

[Donate](#)

[Learn to code — free 3,000-hour curriculum](#)

APRIL 27, 2021/[#PYTHON](#)

# Python Code Examples – Sample Script Coding Tutorial for Beginners



[Estefania Cassingena Navone](#)

Hi! Welcome. If you are learning Python, then this article is for you. You will find a thorough description of Python syntax and lots of code examples to guide you during your coding journey.

## **What we will cover:**

- [Variable Definitions in Python](#)
- [Hello, World! Program in Python](#)
- [Data Types and Built-in Data Structures in Python](#)
- [Python Operators](#)
- [Conditionals in Python](#)
- [For Loops in Python](#)
- [While Loops in Python](#)
- [Nested Loops in Python](#)
- [Functions in Python](#)
- [Recursion in Python](#)
- [Exception Handling in Python](#)

- [Object-Oriented Programming in Python](#)
- [How to Work with Files in Python](#)
- [Import Statements in Python](#)
- [List and Dictionary Comprehension in Python](#)
- and more...

Are you ready? Let's begin! ⚙️

💡 **Tip:** throughout this article, I will use `<>` to indicate that this part of the syntax will be replaced by the element described by the text. For example, `<var>` means that this will be replaced by a variable when we write the code.

## 🔹 Variable Definitions in Python

The most basic building-block of any programming language is the concept of a variable, a name and place in memory that we reserve for a value.

In Python, we use this syntax to create a variable and assign a value to this variable:

```
<var_name> = <value>
```

For example:

```
age = 56
name = "Nora"
color = "Blue"
grades = [67, 100, 87, 56]
```

If the name of a variable has more than one word, then the [Style Guide for Python Code](#) recommends separating words with an underscore "as necessary to improve readability."

For example:

```
my_list = [1, 2, 3, 4, 5]
```

💡 **Tip:** The Style Guide for Python Code (PEP 8) has great suggestions that you should follow to write clean Python code.

## ◆ Hello, World! Program in Python

Before we start diving into the data types and data structures that you can use in Python, let's see how you can write your first Python program.

You just need to call the `print()` function and write `"Hello, World!"` within parentheses:

```
print("Hello, World!")
```

You will see this message after running the program:

```
"Hello, World!"
```

💡 **Tip:** Writing a `"Hello, World!"` program is a tradition in the developer community. Most developers start learning how to code by writing this program.

Great. You just wrote your first Python program. Now let's start learning about the data types and built-in data structures that you can use in Python.

## ◆ Data Types and Built-in Data Structures in Python

We have several basic data types and built-in data structures that we can work with in our programs. Each one has its own particular applications. Let's see them in detail.

### **Numeric Data Types in Python: Integers, Floats, and Complex**

These are the numeric types that you can work with in Python:

#### **Integers**

Integers are numbers without decimals. You can check if a number is an integer with the `type()` function. If the output is `<class 'int'>`, then the number is an integer.

For example:

```
>>> type(1)
<class 'int'>
```

```
>>> type(15)
<class 'int'>
```

```
>>> type(0)
<class 'int'>
```

```
>>> type(-46)
<class 'int'>
```

### **Floats**

Floats are numbers with decimals. You can detect them visually by locating the decimal point. If we call `type()` to check the data type of these values, we will see this as the output:

```
<class 'float'>
```

Here we have some examples:

```
>>> type(4.5)
<class 'float'>
```

```
>>> type(5.8)
<class 'float'>
```

```
>>> type(2342423424.3)
<class 'float'>
```

```
>>> type(4.0)
<class 'float'>
```

```
>>> type(0.0)
<class 'float'>
```

```
>>> type(-23.5)
<class 'float'>
```

## Complex

Complex numbers have a real part and an imaginary part denoted with `j`. You can create complex numbers in Python with `complex()`. The first argument will be the real part and the second argument will be the imaginary part.

These are some examples:

```
>>> complex(4, 5)
(4+5j)
```

```
>>> complex(6, 8)
(6+8j)
```

```
>>> complex(3.4, 3.4)
(3.4+3.4j)
```

```
>>> complex(0, 0)
0j
```

```
>>> complex(5)
(5+0j)
```

```
>>> complex(0, 4)
4j
```

## Strings in Python

Strings are incredibly helpful in Python. They contain a sequence of characters and they are usually used to represent text in the code.

For example:

```
"Hello, World!"
'Hello, World!'
```

We can use both single quotes `' '` or double quotes `" "` to define a string. They are both valid and equivalent, but you should choose one of them and use it consistently throughout the program.

**💡 Tip:** Yes! You used a string when you wrote the `"Hello, World!"` program. Whenever you see a value surrounded by single or double quotes in Python, that is a string.

Strings can contain any character that we can type in our keyboard, including numbers, symbols, and other special characters.

For example:

```
"45678"
```

```
"my_email@email.com"
```

```
"#IlovePython"
```

**💡 Tip:** Spaces are also counted as characters in a string.

### Quotes Within Strings

If we define a string with double quotes `" "`, then we can use single quotes within the string. For example:

```
"I'm 20 years old"
```

If we define a string with single quotes `' '`, then we can use double quotes within the string. For example:

```
'My favorite book is "Sense and Sensibility"'
```


### String Indexing

We can use indices to access the characters of a string in our Python program. An index is an integer that represents a specific position in the string. They are associated to the character at that position.

This is a diagram of the string `"Hello"`:

```
String:  H e l l o
```

```
Index:   0 1 2 3 4
```

 **Tip:** Indices start from 0 and they are incremented by 1 for each character to the right.

For example:

```
>>> my_string = "Hello"
```

```
>>> my_string[0]
'H'
```

```
>>> my_string[1]
'e'
```

```
>>> my_string[2]
'l'
```

```
>>> my_string[3]
'l'
```

```
>>> my_string[4]
'o'
```

We can also use negative indices to access these characters:

```
>>> my_string = "Hello"
```

```
>>> my_string[-1]
'o'
```


```
>>> my_string[-2]
'l'
```

```
>>> my_string[-3]
'l'
```

```
>>> my_string[-4]
'e'
```

```
>>> my_string[-5]
```

```
'H'
```

 **Tip:** we commonly use `-1` to access the last character of a string.

### String Slicing

We may also need to get a slice of a string or a subset of its characters. We can do so with string slicing.

This is the general syntax:

```
<string_variable>[start:stop:step]
```

`start` is the index of the first character that will be included in the slice. By default, it's `0`.

- `stop` is the index of the last character in the slice (this character will **not** be included). By default, it is the last character in the string (if we omit this value, the last character will also be included).
- `step` is how much we are going to add to the current index to reach the next index.

We can specify two parameters to use the default value of `step`, which is `1`. This will include all the characters between `start` and `stop` (not inclusive):

```
<string_variable>[start:stop]
```

For example:

```
>>> freecodecamp = "freeCodeCamp"
```

```
>>> freecodecamp[2:8]
```

```
'eeCode'
```

```
>>> freecodecamp[0:3]
```

```
'fre'
```



```
>>> freecodecamp[0:4]
'free'
```


```
>>> freecodecamp[4:7]
'Cod'
```

```
>>> freecodecamp[4:8]
'Code'
```

```
>>> freecodecamp[8:11]
'Cam'
```

```
>>> freecodecamp[8:12]
'Camp'
```

```
>>> freecodecamp[8:13]
'Camp'
```

 **Tip:** Notice that if the value of a parameter goes beyond the valid range of indices, the slice will still be presented. This is how the creators of Python implemented this feature of string slicing. If we customize the `step`, we will "jump" from one index to the next according to this value.  
For example:

```
>>> freecodecamp = "freeCodeCamp"
```

```
>>> freecodecamp[0:9:2]
'feCdC'
```

```
>>> freecodecamp[2:10:3]
'eoC'
```

```
>>> freecodecamp[1:12:4]
'roa'
```

```
>>> freecodecamp[4:8:2]
'Cd'
```

```
>>> freecodecamp[3:9:2]
'ee'
```

```
>>> freecodecamp[1:10:5]
'rd'
```

We can also use a **negative** step to go from right to left:

```
>>> freecodecamp = "freeCodeCamp"
```

```
>>> freecodecamp[10:2:-1]
'maCedoCe'
```

```
>>> freecodecamp[11:4:-2]
'paeo'
```

```
>>> freecodecamp[5:2:-4]
'o'
```

And we can omit a parameter to use its default value. We just have to include the corresponding colon (:) if we omit `start`, `stop`, or both:

```
>>> freecodecamp = "freeCodeCamp"
```

```
# Default start and step
```

```
>>> freecodecamp[:8]
'freeCode'
```

```
# Default end and step
```

```
>>> freecodecamp[4:]
'CodeCamp'
```

```
# Default start
```

```
>>> freecodecamp[:8:2]
'feCd'
```

```
# Default stop
```

```
>>> freecodecamp[4::3]
```

```
'Cem'
```

```
# Default start and stop
```


```
>>> freecodecamp[::-2]
```

```
'paeoer'
```

```
# Default start and stop
```

```
>>> freecodecamp[::-1]
```

```
'pmaCedoCeerf'
```

 **Tip:** The last example is one of the most common ways to reverse a string.

### **f-Strings**

In Python 3.6 and more recent versions, we can use a type of string called f-string that helps us format our strings much more easily.

To define an f-string, we just add an `f` before the single or double quotes. Then, within the string, we surround the variables or expressions with curly braces `{ }`. This replaces their value in the string when we run the program.

For example:

```
first_name = "Nora"
```

```
favorite_language = "Python"
```

```
print(f'Hi, I'm {first_name}. I'm learning {favorite_language}.')
```

The output is:

```
Hi, I'm Nora. I'm learning Python.
```

Here we have an example where we calculate the value of an expression and replace the result in the string:

```
value = 5
```

```
print(f"{value} multiplied by 2 is: {value * 2}")
```

The values are replaced in the output:

```
5 multiplied by 2 is: 10
```

We can also call methods within the curly braces and the value returned will be replaced in the string when we run the program:

```
freecodecamp = "FREECODECAMP"
```

```
print(f"{freecodecamp.lower()}")
```

The output is:

```
freecodecamp
```

### String Methods

Strings have methods, which represent common functionality that has been implemented by Python developers, so we can use it in our programs directly. They are very helpful to perform common operations.

