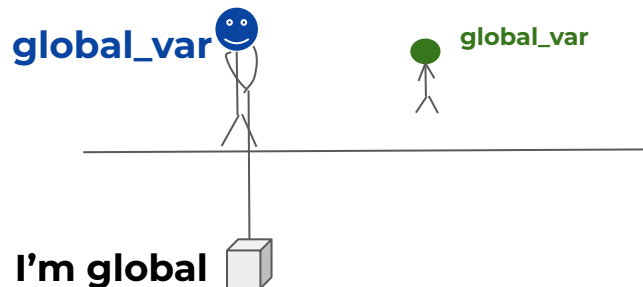
When a new value is assigned to **global_var**, it lets go the previous object and holds to the new one.

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
```

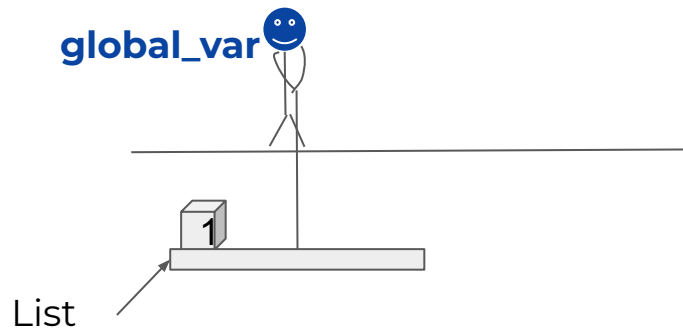
When the function exits, both the local name and object disappear and **global_var** is still holding to the original object.



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]
```

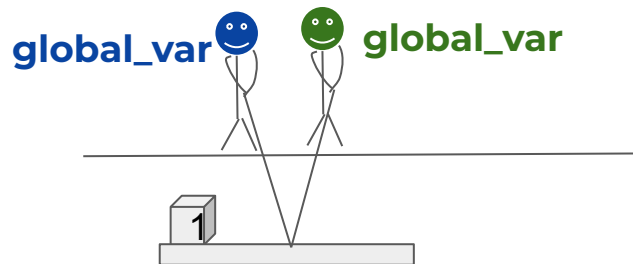


Names, objects and mutability

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]
```

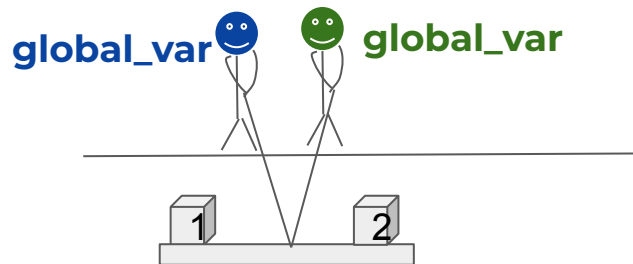


Names, objects and mutability

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]
```



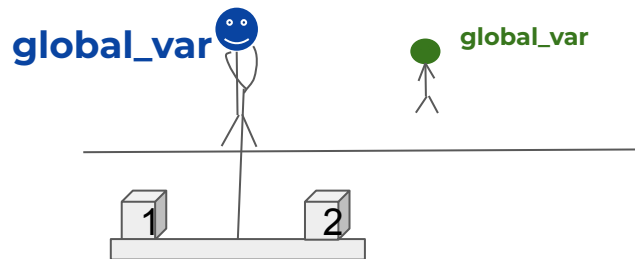
We add an object to the list, but **global_var** is still holding to the same object.

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.

```
>>> 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.

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.

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

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


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
```

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 once all the calls are resolved.

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

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
```



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
```

One same task can sometimes be done recursively and iteratively.

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

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

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

*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

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

Referring functions

As input arguments

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

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

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

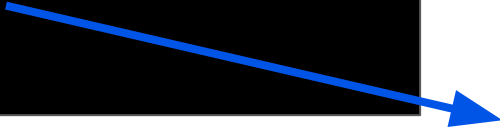
The “alias” of that function in the local scope of `operation` is `func`.

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()`.

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

As output arguments

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



- `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`.

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

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.

Decorators

They give additional functionality to our existing functions without modifying them.

