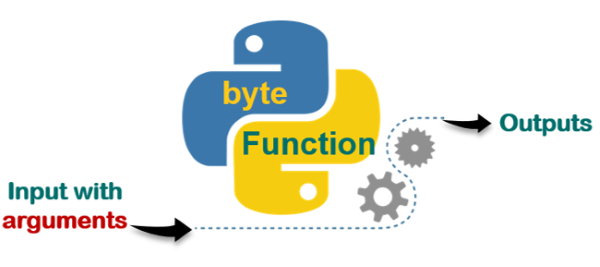
Acquaintance with Functions

Introduction

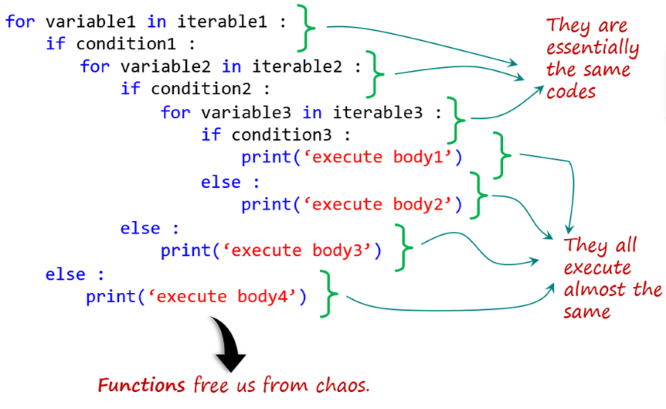
Basically, a **function** is a block of code that executes some logic for you, e.g. prints a text, deletes some data or square a number. In other words, a **function**is a piece of code that only runs when it is called.

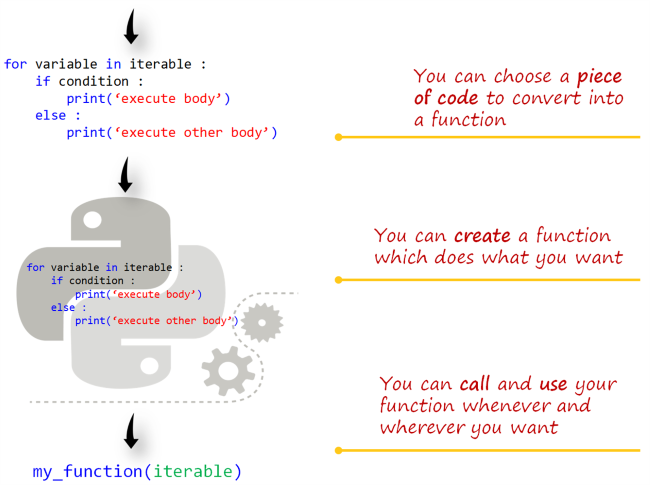
Functions in Python provide organized, reusable and modular code to perform a set of speciﬁc actions. Functions simplify the coding process, prevent redundant logic, and make the code easier to follow.

You can enter or input data, known as **arguments**, into a function and it returns/outputs something good that you want.



In some cases, you may need to create your own function. So that they help eliminate mess in your code because it saves you from unnecessary repetitions. You can **call**and use your function whenever and wherever you want. Follow a sample of flow diagrams of the functionalization process below. You can see below that we have several nested loops and conditional statements. As you noticed there are many repetitions of codes which look a bit messy and hard to understand.





Here we have tried to show you just the logical process of creating a function. As an example, my\_function is the name of the function that we have created. We have chosen a code block which consists of a **for-loop** and an **if-statement** to create a function. You will learn how we create and call a function with full syntaxes in the next lesson. Before **defining** (or creating) a function, let's take a look at how we **call** and use it.

**Q**: What is function in Python?  
**A**: A function is a block of code which is executed only when it is called. To define a Python function, the def keyword is used.

## Acquaintance with Functions

### Calling a Function

#### Calling**a Function Means**Using**It**

In the previous lesson, you have learned the basic philosophy of what a function is. In this lesson, we will examine how we **call** a function or what does 'calling a function' mean.

Reading a function is very easy in Python. For example; multiply(2, 5) or multiply(no1, no2). In this example, **multiply** is the name of the function, 2 and 5 are its **arguments** that we passed into the **parameters** which are the variables (no1 and no2). You can simply grasp that this function multiplies two numbers.

**💡Tips:**

* In fact, according to relevant [Python documents](https://docs.python.org/3/tutorial/controlflow.html#defining-functions), there is no significant difference between the definition and use of **parameter** and **argument** terms in Python. But, they are slightly different from each other.

Actually, **calling** a function means **using** it. When you need a function in your codes, you can simply use it. For example, if you want to multiply two numbers, you can just write the name of that function and the numbers (arguments) inside the parenthesis. By doing this, you have actually called this function. Look at the examples below :

input :

a = 3

b = 5

multiply(3, 5)

output :

15

input :

**a = 3**

b = 5

multiply(a, b)

output :

15

**⚠️Avoid:**

* Here, don't try to run these examples on the Playground otherwise it gives an error. Remember, we haven't defined this function yet.

#### **Calling print() Function**

You have already used print() function dozens of times since the beginning of this course. You have also learned the details of print() function in the **Python Basics** course.  You are now very familiar with it. Nevertheless, let's take a look at what is what.

In fact, what we do is solely writing its name and adding parentheses after it to call the print() function in your code. That's it!  
  
For example, let's consider this code : print("Say : I love you!") Here, in the example, you see the name of the function (print) followed by a sentence in parentheses. We can say that the sentence (Say: I love you!) that you passed into the 👉🏻() is an **argument**. We have a wide range of freedom of movement here. We may use the print() function with no argument besides we can also use it even with multiple arguments :

input :

print('Say: I love you!')

print()

print('me too', 2019)

output :

Say: I love you!

me too 2019

As you can see the outputs of the example above, we called that function (print()) three times. The first call printed a string, the second call printed an empty line and the third call printed two arguments which consist of one string and one integer data.  
At this point, we advise you to examine [this function](https://docs.python.org/3/library/functions.html#print) again.

Acquaintance with Functions

Built-in Functions

Frankly, our intention for this lesson is to inform you of the existence of the 'built-in' functions and to make you familiar with it. We are going to take a short tour of the 'built-in' functions. Besides, throughout the course, we will examine a significant part of the 'built-in' functions under some headings.

If you are considering a function which may do something that you want, it probably exists. You just need to be aware of its existence.

There are a range of functions and types built into the Python interpreter, so they are always usable. By the way, you don't have to worry about the term **interpreter**. We will talk about what it is in the next lessons.

In the latest version Python 3.9 the number of [**built-in functions**](https://docs.python.org/3/library/functions.html#built-in-functions) is 69. So far you have learned and used almost a dozen of these functions for various purposes. Such as : print(), int(), list(), input(), range().

These built-in functions are indeed very useful. They solve most of your needs without having to fall back on elsewhere.

It is a great benefit to have a quick look at the official Python documentation for the built-in functions mentioned below.

* Some of them return bool type according to the conditional algorithm in it. For example; all(*iterable*), any(*iterable*), and callable(*object*).
* Some of them help you convert data types into each other. For example; bool(), float(), int(), and str().
* Some others allow you to create and process the collection types. Such as : dict(), list(), tuple(), set(), len(), frozenset(), zip(), filter(*function*, *iterable*), and enumerate(*iterable*).
* Some others tackle numbers. Such as : max(), min(), sum(), and round().
* The others are built for special purposes. They do some complicated implementations. For example : map(*function*, *iterable*, *...*), eval(*expression*[, *globals*[, *locals*]]), sorted(*iterable*), open(), dir([*object*]), hash(), and help([*object*]).

In conclusion, we had a short tour of built-in functions and there is far more to explore, but don't ever give up!

**Q**: Explain Python functions.  
**A**: A function is a section of the program or a block of code that is written once and can be executed whenever required in the program. A function is a block of self-contained statements which has a valid name, parameters list, and body. Functions make programming more functional and modular to perform modular tasks. Python provides several built-in functions to complete tasks and also allows a user to create new functions as well. There are two types of functions:

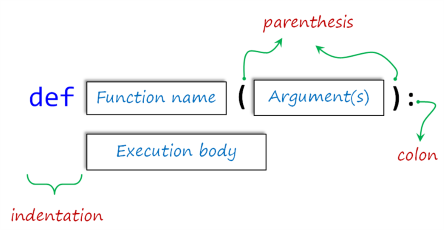
Built-In Functions: copy(), len(), count() are the some built-in functions.

User-defined Functions: Functions which are defined by a user known as user-defined functions.

Defining (Creating) a Function

Introduction

Sometimes when you writing a program, built-in functions maybe not enough for you. Or sometimes you may need to use a block of code in your program repeatedly. Then you can write your own function. That is called the **user-defined function**in Python. Let's take a brief look at how it works.



The keyword **def** introduces the name of the function. It must be followed by the function name and the parenthesized list of formal arguments. The statements that form the body of the function start at the next line and must be indented (leave four spaces).

## Defining (Creating) a Function

### Main Principles of 'Defining'

The basic **formula syntax** of user-defined function is :

def function\_name(arguments) :

execution body

We define a Python function using the **def** statement, providing a name (so as to call it later) for our function and specifying either an empty or multiple argument list within parentheses. The rules for naming variables also apply here. So they should be written in **lowercase with underscores between words**. Argument lists are optional, but the parentheses are NOT. A colon 👉🏻**:** follows the closing parenthesis and indicates the start of our functions execution body. The function’s codes (execution body) MUST be indented under the **def** statement.  
**⚠️Avoid:**

* Remember to put colon 👉🏻: just after the parentheses.
* Remember to leave four-space indentation at the beginning of the execution body lines.

Let's grasp the matter with an example :

def first\_function(argument\_1, argument\_2) :

print(argument\_1\*\*2 + argument\_2\*\*2)

This function, which we defined, gives the sum of the squares of arguments. Let's call and use it.

first\_function(2, 3) # here, the values (2 and 3) are

    allocated to the arguments

output :

13

In the example above, the values (2 and 3) are allocated to the arguments provided at the function call in parentheses.

**💡Tips:**

* When there is**no indentation**, it means that the **definition process** of the function must **end**.

And now, let's define the multiplying function (multiply(a, b)) that you have seen as an example in the previous lesson.

input :

def multiply(a, b) :

print(a \* b)

multiply(3, 5)

multiply(-1, 2.5)

multiply('amazing ', 3) # it's really amazing, right?

output :

15

-2.5

amazing amazing amazing

As we have already stated, we can define a function without using any arguments. Let's give an example by leaving the parentheses empty.

input :

def motto() :

print("Don't hesitate to reinvent yourself!")

motto() # it takes no argument

output :

Don't hesitate to reinvent yourself!

If you want to go deeper, [here](https://docs.python.org/3/tutorial/controlflow.html#defining-functions)what you looking for.

**Q**: How do we write a function in Python?  
**A**: We can create a Python function in the following manner.  
Step-1: to begin the function, start writing with the keyword def and then mention the function name.  
Step-2: We can now pass the arguments and enclose them using the parentheses. A colon, in the end, marks the end of the function header.  
Step-3: After pressing an enter, we can add the desired Python statements for execution.

## Defining (Creating) a Function

### Execution of a Function

The functions you have seen so far did not **return**any types or values but executed some actions. In order to use later the output and data types generated by the functions in our program flow, we need to define our function using the keyword return in addition to def. Let's see what happens in the following example :

input :

def multiply\_1(a, b) :

print(a \* b) # it prints something

def multiply\_2(a, b) :

return a \* b # returns any numeric data type value

multiply\_1(10, 5)

print(multiply\_2(10, 5))

output :

50

50

As you noticed, the outputs are the same. Then what is the difference? Well, the first function just prints some data that you passed into. The second one generates a numeric type value. If you check their types you will see :

input :

print(type(multiply\_1(10, 2)))

print(type(multiply\_2(10, 5)))

output :

20

<class 'NoneType'>

<class 'int'>

So, when we need it in our program, we can't use the result of the first function since it is **NoneType** data. But, the second one is **integer data** that we can use it in the future when we need it. Let's take a look at this subject using Python's best-known function.

input :

shadow\_var = print("It can't be assigned to any variable")

print(shadow\_var) # NoneType value can't be used

output :

It can't be assigned to any variable

None

In the example above, we can't assign the result of print() function to a variable.  
**💡Tips:**

* Note that, if there are more than one keyword return in a function, then the execution of that function will end after the first return.

**Q**: What is the return keyword used for in Python?  
**A**: The purpose of a function is to receive the inputs and return some output. The return is a Python statement which we can use in a function for sending a value back to its caller.

## The Matter of Arguments

### Arguments vs Parameters

We are aware that it may seem a bit difficult to understand clearly what **arguments**are. Likewise, the term **parameters**used in the same way as the arguments may seem also hard to grasp.

In fact, **arguments**are some kind of variable. So you can think of them as aliases for variables. This is exactly what we call the **parameters**. That is, the values ​​you assign to the parameters defined in a function are arguments.

If you look at the Python [documentation](https://docs.python.org/3/tutorial/controlflow.html?highlight=built%20function#more-on-defining-functions)on this topic, you will notice that the terms **parameters**and **arguments**are used almost the same way. But they are different things. You will better understand this topic through an example :

input :

def who(first, last) : # 'first' and 'last' are the parameters(or variables)

print('Your first name is :', first)

print('Your last name is :', last)

who('Guido', 'van Rossum') # 'Guido' and 'van Rossum' are the arguments

print()

who('Marry', 'Bold') # 'Marry' and 'Bold' are also the arguments

output :

Your first name is : Guido

Your last name is : van Rossum

Your first name is : Marry

Your last name is : Bold

As you can see, this function can be executed with different arguments. You can pass any two strings into the parameters.  
  
Since the function given in the example above takes two arguments, we need to pass exactly two arguments into it. If you passed only one argument into the parameter of the  who() function, it gives an error. Consider the following example :

input :

who('Joseph') # we passed only one argument into the function

output :

Traceback (most recent call last):

File "code.py", line 5, in <module>

who('Joseph')

TypeError: who() missing 1 required positional argument:

    'last'

**⚠️Avoid:**

* Be careful. The **order** of **parameters**in the function must match the **order** of **arguments** you passed into.

There is much more to talk about the arguments.

## The Matter of Arguments

### Correct Use of Arguments

We will stick to relevant [Python documents](https://docs.python.org/3/tutorial/controlflow.html#more-on-defining-functions) to make you understand this subject clearly. When calling a function, there are several ways to use arguments.

#### **Positional Arguments**

Actually, the arguments that you learned in the previous lesson is the positional one. Positions (sequence) of the arguments matter. When calling a function with **positional arguments**, they must be passed **in order from left to right**.

Take a look at these additional examples :

input :

def pos\_args(a, b):

print(a, 'is the first argument')

print(b, 'is the second argument')

pos\_args(3,4)

print()

pos\_args(4,3)

output :

3 is the first argument

4 is the second argument

4 is the first argument

3 is the second argument

input :

pos\_args('first','second')

print()

pos\_args('second', 'first')

output :

first is the first argument

second is the second argument

second is the first argument

first is the second argument

#### **Keyword Arguments**

If you do not want to allow the sequences/positions of the arguments to restrict you when you call a function, you can also call these arguments by keywords. Commonly and traditionally, **kwargs**is used as an abbreviation of **keyword arguments**.

The formula syntax is : kwargs=values.

Consider the following example : input :

def who(first, last) : # same structure as the previous one

print('Your first name is :', first)

print('Your last name is :', last)

who(first='Guido', last='van Rossum') # calling the function is different

# we used kwargs to pass the values into the function

output :

Your first name is : Guido

Your last name is : van Rossum

 Consider the example taken from the official Python doc. :

def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):

print("-- This parrot wouldn't", action, end=' ')

print("if you put", voltage, "volts through it.")

print("-- Lovely plumage, the", type)

print("-- It's", state, "!")

accepts one required argument (voltage) and three optional arguments (state, action, and type). This function can be called in any of the following ways:

parrot(1000) # 1 positional argument

parrot(voltage=1000) # 1 keyword argument

parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments

parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments

parrot('a million', 'bereft of life', 'jump') # 3 positional arguments

parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword

**💡Tips:**

* If you have noticed the positions of the parameters voltage and action, sequences or positions don't matter when using **keyword arguments**.

To examine how it works, you'd better try the functions above one by one on the Playground.

Considering the defined functions, all the following calls would be invalid:

parrot() # required argument missing

parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument

parrot(110, voltage=220) # duplicate value for the same argument

parrot(actor='John Cleese') # unknown keyword argument

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. actor is not a valid argument for the parrot function), and their order is not important. This also includes non-optional arguments (e.g. parrot(voltage=1000) is valid too). No argument may receive a value more than once. Here’s an example that fails due to this restriction:

input :

def function(a):

pass # actually, 'pass' does nothing. it just moves to the next line of code

function(0, a=0)

output :

Traceback (most recent call last):

File "code.py", line 4, in

function(0, a=0)

TypeError: function() got multiple values for argument 'a'

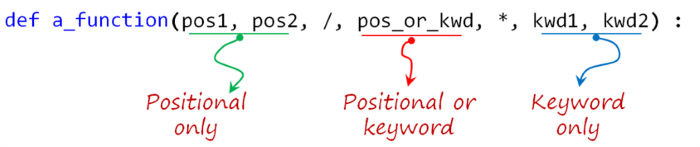
## The Matter of Arguments

### Special Arguments (Optional)

*--It'll be not included in the In-Class--*

According to the official related Python documents: By default, arguments may be passed to a Python function either by **position** or explicitly by **keyword**. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:



where 👉🏻/ and 👉🏻\* are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: *positional-only*, *positional-or-keyword*, and *keyword-only*. Keyword arguments are also referred to as *named arguments* or *named parameters.*

#### **Positional-or-Keyword Arguments**

If 👉🏻/ and 👉🏻\* are not present in the function definition, arguments may be passed to a function by position or by keyword. These are the arguments that we have already mentioned in the previous lesson.

#### **Positional-Only Arguments**

Looking at this in a bit more detail, it is possible to mark certain parameters as positional-only. If positional-only, the parameters’ order matters and the parameters cannot be passed by keyword. Positional-only parameters are placed before a 👉🏻/ (forward-slash). The 👉🏻/ is used to logically separate the positional-only parameters from the rest of the parameters. If there is no 👉🏻/ in the function definition, there are no positional-only parameters.

Parameters following the 👉🏻/ may be positional-or-keyword or keyword-only.

#### **Keyword-Only Arguments**

To mark parameters as keyword-only, indicating the parameters must be passed by keyword argument, place an 👉🏻\* in the arguments list just before the first keyword-only parameter.  
avoid:Do not confuse. When you use all kinds of arguments in a function definition, you have to pay attention the order of them.

## The Matter of Arguments

### Arbitrary Number of Arguments

#### **Default Arguments**

Before moving on to the subject of the **arbitrary number of arguments**, it is better to briefly mention the **default arguments**, which are one of the kinds of arguments.  
The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow.

When calling a function defined by **parameters with default values**, there is no obligation to pass any arguments into the function. Let's see how it works in an example :  
input :

def city(capital, continent='Europe'):

print(capital, 'in', continent)

city('Athens') # we don't have to pass any arguments into 'continent'

city('Ulaanbaatar', continent='Asia') # we can change the default value by kwargs

city('Cape Town', 'Africa') # we can change the default value by positional args.

output :

Athens in Europe

Ulaanbaatar in Asia

Cape Town in Africa

As you see in the example, new values can be assigned to parameters either by name or by position. This kind of usage simplify the calling of a function, otherwise defining the function with default values of parameters makes no sense.

#### **\*args and \*\*kwargs**

The arguments we have used in the functions so far limit us to a certain extent. If we don't define the default values for the arguments, we will always need to pass the exact numbers of the arguments to match the number of the parameters defined in the function. But, there will be some situations when you might want to pass an arbitrary number of arguments.  
  
Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur.   
The formula syntax is : \*args.

Normally, these variadic arguments will be last in the list of formal parameters because they scoop up all remaining input arguments that are passed into the function. Any formal parameters which occur after the \*args parameter are ‘keyword-only’ arguments, meaning that they can only be used as keywords rather than positional arguments.

For example, let's define a function which takes two kinds of fruit and prints them.

input :

def fruiterer(fruit1, fruit2) :

print('I want to get', fruit1, 'and', fruit2)

fruiterer('orange', 'banana')

output :

I want to get orange and banana

What if the user wants to get more than two kinds of fruit? Since we don't know how many kinds of fruit each user will enter, using '**arbitrary numbers of arguments**' is the most intelligent method. Consider the following example :

input :

def fruiterer(\*fruit) :

print('I want to get :')

for i in fruit :

print('-', i)

fruiterer('orange', 'banana', 'melon', 'ananas')

output :

I want to get :

- orange

- banana

- melon

- ananas

As you can see above, we passed a list of fruits (arguments) into one parameter (fruit). Isn't it very useful?

If you need to prefer to use arbitrary keyword arguments (\*\*kwargs), you can use it in the same way.

The formula syntax is : \*\*kwargs.

You can examine the following example :

input :

def animals(\*\*kwargs):

for key, value in kwargs.items():

print(value, "are", key)

animals(Carnivores="Lions", Omnivores="Bears", Herbivores="Deers", Nomnivores="Human")

output :

Lions are Carnivores

Bears are Omnivores

Deers are Herbivores

Human are Nomnivores

As you can see in the example, in this type of argument (\*\*kwargs), we can determine the number of the arguments and their assigned value pairs by ourselves. In the next call of this function, we can use different arguments both in number and value from the above argument pairs.

**💡Tips:** Traditionally, people in the world of computer programming use \*args for the arbitrary number of positional arguments and \*\*kwargs for the arbitrary number of keyword arguments.

When calling a function defined by multiple positional parameters, using \*arg syntax in parentheses, we can pass all arguments into the function with a single variable. Likewise; When calling a function defined by multiple keyword arguments, using \*\*kwargs syntax in parentheses, we can pass all arguments which are in a dictionary form into the function with a single variable. Carefully examine the following examples :input :

def brothers(bro1, bro2, bro3):

print('Here are the names of brothers :')

print(bro1, bro2, bro3, sep='\n')

family = ['tom', 'sue', 'tim']

brothers(\*family)

output :

Here are the names of brothers :

tom

sue

tim

input :

def gene(x, y): # defined by positional args

print(x, "belongs to Generation X")

print(y, "belongs to Generation Y")

dict\_gene = {'y' : "Marry", 'x' : "Fred"}

gene(\*\*dict\_gene) # we call the function by a single argument(variable)

output :

Fred belongs to Generation X

Marry belongs to Generation Y

input :

def gene(x='Solomon', y='David'): # defined by kwargs (default values assigned to x and y)

print(x, "belongs to Generation X")

print(y, "belongs to Generation Y")

dict\_gene = {'y' : "Marry", 'x' : "Fred"}

gene(\*\*dict\_gene)

output :

Fred belongs to Generation X

Marry belongs to Generation Y

Let's strengthen your knowledge of the 'arguments' thoroughly with more examples.

**Q**: What does this mean: \*args, \*\*kwargs? And why would we use it?  
**A**: We use \*args when we aren’t sure how many arguments are going to be passed to a function, or if we want to pass a stored list or tuple of arguments to a function.

\*\*kwargs is used when we don’t know how many keyword arguments will be passed to a function, or it can be used to pass the values of a dictionary as keyword arguments. The identifiers args and kwargs are a convention, you could also use \*bob and \*\*billy but that would not be wise.

### Recapitulation (Optional)

The use case will determine which parameters to use in the function definition:

def a\_function(pos1, pos2, /, pos\_or\_kwarg, \*, kwarg1, kwarg2, \*\*kwargs) :

As guidance:

* Use **positional-only** if you want the name of the parameters to not be available to the user. This is useful when parameter names have no real meaning, if you want to enforce the order of the arguments when the function is called or if you need to take some positional parameters and arbitrary keywords.
* Use **keyword-only** when names have meaning and the function definition is more understandable by being explicit with names or you want to prevent users relying on the position of the argument being passed.
* Use **arbitrary numbers of arguments** (\*args) when you can't determine how many arguments your function needs. **\*args**enables you to have interoperability with a list of positional arguments in your function.
* You can use **\*\*kwargs**, when you don't know the exact number of keyword arguments in your function. **\*\*kwargs** enables you to have interoperability with a dictionary of key-value pairs.
* The order of the **parameters** you use when defining the function is as important as the order of the **arguments** you pass into when you call the function.

**✏️Homework:**

* Do a research on the definition and the usage of the **API**s

## Scope of the Variables (Optional)

### Theoretical Definitions

*--It'll not be included in the In-Class--*

Let us give you some theoretical explanations about the term **namespace** and **scope** in Python. We will again stick to relevant [Python documents](https://docs.python.org/3/tutorial/classes.html?python-scopes-and-namespaces#python-scopes-and-namespaces) to make you understand this subject clearly. Besides, we think that examining these Python documents will be of great benefit to you.

#### What is Namespace?

A **namespace** is a system in which each **object** in Python has a *separate name*. An object could be a *method* or a *variable*. In other words, a **namespace** is a mapping from names to objects. Most namespaces are currently implemented as Python **dictionaries**, but that’s normally not noticeable in any way (except for performance), and it may change in the future.

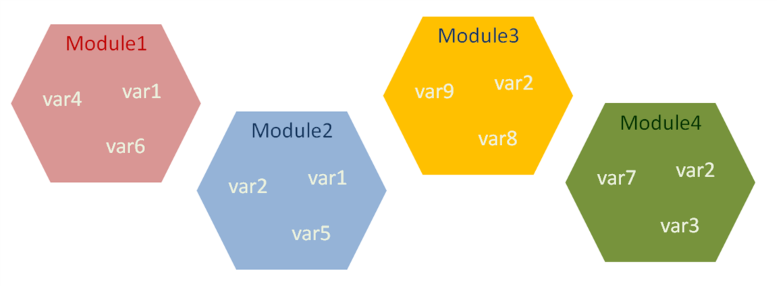
*For example*: imagine that you have two files named 'my\_python' on your computer both on **drive-C** and **drive-D**. You can easily access which file you want, by the file-path system. It can be easily understood which file is on which drive by looking at their file path.

In the program stream, the Python interpreter understands what specific method or variable one is trying to access in the code, depending on the namespaces.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up and is never deleted.

In the following figure, you can see some variables which have the same names are in the different *modules* (**namespaces**) at the same time. You can work with the variable that you want using this syntax : module.variable. Considering this figure, we can call var1 in Module2 like : module2.var1

You don't need to know what the module is and its definition for now. You will examine it in detail in the next lessons.



#### What is Scope?

A **scope** is a concept describes where or in whic **space** the **variables** are defined in the program stream. This concept has a significant place in programming. In other words, a **scope** is a *textual region*of a Python program where a *namespace*is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

The term **scope** is mostly related to nested functions, modules, and the main program flow in accordance with the use of variables. It describes the accessibility and the existence of a variable.

A **scope** defines the **hierarchical order** in which the names of the variables have to exist in order to match **names** with the **objects**(variables).

Now, let's put all these definitions into practice with a simple example :

input :

my\_var = 'outer variable'

def func\_var():

my\_var= 'inner variable'

print(my\_var)

func\_var()

print(my\_var)

output :

inner variable

outer variable

As you can see in the example, the name of the variable (my\_var) has been used both in the function (func\_var) and at the top of the main program stream. When you call the function (func\_var) or print directly the variable (my\_var), you were probably noticed that the same variable produces different outputs. This is because of the location (space) of that variable, that is, where or in which space it is defined in the program flow.

After learning what the concept of scope theoretically is, let's examine the **global** and **local** variables.

**Q**: What is the namespace in Python?  
**A**: The namespace is a fundamental idea to structure and organize the code that is more useful in large projects. A namespace is defined as a simple system to control the names in a program. It ensures that names are unique and won't lead to any conflict. Also, Python implements namespaces in the form of dictionaries and maintains name-to-object mapping where names act as keys and the objects as values.

## Scope of the Variables (Optional)

### Global and Local Variables

*--It'll not be included in the In-Class--*

When you define a variable in the Python program stream it is global or local, depending on in which space it is defined.

#### **Global Variable**

If the variable you define is at the highest level of a module, that variable becomes **global**. So you have the freedom to use this global **variable** in a block of code anywhere in your program.

Global variables allow us to make some interactions between functions. **For example,** suppose we store the credentials of a student who has applied for Clarusway in a **global variable**. Let's assume that we use this global variable many times in 3 **different functions** that we have defined regarding course activities. The **global variable** provides us with convenience when the credentials of the person change. Only when we rearrange the information in this global variable will our variables in all functions be rearranged.

#### **Local Variable**

The variables you have defined **in a function** body are **local**. The name of this variable is therefore **only valid** in the function body to which it is located.**Local variables** eliminate some of the confusion risks that global variables can cause.

Let's take a look at this example to grasp the difference between global and local variables:

input :

text = "I am the global one"

def global\_func():

print(text) # we can use 'text' in a function

# because it's a global variable

global\_func() # 'I am the global one' will be printed

print(text) # it can also be printed outside of the function

text = "The globals are valid everywhere "

global\_func() # we changed the value of 'text'

# 'The globals are valid everywhere' will be printed

def local\_func():

local\_text = "I am the local one"

print(local\_text) # local\_text is a local variable

local\_func() # 'I am the local one' will be printed as expected

print(local\_text) # NameError will be raised

# because we can't use local variable outside of its function

output :

I am the global one

I am the global one

The globals are valid everywhere

I am the local one

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

NameError: name 'local\_text' is not defined

In the above example, we have seen that a *global variable* can be accessed not only from the top-level of the module but also from the body of the function. On the other hand, a *local variable* is valid only in the function's body it is defined. So, it is accessible from inside the nearest scope level and can not be accessed from the outside.

**💡Tips:**

* You might have a question about where you will need to use these issues. But, if you are writing a relatively long algorithm, you will eventually need to work with the nested functions and modules.

**Q**: What are local variables and global variables in Python?  
**A**: Variables declared outside a function or in global space are called global variables. These variables can be accessed by any function in the program. Any variable declared inside a function is known as a local variable. This variable is present in the local space and not in the global space. When you try to access the local variable outside the function, it will give an error.

Scope of the Variables (Optional)

LEGB Ranking Rule

*--It'll not be included in the In-Class--*

When you call an **object** (*method or variable*), the *interpreter*looks for its name in the following order:

1. **Locals**. The space which is searched first, contains the local names defined in a function body.
2. **Enclosing**. The scopes of any enclosing functions, which are searched starting with the nearest enclosing scope (from inner to outer), contains non-local, but also non-global names.
3. **Globals**. It contains the current module’s global names. The variables defined at the top-level of its module.
4. **Built-in**. The outermost scope (searched last) is the namespace containing built-in names.

The order given above is known as **LEGB** Ranking Rule. Let's see how it works in an example :

input :

variable = "global"

def func\_outer():

variable = "enclosing outer local"

def func\_inner():

variable = "enclosing inner local"

def func\_local():

variable = "local"

print(variable)

func\_local()

func\_inner()

func\_outer() # prints 'local' defined in the innermost function

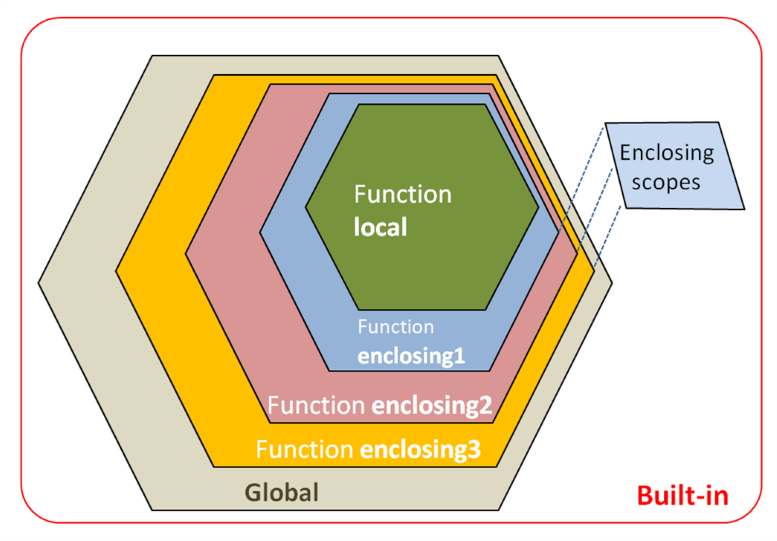
print(variable) # 'global' level variable holds its value

output :

local

global

In this example, during the execution of the code lines, the interpreter has to resolve the name '*variable*'.  The searching order of the variable names will be as follows : 'local' in func\_local, 'enclosing inner local' in func\_inner, 'enclosing outer local' in func\_outer, globals and built-in names. You can examine LEGB Rule in the following figure. 



## Scope of the Variables (Optional)

### 'global' and 'nonlocal'

*--It'll not be included in the In-Class--*

You know from the previous lesson that a variable defined in a function body becomes local. In some cases, we want to work with the variables defined as a global scope in the function body. Normally they are perceived *globally* and processed accordingly.

Or we may need to work with the nonlocal variables in the function body. The keywords global and nonlocal save us from these restrictions.

#### **Keyword 'global'**

You can not change the value assigned to a globally defined variable within a function. To do this we use the keyword global. If you examine the example below you will understand better.

input :

count = 1

def print\_global():

print(count)

print\_global()

def counter():

print(count)

count += 1 # we're trying to change its value

print() # just empty line

counter()

output :

1

Traceback (most recent call last):

File "code.py", line 11, in <module>

counter()

File "code.py", line 8, in counter

print(count)

UnboundLocalError: local variable 'count' referenced before assignment

As you can see in the example above, if you try to assign a value **contains local variable expressions** to a **global variable** within a function, *UnboundLocalError* will raise. We've tried to assign a value to the count variable using an expression that contains the count variable. This is because the interpreter can't find this *variable* in the **local scope**. So, let's use the keyword global to solve this problem.

input :

count = 1

def counter():

global count # we've changed its scope

print(count) # it's global anymore

count += 1

counter()

counter()

counter()

output :

1

2

3

The reason for the error in the previous program is that the variable (count) we tried to modify could not be found by the interpreter in the local scope. It's because we used a *global variable* in the *local scope*.

**Keyword 'nonlocal'**

On the other hand, you can use the keyword nonlocal to extend the scope of the *local variable* to an upper scope. Consider the examples of nonlocalization :

input :

def func\_enclosing1():

x = 'outer variable'

def func\_enclosing2():

x = 'inner variable'

print("inner:", x)

func\_enclosing2()

print("outer:", x)

func\_enclosing1()

output :

inner: inner variable

outer: outer variable

We will make the variable x nonlocal so we can use its inner-value in the outer function (scope). Let's see.

input :

def enclosing\_func1():

x = 'outer variable'

def enclosing\_func2():

nonlocal x # its inner-value can be used in the outer scope

x = 'inner variable'

print("inner:", x)

enclosing\_func2()

print("outer:", x)

enclosing\_func1()

output :

inner: inner variable

outer: inner variable

**💡Tips:**

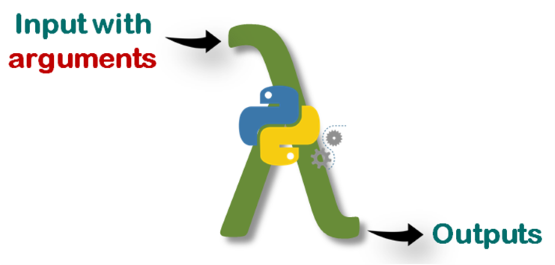
* Frankly, these keywords are not widely used in programming but are worth discussing.

## Lambda Functions

### Defining a Lambda Function

Another way to define functions in Python is lambda functions. Lambda functions are also called **anonymous**functions since they have no name. We use keyword lambda to define a function.

**The formula syntax is**: lambda parameters : expression



**Why we need lambda functions?**

If you need to use a one-time function, defining a lambda function is the best option. In some cases, you may need to define a function only once without having to use it later. For instance; let's square given numbers with a function. First, we're going to use def :

def square(x)

return x\*\*2

And now we'll define lambda function to do the same.

lambda x: x\*\*2

As you see, lambda is very simple and has a single line with a single expression. On the other hand, these two functions do exactly the same thing.

A lambda function can take multiple arguments separated by commas, but it **must** be defined with a *single expression*. This expression is evaluated and the result is returned

**⚠️Avoid:**

* Note that you do not need to use return statement in lambda functions.

Consider the following example of multiple arguments. Let's calculate the arithmetic mean of two numbers :

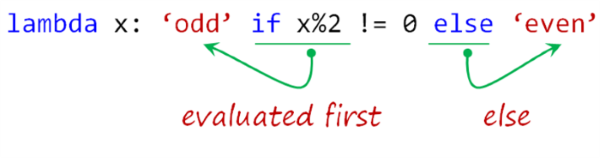
lambda x, y: (x+y)/2 # takes two numbers, returns the result

What if we need to use conditional statements within the lambda definition? Here how we do it :

lambda x: 'odd' if x % 2 != 0 else 'even'

**The formula syntax of conditional lambda statement is**:

lambda parameters : **first\_result** if conditional statement else **second\_result**



**⚠️Avoid:**

* Note that you can't use the [usual conditional statement](https://lms.clarusway.com/mod/lesson/view.php?id=18&pageid=57) with lambda definition.

If lambda is doing the same things with def then you might think of why and where do we use **lambda**? In the next lesson, we will try to find the answer to this question.

**Q**: What is a lambda function?  
**A**: A lambda function is an anonymous function (a function that does not have a name) in Python. To define anonymous functions, we use the ‘lambda’ keyword instead of the ‘def’ keyword, hence the name ‘lambda function’. Lambda functions can have any number of arguments but only one statement.

Lambda Functions

Uses of the Lambda Functions

So far you have seen the definition of lambda function and some of its features. Well, unlike def, where do we use lambda? If we need, how do we use the **lambda functions** in our code stream? Moreover, they don't even have names, so how can we call them? In this and the next lesson, we're going to try to find out the answer to these questions.

Lambda's most important advantages and uses are:

1. You can use it with its own syntax using ***parentheses***,
2. You can also *assign* it to a ***variable***,
3. You can use it in several ***built-in*** functions,
4. It can be useful inside **user-defined** functions (def).

Let's see how these work:

* **By enclosing the function in parentheses** :

(lambda x: x\*\*2)(2) # squares '2'

**The formula syntax is**: (lambda parameters : expression)(arguments)

Let's print the output :

input :

print((lambda x: x\*\*2)(2))

output :

4

Or you can use multiple arguments using the same syntax :input :

print((lambda x, y: (x+y)/2)(3, 5)) # takes two int, returns mean of them

output :

4.0

You can also assign the lambda statement in parentheses to a variable :

input :

average = (lambda x, y: (x+y)/2)(3, 5)

print(average)

output :

4.0

* **By assigning a function object to a variable**:

Alternatively, you can assign the lambda function definition to a variable then you can call it : input :

average = lambda x, y: (x+y)/2

print(average(3, 5)) # we call

output :

4.0

You will see how we use lambda definition within some **built-in** or **user-defined**functions in the next lesson.

**Q**: What Are The Principal Differences Between The Lambda And Def?  
- Def can hold multiple expressions while lambda is a uni-expression function.  
- Def generates a function and designates a name to call it later. Lambda forms a function object and returns it.  
- Def can have a return statement. Lambda can’t have return statements.   
- Lambda supports to get used inside a list and dictionary.

Lambda Functions

Lambda within Built-in (map()) Functions-1

When using some built-in functions we may need additional functions inside them. This can be done by using def, but when we do the same thing with lambda we save both time and additional lines of code and we make it clear to read.

* **Lambda within map() function :**

map() returns a list of the outputs after applying the given function to each element of a given iterable object such as list, tuple, etc.

**The basic formula syntax is**: map(function, iterable)

Let's square all the numbers in the list using map() and lambda. Consider this example :input :

iterable = [1, 2, 3, 4, 5]

map(lambda x:x\*\*2, iterable)

result = map(lambda x:x\*\*2, iterable)

print(type(result)) # it's a map type

print(list(result)) # we've converted it to list type to print

print(list(map(lambda x:x\*\*2, iterable))) # you can print directly

output :

<class 'map'>

[1, 4, 9, 16, 25]

[1, 4, 9, 16, 25]

☝ Discuss it in-class!

If you try to do the same thing using def, it is likely that the lines of code similar to the following occur. As you can see below, there are at least two additional lines of code. Moreover, we will not use the square function again because we only need to use it inside the map() function.

def square(n): # at least two additional lines of code

return n\*\*2

iterable = [1, 2, 3, 4, 5]

result = map(square, iterable)

print(list(result))

Now, let's try to give an example with multiple arguments in **lambda function** using map() :input :

letter1 = ['o', 's', 't', 't']

letter2 = ['n', 'i', 'e', 'w']

letter3 = ['e', 'x', 'n', 'o']

numbers = map(lambda x, y, z: x+y+z, letter1, letter2, letter3)

print(list(numbers))

output :

['one', 'six', 'ten', 'two']

In the above example, we have combined three strings using 👉🏻**+** operator in the lambda definition. 

**💡Tips :**

* Note that map() takes each element from iterable objects one by one and in order.

**Q**: What is map function in Python?  
**A**: map function executes the function given as the first argument on all the elements of the iterable given as the second argument. If the function given takes in more than 1 arguments, then many iterables are given.

Lambda Functions

Lambda within Built-in (filter()) Functions-2

* **Lambda within filter() function :**

filter() filters the given sequence (iterable objects) with the help of a function (lambda) that tests each element in the sequence to be true or not.

**The basic formula syntax is**: filter(function, sequence)

Let's grasp the subject with an example in which we'll filter the even numbers in a list.

input :

first\_ten = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

even = filter(lambda x:x%2==0, first\_ten)

print(type(even)) # it's 'filter' type,

# in order to print the result,

# we'd better convert it into the list type

print('Even numbers are :', list(even))

output :

<class 'filter'>

Even numbers are : [0, 2, 4, 6, 8]

**💡Tips :**

* Note that filter() filters each element in the iterable object, depending on whether the function's result is True or False.

This time, we'll filter the vowels from the first ten letters in the list.

input :

vowel\_list = ['a', 'e', 'i', 'o', 'u']

first\_ten = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

vowels = filter(lambda x: True if x in vowel\_list else False, first\_ten)

print('Vowels are :', list(vowels))

output :

Vowels are : ['a', 'e', 'i']

☝ Discuss it in-class!

We draw your attention to this issue that *lambda definition* we use in this example gives only True or False as a result.

Lambda Functions

Lambda within User-Defined Functions

* **Lambda within def :**

Using a lambda statement in a user-defined function provides us useful opportunities. We can define a group of functions that we may use later in our program flow. Take a look at the following example :input :

def modular\_function(n):

return lambda x: x \*\* n

power\_of\_2 = modular\_function(2) # first sub-function derived from def

power\_of\_3 = modular\_function(3) # second sub-function derived from def

power\_of\_4 = modular\_function(4) # third sub-function derived from def

print(power\_of\_2(2)) # 2 to the power of 2

print(power\_of\_3(2)) # 2 to the power of 3

print(power\_of\_4(2)) # 2 to the power of 4

output :

4

8

16

The modular\_function takes one argument, number ***n***, and returns a function that takes the power of any given number ***x*** by that ***n***.

This usage enabled us to use a function as flexible. Thanks to lambda, we could use a single def in different ways with the arguments we wanted. We've created three sub-functions derived from a single def. This is flexibility!

We can define a function with the same logic as the previous example that repeats the string passed into it.

input :

def repeater(n):

return lambda x: x \* n

repeat\_2\_times = repeater(2) # repeats 2 times

repeat\_3\_times = repeater(3) # repeats 3 times

repeat\_4\_times = repeater(4) # repeats 4 times

print(repeat\_2\_times('alex '))

print(repeat\_3\_times('lara '))

print(repeat\_4\_times('linda '))

output :

alex alex

lara lara lara

linda linda linda linda

## Loading Modules

### Fundamentals of Modules

We give some explanations about the subject by staying within the frame drawn by [related Python documents](https://docs.python.org/3/tutorial/modules.html?#modules). *Scripts* and *modules* have essentially identical structures in terms of creation and are the files with a **.py** extension, containing some Python codes, statements, operations, and functions.

**💡Tips :**  
The difference between these two terms :

* In fact, the difference between them depends on **how** and **for what purpose** you use this file with **.py** extension.

#### **What is Script?**

While working on your simple and short projects, you normally type and run Python codes directly into the interpreter. It would be a good option to save them in a file. And later when you need to run or work with them within a text editor you can easily access them using this file you created. So that every time you quit from the Python interpreter and enter again you won't lose your codes (functions and variables) typed before. This option will be vital to you, especially when working with programs with long blocks of code. The file you created consisting of codes, definitions and a list of operations that can be read, edited and interpreted in the future is known as **script**.

#### **What is Module?**

As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you’ve written in several programs without copying its definition into each program. To support this, Python has a way to put definitions in a file and use them in a **script** or in an interactive instance of the interpreter. Such a file is called a module.

You may need to use a function that you type in a program at another time and in another program stream. Or you may even need to reuse some code someone else wrote before. You can solve this issue by copying the code you have already written and pasting it into another program stream. But this does not bring a fundamental solution. Because as the functions and code blocks you want to work with increase, the structure and readability of your program will seriously deteriorate. As we said before, Python is a user-friendly programming language and you can be sure that there is a solution to handle this issue as well. And the solution is **modules**.