This is the general syntax to call a string method:

```
<string_variable>.<method_name>(<arguments>)
```

For example:

```
>>> freecodecamp = "freeCodeCamp"
```

```
>>> freecodecamp.capitalize()
```

```
'Freecodecamp'
```

```
>>> freecodecamp.count("C")
```

```
2
```

```
>>> freecodecamp.find("e")
```

```
2
```

```
>>> freecodecamp.index("p")
```

```
11
```

```
>>> freecodecamp.isalnum()
```

```
True
```

```
>>> freecodecamp.isalpha()
```

```
True
```

```
>>> freecodecamp.isdecimal()
```

```
False
```

```
>>> freecodecamp.isdigit()
```

```
False
```

```
>>> freecodecamp.isidentifier()
```

```
True
```

```
>>> freecodecamp.islower()
```

```
False
```

```
>>> freecodecamp.isnumeric()
```

```
False
```

```
>>> freecodecamp.isprintable()
```

```
True
```

```
>>> freecodecamp.isspace()
```

```
False
```

```
>>> freecodecamp.istitle()
```

```
False
```

```
>>> freecodecamp.isupper()
```

```
False
```

```
>>> freecodecamp.lower()
```

```
'freecodecamp'
```

```
>>> freecodecamp.lstrip("f")
```

```
'reeCodeCamp'
```

```
>>> freecodecamp.rstrip("p")
```

```
'freeCodeCam'
```

```
>>> freecodecamp.replace("e", "a")
```

```
'freaaCodaCamp'
```

```
>>> freecodecamp.split("C")
```

```
['free', 'ode', 'amp']
```

```
>>> freecodecamp.swapcase()
```

```
'FREEcODEcAMP'
```


```
>>> freecodecamp.title()
```

```
'Freecodecamp'
```

```
>>> freecodecamp.upper()
```

```
'FREECODECAMP'
```

To learn more about Python methods, I would recommend reading [this article](#) from the Python documentation.

 **Tip:** All string methods return copies of the string. They do not modify the string because strings are immutable in Python.

## Booleans in Python

Boolean values are `True` and `False` in Python. They must start with an uppercase letter to be recognized as a boolean value.

For example:

```
>>> type(True)
<class 'bool'>
```

```
>>> type(False)
<class 'bool'>
```

If we write them in lowercase, we will get an error:


```
>>> type(true)
Traceback (most recent call last):
  File "<pyshell#92>", line 1, in <module>
    type(true)
NameError: name 'true' is not defined
```

```
>>> type(false)
Traceback (most recent call last):
  File "<pyshell#93>", line 1, in <module>
    type(false)
NameError: name 'false' is not defined
```

## Lists in Python

Now that we've covered the basic data types in Python, let's start covering the built-in data structures. First, we have lists.

To define a list, we use square brackets `[]` with the elements separated by a comma.

 **Tip:** It's recommended to add a space after each comma to make the code more readable.

For example, here we have examples of lists:

```
[1, 2, 3, 4, 5]
["a", "b", "c", "d"]
[3.4, 2.4, 2.6, 3.5]
```

Lists can contain values of different data types, so this would be a valid list in Python:

```
[1, "Emily", 3.4]
```

We can also assign a list to a variable:

```
my_list = [1, 2, 3, 4, 5]
letters = ["a", "b", "c", "d"]
```

### **Nested Lists**

Lists can contain values of any data type, even other lists. These inner lists are called **nested lists**.

```
[[1, 2, 3], [4, 5, 6]]
```

In this example, `[1, 2, 3]` and `[4, 5, 6]` are nested lists.

Here we have other valid examples:

```
[["a", "b", "c"], ["d", "e", "f"], ["g", "h", "i"]]
[1, [2, 3, 4], [5, 6, 7], 3.4]
```

We can access the nested lists using their corresponding index:

```
>>> my_list = [[1, 2, 3], [4, 5, 6]]
```

```
>>> my_list[0]
[1, 2, 3]
```

```
>>> my_list[1]
[4, 5, 6]
```

Nested lists could be used to represent, for example, the structure of a simple 2D game board where each number could represent a different element or tile:

```
# Sample Board where:
# 0 = Empty tile
# 1 = Coin
# 2 = Enemy
# 3 = Goal
board = [[0, 0, 1],
         [0, 2, 0],
```



```
[1, 0, 3]
```

### List Length

We can use the `len()` function to get the length of a list (the number of elements it contains).

For example:

```
>>> my_list = [1, 2, 3, 4]
```

```
>>> len(my_list)
```

```
4
```

### Update a Value in a List

We can update the value at a particular index with this syntax:

```
<list_variable>[<index>] = <value>
```

For example:

```
>>> letters = ["a", "b", "c", "d"]
```

```
>>> letters[0] = "z"
```

```
>>> letters
```

```
['z', 'b', 'c', 'd']
```

### Add a Value to a List

We can add a new value to the end of a list with the `.append()` method.

For example:

```
>>> my_list = [1, 2, 3, 4]
```

```
>>> my_list.append(5)
```

```
>>> my_list
```

```
[1, 2, 3, 4, 5]
```

### Remove a Value from a List


We can remove a value from a list with the `.remove()` method.  
For example:

```
>>> my_list = [1, 2, 3, 4]
```

```
>>> my_list.remove(3)
```

```
>>> my_list
```

```
[1, 2, 4]
```

 **Tip:** This will only remove the first occurrence of the element.  
For example, if we try to remove the number 3 from a list that has two number 3s, the second number will not be removed:

```
>>> my_list = [1, 2, 3, 3, 4]
```

```
>>> my_list.remove(3)
```

```
>>> my_list
```

```
[1, 2, 3, 4]
```

### List Indexing

We can index a list just like we index strings, with indices that start from 0:

```
>>> letters = ["a", "b", "c", "d"]
```

```
>>> letters[0]
```

```
'a'
```

```
>>> letters[1]
```

```
'b'
```

```
>>> letters[2]
```

```
'c'
```

```
>>> letters[3]
```

```
'd'
```

## List Slicing

We can also get a slice of a list using the same syntax that we used with strings and we can omit the parameters to use their default values. Now, instead of adding characters to the slice, we will be adding the elements of the list.

```
<list_variable>[start:stop:step]
```

For example:

```
>>> my_list = ["a", "b", "c", "d", "e", "f", "g", "h", "i"]
```

```
>>> my_list[2:6:2]  
['c', 'e']
```

```
>>> my_list[2:8]  
['c', 'd', 'e', 'f', 'g', 'h']
```

```
>>> my_list[1:10]  
['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

```
>>> my_list[4:8:2]  
['e', 'g']
```

```
>>> my_list[::-1]  
['i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
```

```
>>> my_list[::-2]  
['i', 'g', 'e', 'c', 'a']
```

```
>>> my_list[8:1:-1]  
['i', 'h', 'g', 'f', 'e', 'd', 'c']
```

## List Methods

Python also has list methods already implemented to help us perform common list operations. Here are some examples of the most commonly used list methods:

```
>>> my_list = [1, 2, 3, 3, 4]

>>> my_list.append(5)
>>> my_list
[1, 2, 3, 3, 4, 5]

>>> my_list.extend([6, 7, 8])
>>> my_list
[1, 2, 3, 3, 4, 5, 6, 7, 8]

>>> my_list.insert(2, 15)
>>> my_list
[1, 2, 15, 3, 3, 4, 5, 6, 7, 8, 2, 2]

>>> my_list.remove(2)
>>> my_list
[1, 15, 3, 3, 4, 5, 6, 7, 8, 2, 2]

>>> my_list.pop()
2

>>> my_list.index(6)
6

>>> my_list.count(2)
1

>>> my_list.sort()
>>> my_list
[1, 2, 3, 3, 4, 5, 6, 7, 8, 15]

>>> my_list.reverse()
>>> my_list
[15, 8, 7, 6, 5, 4, 3, 3, 2, 1]

>>> my_list.clear()
```

```
>>> my_list
```

```
[]
```

To learn more about list methods, I would recommend reading [this article](#) from the Python documentation.

## Tuples in Python

To define a tuple in Python, we use parentheses `()` and separate the elements with a comma. It is recommended to add a space after each comma to make the code more readable.

```
(1, 2, 3, 4, 5)
```

```
("a", "b", "c", "d")
```

```
(3.4, 2.4, 2.6, 3.5)
```

We can assign tuples to variables:

```
my_tuple = (1, 2, 3, 4, 5)
```

## Tuple Indexing

We can access each element of a tuple with its corresponding index:

```
>>> my_tuple = (1, 2, 3, 4)
```

```
>>> my_tuple[0]
```

```
1
```

```
>>> my_tuple[1]
```

```
2
```

```
>>> my_tuple[2]
```

```
3
```

```
>>> my_tuple[3]
```

```
4
```

We can also use negative indices:

```
>>> my_tuple = (1, 2, 3, 4)
```

```
>>> my_tuple[-1]
```

```
4
```

```
>>> my_tuple[-2]
```

```
3
```

```
>>> my_tuple[-3]
```

```
2
```

```
>>> my_tuple[-4]
```

```
1
```

### Tuple Length

To find the length of a tuple, we use the `len()` function, passing the tuple as argument:

```
>>> my_tuple = (1, 2, 3, 4)
```

```
>>> len(my_tuple)
```

```
4
```

### Nested Tuples

Tuples can contain values of any data type, even lists and other tuples. These inner tuples are called **nested tuples**.

```
[[1, 2, 3], (4, 5, 6)]
```

In this example, we have a nested tuple `(4, 5, 6)` and a list. You can access these nested data structures with their corresponding index.

For example:

```
>>> my_tuple = ([1, 2, 3], (4, 5, 6))
```

```
>>> my_tuple[0]
```

```
[1, 2, 3]
```

```
>>> my_tuple[1]
```

```
(4, 5, 6)
```

### **Tuple Slicing**

We can slice a tuple just like we sliced lists and strings. The same principle and rules apply.

This is the general syntax:

```
<tuple_variable>[start:stop:step]
```

For example:

```
>>> my_tuple = (4, 5, 6, 7, 8, 9, 10)
```

```
>>> my_tuple[3:8]
```

```
(7, 8, 9, 10)
```

```
>>> my_tuple[2:9:2]
```

```
(6, 8, 10)
```

```
>>> my_tuple[:8]
```

```
(4, 5, 6, 7, 8, 9, 10)
```

```
>>> my_tuple[:6]
```

```
(4, 5, 6, 7, 8, 9)
```

```
>>> my_tuple[:4]
```

```
(4, 5, 6, 7)
```

```
>>> my_tuple[3:]
```

```
(7, 8, 9, 10)
```

```
>>> my_tuple[2:5:2]
```

```
(6, 8)
```

```
>>> my_tuple[::2]
```

```
(4, 6, 8, 10)
```

```
>>> my_tuple[::-1]
(10, 9, 8, 7, 6, 5, 4)
```

```
>>> my_tuple[4:1:-1]
(8, 7, 6)
```


## Tuple Methods

There are two built-in tuple methods in Python:

```
>>> my_tuple = (4, 4, 5, 6, 6, 7, 8, 9, 10)
```

```
>>> my_tuple.count(6)
2
```

```
>>> my_tuple.index(7)
5
```

 **Tip:** tuples are immutable. They cannot be modified, so we can't add, update, or remove elements from the tuple. If we need to do so, then we need to create a new copy of the tuple.

## Tuple Assignment

In Python, we have a really cool feature called Tuple Assignment. With this type of assignment, we can assign values to multiple variables on the same line.

The values are assigned to their corresponding variables in the order that they appear. For example, in `a, b = 1, 2` the value `1` is assigned to the variable `a` and the value `2` is assigned to the variable `b`.

For example:

```
# Tuple Assignment
```

```
>>> a, b = 1, 2
```


```
>>> a
```



1

```
>>> b
```

2

 **Tip:** Tuple assignment is commonly used to swap the values of two variables:

```
>>> a = 1
```

```
>>> b = 2
```

```
# Swap the values
```

```
>>> a, b = b, a
```

```
>>> a
```

2

```
>>> b
```

1

## Dictionaries in Python

Now let's start diving into dictionaries. This built-in data structure lets us create pairs of values where one value is associated with another one.

To define a dictionary in Python, we use curly brackets `{ }` with the key-value pairs separated by a comma.

The key is separated from the value with a colon `:`, like this:

```
{"a": 1, "b": 2, "c": 3}
```

You can assign the dictionary to a variable:

```
my_dict = {"a": 1, "b": 2, "c": 3}
```

The keys of a dictionary must be of an immutable data type. For example, they can be strings, numbers, or tuples but not lists since lists are mutable.

- **Strings:** {"City 1": 456, "City 2": 577, "City 3": 678}
- **Numbers:** {1: "Move Left", 2: "Move Right", 3: "Move Up", 4: "Move Down"}
- **Tuples:** {(0, 0): "Start", (2, 4): "Goal"}

The values of a dictionary can be of any data type, so we can assign strings, numbers, lists, tuple, sets, and even other dictionaries as the values. Here we have some examples:

```
{"product_id": 4556, "ingredients": ["tomato", "cheese", "mushrooms"],
"price": 10.67}
{"product_id": 4556, "ingredients": ("tomato", "cheese", "mushrooms"), "price": 10.67}
{"id": 567, "name": "Emily", "grades": {"Mathematics": 80, "Biology": 74, "English": 97}}
```

### Dictionary Length

To get the number of key-value pairs, we use the `len()` function:

```
>>> my_dict = {"a": 1, "b": 2, "c": 3, "d": 4}
```

```
>>> len(my_dict)
```

```
4
```

### Get a Value in a Dictionary

To get a value in a dictionary, we use its key with this syntax:

```
<variable_with_dictionary>[<key>]
```

This expression will be replaced by the value that corresponds to the key.

For example:

```
my_dict = {"a": 1, "b": 2, "c": 3, "d": 4}
```

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

The output is the value associated to "a":

```
1
```

### Update a Value in a Dictionary

To update the value associated with an existing key, we use the same syntax but now we add an assignment operator and the value:

```
<variable_with_dictionary>[<key>] = <value>
```

For example:

```
>>> my_dict = {"a": 1, "b": 2, "c": 3, "d": 4}
```

```
>>> my_dict["b"] = 6
```

Now the dictionary is:

```
{'a': 1, 'b': 6, 'c': 3, 'd': 4}
```

### **Add a Key-Value Pair to a Dictionary**

The keys of a dictionary have to be unique. To add a new key-value pair we use the same syntax that we use to update a value, but now the key has to be new.

```
<variable_with_dictionary>[<new_key>] = <value>
```

For example:

```
>>> my_dict = {"a": 1, "b": 2, "c": 3, "d": 4}
```

```
>>> my_dict["e"] = 5
```

Now the dictionary has a new key-value pair:

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

### **Delete a Key-Value Pair in a Dictionary**

To delete a key-value pair, we use the `del` statement:

```
del <dictionary_variable>[<key>]
```

For example:

```
>>> my_dict = {"a": 1, "b": 2, "c": 3, "d": 4}
```

```
>>> del my_dict["c"]
```

Now the dictionary is:

```
{'a': 1, 'b': 2, 'd': 4}
```

## Dictionary Methods

These are some examples of the most commonly used dictionary methods:

```
>>> my_dict = {"a": 1, "b": 2, "c": 3, "d": 4}
```

```
>>> my_dict.get("c")
```

```
3
```

```
>>> my_dict.items()
```

```
dict_items([('a', 1), ('b', 2), ('c', 3), ('d', 4)])
```

```
>>> my_dict.keys()
```

```
dict_keys(['a', 'b', 'c', 'd'])
```

```
>>> my_dict.pop("d")
```

```
4
```

```
>>> my_dict.popitem()
```

```
('c', 3)
```

```
>>> my_dict.setdefault("a", 15)
```

```
1
```

```
>>> my_dict
```

```
{'a': 1, 'b': 2}
```

```
>>> my_dict.setdefault("f", 25)
```

```
25
```

```
>>> my_dict
```

```
{'a': 1, 'b': 2, 'f': 25}
```

```
>>> my_dict.update({"c": 3, "d": 4, "e": 5})
```

```
>>> my_dict.values()
dict_values([1, 2, 25, 3, 4, 5])
```

```
>>> my_dict.clear()
```

```
>>> my_dict
```

```
{}
```

To learn more about dictionary methods, I recommend [reading this article](#) from the documentation.

## ◆ Python Operators

Great. Now you know the syntax of the basic data types and built-in data structures in Python, so let's start diving into operators in Python. They are essential to perform operations and to form expressions.

### Arithmetic Operators in Python

These operators are:

**Addition: +**

```
>>> 5 + 6
```

```
11
```

```
>>> 0 + 6
```

```
6
```

```
>>> 3.4 + 5.7
```


```
9.1
```

```
>>> "Hello" + ", " + "World"
```

```
'Hello, World'
```

```
>>> True + False
```

```
1
```

 **Tip:** The last two examples are curious, right? This operator behaves differently based on the data type of the operands. When they are strings, this operator concatenates the strings and when they are Boolean values, it performs a particular operation.

In Python, `True` is equivalent to `1` and `False` is equivalent to `0`. This is why the result is  $1 + 0 = 1$

**Subtraction: -**

```
>>> 5 - 6
```

```
-1
```

```
>>> 10 - 3
```

```
7
```

```
>>> 5 - 6
```

```
-1
```

```
>>> 4.5 - 5.6 - 2.3
```

```
-3.3999999999999995
```

```
>>> 4.5 - 7
```

```
-2.5
```

```
>>> - 7.8 - 6.2
```

```
-14.0
```

**Multiplication: \***

```
>>> 5 * 6
```

```
30
```

```
>>> 6 * 7
```

```
42
```

```
>>> 10 * 100
```

```
1000
```

```
>>> 4 * 0
```

```
0
```

```
>>> 3.4 * 6.8
```

```
23.119999999999997
```

```
>>> 4 * (-6)
```

```
-24
```

```
>>> (-6) * (-8)
```

```
48
```

```
>>> "Hello" * 4
```


```
'HelloHelloHelloHello'
```

```
>>> "Hello" * 0
```

```
''
```

```
>>> "Hello" * -1
```

```
''
```

 **Tip:** you can "multiply" a string by an integer to repeat the string a given number of times.

**Exponentiation: \*\***

```
>>> 6 ** 8
```

```
1679616
```

```
>>> 5 ** 2
```

```
25
```

```
>>> 4 ** 0
```

```
1
```

```
>>> 16 ** (1/2)
4.0
```

```
>>> 16 ** (0.5)
4.0
```

```
>>> 125 ** (1/3)
4.999999999999999
```

```
>>> 4.5 ** 2.3
31.7971929089206
```

```
>>> 3 ** (-1)
0.3333333333333333
```

**Division: /**

```
>>> 25 / 5
5.0
```

```
>>> 3 / 6
0.5
```

```
>>> 0 / 5
0.0
```

```
>>> 2467 / 4673
0.5279263856195163
```

```
>>> 1 / 2
0.5
```

```
>>> 4.5 / 3.5
1.2857142857142858
```

```
>>> 6 / 7
```



```
0.8571428571428571
```

```
>>> -3 / -4  
0.75
```

```
>>> 3 / -4  
-0.75
```

```
>>> -3 / 4  
-0.75
```

💡 **Tip:** this operator returns a `float` as the result, even if the decimal part is `.0`

If you try to divide by `0`, you will get a `ZeroDivisionError`:

```
>>> 5 / 0  
Traceback (most recent call last):  
  File "<pyshell#109>", line 1, in <module>  
    5 / 0
```

```
ZeroDivisionError: division by zero
```

### Integer Division: //

This operator returns an integer if the operands are integers. If they are floats, the result will be a float with `.0` as the decimal part because it truncates the decimal part.

```
>>> 5 // 6  
0
```

```
>>> 8 // 2  
4
```

```
>>> -4 // -5  
0
```

```
>>> -5 // 8  
-1
```

```
>>> 0 // 5
```

0

```
>>> 156773 // 356
```

440

**Modulo: %**

```
>>> 1 % 5
```

1

```
>>> 2 % 5
```

2

```
>>> 3 % 5
```

3

```
>>> 4 % 5
```

4

```
>>> 5 % 5
```

0

```
>>> 5 % 8
```

5

```
>>> 3 % 1
```

0

```
>>> 15 % 3
```

0

```
>>> 17 % 8
```

1

```
>>> 2568 % 4
```

0

```
>>> 245 % 15
```

```
5
```

```
>>> 0 % 6
```

```
0
```

```
>>> 3.5 % 2.4
```

```
1.1
```

```
>>> 6.7 % -7.8
```

```
-1.0999999999999996
```

```
>>> 2.3 % 7.5
```

```
2.3
```

### **Comparison Operators**

These operators are:

- Greater than: >
- Greater than or equal to: >=
- Less than: <
- Less than or equal to: <=
- Equal to: ==
- Not Equal to: !=

These comparison operators make expressions that evaluate to either `True` or `False`. Here we have are some examples:

```
>>> 5 > 6
```

```
False
```

```
>>> 10 > 8
```

```
True
```

```
>>> 8 > 8
```

```
False
```

```
>>> 8 >= 5
```

```
True
```

```
>>> 8 >= 8
```

```
True
```

```
>>> 5 < 6
```

```
True
```

```
>>> 10 < 8
```

```
False
```

```
>>> 8 < 8
```

```
False
```

```
>>> 8 <= 5
```

```
False
```

```
>>> 8 <= 8
```

```
True
```

```
>>> 8 <= 10
```

```
True
```

```
>>> 56 == 56
```

```
True
```

```
>>> 56 == 78
```

```
False
```

```
>>> 34 != 59
```

```
True
```

```
>>> 67 != 67
```

```
False
```

We can also use them to compare strings based on their alphabetical order:

```
>>> "Hello" > "World"
False
>>> "Hello" >= "World"
False
>>> "Hello" < "World"
True
>>> "Hello" <= "World"
True
>>> "Hello" == "World"
False
>>> "Hello" != "World"
True
```

We typically use them to compare the values of two or more variables:

```
>>> a = 1
>>> b = 2

>>> a < b
True

>>> a <= b
True

>>> a > b
False

>>> a >= b
False

>>> a == b
False
```

```
>>> a != b
```

```
True
```

💡 **Tip:** notice that the comparison operator is `==` while the assignment operator is `=`. Their effect is different. `==` returns `True` or `False` while `=` assigns a value to a variable.

### Comparison Operator Chaining

In Python, we can use something called "comparison operator chaining" in which we chain the comparison operators to make more than one comparison more concisely.

For example, this checks if `a` is less than `b` and if `b` is less than `c`:

```
a < b < c
```

Here we have some examples:

```
>>> a = 1
```

```
>>> b = 2
```

```
>>> c = 3
```

```
>>> a < b < c
```

```
True
```

```
>>> a > b > c
```

```
False
```

```
>>> a <= b <= c
```

```
True
```

```
>>> a >= b >= c
```

```
False
```

```
>>> a >= b > c
```

```
False
```

```
>>> a <= b < c
```

```
True
```

## Logical Operators

There are three logical operators in Python: `and`, `or`, and `not`. Each one of these operators has its own truth table and they are essential to work with conditionals.

The `and` operator:

```
>>> True and True
```

```
True
```

```
>>> True and False
```

```
False
```

```
>>> False and True
```

```
False
```

```
>>> False and False
```

```
False
```

The `or` operator:

```
>>> True or True
```

```
True
```

```
>>> True or False
```

```
True
```

```
>>> False or True
```

```
True
```

```
>>> False or False
```

```
False
```

The `not` operator:

```
>>> not True
```

```
False
```

```
>>> not False
```

```
True
```

These operator are used to form more complex expressions that combine different operators and variables.

For example:

```
>>> a = 6
```

```
>>> b = 3
```

```
>>> a < 6 or b > 2
```

```
True
```

```
>>> a >= 3 and b >= 1
```

```
True
```

```
>>> (a + b) == 9 and b > 1
```

```
True
```

```
>>> ((a % 3) < 2) and ((a + b) == 3)
```

```
False
```

### Assignment Operators

Assignment operators are used to assign a value to a variable.

They are: =, +=, -=, \*=, /=, //=, \*\*=

- The = operator assigns the value to the variable.
- The other operators perform an operation with the current value of the variable and the new value and assigns the result to the same variable.

For example:

```
>>> x = 3
```

```
>>> x
```

```
3
```



```
>>> x += 15
```

```
>>> x
```

```
18
```

```
>>> x -= 2
```

```
>>> x
```

```
16
```

```
>>> x *= 2
```

```
>>> x
```

```
32
```

```
>>> x %= 5
```

```
>>> x
```

```
2
```

```
>>> x /= 1
```

```
>>> x
```

```
2.0
```

```
>>> x //= 2
```


```
>>> x
```

```
1.0
```

```
>>> x **= 5
```

```
>>> x
```

```
1.0
```

 **Tips:** these operators perform bitwise operations before assigning the result to the variable: `&=`, `|=`, `^=`, `>>=`, `<<=`.

### Membership Operators

You can check if an element is in a sequence or not with the operators: `in` and `not in`. The result will be either `True` or `False`. For example:

```
>>> 5 in [1, 2, 3, 4, 5]
True
```

```
>>> 8 in [1, 2, 3, 4, 5]
False
```

```
>>> 5 in (1, 2, 3, 4, 5)
True
```

```
>>> 8 in (1, 2, 3, 4, 5)
False
```

```
>>> "a" in {"a": 1, "b": 2}
True
```

```
>>> "c" in {"a": 1, "b": 2}
False
```

```
>>> "h" in "Hello"
False
```

```
>>> "H" in "Hello"
True
```

```
>>> 5 not in [1, 2, 3, 4, 5]
False
```

```
>>> 8 not in (1, 2, 3, 4, 5)
True
```

```
>>> "a" not in {"a": 1, "b": 2}
False
```

```
>>> "c" not in {"a": 1, "b": 2}
True
```

```
>>> "h" not in "Hello"
True
```

```
>>> "H" not in "Hello"
False
```

We typically use them with variables that store sequences, like in this example:

```
>>> message = "Hello, World!"

>>> "e" in message
True
```

## ◆ Conditionals in Python

Now let's see how we can write conditionals to make certain parts of our code run (or not) based on whether a condition is `True` or `False`.

### `if` statements in Python

This is the syntax of a basic `if` statement:

```
if <condition>:
    <code>
```

If the condition is `True`, the code will run. Else, if it's `False`, the code will not run.

💡 **Tip:** there is a colon (`:`) at the end of the first line and the code is indented. This is essential in Python to make the code belong to the conditional.

Here we have some examples:

### False Condition

```
x = 5

if x > 9:
    print("Hello, World!")
```

The condition is `x > 9` and the code is `print("Hello, World!")`.

In this case, the condition is `False`, so there is no output.

### True Condition

Here we have another example. Now the condition is `True`:

```
color = "Blue"
```

```
if color == "Blue":  
    print("This is my favorite color")
```

The output is:

```
"This is my favorite color"
```

### Code After the Conditional

Here we have an example with code that runs after the conditional has been completed. Notice that the last line is not indented, which means that it doesn't belong to the conditional.

```
x = 5
```

```
if x > 9:  
    print("Hello!")
```

```
print("End")
```

In this example, the condition `x > 9` is `False`, so the first print statement doesn't run but the last print statement runs because it is not part of the conditional, so the output is:

```
End
```

However, if the condition is `True`, like in this example:

```
x = 15
```

```
if x > 9:  
    print("Hello!")
```

```
print("End")
```

The output will be:

```
Hello!
```

```
End
```

## Examples of Conditionals

This is another example of a conditional:

```
favorite_season = "Summer"
```

```
if favorite_season == "Summer":  
    print("That is my favorite season too!")
```

In this case, the output will be:

```
That is my favorite season too!
```

But if we change the value of `favorite_season`:

```
favorite_season = "Winter"
```

```
if favorite_season == "Summer":  
    print("That is my favorite season too!")
```


There will be no output because the condition will be `False`.

## `if/else` statements in Python

We can add an `else` clause to the conditional if we need to specify what should happen when the condition is `False`.

This is the general syntax:

```
if <condition>:  
    <code>  
else:  
    <code>
```

 **Tip:** notice that the two code blocks are indented (`if` and `else`). This is essential for Python to be able to differentiate between the code that belongs to the main program and the code that belongs to the conditional.

Let's see an example with the `else` clause:

## True Condition

```
x = 15
```

```
if x > 9:
    print("Hello!")
else:
    print("Bye!")
```

```
print("End")
```

The output is:

```
Hello!
```

```
End
```

When the condition of the `if` clause is `True`, this clause runs.

The `else` clause doesn't run.

### False Condition

Now the `else` clause runs because the condition is `False`.

```
x = 5
```

```
if x > 9:
    print("Hello!")
else:
    print("Bye!")
```

```
print("End")
```

Now the output is:

```
Bye!
```

```
End
```

### `if/elif/else` statements in Python

To customize our conditionals even further, we can add one or more `elif` clauses to check and handle multiple conditions. Only the code of the first condition that evaluates to `True` will run.

 **Tip:** `elif` has to be written after `if` and before `else`.

### First Condition True

```
x = 5
```

```
if x < 9:
    print("Hello!")
elif x < 15:
    print("It's great to see you")
else:
    print("Bye!")
```

```
print("End")
```

We have two conditions  $x < 9$  and  $x < 15$ . Only the code block from the first condition that is `True` from top to bottom will be executed.

In this case, the output is:

```
Hello!
```

```
End
```

Because the first condition is `True`:  $x < 9$ .

### **Second Condition True**

If the first condition is `False`, then the second condition will be checked.

In this example, the first condition  $x < 9$  is `False` but the second condition  $x < 15$  is `True`, so the code that belongs to this clause will run.

```
x = 13
```

```
if x < 9:
    print("Hello!")
elif x < 15:
    print("It's great to see you")
else:
    print("Bye!")
```

```
print("End")
```

The output is:

```
It's great to see you
End
```

### All Conditions are False

If all conditions are `False`, then the `else` clause will run:

```
x = 25
```

```
if x < 9:
    print("Hello!")
elif x < 15:
    print("It's great to see you")
else:
    print("Bye!")
```

```
print("End")
```

The output will be:

```
Bye!
End
```

### Multiple `elif` Clauses

We can add as many `elif` clauses as needed. This is an example of a conditional with two `elif` clauses:

```
if favorite_season == "Winter":
    print("That is my favorite season too")
elif favorite_season == "Summer":
    print("Summer is amazing")
elif favorite_season == "Spring":
    print("I love spring")
else:
    print("Fall is my mom's favorite season")
```

Each condition will be checked and only the code block of the first condition that evaluates to `True` will run. If none of them are `True`, the `else` clause will run.

## ◆ For Loops in Python



Now you know how to write conditionals in Python, so let's start diving into loops. For loops are amazing programming structures that you can use to repeat a code block a specific number of times.


This is the basic syntax to write a for loop in Python:

```
for <loop_variable> in <iterable>:  
    <code>
```

The iterable can be a list, tuple, dictionary, string, the sequence returned by range, a file, or any other type of iterable in Python. We will start with `range()`.

### **The `range()` function in Python**

This function returns a sequence of integers that we can use to determine how many iterations (repetitions) of the loop will be completed. The loop will complete one iteration per integer.

 **Tip:** Each integer is assigned to the loop variable one at a time per iteration.

This is the general syntax to write a for loop with `range()`:

```
for <loop_variable> in range(<start>, <stop>, <step>):  
    <code>
```

As you can see, the range function has three parameters:

- **start:** where the sequence of integers will start. By default, it's 0.
- **stop:** where the sequence of integers will stop (without including this value).
- **step:** the value that will be added to each element to get the next element in the sequence. By default, it's 1.

You can pass 1, 2, or 3 arguments to `range()`:

- With 1 argument, the value is assigned to the `stop` parameter and the default values for the other two parameters are used.

- With 2 arguments, the values are assigned to the `start` and `stop` parameters and the default value for `step` is used.
- With 3 arguments, the values are assigned to the `start`, `stop`, and `step` parameters (in order).

Here we have some examples with **one parameter**:

```
for i in range(5):  
    print(i)
```

Output:

```
0  
1  
2  
3  
4
```



**Tip:** the loop variable is updated automatically.

