

# Learning Python: From Zero to Hero



TK

Sep 30, 2017 · 11 min read



This post was originally published at *TK's Blog*.

First of all, what is Python? According to its creator, Guido van Rossum, Python is a:

“high-level programming language, and its core design philosophy is all about code readability and a syntax which allows programmers to express concepts in a few lines of code.”

For me, the first reason to learn Python was that it is, in fact, a beautiful programming language. It was really natural to code in it and express my thoughts.

Another reason was that we can use coding in Python in multiple ways: data science, web development, and machine learning all shine here. Quora, Pinterest and Spotify all use Python for their backend web development. So let's learn a bit about it.

## The Basics

### 1. Variables

You can think about variables as words that store a value. Simple as that.

In Python, it is really easy to define a variable and set a value to it. Imagine you want to store number 1 in a variable called “one.” Let's do it:

```
1 one = 1
```

variable.py hosted with ❤ by GitHub

[view raw](#)

How simple was that? You just assigned the value 1 to the variable “one.”

```
1 two = 2
2 some_number = 10000
```

variables.py hosted with ❤ by GitHub

[view raw](#)

And you can assign any other **value** to whatever other **variables** you want. As you see in the table above, the variable “**two**” stores the integer **2**, and “**some\_number**” stores **10,000**.

Besides integers, we can also use booleans (True / False), strings, float, and so many other data types.

```
1 # booleans
2 true_boolean = True
3 false_boolean = False
4
5 # string
6 my_name = "Leandro Tk"
7
8 # float
9 book_price = 15.80
```

other\_data\_types.py hosted with ❤ by GitHub

[view raw](#)

## 2. Control Flow: conditional statements

“**If**” uses an expression to evaluate whether a statement is True or False. If it is True, it executes what is inside the “if” statement. For example:

```
1  if True:
2      print("Hello Python If")
3
4  if 2 > 1:
5      print("2 is greater than 1")
```

if.py hosted with ❤ by GitHub

[view raw](#)

2 is greater than 1, so the “**print**” code is executed.

The “**else**” statement will be executed if the “if” expression is **false**.

```
1  if 1 > 2:
2      print("1 is greater than 2")
3  else:
4      print("1 is not greater than 2")
```

if\_else.py hosted with ❤ by GitHub

[view raw](#)

1 is not greater than 2, so the code inside the “**else**” statement will be executed.

You can also use an “**elif**” statement:

```
1  if 1 > 2:
2      print("1 is greater than 2")
3  elif 2 > 1:
4      print("1 is not greater than 2")
5  else:
6      print("1 is equal to 2")
```

if\_elif\_else.py hosted with ❤ by GitHub

[view raw](#)

## 3. Looping / Iterator

In Python, we can iterate in different forms. I’ll talk about two: **while** and **for**.

**While Looping:** while the statement is True, the code inside the block will be executed. So, this code will print the number from 1 to 10.

```
1 num = 1
2
3 while num <= 10:
4     print(num)
5     num += 1
```

while.py hosted with ❤ by GitHub

[view raw](#)

The **while** loop needs a “**loop condition**.” If it stays True, it continues iterating. In this example, when `num` is 11 the **loop condition** equals `False`.

Another basic bit of code to better understand it:

```
1 loop_condition = True
2
3 while loop_condition:
4     print("Loop Condition keeps: %s" %(loop_condition))
5     loop_condition = False
```

while\_basic.py hosted with ❤ by GitHub

[view raw](#)

The **loop condition** is `True` so it keeps iterating — until we set it to `False`.

**For Looping:** you apply the variable “**num**” to the block, and the “**for**” statement will iterate it for you. This code will print the same as **while** code: from 1 to 10.

```
1 for i in range(1, 11):
2     print(i)
```

for.py hosted with ❤ by GitHub

[view raw](#)

See? It is so simple. The range starts with 1 and goes until the 11th element (10 is the 10th element).

## List: Collection | Array | Data Structure

Imagine you want to store the integer 1 in a variable. But maybe now you want to store 2. And 3, 4, 5 ...