These simple files, usually with a **.py** extension and containing Python statements and definitions, are called **modules**. The ability to load or import one module from another in Python is a unique feature that significantly reduces our programming processes, and that is what makes the Module system really powerful.

**💡Tips :**

* If you open and use this file (with a **.py** extension) *directly*, that is **script**, and
* If you *load* (import)this file (with a **.py** extension) and call any function from it, that's a **module**this time.

How to Load a Module?

For loading a module, we use keyword import and the **name** of the module.

**The formula syntax is**: import module\_name

For instance, suppose we have a module called my\_module. Let's take a look at how we load it :

import my\_module # we've loaded my\_module

my\_module.my\_function() # we've called a function defined in my\_module

print(my\_module.my\_variable) # we've accessed a variable defined in my\_module

my\_module is the name of the module we imported. When loading a **module**you can also use an **abbreviated nickname** for modules by using a keyword as. Let's give a nickname to my\_module to load it :

import my\_module as mym # loads my\_module, we give a nickname to it

mym.my\_function() # we can use it the same way

print(mym.my\_variable)

In the example above, mym stands for the module my\_module. For instance, imagine that there is a file called my\_module.py named my\_module. And for being importable, this file should be placed in the same directory as the file you are working on.

**Initially**, the Python importing mechanism **searches for** a module in the **current directory**, **then** the **built-in modules**are inspected and an error will be raised if nothing is found. The module becomes available under **its name or alias** after importing and you can use the **dot notation** to access the functions and variables defined in it.

Importing a function or variable defined in a module is a very common and useful method. We use the keyword from to use this option. Let's see how it works :

from my\_module import my\_function # we've loaded only my\_function from my\_module

my\_function() # my\_function can be used directly now at the current module

**💡Tips:**

* Note that in the example above, we imported only my\_function. So if we try to use this syntax : my\_module.my\_function an error will be raise.

You can also use keyword as here the same way as well. Consider this example :

from my\_module import my\_function as mfnc # we've imported my\_function named mfnc

mfnc() # we use the my\_function's alias directly

It is traditionally best to type each import syntax in separate lines and put them all at the beginning of the current module. Let's see it in an example :

import module\_1

import module\_2

import module\_3

# The code stream of the current module starts here

**Q**: How to import modules in Python?  
**A**: Modules can be imported using the import keyword. You can import modules in three ways. Example:

import array # importing using the original module name

import array as arr # importing using an alias name

from array import \* # imports everything present in the array name

### Built-in Modules

Python comes with a huge library of standard modules many of which are built into the interpreter. These modules make our code more effective by providing useful functions and data structures. Another advantage of built-in modules which is invaluable to the user is that you can access these standard libraries from all operating systems in which Python is installed. Officially published standard library list (Python Module Index) can be found [here](https://docs.python.org/3/py-modindex.html).

Let's have a look at some of them if you want :

* **math** is one of the most known and used modules. This module allows us to work with mathematical functions.

input :

import math

print(dir(math)) # you can find out all names defined in this module

output :

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

**💡Tips:**

* The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings.

Let's use some of them in the examples :

input :

from math import pi, factorial, log10 # we'll use the functions directly

print(pi) # it also contains several arithmetic constants

print(factorial(4)) # gives the value of 4!

print(log10(1000)) # prints the common logarithm of 1000

output :

3.141592653589793

24

3.0

As you can see in the example above, we can import several functions by writing function names separated by commas at the same time.

**✏️Homework:**

* Define your own factorial function named factor using def.

def factor(x):

result = 1

for i in range(x):

result \*= (i+1)

return print(result)

Show the Answer

* **string** is used for common string operations.

input :

import string as stg # we've used alias for 'string' module

print(stg.punctuation) # prints all available punctuation marks

print(stg.digits) # prints all the digits

output :

!"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~

0123456789

* **datetime** is commonly used when working with date and time types.

input :

import datetime

print(datetime.date.today()) # prints today's date (yyyy-mm-dd)

print(datetime.datetime.now()) # prints the current time in microseconds

output :

2019-12-31

2019-12-31 15:03:31.303994

**💡Tips:**

* In the example above, we used the functions (now() and today()) defined under the date and datetime objects from the datetime module according to the **dot notation**.
* **random**is a module that contains functions that allow us to select randomly from various data types.

input :

from random import choice

city = ['Stockholm', 'Istanbul', 'Seul', 'Cape Town']

print(choice(city))

output :

Istanbul

**Q**: What are python modules? Name some commonly used built-in modules in Python.  
**A**: Python modules are files containing Python code. This code can either be functions classes or variables. A Python module is a .py file containing executable code. Some of the commonly used built-in modules are: os, sys, math, random, datetime.

- **Interview Q&A**

**Q**: How can you generate random numbers in Python?  
**A**: Random module is the standard module that is used to generate a random number. The method is defined as:

import random  
print(random.random())

The statement "random.random()" method return the floating point number that is in the range of (0, 1).  
The function generates random float numbers.

## Creating and Working with Modules

### Scripts & Modules Initialization

*Scripts* and *modules* have essentially identical structures in terms of creation and are the files with a **.py** extension, containing some Python codes, statements, operations, and functions.

As we stated before, the file with the **.py** extension, which contains all the codes, expressions, definitions and functions in a program you saved for later work, is actually the **script**. Then what is the difference between the *scripts* and the *modules*?

In fact, if you're using an advanced IDLE, such as **Jupyter Notebook**/**Lab** (which we are), all these issues about the *scripts* and the *modules* don't make much sense. So, these applications have a user-friendly menu on such issues.

**💡Tips:**

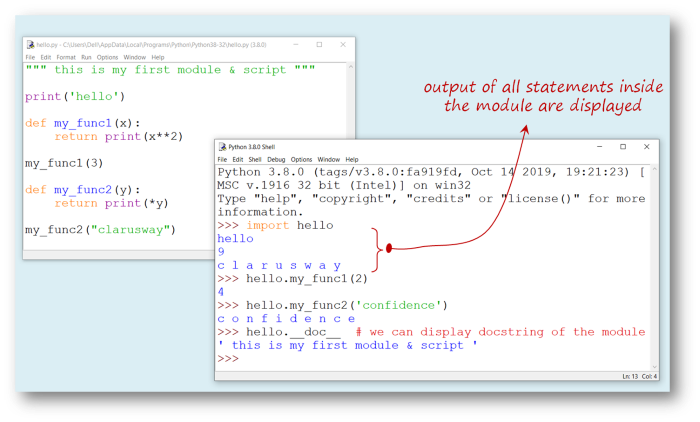
* When using Jupyter Lab / Notebook, you will almost always work with files with a **.ipynb** extension.

Let's take a look at an [**interactive lesson**](https://lms.clarusway.com/mod/hvp/view.php?id=910&forceview=1) to learn them in detail... ***Note :*** When you have finished this interactive lesson, ***close*** the opened window and ***return to this page*** to proceed.

### Working with the Modules (Optional)

#### **Acting of a Module as a Script**

In accordance with the previous lesson, during working with modules and scripts you should have noticed that when you load (Only when you first import) the module with the keyword import, the outputs of all the operations typed in it is displayed collectively. Take a look at this image again :



As you see, when you want to import this file as a *module*, it acts as a *script* for the **first** importing, which is undesirable. It is not normal for a module to generate output when imported. Then why it happens?

Well. As a Pythonic rule, when the file you created with **.py** extension is imported as a *module*, Python sets the specific variable **\_\_name\_\_** to the name of the *module*. But, if the file is run as a *script*, variable **\_\_name\_\_** is set to the string value of **"\_\_main\_\_"**. So, using this Pythonic rule, we can fix this issue.

#### **\_\_name\_\_, "\_\_main\_\_" Method**

If we collect the output-generating statements which are in our module under if \_\_name\_\_ == "\_\_main\_\_" : statement we will solve the problem. Let's do it and see what will happen :

hello.py :

""" this is my first module & script """

def my\_func1(x):

return print(x\*\*2)

def my\_func2(y):

return print(\*y)

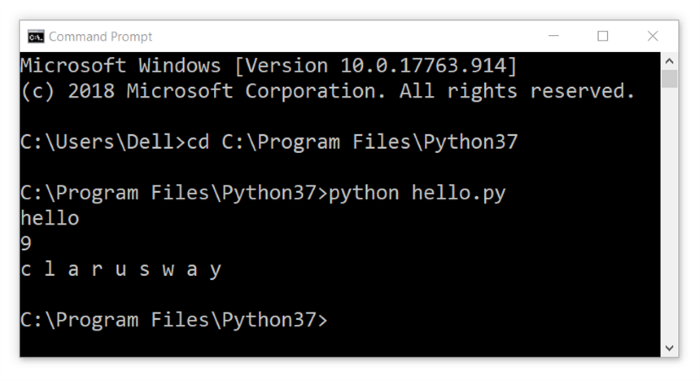
if \_\_name\_\_ == '\_\_main\_\_': # output-generating statements are here

print('hello')

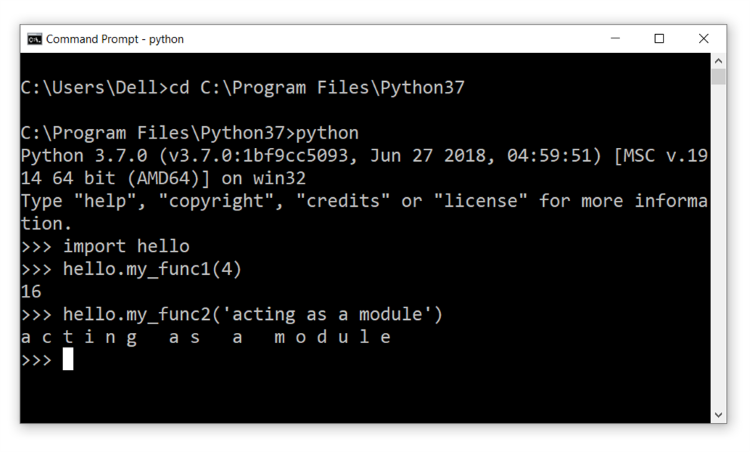
my\_func1(3)

my\_func2("clarusway")

Let's run it on the Command Prompt (console) as a *script*:



Let's run it on the Command Prompt (console) as a *module*:



We've solved this issue that when we import the module to use it, there is no undesired output anymore.

## Packages

### Package Initialization

According to the [official document](https://docs.python.org/3/tutorial/modules.html?#packages) of Python, packages are basically a way of structuring Python’s module namespace by using “**dotted module names**”.

Imagine designing a very large project with many modules. As the number of modules you work on and create in the project increases, it becomes more and more difficult to follow them systematically and use them regularly and eventually becomes inextricable. This is especially so if they have similar *names* or *functions*. In order to make the modules more systematically organized, we can use **packages**.

Examine the basic structure of the package system:

earth/ # Top-level package

\_\_init\_\_.py # Initialize the earth package

asia/ # Subpackage for file asia

\_\_init\_\_.py

japan.py

mongolia.py # A module under a subpackage

pakistan.py

taiwan.py

...

europe/ # Subpackage for file europe

\_\_init\_\_.py

germany.py # A module under a subpackage

england.py

turkey.py

kosovo.py

...

america/ # Subpackage for file america

\_\_init\_\_.py

canada.py

ustates.py

mexico.py

peru.py # A module under a subpackage

...

The *hierarchical model*of **dot notation** used to access and work with a module works as follows. The importing syntax which shows the entire hierarchy is so-called **absolute** importing. It looks like :

import earth.europe.kosovo # importing with naming package, subpackage and module

earth.europe.kosovo.a\_function() # we want to access a function defined in kosovo module

from earth import europe.kosovo # importing with naming subpackage and module

europe.kosovo.a\_function() # we want to access a function defined in kosovo module

from earth.europe import kosovo # importing without naming package and subpackage

kosovo.a\_function() # we want to access a function defined in kosovo module

from earth.europe.kosovo import a\_function # importing without any naming

a\_function() # we use directly the function's name

**💡Tips:**

* Which style you should use depends on your needs. But the key point is readability!

| **modules** |
| --- |
| Structure of a Package |

**⚠️Don't forget:**

* For Python to recognize the folders you created as packages / subpackages, you need to create an empty file named \_\_init\_\_.py in both the package and subpackage folders.
* They are usually empty, but may contain some initialization code of the package.

When you need to reorganize your modules with the packaging system, you need to create package/subpackage folders in the directory where Python is installed. Of course, keep in mind that you have to put a file named **\_\_init\_\_.py** in the folders you will create.

**💡Tips:**

* **Note that** when using from package import item, the item can be either a *submodule (or subpackage)* of the *package*, or some other name defined in the package, like a function, class or variable.
* The import statement **first** tests whether the item is defined in the package; if not, it assumes it is a **module** and attempts to load it. If it fails to find it, an *ImportError* exception is raised.

Let's take a look at an [**interactive lesson**](https://lms.clarusway.com/mod/hvp/view.php?id=971) to learn them in detail... ***Note :*** When you have finished this interactive lesson, ***close*** the opened window and ***return to this page*** to proceed.

**Q**: What is module and package in Python?  
**A**: In Python, module is the way to structure the program. Each Python program file is a module, which imports other modules like objects and attributes. The folder of Python program is a package of modules. A package can have modules or subfolders.

### Importing \* From a Package (Optional)

In connection with this subject, we can say that there is another and practical way to import packages / subpackages / modules. Consider this method : from package.subpackage import \* . This syntax allows us to import all modules that the subpackage has.

Although this way of importing all content is not a highly preferred method, when you add \_\_all\_\_ = [list of content] statement into the \_\_init\_\_.py file, you can display the subpackages/modules under package / subpackage in a list without any mess.

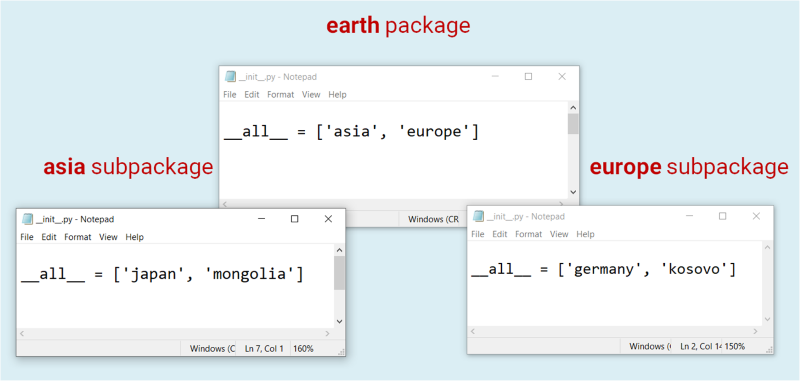
Well, how we do that? Take a look at the following images and explanations :

In order to display the list of *subpackages/modules* inside the *package*, we must add the following syntax into the **\_\_init\_\_.py** files:

For the **package**: \_\_all\_\_ = ['subpackage1', 'subpackage2']

For the **subpackage**: \_\_all\_\_ = ['module1', 'module2']

Let's see the creation of these \_\_init\_\_.py files below :



This image shows that when you type a **dot** immediately after the package name, the list of subpackages appears on the IDLE.



And the next image shows that when you type a **dot** immediately after the subpackage name, the list of modules appears on the IDLE.



Working with Inter & Intra-Packages/Subpackages

You can feel the power of Python and its user-friendliness in working with intra&inter-packages/subpackages. If you need to work with a module defined in another subpackage, you can simply import it into the current module file. And of course, you can either import and use another *module*from the current *subpackage* you're working on.

For example: while you're working on earth.europe.germany in **germany.py** file, you can import and use *module*'**japan**' from *subpackage* **asia** by using a similar syntax : **from** earth.asia **import** japan. We've stated before that this kind of syntax is **so-called absolute** importing. But there is another simple way to do the same thing. This style of syntax is so-called **relative imports**.

You can write **relative imports**, with the from module import name form of the *import statement*. These imports use leading **dots** to indicate the *current* and *parent packages* involved in the relative import. Let's see how it works for the *module***japan**:

from . import mongolia # one dot means addressing to a current package/subpackage

from .. import europe # two dots mean addressing to a parent package/subpackage

from ..europe import kosovo # subpackage name comes immediately after two dots

The modified **germany.py** file is below. We're importing and using module **japan**inside the **germany.py** file :

""" this is my first module & script """

def my\_func1(x):

return print(x\*\*2)

def my\_func2(y):

return print(\*y)

from ..asia import japan # we've added importing syntax using two dots

japan.my\_func1(6) # and the name of parent subpackage (asia)

if \_\_name\_\_ == '\_\_main\_\_':

print('hello')

my\_func1(3)

my\_func2("clarusway")

**⚠️Avoid:**

* Using the syntax from name import **\*** is usually considered bad practice.
* While working on Python codes, keep this in your mind : **Readability counts**!

When we import the module **germany,**we can see the result like:

input :

import earth.europe.germany

output :

36

**💡Tips:**

* **Absolute imports** syntax are preferred, as they are usually more readable.
* When dealing with very complex and sophisticated packages, it is preferable to use **relative imports**syntax, since using absolute imports will result in unnecessary redundancy.

### pip - The Package Manager for Python

#### **What is pip?**

After you develop yourself in the Python programming, you will start to take part in more professional projects. Over time you will see that you will have to benefit from the packages in almost all of your programs. This rich package library is one of the factors that make Python programming language powerful.

In fact, the package library we're talking about here isn't built-in and you need to install it externally. That's where the **pip** comes in. pip which is also a preinstalled program in Python is the standard package manager for Python. It allows you to install/uninstall, and manage additional packages that are not part of the [Python standard modules](https://docs.python.org/3/py-modindex.html). The acronym of **Pip Installs Packages** is mostly known as pip.

You can use pip not only to give additional functionality to the standard library by installing additional packages on your computer, but you can also use it to help you contribute to Python's development by sharing your own projects.

Now open your *command prompt* and run the following syntax to make sure that you have pip installed.

pip --version

This code should display your valid **pip** version which is **19.3.1** currently. The output will be :

pip 19.3.1

If you have problems with installing or upgrading **pip**, you can follow the [official guide](https://pip.pypa.io/en/stable/installing/) for the best practice.

**💡Tips:**

* When you install the **Anaconda-3** package program, you will also automatically install hundreds of packages in addition to Python's standard library.
* Therefore, if you installed the Anaconda-3 package program, you will not actually have much work with pip.

Now, let's learn more about this useful tool for managing packages.

#### **Working with pip**

**The formula syntax is** : pip command options

* **install**

The most common and essential command of **pip** is of course install. The most common syntax is :

pip install my\_package

If you want, you can use this command by adding the version number to the end of the syntax as follows :

pip install my\_package==3.2.1

For Python's current version you can use the following command. Although Python is not actually a *package*, you can also install it as follows. You do not need to try it because it will be faster if you [download](https://www.python.org/downloads/release/python-381/)and install it from its website.

pip install python==3.9.0

* **list**

Another important command you should learn is list. It lists all the packages you have installed on your computer in alphabetical order and in two columns.

pip list

* **show**

The other useful command we can mention is show.

pip show my\_package

It's used to view some information about the packages. These information about a package will be : Name, Version, Summary, Home-page, Author, Author-email, License, Location on PC.

* **uninstall**

And the last command we want to show you is uninstall. It uninstalls the installed packages from your computer.

pip uninstall my\_package

If you want to examine pip in detail visit [here](https://pip.pypa.io/en/stable/reference/pip/#pip).

## Program Execution

### Introduction

In this section, we will try to understand how Python codes work, in other words, we will take a look at what's happening when you run the codes.

Have you ever thought, after writing lines of Python code, how they do amazing things? We just write codes and do not care about the rest, but after we press the run button, we can say that very complex things are performed very quickly.

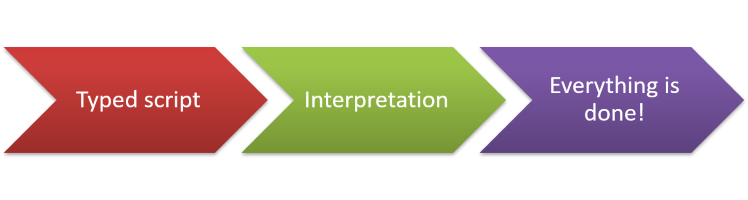
Even if we run the simple conventional Python syntax : print('Hello World!'), the same complex things are performed in the same way. The Python scripts (a file with **.py** extension) contains Python codes and we can run this file via several **interpreter**tools such as Python Shell, OS Console or IDEs. We don't care what's happening when we run our set of Python statements. This is normal but it's worth learning how it works.

**✏️Homework:**

* Do a research on which IDEs can be used for Python codes in the programming world.

Since the very beginning of this course, you have probably heard terms such as **interpreted** or **compiled**languages. And most likely, you learned that Python was an interpreted type of programming language. What does all this mean? Let's dive into the Pythonic world.

The process of program execution (the program flow) basically looks like this :



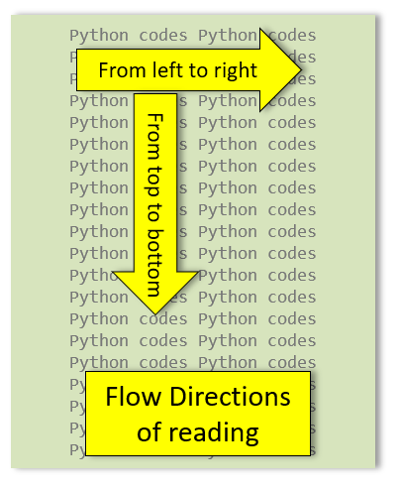
The step we are interested in is the middle one : **Interpretation**. Yes, Python is an interpreted programming language. We know that but superficially. Let's take a closer look at what this means.

The Interpretation

What is the interpretation process? Simply, we can say that this is like reading your codes. In other words, you can think that a specific-purpose software reads your codes line by line and executes what you typed.

This magical software is known as Interpreter and is a part of the standard Python installation package. And of course, the other default installation part of Python is built-in libraries.

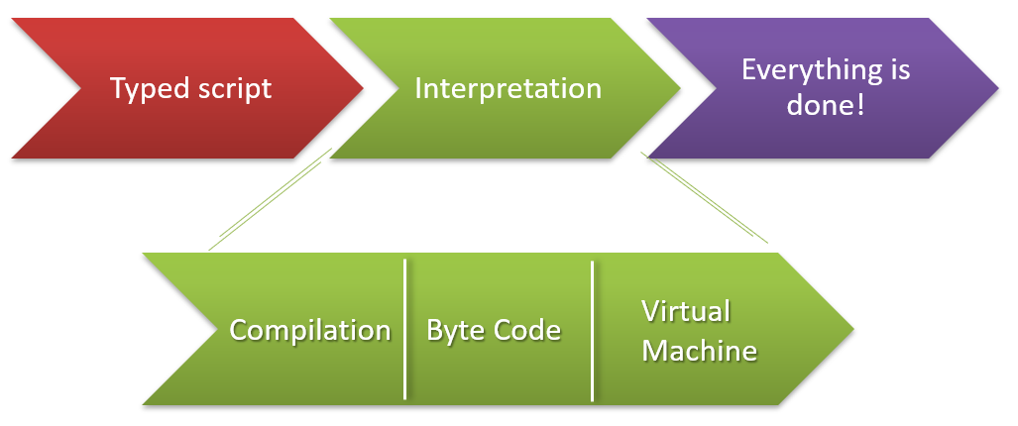
The compiler translates Python codes to byte-codes that come into it by reading line by line from **top to bottom** and from **left to right**.



The interesting thing is that the interpreter we mentioned here can be written in any programming language. Python's default interpreter is a software written in the **C** programming language known as **CPython**. There are several other interpreters available such as :

* **Jython** is an interpreter that works with a Java-based algorithm and converts Python codes into Java-compatible byte code, which will be executed later by the **Java Virtual Machine**.
* **PyPy** is a replacement for CPython. It is built using the RPython language that was co-developed with it. The main reason to use it instead of CPython is speed: **RPython**(Restricted Python) provides some restrictions to the usual Python code.
* **IronPython** is an open-source implementation of the Python programming language which is tightly integrated with the **.NET Framework**.

Now let's try to figure out the interpretation process in detail. The interpretation step consists of basically three sub-steps which are :



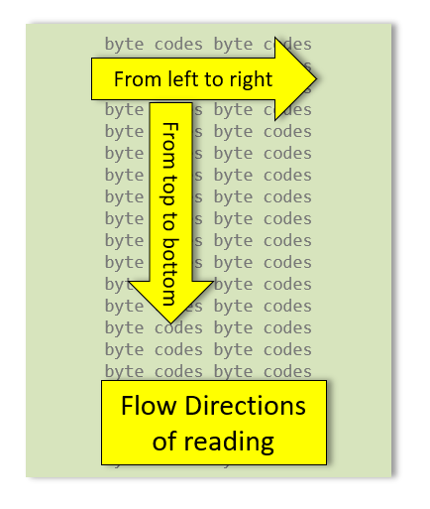
The **compiler**translates your Python statements (source code) into byte-code. Different from the binary machine code, the **byte code** is platform-independent, primitive level and **specific**version of the Python source code. That's why some Python programs are executed not as fast as in **C-based** conventional compiled languages.

After **compilation**of Python statements into **byte-code**, it is now time for **Python Virtual Machine** to run.

### Virtual Machine

When you say Python Virtual Machine, you should not think of a mechanical machine or hardware. It is actually nothing but software made up of a large piece of code.

Just like in compiler, Virtual Machine executes byte-codes that come into it by reading line by line from **top to bottom** and from **left to right**.



As we mentioned before, all this is done automatically by itself. It does not require a separate library or program installation. You do not need to know how the codes that go from the Python statements to the byte-code and from the byte-code to the Virtual Machine are created and executed. Just grasping the order of flow and the logic of these processes will be sufficient at this stage.

Lastly, we can say that the actual execution of the Python codes we typed is done by Python Virtual Machine as the last step. In fact, other previous steps are the process of converting Python statements into a format that Python Virtual Machine can understand and execute.

## Errors

### Introduction

Keep in mind, writing code in Python is easy but requires high attention. Even the slightest punctuation, a comma or a space character will cause you to receive an error warning and your program not to run.

Sometimes you can see that the codes with the simplest syntax can give an error when you never expected. Consider the following example :

input :  
print("Don't say 'I never make a mistake'"

output :

Traceback (most recent call last):

File "code.py", line 1

print("Don't say 'I never make a mistake'"

^

SyntaxError: unexpected EOF while parsing

In the error message appeared on the screen, there is the term **Traceback. It** is actually a [module](https://docs.python.org/3/library/traceback.html#module-traceback) of [614 lines of Python code](https://github.com/python/cpython/blob/3.8/Lib/traceback.py). This module provides a standard interface to extract, format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. In this way, it allows you to follow the line and character of the error and trace it.

**💡Tips:**

* Concentrate on the **last lines** of the error messages.

The name of module - *Traceback* - appears when your code causes an error and it reports detailed information on that specific error, demonstrating the particular files in which the error occurred. In these error messages, the most important thing that a programmer should be interested in is the last lines in the most cases. In this example, the last two lines indicate that this error type is a Syntax error and it also indicates in which line and in which character (with the **^** sign) the error raised.

**⚠️Attention:**Do not panic when you see those error lines. Do not hesitate to read carefully what they are saying to you.

Syntax Errors

In the previous example (shown below), you must have seen the mysterious word **SyntaxError**, which you will likely encounter frequently during your time in Python.

Traceback (most recent call last):

File "code.py", line 1

print("Don't say 'I never make a mistake'"

^

SyntaxError: unexpected EOF while parsing

A wide variety of errors in Python are called SyntaxError. Typically, they indicate a problem that Python encountered when trying to compile your program, or that your code could not be run.

Every syntax error has a text value (known as **associated value**) that describes the error in detail. In this example, the message "**SyntaxError: unexpected EOF while parsing**" means that something else had been expected by the interpreter after your statement, but you didn't pass it to the interpreter. In our mistaken code, there should have been a closing parenthesis mark "**)**" which is missing.

**💡Tips:**Focusing on the **associated value** of the SyntaxError often helps solve your problem.

Error messages may not always be very clear. To find the source of your error, it is sufficient to know that the error you received is in the syntax.

### Common Errors

The most common syntax errors made by programmers can be listed as follows.

**Quotes** :

In the [Matter of Quotes](https://lms.clarusway.com/mod/lesson/view.php?id=3&pageid=6) lesson, we have elaborated on how sensitive programming language Python is to quotation marks. So it's critical not to forget to enclose a string in quotes of the same type.

**💡Tips:**

* Keep this simple advice in your mind; triple quotes for multi-line strings, double or single quotes for ordinary strings.

**Wrong Parentheses :**

One of the most common syntax errors is the wrong number of parentheses in function calls. e.g. print(len(my\_list[2:]).

**Wrong Spelling & Typo :**

Yes, it may sound strange to you, but the most common mistake made by programmers is the wrong spelling keywords, function names, and variable names. e.g. True and true, print and prit or pirnt.

**⚠️Avoid ! :**

* Do not confuse uppercase and lowercase letter of the keywords. Keep in your mind that Python is a case-sensitive programing language.

**Indents :**

Indents are also very common errors for programmers.

**⚠️Avoid ! :**

* Do not forget to put the appropriate indent where necessary. Keep in your mind that Python is a indent-sensitive programing language.

Modern IDEs tend to check everything you might be able to make mistakes and to gently highlight the places you made mistakes or typos with various cursors and indicators while typing. This will make your job significantly easier, but don't rely too much on it and be prepared to read your traceback yourself.

## Exceptions

### Introduction

So far, your code has been syntactically error-free and should be executed smoothly. However, in some cases, you will find that even if you have not made a syntax error, your program will be executed to a certain line and then give some warning messages. This means that your program is *partially executed*. Well, let's figure out what is going on.

If your program is running but you still get some error messages for several reasons, you probably get an error warning called an **exception**. This is because of defective line of codes in your program.

What is the difference between Syntax errors and Exception errors?

| **Difference** | |
| --- | --- |
| **Syntax Error** | **Exception Error** |
| These types of errors are detected during **compiling** the program into byte-code. | These types of errors are detected during the program execution (**interpretation**) process. |

Now, let's examine the following example :  
input :

print('Here we go!')

print('I will be the second text')

a = '3'

b = 5

print('It is time for an error message :(')

print(a + b) # it won't be printed

print("Sorry, but I won't be printed") # it won't be printed

output :

Here we go!

I will be the second text

It is time for an error message :(

Traceback (most recent call last):

File "code.py", line 6, in <module>

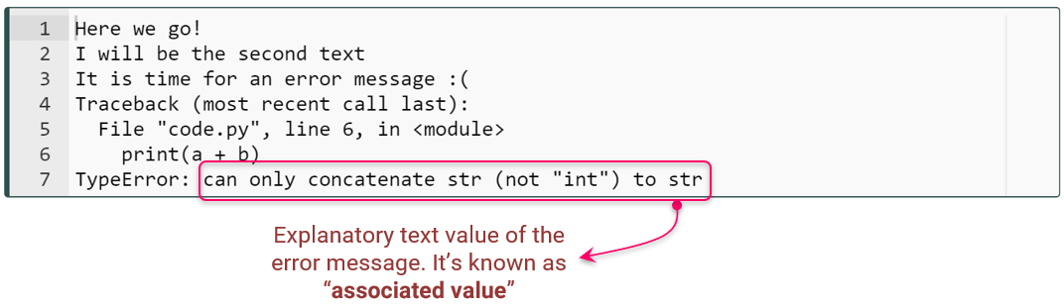
print(a + b)

TypeError: can only concatenate str (not "int") to str

Yes, as you can see, the first three print() functions printed their own content, but the last two print() functions did not work and we received an exception error message (that is our program was partially executed). As we stated before, you should look at the last line of the error message which says that this is the **TypeError**. It's obvious, right? There is a wrong use of types of variables (a is a str type and b is an int).

**💡Tips:** The exceptions are not the end of the world: you will learn how to deal with them later.

Similar to the syntax errors, almost all exception error messages also have a text value following the error message explaining what the error is related to.



**Q**: What are the Errors and Exeptions in Python?   
**A**:

In Python, there are two types of errors: syntax error and exceptions.

**Syntax Error:**It is also known as parsing errors. Errors are issues in a program which may cause it to exit abnormally. When an error is detected, the parser repeats the offending line and then displays an arrow which points at the earliest point in the line.

**Exceptions:**Exceptions take place in a program when the normal flow of the program is interrupted due to the occurrence of an external event. Even if the syntax of the program is correct, there are chances of detecting an error during execution, this error is nothing but an exception. Some of the examples of exceptions are - ZeroDivisionError, TypeError and NameError.

### Common Exceptions

In this lesson, we will take a look at the most common [exception errors](https://docs.python.org/3/library/exceptions.html#built-in-exceptions) a programmer can encounter.

**ValueError** :

Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as IndexError. In other words; to encounter a ValueError in Python means that it is a problem with the content of the object you tried to assign the value to. This should not be confused with types in Python.



Imagine, this is like forcing a fish to live in a chicken coop. Consider the following example :

input :  
print(int('ten'))

output :  
Traceback (most recent call last):

File "code.py", line 1, in <module>

print(int('ten'))

ValueError: invalid literal for int() with base 10: 'ten'

As you can see in the output; when you are trying to make an integer of a string which has no integer value, the **ValueError** raises.

**NameError** :

This error is usually raised if a variable you use in the code stream is not pre-defined or not properly defined. In other words, it is raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found. For better understanding, let's take a look at this example :

input :  
print(variable)

variable = "Don't ever give up!"

output :

Traceback (most recent call last):

File "code.py", line 1, in <module>

print(variable)

NameError: name 'variable' is not defined

**⚠️Attention :** Note that the NameError often raises due to the lack of attention to these two things: **case-sensitivity** of Python and **pre-defines** of the variables.

**TypeError** :

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch. Let's examine this example :

input :  
for i in range('x'):

print(i)

output :  
Traceback (most recent call last):

File "code.py", line 1, in <module>

for i in range('x'):

TypeError: 'str' object cannot be interpreted as an integer

As in the example; although it requires int, we have tried to iterate the wrong type - str - object in the range() function.