```
>>> for j in range(15):  
    print(j * 2)
```

Output:

```
0  
2  
4  
6  
8  
10  
12  
14  
16  
18  
20  
22  
24  
26  
28
```

In the example below, we repeat a string as many times as indicated by the value of the loop variable:

```
>>> for num in range(8):  
    print("Hello" * num)
```

Output:


```
Hello  
HelloHello  
HelloHelloHello  
HelloHelloHelloHello  
HelloHelloHelloHelloHello  
HelloHelloHelloHelloHelloHello  
HelloHelloHelloHelloHelloHelloHello  
HelloHelloHelloHelloHelloHelloHelloHello
```

We can also use for loops with built-in data structures such as lists:

```
>>> my_list = ["a", "b", "c", "d"]  
  
>>> for i in range(len(my_list)):  
    print(my_list[i])
```

Output:

```
a  
b  
c  
d
```

 **Tip:** when you use `range(len(<seq>))`, you get a sequence of numbers that goes from 0 up to `len(<seq>) - 1`. This represents the sequence of valid indices.

These are some examples with **two parameters**:

```
>>> for i in range(2, 10):  
    print(i)
```

Output:

2  
3  
4  
5  
6  
7  
8  
9

### Code:

```
>>> for j in range(2, 5):  
    print("Python" * j)
```

### Output:

PythonPython

PythonPythonPython

PythonPythonPythonPython

### Code:

```
>>> my_list = ["a", "b", "c", "d"]  
  
>>> for i in range(2, len(my_list)):  
    print(my_list[i])
```

### Output:

c

d

### Code:

```
>>> my_list = ["a", "b", "c", "d"]  
  
>>> for i in range(2, len(my_list)-1):  
    my_list[i] *= i
```

Now the list is: ['a', 'b', 'cc', 'd']

These are some examples with **three parameters**:

```
>>> for i in range(3, 16, 2):  
    print(i)
```

### Output:

3  
5  
7  
9  
11  
13  
15

### Code:

```
>>> for j in range(10, 5, -1):  
    print(j)
```

### Output:

10  
9  
8  
7  
6

### Code:

```
>>> my_list = ["a", "b", "c", "d", "e", "f", "g"]  
  
>>> for i in range(len(my_list)-1, 2, -1):  
    print(my_list[i])
```

### Output:

g  
f  
e  
d

## How to Iterate over Iterables in Python

We can iterate directly over iterables such as lists, tuples, dictionaries, strings, and files using for loops. We will get each one of their elements one at a time per iteration. This is very helpful to work with them directly.

Let's see some examples:

### Iterate Over a String

If we iterate over a string, its characters will be assigned to the loop variable one by one (including spaces and symbols).

```
>>> message = "Hello, World!"
```

```
>>> for char in message:  
    print(char)
```

```
H  
e  
l  
l  
o  
,  
  
W  
o  
r  
l  
d  
!
```

We can also iterate over modified copies of the string by calling a string method where we specify the iterable in the for loop. This will assign the copy of the string as the iterable that will be used for the iterations, like this:

```
>>> word = "Hello"
```

```
>>> for char in word.lower(): # calling the string method  
    print(char)
```

h  
e  
l  
l  
o

```
>>> word = "Hello"
```

```
>>> for char in word.upper(): # calling the string method
    print(char)
```

H  
E  
L  
L  
O

## Iterate Over Lists and Tuples

```
>>> my_list = [2, 3, 4, 5]
```

```
>>> for num in my_list:
    print(num)
```

The output is:

2  
3  
4  
5

## Code:

```
>>> my_list = (2, 3, 4, 5)
```

```
>>> for num in my_list:
    if num % 2 == 0:
        print("Even")
    else:
```

```
print("Odd")
```

Output:

Even

Odd

Even

Odd


## Iterate Over the Keys, Values, and Key-Value Pairs of Dictionaries

We can iterate over the keys, values, and key-value pairs of a dictionary by calling specific dictionary methods. Let's see how.

**To iterate over the keys, we write:**

```
for <var> in <dictionary_variable>:  
    <code>
```

We just write the name of the variable that stores the dictionary as the iterable.

 **Tip:** you can also write `<dictionary_variable>.keys()` but writing the name of the variable directly is more concise and it works exactly the same.

For example:

```
>>> my_dict = {"a": 1, "b": 2, "c": 3}
```

```
>>> for key in my_dict:  
    print(key)
```

a

b

c

 **Tip:** you can assign any valid name to the loop variable.

**To iterate over the values, we use:**

```
for <var> in <dictionary_variable>.values():
```



<code>

For example:

```
>>> my_dict = {"a": 1, "b": 2, "c": 3}
```

```
>>> for value in my_dict.values():  
    print(value)
```


1

2

3

To **iterate over the key-value pairs**, we use:

```
for <key>, <value> in <dictionary_variable>.items():  
    <code>
```

 **Tip:** we are defining two loop variables because we want to assign the key and the value to variables that we can use in the loop.

```
>>> my_dict = {"a": 1, "b": 2, "c": 3}
```

```
>>> for key, value in my_dict.items():  
    print(key, value)
```

a 1

b 2

c 3

If we define only one loop variable, this variable will contain a tuple with the key-value pair:

```
>>> my_dict = {"a": 1, "b": 2, "c": 3}  
>>> for pair in my_dict.items():  
    print(pair)
```

```
('a', 1)
('b', 2)
('c', 3)
```

## Break and Continue in Python

Now you know how to iterate over sequences in Python. We also have loop control statements to customize what happens when the loop runs: `break` and `continue`.

### The Break Statement

The `break` statement is used to stop the loop immediately. When a `break` statement is found, the loop stops and the program returns to its normal execution beyond the loop. In the example below, we stop the loop when an even element is found.

```
>>> my_list = [1, 2, 3, 4, 5]

>>> for elem in my_list:
    if elem % 2 == 0:
        print("Even:", elem)
        print("break")
        break
    else:
        print("Odd:", elem)
```

```
Odd: 1
Even: 2
break
```

### The Continue Statement

The `continue` statement is used to skip the rest of the current iteration.

When it is found during the execution of the loop, the current iteration stops and a new one begins with the updated value of the loop variable.

In the example below, we skip the current iteration if the element is even and we only print the value if the element is odd:

```
>>> my_list = [1, 2, 3, 4, 5]
```

```
>>> for elem in my_list:
    if elem % 2 == 0:
        print("continue")
        continue
    print("Odd:", elem)
```

```
Odd: 1
continue
Odd: 3
continue
Odd: 5
```

### The zip() function in Python

`zip()` is an amazing built-in function that we can use in Python to iterate over multiple sequences at once, getting their corresponding elements in each iteration.

We just need to pass the sequences as arguments to the `zip()` function and use this result in the loop.

For example:

```
>>> my_list1 = [1, 2, 3, 4]
```


```
>>> my_list2 = [5, 6, 7, 8]
```

```
>>> for elem1, elem2 in zip(my_list1, my_list2):
    print(elem1, elem2)
```

```
1 5
2 6
3 7
4 8
```

## The enumerate() Function in Python

You can also keep track of a counter while the loop runs with the `enum()` function. It is commonly used to iterate over a sequence and get the corresponding index.

 **Tip:** By default, the counter starts at 0.

For example:

```
>>> my_list = [5, 6, 7, 8]
```

```
>>> for i, elem in enumerate(my_list):
    print(i, elem)
```

```
0 5
1 6
2 7
3 8
```

```
>>> word = "Hello"
```

```
>>> for i, char in enumerate(word):
    print(i, char)
```

```
0 H
1 e
2 l
3 l
4 o
```

If you start the counter from 0, you can use the index and the current value in the same iteration to modify the sequence:

```
>>> my_list = [5, 6, 7, 8]
```

```
>>> for index, num in enumerate(my_list):  
    my_list[index] = num * 3
```

```
>>> my_list  
[15, 18, 21, 24]
```

You can start the counter from a different number by passing a second argument to `enumerate()`:


```
>>> word = "Hello"
```

```
>>> for i, char in enumerate(word, 2):  
    print(i, char)
```

```
2 H  
3 e  
4 l  
5 l  
6 o
```

### The else Clause

For loops also have an `else` clause. You can add this clause to the loop if you want to run a specific block of code when the loop completes all its iterations without finding the `break` statement.

 **Tip:** if `break` is found, the `else` clause doesn't run and if `break` is not found, the `else` clause runs.

In the example below, we try to find an element greater than 6 in the list. That element is not found, so `break` doesn't run and the `else` clause runs.

```
my_list = [1, 2, 3, 4, 5]
```

```
for elem in my_list:
```

```
if elem > 6:
    print("Found")
    break
else:
    print("Not Found")
```

The output is:

Not Found

However, if the `break` statement runs, the `else` clause doesn't run. We can see this in the example below:

```
my_list = [1, 2, 3, 4, 5, 8] # Now the list has the value 8
```

```
for elem in my_list:
    if elem > 6:
        print("Found")
        break
else:
    print("Not Found")
```

The output is:

Found

## ◆ While Loops in Python

While loops are similar to for loops in that they let us repeat a block of code. The difference is that while loops run while a condition is `True`.

In a while loop, we define the condition, not the number of iterations. The loop stops when the condition is `False`.

This is the general syntax of a while loop:

```
while <condition>:
    <code>
```

💡 **Tip:** in while loops, you must update the variables that are part of the condition to make sure that the condition will eventually become `False`.

For example:

```
>>> x = 6
```

```
>>> while x < 15:  
    print(x)  
    x += 1
```

```
6
```

```
7
```

```
8
```

```
9
```

```
10
```

```
11
```

```
12
```

```
13
```

```
14
```

```
>>> x = 4
```

```
>>> while x >= 0:  
    print("Hello" * x)  
    x -= 1
```

```
HelloHelloHelloHello
```

```
HelloHelloHello
```

```
HelloHello
```

```
Hello
```

```
>>> num = 5
```

```
>>> while num >= 1:  
    print("*" * num)  
    num -= 2
```

```
*****
```

```
***
```

```
*
```

## Break and Continue

We can also use `break` and `continue` with while loops and they both work exactly the same:

- `break` stops the while loop immediately.
- `continue` stops the current iteration and starts the next one.

For example:

```
>>> x = 5
```

```
>>> while x < 15:
    if x % 2 == 0:
        print("Even:", x)
        break
    print(x)
    x += 1
```

```
5
```

```
Even: 6
```

```
>>> x = 5
```

```
>>> while x < 15:
    if x % 2 == 0:
        x += 1
        continue
    print("Odd:", x)
    x += 1
```

```
Odd: 5
```

```
Odd: 7
```



Odd: 9

Odd: 11

Odd: 13

### The `else` Clause

We can also add an `else` clause to a `while` loop. If `break` is found, the `else` clause doesn't run but if the `break` statement is not found, the `else` clause runs.

In the example below, the `break` statement is not found because none of the numbers are even before the condition becomes `False`, so the `else` clause runs.