A closure that **uses a function** as an argument and executes it (or not) in the enclosed function.

```
>>> def make_pretty(func):
...     def inner():
...         print("I'm decorated")
...         func()
...     return inner
...
>>> def ordinary():
...     print("Hello World!")
...
>>> ordinary()
Hello World!
>>> pretty = make_pretty(ordinary)
>>> pretty()
I'm decorated
Hello World!
```

`make_pretty()` is a decorator that prints some text before executing the function passed to it.

Decorators are a way to add a specific functionality to our functions. They are also, themselves, functions.

We can apply the decorator temporarily into a new variable name simply by calling it and passing the function we want to decorate.

Shortcut

```
>>> def make_uppercase(func):  
...     def inner():  
...         return func().upper()  
...     return inner  
...  
>>> @make_uppercase  
>>> def greeting():  
...     return "Hello World!"  
...  
>>> print( greeting() )  
HELLO WORLD!
```

Our functions can be permanently decorated by preceding their definitions with `@name_decorator`.

This is equivalent to doing `greeting = make_uppercase(greeting)` right after the definition of `greeting`.

Multiple decorators

```
>>> def make_count(func):
...     def inner():
...         return str(len(func()))
...     return inner
...
>>> def make_split(func):
...     def inner():
...         return func().split()
...     return inner
...
>>> @make_count
>>> @make_split
>>> def ordinary():
...     return "Hello World!"
...
>>> print(ordinary())
2
```

The decorators will be applied starting from the closest to the definition, in this case

@make_split.

We first split the sentence into a list of words and then count the resulting list. Our sentence has **2** words.

Inverting the decorators will count the characters in the string, convert them to text and split that text into a list, returning **['12']**.

Using function arguments

```
>>> def make_capitalized_arguments(func):  
...     def inner(*args):  
...         capitalized = [w.capitalize()  
...             for w in args]  
...         return func(*capitalized)  
...     return inner  
...  
>>> @make_capitalized_arguments  
>>> def greeting(first, last):  
...     return f"Hello, {first} {last}!"  
...  
>>> print( greeting("jAMES", "bROWN") )  
Hello, James Brown!
```

If the function to be decorated has parameters we must include them in our decorator's **inner** function and include them in the **func** call.

We can operate, or not, with those arguments.

Using decorator arguments

```
>>> def make_capitalized_arguments(*deco_args):  
...     def decorator(func):  
...         def inner(**kwargs):  
...             data = []  
...             for k, w in kwargs.items():  
...                 if k in deco_args:  
...                     data.append(w.capitalize())  
...                 else:  
...                     data.append(w)  
...             return func(*data)  
...         return inner  
...     return decorator  
...  
>>> @make_capitalized_arguments("first")  
>>> def greeting(first, last):  
...     return f"Hello, {first} {last}!"  
...  
>>> print( greeting(first="jAMES", last="bROWN") )  
Hello, James bROWN!
```

We need to wrap another function around to hold the arguments of the decorator.

And return the inner **decorator**.

We can now specify which of the function keyword arguments we want to capitalize.

Lambda functions

Lambda functions

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

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.

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

Lambda functions

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



Input parameters. There can be any number or 0.

Output parameter.

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.

Lambda functions

```
>>> def make_uppercase(func):  
...     return lambda x: func().upper()  
...  
>>> @make_uppercase  
>>> def greeting():  
...     return "Hello World!"  
...  
>>> print( greeting() )  
HELLO WORLD!
```

Often used in **closures** and **decorators** or as arguments to be passed to *higher-order functions*.

They help keep the code a little more concise and prevents the creation of a variable name that is not required.

We learned ...

- That we can call functions from inside another function as well as from the same function.
- That recursive functions need to identify a base case to halt the recursion.
- That they operate in two steps: drill-down and backwards calculations.
- That functions can be nested and contained within other functions.
- That functions are first-class citizens and can be assigned to variables and passed as arguments.
- That closures are functions that return a function and this one has access to the first function scope.
- That decorators are closures that take a function as an argument.
- How to work with function and decorator arguments.
- How to create anonymous functions by using lambda functions.

A large group of people, mostly young adults, are posing for a group photo in a room with a projector screen in the background. They are arranged in several rows, with some people sitting on the floor in the front. Many are making peace signs or other celebratory gestures. The image has a semi-transparent dark overlay.

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH