

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

Glossary

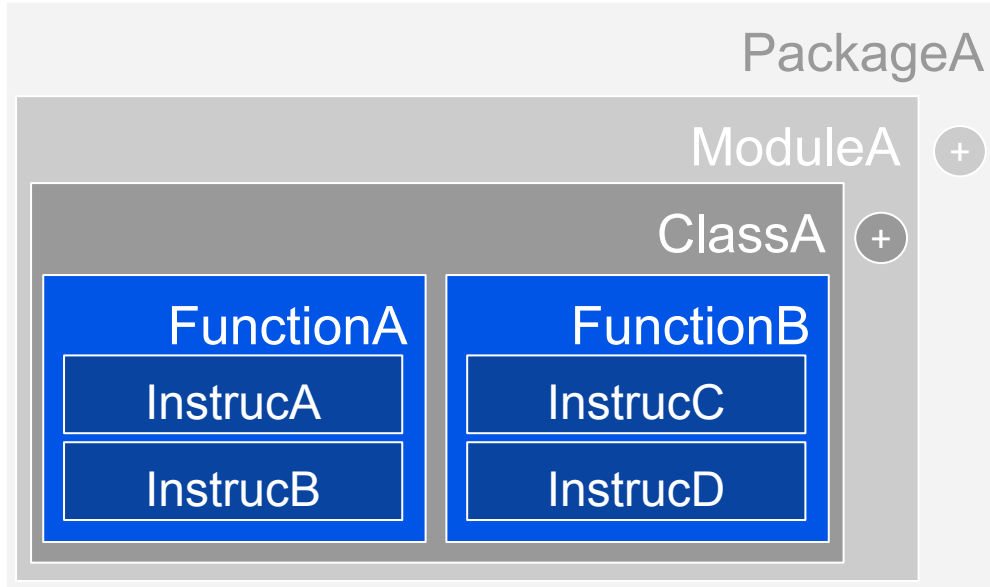
DLI

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

Introduction to Python Functions

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

DLI

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.*

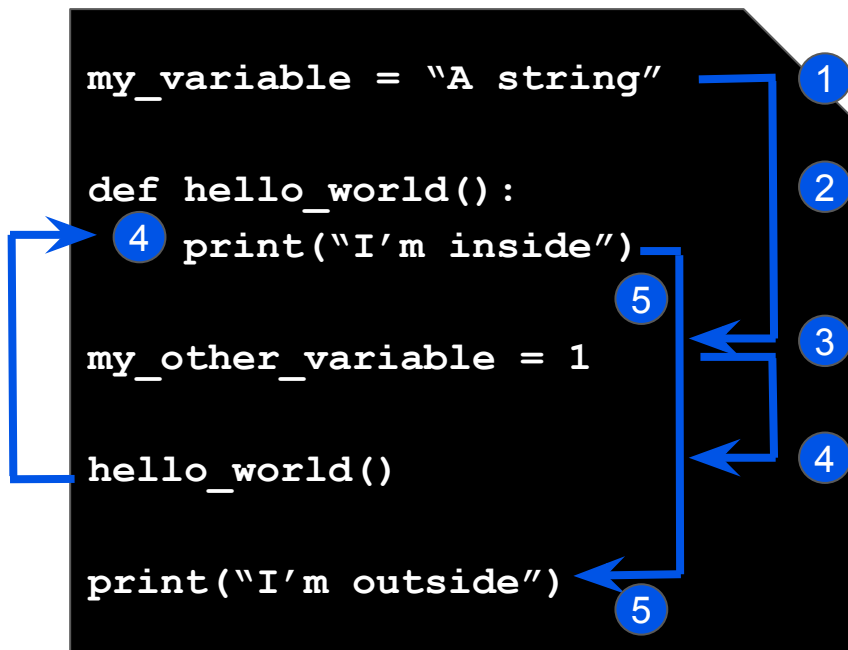
Using a function

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



Execution →

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

DLI

```
>>> 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

DLI

```
>>> 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

DLI

```
>>> 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.

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

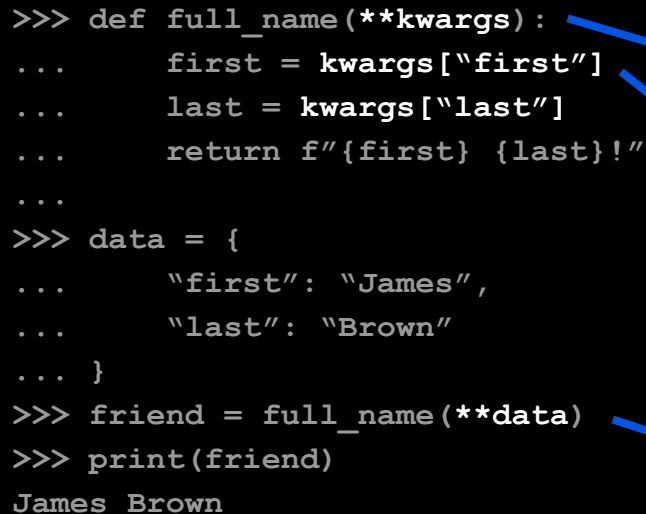
We can automatically **pack** all incoming **positional arguments** into a **list** with the ***** operator.

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.

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

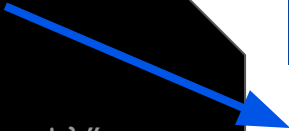
We can automatically **pack** all incoming **keyword arguments** into a **dictionary** with the ****** operator.

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.