```
x = 5
```

```
while x < 15:
    if x % 2 == 0:
        print("Even number found")
        break
    print(x)
    x += 2
else:
    print("All numbers were odd")
```

This is the output:

5

7

9

11

13

All numbers were odd

But in this version of the example, the `break` statement is found and the `else` clause doesn't run:

```
x = 5
```

```
while x < 15:
    if x % 2 == 0:
        print("Even number found")
```

```

        break
    print(x)
    x += 1 # Now we are incrementing the value by 1
else:
    print("All numbers were odd")

```


The output is:

5

Even number found

### Infinite While Loops

When we write and work with while loops, we can have something called an "infinite loop." If the condition is never `False`, the loop will never stop without external intervention. This usually happens when the variables in the condition are not updated properly during the execution of the loop.

 **Tip:** you must make the necessary updates to these variables to make sure that the condition will eventually evaluate to `False`. For example:

```
>>> x = 5
```

```
>>> while x > 2:
    print(x)
```

5  
5  
5  
5  
5  
5  
5  
5

5

.  
. .  
.

# The output continues indefinitely

💡 **Tip:** to stop this process, type `CTRL + C`. You should see a `KeyboardInterrupt` message.

## ◆ Nested Loops in Python

We can write for loops within for loops and while loops within while loops. These inner loops are called nested loops.

💡 **Tip:** the inner loop runs for each iteration of the outer loop.

### Nested For Loops in Python

```
>>> for i in range(3):  
    for j in range(2):  
        print(i, j)
```

```
0 0  
0 1  
1 0  
1 1  
2 0  
2 1
```

If we add print statements, we can see what is happening behind the scenes:

```
>>> for i in range(3):  
    print("==> Outer Loop")  
    print(f'i = {i}')  
    for j in range(2):  
        print("Inner Loop")  
        print(f'j = {j}')
```

====> Outer Loop

i = 0

Inner Loop

j = 0

Inner Loop

j = 1

====> Outer Loop

i = 1

Inner Loop

j = 0

Inner Loop

j = 1

====> Outer Loop

i = 2

Inner Loop

j = 0

Inner Loop

j = 1

The inner loop completes two iterations per iteration of the outer loop. The loop variables are updated when a new iteration starts.

This is another example:

```
>>> num_rows = 5
```

```
>>> for i in range(5):
    for num_cols in range(num_rows-i):
        print("*", end="")
    print()
```

\*\*\*\*\*

\*\*\*\*

```
***  
**  
*
```

## Nested While Loops in Python

Here we have an example of nested while loops. In this case, we have to update the variables that are part of each condition to guarantee that the loops will stop.

```
>>> i = 5  
  
>>> while i > 0:  
    j = 0  
    while j < 2:  
        print(i, j)  
        j += 1  
    i -= 1
```


```
5 0  
5 1  
4 0  
4 1  
3 0  
3 1  
2 0  
2 1  
1 0  
1 1
```

 **Tip:** we can also have for loops within while loops and while loops within for loops.

## ◆ Functions in Python

In Python, we can define functions to make our code reusable, more readable, and organized. This is the basic syntax of a Python function:

```
def <function_name>(<param1>, <param2>, ...):  
    <code>
```

 **Tip:** a function can have zero, one, or multiple parameters.


### Function with No Parameters in Python

A function with no parameters has an empty pair of parentheses after its name in the function definition. For example:

```
def print_pattern():  
    size = 4  
    for i in range(size):  
        print("*" * size)
```

This is the output when we call the function:

```
>>> print_pattern()  
****  
  
****  
  
****  
  
****
```

 **Tip:** You have to write an empty pair of parentheses after the name of the function to call it.

### Function with One Parameter in Python

A function with one or more parameters has a list of parameters surrounded by parentheses after its name in the function definition:

```
def welcome_student(name):  
    print(f"Hi, {name}! Welcome to class.")
```

When we call the function, we just need to pass one value as argument and that value will be replaced where we use the parameter in the function definition:

```
>>> welcome_student("Nora")
```

```
Hi, Nora! Welcome to class.
```

Here we have another example – a function that prints a pattern made with asterisks. You have to specify how many rows you want to print:

```
def print_pattern(num_rows):  
    for i in range(num_rows):  
        for num_cols in range(num_rows-i):  
            print("*", end="")  
        print()
```

You can see the different outputs for different values of `num_rows`:

```
>>> print_pattern(3)
```

```
***  
**  
*
```

```
>>> print_pattern(5)
```

```
*****  
****  
***  
**  
*
```

```
>>> print_pattern(8)
```

```
*****  
*****  
*****  
*****  
****  
***
```

```
**  
*
```

## Functions with Two or More Parameters in Python

To define two or more parameters, we just separate them with a comma:

```
def print_sum(a, b):  
    print(a + b)
```

Now when we call the function, we must pass two arguments:

```
>>> print_sum(4, 5)  
9
```

```
>>> print_sum(8, 9)  
17
```

```
>>> print_sum(0, 0)  
0
```

```
>>> print_sum(3, 5)  
8
```

We can adapt the function that we just saw with one parameter to work with two parameters and print a pattern with a customized character:

```
def print_pattern(num_rows, char):  
    for i in range(num_rows):  
        for num_cols in range(num_rows-i):  
            print(char, end="")  
        print()
```

You can see the output with the customized character is that we call the function passing the two arguments:

```
>>> print_pattern(5, "A")
```



```
AAAAA
```

```
AAAA
```

```
AAA
```

```
AA
```

```
A
```

```
>>> print_pattern(8, "%")
```

```
%%%%%%%%%
```

```
%%%%%%%%%
```

```
%%%%%%%%%
```

```
%%%%%%
```

```
%%%%%%
```

```
%%%%
```

```
%%%
```

```
%
```

```
>>> print_pattern(10, "#")
```

```
#####
```

```
#####
```

```
#####
```

```
#####
```

```
#####
```

```
#####
```

```
####
```

```
###
```

```
##
```

```
#
```

## How to Return a Value in Python

Awesome. Now you know how to define a function, so let's see how you can work with return statements.

We will often need to return a value from a function. We can do this with the `return` statement in Python. We just need to write this in the function definition:

```
return <value_to_return>
```

 **Tip:** the function stops immediately when `return` is found and the value is returned.

Here we have an example:

```
def get_rectangle_area(length, width):  
    return length * width
```

Now we can call the function and assign the result to a variable because the result is returned by the function:

```
>>> area = get_rectangle_area(4, 5)  
>>> area
```

20

We can also use `return` with a conditional to return a value based on whether a condition is `True` or `False`.


In this example, the function returns the first even element found in the sequence:

```
def get_first_even(seq):  
    for elem in seq:  
        if elem % 2 == 0:  
            return elem  
    else:  
        return None
```

If we call the function, we can see the expected results:

```
>>> value1 = get_first_even([2, 3, 4, 5])  
>>> value1  
2  
>>> value2 = get_first_even([3, 5, 7, 9])  
>>> print(value2)
```

None


 **Tip:** if a function doesn't have a `return` statement or doesn't find one during its execution, it returns `None` by default.

The [Style Guide for Python Code](#) recommends using return statements consistently. It mentions that we should:

Be consistent in return statements. Either all return statements in a function should return an expression, or none of them should. If any return statement returns an expression, any return statements where no value is returned should explicitly state this as return None, and an explicit return statement should be present at the end of the function (if reachable)

### Default Arguments in Python

We can assign default arguments for the parameters of our function. To do this, we just need to write `<parameter>=<value>` in the list of parameters.

 **Tip:** The [Style Guide for Python Code](#) mentions that we shouldn't "use spaces around the = sign when used to indicate a keyword argument."

In this example, we assign the default value 5 to the parameter `b`. If we omit this value when we call the function, the default value will be used.

```
def print_product(a, b=5):  
    print(a * b)
```

If we call the function without this argument, you can see the output:

```
>>> print_product(4)
```


```
20
```

We confirm that the default argument 5 was used in the operation.

But we can also assign a custom value for `b` by passing a second argument:

```
>>> print_product(3, 4)
```

```
12
```

 **Tip:** parameters with default arguments have to be defined at the end of the list of parameters. Else, you will see this

**error:** `SyntaxError: non-default argument follows default argument.`

Here we have another example with the function that we wrote to print a pattern. We assign the default value "\*" to the `char` parameter.

```
def print_pattern(num_rows, char="*"):  
    for i in range(num_rows):  
        for num_cols in range(num_rows-i):  
            print(char, end="")  
        print()
```

Now we have the option to use the default value or customize it:

```
>>> print_pattern(5)
```

```
*****  
****  
***  
**  
*
```

```
>>> print_pattern(6, "&")
```

```
&&&&&&  
&&&&&  
&&&&  
&&&  
&&  
&  
&
```

## ◆ Recursion in Python

A recursive function is a function that calls itself. These functions have a base case that stops the recursive process and a recursive case that continues the recursive process by making another recursive call.

Here we have some examples in Python:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

## Recursive Factorial Function

```
def fibonacci(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

## The Fibonacci Function

```
def find_power(a, b):
    if b == 0:
        return 1
    else:
        return a * find_power(a, b-1)
```

## Find a Power Recursively

### ◆ Exception Handling in Python

An error or unexpected event that occurs while a program is running is called an **exception**. Thanks to the elements that we will see in just a moment, we can avoid terminating the program abruptly when this occurs.

Let's see the types of exceptions in Python and how we can handle them.

### Common Exceptions in Python

This is a list of common exceptions in Python and why they occur:

- **ZeroDivisionError:** raised when the second argument of a division or modulo operation is zero.

```
>>> 5 / 0
```

Traceback (most recent call last):

```
File "<pyshell#0>", line 1, in <module>
```

```
5 / 0
```

ZeroDivisionError: division by zero

```
>>> 7 // 0
```

Traceback (most recent call last):

File "<pyshell#1>", line 1, in <module>

```
7 // 0
```

ZeroDivisionError: integer division or modulo by zero

```
>>> 8 % 0
```

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

```
8 % 0
```

ZeroDivisionError: integer division or modulo by zero

- **IndexError:** raised when we try to use an invalid index to access an element of a sequence.

```
>>> my_list = [3, 4, 5, 6]
```

```
>>> my_list[15]
```

Traceback (most recent call last):

File "<pyshell#4>", line 1, in <module>

```
my_list[15]
```

IndexError: list index out of range

- **KeyError:** raised when we try to access a key-value pair that doesn't exist because the key is not in the dictionary.

```
>>> my_dict = {"a": 1, "b": 2, "c": 3}
```

```
>>> my_dict["d"]
```

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

```
my_dict["d"]
```

KeyError: 'd'

- **NameError:** raised when we use a variable that has not been defined previously.

```
>>> b
```

Traceback (most recent call last):

```
File "<pyshell#8>", line 1, in <module>
    b
```

NameError: name 'b' is not defined

- **RecursionError:** raised when the interpreter detects that the maximum recursion depth is exceeded. This usually occurs when the process never reaches the base case.

In the example below, we will get a `RecursionError`.

The `factorial` function is implemented recursively but the argument passed to the recursive call is `n` instead of `n-1`. Unless the value is already 0 or 1, the base case will not be reached because the argument is not being decremented, so the process will continue and we will get this error.

```
>>> def factorial(n):
        if n == 0 or n == 1:
            return 1
        else:
            return n * factorial(n)
```

```
>>> factorial(5)
```

Traceback (most recent call last):

```
File "<pyshell#6>", line 1, in <module>
    factorial(5)
File "<pyshell#5>", line 5, in factorial
    return n * factorial(n)
File "<pyshell#5>", line 5, in factorial
    return n * factorial(n)
File "<pyshell#5>", line 5, in factorial
    return n * factorial(n)
[Previous line repeated 1021 more times]
File "<pyshell#5>", line 2, in factorial
    if n == 0 or n == 1:
```

RecursionError: maximum recursion depth exceeded in comparison

💡 **Tip:** to learn more about these exceptions, I recommend reading [this article](#) from the documentation.