**💡Tips:**

* The most useful way you can do about all the errors you can't deal with yourself is to search for the error message on the internet search engine.
* You can make sure that the errors that you will encounter and their solutions have been experienced by someone previously.

**Q**: Give some examples of standard errors that occour in Python.  
**A**:

* **TypeError-** It occurs when the expected type does not match with the given type of a variable.
* **ValueError-**It occurs when an expected value is not given, suppose you are expecting 6 elements in a list and you gave 2.
* **NameError-** It occurs when you are trying to access an undefined variable or a function.
* **IOError-** It occurs when you are trying to access a file that does not exist.
* **IndexError-** It occurs when you are trying to access an invalid index of a sequence.
* **KeyError-** It occurs when you use an invalid key to access a value in the dictionary.

## Exception Handling

### Introduction

In this lesson, you will learn the strategies we need to prevent exception errors from unnecessarily disrupting our program flow.

In some cases, you may need to anticipate exceptions that might disrupt your program flow and take action in advance. This is where exception handling comes into play.

For example, suppose we have a block of code that takes two numbers from the user, divides the first number by the second number and prints the output of the division. Let's see what happens.

input :  
while True:

no\_one = int(input("The first number please : "))

no\_two = int(input("The second number please : "))

division = no\_one / no\_two

print("The result of the division is : ", division)

break

Let's run it for the first time :

output-1 :

The first number please : 6

The second number please : 3

The result of the division is : 2.0

Let's run it again :

output-2 :  
The first number please : 7

The second number please : 2

The result of the division is : 3.5

This time, let's make a mistake intentionally and enter 0 as the second number :

output-3 :  
First number please : 4

Second number please : 0

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

Traceback (most recent call last)

**<ipython-input-5-537f57020b40>** in <module>

2 no\_one = int(input("First number please : "))

3 no\_two = int(input("Second number please : "))

----> 4 division = no\_one / no\_two

5 print("The result of the divison is : ", division)

**ZeroDivisionError**: division by zero

Yes, we received an exception error message in our last attempt. So it's time to show you the way to solve it. We can ensure that our program continues without hesitation by using the **try-except** **statement**. The basic structure of **try-except** **statement** looks like :

try:

code block to be normally executed

except:

code block to be exceptionally executed

Let's fix this exception problem using **try-except** **statement** as follows :

input :

while True:

no\_one = int(input("The first number please : "))

no\_two = int(input("The second number please : "))

try:

division = no\_one / no\_two

print("The result of the division is : ", division)

break

except:

print("Something went wrong...Try again.")

When we enter 0 as a second number, then let's see what will happen :  
output :

The first number please : 5

The second number please : 0

Something went wrong...Try again.

As you can see the output, there is no exception error message or breaking of the program. We can specify the exception error message in our **try-except** **statement**.

**💡Tips:**

* Of course, when do this, we anticipate which exception error we will encounter.

Consider this block of code :

input :  
while True:

no\_one = int(input("The first number please : "))

no\_two = int(input("The second number please : "))

try:

division = no\_one / no\_two

print("The result of the division is : ", division)

break

except ZeroDivisionError:

print("You can't divide by zero! Try again.")

output :

The first number please : 5

The second number please : 0

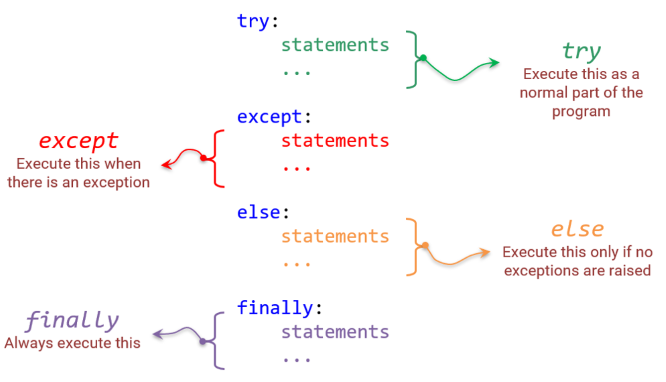
You can't divide by zero! Try again.

Let's elaborate on this and the other examples with a few more keywords in the next lesson.

### Full 'Exception Handling Block'

There are a few more keywords used with **try-except** **statement**. By using them, we increase our useful options in our program and create more effective and quality programs.

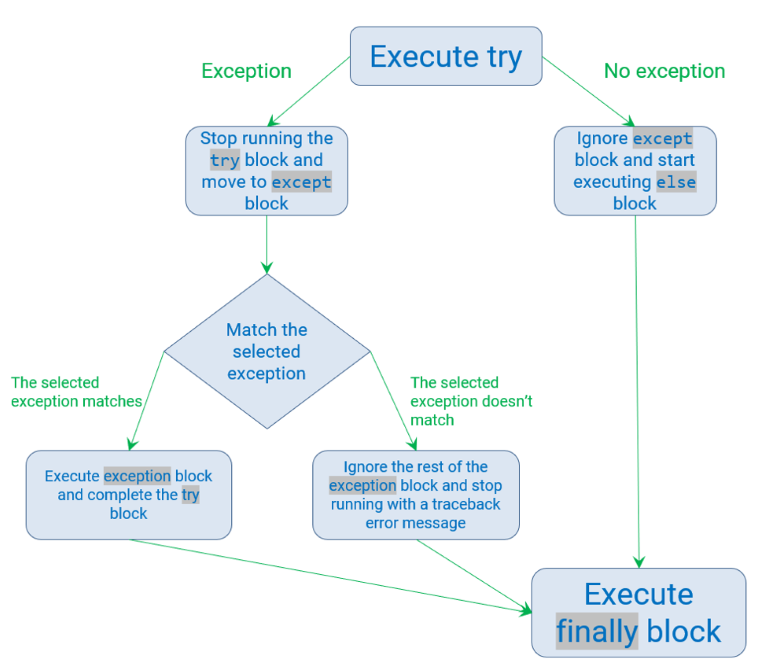
The full version of **try-except** **statement** is as follows :



Let's make some **explanations**about the **process** described above :

* First, Python executes all statements between try and except.
* If there is **no exception**, then the try **block** is successfully executed and finished.
* If an **exception raises**, the rest of the try **block** is **ignored**. After that, Python interpreter checks if the type of exception matches the exception specified after the except keyword, it executes the except **block** and resumes executing the program **after** the try-except **block**.
* If an exception doesn’t match the exception named in the except **syntax**, it is known an unhandled exception and execution of your program stops with a **traceback error message**.
* The else**-block** is only executed if there raised no exceptions.
* We can use a finally **syntax** at the end of the block. A finally **syntax** is always executed before leaving the try-except **block**, whether an exception was raised or not.

You can follow the flow of these processes below :



You can examine the details of [Handling Exception](https://docs.python.org/3/tutorial/errors.html#handling-exceptions) in the official Python documentation.

Implementation of the Full 'Exception Handling Block'-1

Let's reorganize our previous code according to the structure described in the previous lesson. Consider this block of code :

input :

while True:

no\_one = int(input("The first number please : "))

no\_two = int(input("The second number please : "))

try:

division = no\_one / no\_two # normal part of the program

except ZeroDivisionError:

print("You can't divide by zero! Try again.") # executes when division by zero

else:

print("The result of the division is : ", division) # executes if there is no exception

finally:

print("Thanks for using our mini divison calculator! Come again!")

break # exits the while loop

output - 1 : 

The first number please : 7

The second number please : 2

The result of the division is : 3.5

Thanks for using our mini divison calculator! Come again!

output - 2 :  
Let's enter 0 as a second number.

The first number please : 7

The second number please : 0

You can't divide by zero! Try again.

Thanks for using our mini divison calculator! Come again!

output - 3 :  
This time, let's enter str type as a second number. The selected exception (ZeroDivisionError) will not match so traceback error will be raised. 

The first number please : 7

The second number please : k

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

ValueError Traceback (most recent call last)

<ipython-input-2-5fd297d1c219> in <module>

1 while True:

2 no\_one = int(input("The first number please : "))

----> 3 no\_two = int(input("The second number please : "))

4 try:

5 division = no\_one / no\_two # normal part of the program

ValueError: invalid literal for int() with base 10: 'k'

No matter what exception we encounter, there is a way to see it. Using the keyword Exception. Let's take a look at the following example :

input :

while True:

no\_one = int(input("The first number please : "))

no\_two = int(input("The second number please : "))

try:

division = no\_one / no\_two

print("The result of the division is : ", division)

break

except Exception as e:

print("Something went wrong...Try again.")

print("Probably it is because of '{}' error".format(e))

break

output :

The first number please : 4

The second number please : 0

Something went wrong...Try again.

Probably it is because of 'division by zero' error.

Yes, as you can see the output, by using the keyword Exception, we can display the name of the exception type. Consider the other example :

input :

try :

a = 10

b = 2

print("The result of division is :", c)

except Exception as e:

print("The error message is : ", e)

output :

The error message is : name 'c' is not defined

In this case, we used the variable "c" and since we didn't define it previously, it raised a **NameError** of which associated value is displayed in the output cell above : "*name 'c' is not defined*".

**💡Tips:**

* Apart from GeneratorExit, KeyboardInterrupt, SystemExit, the Exception keyword you learned in the previous lesson covers all exceptions.

### Implementation of the Full 'Exception Handling Block'-2

In this second part of our lesson, we will continue to examine the full exception block through various examples. While examining the samples, try to understand the results by experimenting on the **Playground** yourself.

The **try…except** block has an optional else clause. The else clause is executed only if no exceptions are raised. Consider the following example of the else clause :

input :  
try:

x = 4 / 1

except:

print('Something went wrong')

else:

print('Nothing went wrong')

output :  
Nothing went wrong

The **try…except** block has another optional finally clause. The finally clause is always executed, whether an exception has occurred or not. Consider the following example of the finally clause :

input :  
try:

x = 3 / 0

except:

print('Something went wrong')

finally:

print('Always execute this')

output :  
Something went wrong

Always execute this

Use finally clause to define **clean-up actions** that must be executed under all circumstances e.g. *closing a file*. :

input :  
try:

f = open('myfile.txt')

print(f.read())

except:

print("Something went wrong")

finally:

f. close()

### Several Handling Scenarios

In this part of our lesson, we will continue to examine this issue through various examples. While examining the samples, try to understand the results by experimenting on the **Playground** yourself.

Take a look at the first one. In this example you can define as many except blocks as you want, to catch and handle specific exceptions :

input :  
try:

x = 2/0

except ZeroDivisionError:

print('Attempt to divide by zero')

except:

print('Something else went wrong')

output :  
Attempt to divide by zero

Since all the built-in exceptions are formed a [**hierarchical structure**](https://docs.python.org/3/library/exceptions.html#exception-hierarchy), you can use the following exception syntaxes. To catch several possibilities, it’d be better to use two or more except blocks for different exceptions.

input-1: you can identify no specific exception :  
....

except:

print("Unknown exception error occured. Tyr again please.")

input-2: different exception blocks :

....

except ZeroDivisionError:

print("You can't divide by zero!!")

except ValueError:

print("You can only enter numbers consisting of digits, not text!!")

input-3: an except syntax also may contain multiple exceptions separated by commas and enclosed by parenthesis.

....

except (ValueError, TypeError):

print("You can only enter numbers consisting of digits, not text!!")

input-4: because of the **hierarchical structure**, one exception can actually catch various exceptions.

....

except ArithmeticError:

print("I will also catch OverflowError, FloatingPointError and ZeroDivisionError")

You can find all the built-in Python exceptions in the [official documentation](https://docs.python.org/3/library/exceptions.html#built-in-exceptions).