### **try / except in Python**

We can use try/except in Python to catch the exceptions when they occur and handle them appropriately. This way, the problem can terminate appropriately or even recover from the exception.

This is the basic syntax:

```
try:
    <code_that_may_raise_an_exception>
except:
    <code_to_handle_the_exception_if_it_occurs>
```

For example, if we take user input to access an element in a list, the input might not be a valid index, so an exception could be raised:

```
index = int(input("Enter the index: "))
```

```
try:
    my_list = [1, 2, 3, 4]
    print(my_list[index])
except:
    print("Please enter a valid index.")
```

If we enter an invalid value like 15, the output will be:

```
Please enter a valid index.
```

Because the `except` clause runs. However, if the value is valid, the code in `try` will run as expected.

Here we have another example:

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))
```



```

try:
    division = a / b
    print(division)
except:
    print("Please enter valid values.")

```

The output is:

```

Enter a: 5
Enter b: 0

```

```

Please enter valid values.

```

## How to Catch a Specific Type of Exception in Python

Instead of catching and handling all possible exceptions that could occur in the `try` clause, we could catch and handle a specific type of exception. We just need to specify the type of the exception after the `except` keyword:

```

try:
    <code_that_may_raise_an_exception>
except <exception_type>:
    <code_to_handle_an_exception_if_it_occurs>

```

For example:

```

index = int(input("Enter the index: "))

```

```

try:
    my_list = [1, 2, 3, 4]
    print(my_list[index])
except IndexError: # specify the type
    print("Please enter a valid index.")
a = int(input("Enter a: "))
b = int(input("Enter b: "))

```

```

try:
    division = a / b
    print(division)

```

```
except ZeroDivisionError: # specify the type
    print("Please enter valid values.")
```

## How to Assign a Name to the Exception Object in Python

We can specify a name for the exception object by assigning it to a variable that we can use in the `except` clause. This will let us access its description and attributes.

We only need to add `as <name>`, like this:

```
try:
    <code_that_may_raise_an_exception>
except <exception_type> as <name>:
    <code_to_handle_an_exception_if_it_occurs>
```

For example:

```
index = int(input("Enter the index: "))
```

```
try:
    my_list = [1, 2, 3, 4]
    print(my_list[index])
except IndexError as e:
    print("Exception raised:", e)
```

This is the output if we enter 15 as the index:

```
Enter the index: 15
Exception raised: list index out of range
```

This is another example:

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))

try:
    division = a / b
    print(division)
except ZeroDivisionError as err:
    print("Please enter valid values.", err)
```

This is the output if we enter the value 0 for b:

Please enter valid values. division by zero

### **try / except / else in Python**

We can add an `else` clause to this structure after `except` if we want to choose what happens when no exceptions occur during the execution of the `try` clause:

```
try:
    <code_that_may_raise_an_exception>
except:
    <code_to_handle_an_exception_if_it_occurs>
else:
    <code_that_only_runs_if_no_exception_in_try>
```

For example:

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))

try:
    division = a / b
    print(division)
except ZeroDivisionError as err:
    print("Please enter valid values.", err)
else:
    print("Both values were valid.")
```

If we enter the values 5 and 0 for `a` and `b` respectively, the output is:

Please enter valid values. division by zero

But if both values are valid, for

example 5 and 4 for `a` and `b` respectively, the `else` clause runs after `try` is completed and we see:

1.25

Both values were valid.

### **try / except / else / finally in Python**

We can also add a `finally` clause if we need to run code that should always run, even if an exception is raised in `try`.

For example:

```

a = int(input("Enter a: "))
b = int(input("Enter b: "))

try:
    division = a / b
    print(division)
except ZeroDivisionError as err:
    print("Please enter valid values.", err)
else:
    print("Both values were valid.")
finally:
    print("Finally!")

```

If both values are valid, the output is the result of the division and:

```

Both values were valid.
Finally!

```

And if an exception is raised because `b` is 0, we see:

```

Please enter valid values. division by zero
Finally!

```

The `finally` clause always runs.

💡 **Tip:** this clause can be used, for example, to close files even if the code throws an exception.

## ◆ Object-Oriented Programming in Python

In Object-Oriented Programming (OOP), we define classes that act as blueprints to create objects in Python with attributes and methods (functionality associated with the objects).

This is a general syntax to define a class:

```

class <className>:

    <class_attribute_name> = <value>

    def __init__(self, <param1>, <param2>, ...):

```

```

        self.<attr1> = <param1>
        self.<attr2> = <param2>


        .
        .
        .

        # As many attributes as needed

def <method_name>(self, <param1>, ...):
    <code>

    # As many methods as needed

```

 **Tip:** `self` refers to an instance of the class (an object created with the class blueprint).

As you can see, a class can have many different elements so let's analyze them in detail:


## Class Header

The first line of the class definition has the `class` keyword and the name of the class:

```

class Dog:
class House:
class Ball:

```

 **Tip:** If the class inherits attributes and methods from another class, we will see the name of the class within parentheses:

```

class Poodle(Dog):
class Truck(Vehicle):
class Mom(FamilyMember):

```

In Python, we write class name in Upper Camel Case (also known as Pascal Case), in which each word starts with an uppercase letter. For example: `FamilyMember`

## `__init__` and instance attributes

We are going to use the class to create object in Python, just like we build real houses from blueprints.


The objects will have attributes that we define in the class. Usually, we initialize these attributes in `__init__`. This is a method that runs when we create an instance of the class. This is the general syntax:

```
def __init__(self, <parameter1>, <parameter2>, ...):  
    self.<attribute1> = <parameter1> # Instance attribute  
    self.<attribute2> = <parameter2> # Instance attribute  
    .  
    .  
    .  
    # As many instance attributes as needed
```

We specify as many parameters as needed to customize the values of the attributes of the object that will be created.

Here is an example of a `Dog` class with this method:

```
class Dog:  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

 **Tip:** notice the double leading and trailing underscore in the name `__init__`.

### How to Create an Instance

To create an instance of `Dog`, we need to specify the name and age of the dog instance to assign these values to the attributes:

```
my_dog = Dog("Nora", 10)
```

Great. Now we have our instance ready to be used in the program.


Some classes will not require any arguments to create an instance. In that case, we just write empty parentheses. For example:

```
class Circle:
```

```
def __init__(self):  
    self.radius = 1
```

To create an instance:

```
>>> my_circle = Circle()
```

 **Tip:** `self` is like a parameter that acts "behind the scenes", so even if you see it in the method definition, you shouldn't consider it when you pass the arguments.

### Default Arguments

We can also assign default values for the attributes and give the option to the user if they would like to customize the value.

In this case, we would write `<attribute>=<value>` in the list of parameters.

This is an example:

```
class Circle:  
  
    def __init__(self, radius=1):  
        self.radius = radius
```

Now we can create a `Circle` instance with the default value for the radius by omitting the value or customize it by passing a value:

# Default value

```
>>> my_circle1 = Circle()
```

# Customized value

```
>>> my_circle2 = Circle(5)
```

### How to Get an Instance Attribute

To access an instance attribute, we use this syntax:

```
<object_variable>.<attribute>
```

For example:

# Class definition

```
>>> class Dog:
```

```
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

# Create instance

```
>>> my_dog = Dog("Nora", 10)
```

# Get attributes

```
>>> my_dog.name
```

```
'Nora'
```

```
>>> my_dog.age
```

```
10
```

## How to Update an Instance Attribute

To update an instance attribute, we use this syntax:

```
<object_variable>.<attribute> = <new_value>
```

For example:

```
>>> class Dog:
```

```
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
>>> my_dog = Dog("Nora", 10)
```

```
>>> my_dog.name
```

```
'Nora'
```

# Update the attribute



```
>>> my_dog.name = "Norita"
```

```
>>> my_dog.name  
'Norita'
```

## How to Remove an Instance Attribute

To remove an instance attribute, we use this syntax:

```
del <object_variable>.<attribute>
```

For example:

```
>>> class Dog:
```

```
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

```
>>> my_dog = Dog("Nora", 10)
```

```
>>> my_dog.name  
'Nora'
```

```
# Delete this attribute
```

```
>>> del my_dog.name
```

```
>>> my_dog.name
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#77>", line 1, in <module>
```

```
    my_dog.name
```

```
AttributeError: 'Dog' object has no attribute 'name'
```

## How to Delete an Instance

Similarly, we can delete an instance using `del`:

```
>>> class Dog:
```

```

def __init__(self, name, age):
    self.name = name
    self.age = age

>>> my_dog = Dog("Nora", 10)

>>> my_dog.name
'Nora'

# Delete the instance
>>> del my_dog

>>> my_dog
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    my_dog
NameError: name 'my_dog' is not defined

```

## Public vs. Non-Public Attributes in Python

In Python, we don't have access modifiers to functionally restrict access to the instance attributes, so we rely on naming conventions to specify this.

For example, by adding a leading underscore, we can signal to other developers that an attribute is meant to be non-public.

For example:

```

class Dog:

    def __init__(self, name, age):
        self.name = name # Public attribute
        self._age = age # Non-Public attribute

```

The Python documentation mentions:


Use one leading underscore only for non-public methods and instance variables.

Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed. - [source](#)

However, as the documentation also mentions:

We don't use the term "private" here, since no attribute is really private in Python (without a generally unnecessary amount of work). - [source](#)

 **Tip:** technically, we can still access and modify the attribute if we add the leading underscore to its name, but we shouldn't.

### **Class Attributes in Python**

Class attributes are shared by all instances of the class. They all have access to this attribute and they will also be affected by any changes made to these attributes.

```
class Dog:

    # Class attributes
    kingdom = "Animalia"
    species = "Canis lupus"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

 **Tip:** usually, they are written before the `__init__` method.

### How to Get a Class Attribute

To get the value of a class attribute, we use this syntax:

```
<class_name>.<attribute>
```

For example:

```
>>> class Dog:
```

```
    kingdom = "Animalia"
```

```
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
>>> Dog.kingdom
```

```
'Animalia'
```

 **Tip:** You can use this syntax within the class as well.

### How to Update a Class Attribute

To update a class attribute, we use this syntax:

```
<class_name>.<attribute> = <value>
```

For example:

```
>>> class Dog:
```

```
    kingdom = "Animalia"
```

```
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
>>> Dog.kingdom
```

```
'Animalia'
```

```
>>> Dog.kingdom = "New Kingdom"
```

```
>>> Dog.kingdom
```

```
'New Kingdom'
```

## How to Delete a Class Attribute

We use `del` to delete a class attribute. For example:

```
>>> class Dog:
```

```
    kingdom = "Animalia"
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
>>> Dog.kingdom
```

```
'Animalia'
```

```
# Delete class attribute
```

```
>>> del Dog.kingdom
```

```
>>> Dog.kingdom
```

```
Traceback (most recent call last):
```


```
  File "<pyshell#88>", line 1, in <module>
```

```
    Dog.kingdom
```

```
AttributeError: type object 'Dog' has no attribute 'kingdom'
```

## How to Define Methods

Methods represent the functionality of the instances of the class.

 **Tip:** Instance methods can work with the attributes of the instance that is calling the method if we write `self.<attribute>` in the method definition.

This is the basic syntax of a method in a class. They are usually located below `__init__`:

```
class <ClassName>:

    # Class attributes

    # __init__

    def <method_name>(self, <param1>, ...):
        <code>
```

They may have zero, one, or more parameters if needed (just like functions!) but instance methods must always have `self` as the first parameter.

For example, here is a `bark` method with no parameters (in addition to `self`):

```
class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f'woof-woof. I'm {self.name}')
```

To call this method, we use this syntax:

```
<object_variable>.<method>(<arguments>)
```

For example:

```
# Create the instance
>>> my_dog = Dog("Nora", 10)

# Call the method
>>> my_dog.bark()
woof-woof. I'm Nora
```

Here we have a `Player` class with an `increment_speed` method with one parameter:

```
class Player:
```

```
    def __init__(self, name):
        self.name = name
        self.speed = 50
```

```
    def increment_speed(self, value):
        self.speed += value
```

To call the method:

```
# Create instance
>>> my_player = Player("Nora")

# Check initial speed to see the change
>>> my_player.speed
50

# Increment the speed
>>> my_player.increment_speed(5)

# Confirm the change
>>> my_player.speed
55
```

 **Tip:** to add more parameters, just separate them with a comma. It is recommended to add a space after the comma.

### Properties, Getters and Setters in Python

Getters and setters are methods that we can define to get and set the value of an instance attribute, respectively. They work as intermediaries to "protect" the attributes from direct changes.

In Python, we typically use properties instead of getters and setters. Let's see how we can use them.

To define a property, we write a method with this syntax:

```
@property
def <property_name>(self):
    return self.<attribute>
```


This method will act as a getter, so it will be called when we try to access the value of the attribute.

Now, we may also want to define a setter:

```
@<property_name>.setter
def <property_name>(self, <param>):
    self.<attribute> = <param>
```

And a deleter to delete the attribute:

```
@<property_name>.deleter
def <property_name>(self):
    del self.<attribute>
```

 **Tip:** you can write any code that you need in these methods to get, set, and delete an attribute. It is recommended to keep them as simple as possible.

This is an example:

```
class Dog:

    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, new_name):
        self._name = new_name
```



```
@name.deleter
def name(self):
    del self._name
```

If we add descriptive print statements, we can see that they are called when we perform their operation:

```
>>> class Dog:

    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        print("Calling getter")
        return self._name

    @name.setter
    def name(self, new_name):
        print("Calling setter")
        self._name = new_name

    @name.deleter
    def name(self):
        print("Calling deleter")
        del self._name
```

```
>>> my_dog = Dog("Nora")
```

```
>>> my_dog.name
Calling getter
'Nora'
```

```
>>> my_dog.name = "Norita"
```

Calling setter

```
>>> my_dog.name
```

Calling getter

```
'Norita'
```

```
>>> del my_dog.name
```

Calling deleter

## ◆ How to Work with Files in Python

Working with files is very important to create powerful programs. Let's see how you can do this in Python.

### How to Read Files in Python

In Python, it's recommended to use a `with` statement to work with files because it opens them only while we need them and it closes them automatically when the process is completed.

To read a file, we use this syntax:

```
with open("<file_path>") as <file_var>:
```

```
<code>
```

We can also specify that we want to open the file in read mode with an `"r"`:

```
with open("<file_path>", "r") as <file_var>:
```

```
<code>
```

But this is already the default mode to open a file, so we can omit it like in the first example.

This is an example:

```
with open("famous_quotes.txt") as file:
```

```
    for line in file:
```

```
        print(line)
```

OR...

```
with open("famous_quotes.txt", "r") as file:
    for line in file:
        print(line)
```

💡 **Tip:** that's right! We can iterate over the lines of the file using a for loop. The file path can be relative to the Python script that we are running or it can be an absolute path.

## How to Write to a File in Python

There are two ways to write to a file. You can either replace the entire content of the file before adding the new content, or append to the existing content.

```
with open("<file_path>", "w") as <file_var>:
    <code>
```

To replace the content completely, we use the "w" mode, so we pass this string as the second argument to `open()`. We call the `.write()` method on the file object passing the content that we want to write as argument.

For example:

```
words = ["Amazing", "Green", "Python", "Code"]
```

```
with open("famous_quotes.txt", "w") as file:
    for word in words:
        file.write(word + "\n")
```

When you run the program, a new file will be created if it doesn't exist already in the path that we specified.

This will be the content of the file:

```
Amazing
Green
Python
Code
```

## How to Append to a File in Python

However, if you want to append the content, then you need to use the "a" mode:

```
with open("<file_path>", "a") as <file_var>:  
    <code>
```

For example:

```
words = ["Amazing", "Green", "Python", "Code"]
```

```
with open("famous_quotes.txt", "a") as file:  
    for word in words:  
        file.write(word + "\n")
```

This small change will keep the existing content of the file and it will add the new content to the end.

If we run the program again, these strings will be added to the end of the file:

```
Amazing  
Green  
Python  
Code  
Amazing  
Green  
Python  
Code
```

## How to Delete a File in Python

To delete a file with our script, we can use the `os` module. It is recommended to check with a conditional if the file exists before calling the `remove()` function from this module:

```
import os  
  
if os.path.exists("<file_path>"):  
    os.remove("<file_path>")
```

```
else:
```

```
<code>
```

For example:

```
import os
```

```
if os.path.exists("famous_quotes.txt"):
```

```
    os.remove("famous_quotes.txt")
```

```
else:
```

```
    print("This file doesn't exist")
```

You might have noticed the first line that says `import os`. This is an import statement. Let's see why they are helpful and how you can work with them.

## ◆ Import Statements in Python

Organizing your code into multiple files as your program grows in size and complexity is good practice. But we need to find a way to combine these files to make the program work correctly, and that is exactly what import statements do.

By writing an import statement, we can import a module (a file that contains Python definitions and statements) into another file.

These are various alternatives for import statements:

### First Alternative:

```
import <module_name>
```

For example:

```
import math
```

💡 **Tip:** `math` is a built-in Python module.

If we use this import statement, we will need to add the name of the module before the name of the function or element that we are referring to in our code:

```
>>> import math
>>> math.sqrt(25)
```

5.0

We explicitly mention in our code the module that the element belongs to.

### Second Alternative:

```
import <module> as <new_name>
```

For example:

```
import math as m
```

In our code, we can use the new name that we assigned instead of the original name of the module:

```
>>> import math as m
>>> m.sqrt(25)
```

5.0

### Third Alternative:

```
from <module_name> import <element>
```

For example:

```
from math import sqrt
```

With this import statement, we can call the function directly without specifying the name of the module:

```
>>> from math import sqrt
>>> sqrt(25)
5.0
```

### Fourth Alternative:

```
from <module_name> import <element> as <new_name>
```

For example:

```
from math import sqrt as square_root
```

With this import statement, we can assign a new name to the element imported from the module:

```
>>> from math import sqrt as square_root
>>> square_root(25)
5.0
```

### Fifth Alternative:

```
from <module_name> import *
```

This statement imports all the elements of the module and you can refer to them directly by their name without specifying the name of the module.

For example:


```
>>> from math import *

>>> sqrt(25)
5.0

>>> factorial(5)
120

>>> floor(4.6)
4

>>> gcd(5, 8)
1
```

 **Tip:** this type of import statement can make it more difficult for us to know which elements belong to which module, particularly when we are importing elements from multiple modules.

According to the [Style Guide for Python Code](#):

**Wildcard imports** (from <module> import \*) should be avoided, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools.

## ◆ List and Dictionary Comprehension in Python

A really nice feature of Python that you should know about is list and dictionary comprehension. This is just a way to create lists and dictionaries more compactly.

### List Comprehension in Python

The syntax used to define list comprehensions usually follows one of these four patterns:

```
[<value_to_include> for <var> in <sequence>]
[<value_to_include> for <var1> in <sequence1> for <var2> in <sequence2>]
[<value_to_include> for <var> in <sequence> if <condition>]
[<value> for <var1> in <sequence1> for <var2> in <sequence2> if <condition>]
```

**💡 Tip:** you should only use them when they do not make your code more difficult to read and understand.

Here we have some examples:

```
>>> [i for i in range(4, 15)]
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

```
>>> [chr(i) for i in range(67, 80)]
['C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O']
```

```
>>> [i**3 for i in range(2, 5)]
[8, 27, 64]
```

```
>>> [i + j for i in range(5, 8) for j in range(3, 6)]
[8, 9, 10, 9, 10, 11, 10, 11, 12]
```

```
>>> [k for k in range(3, 35) if k % 2 == 0]
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34]
```

```
>>> [i * j for i in range(2, 6) for j in range(3, 7) if i % j == 0]
[9, 16, 25]
```



## List Comprehensions vs. Generator Expressions in Python

List comprehensions are defined with square brackets `[]`. This is different from generator expressions, which are defined with parentheses `()`. They look similar but they are quite different. Let's see why.

- **List comprehensions** generate the entire sequence at once and store it in memory.
- **Generator expressions** yield the elements one at a time when they are requested.

We can check this with the `sys` module. In the example below, you can see that their size in memory is very different:

```
>>> import sys
>>> sys.getsizeof([i for i in range(500)])
2132
>>> sys.getsizeof((i for i in range(500)))
56
```

We can use generator expressions to iterate in a for loop and get the elements one at a time. But if we need to store the elements in a list, then we should use list comprehension.

## Dictionary Comprehension in Python

Now let's dive into dictionary comprehension. The basic syntax that we need to use to define a dictionary comprehension is:

```
{<key_value>: <value> for <var> in <sequence>}
{<key_value>: <value> for <var> in <sequence> if <condition>}
```

Here we have some examples of dictionary comprehension:

```
>>> {num: num**3 for num in range(3, 15)}
{3: 27, 4: 64, 5: 125, 6: 216, 7: 343, 8: 512, 9: 729, 10: 1000, 11: 1331,
12: 1728, 13: 2197, 14: 2744}

>>> {x: x + y for x in range(4, 8) for y in range(3, 7)}
{4: 10, 5: 11, 6: 12, 7: 13}
```

This is an example with a conditional where we take an existing dictionary and create a new dictionary with only the students who earned a passing grade greater than or equal to 60:

```
>>> grades = {"Nora": 78, "Gino": 100, "Talina": 56, "Elizabeth": 45, "Lulu": 67}
```

```
>>> approved_students = {student: grade for (student, grade) in  
grades.items() if grade >= 60}
```

```
>>> approved_students  
{'Nora': 78, 'Gino': 100, 'Lulu': 67}
```

---