Do I have another way to store all the integers that I want, but not in **millions of variables**? You guessed it — there is indeed another way to store them.

`List` is a collection that can be used to store a list of values (like these integers that you want). So let's use it:

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

list.py hosted with ❤ by GitHub

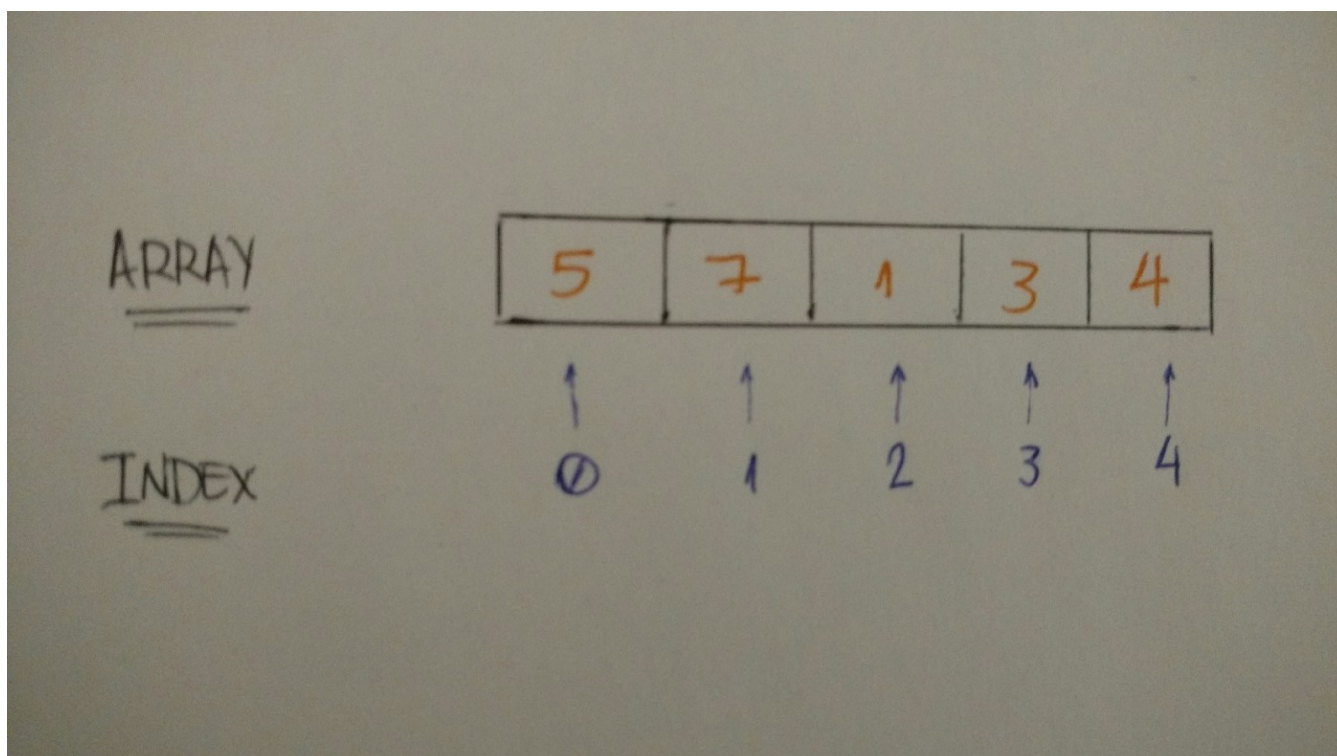
[view raw](#)

It is really simple. We created an array and stored it on **my\_integer**.

But maybe you are asking: “How can I get a value from this array?”

Great question. `List` has a concept called **index**. The first element gets the index 0 (zero). The second gets 1, and so on. You get the idea.

To make it clearer, we can represent the array and each element with its index. I can draw it:



Using the Python syntax, it's also simple to understand:

```
1 my_integers = [5, 7, 1, 3, 4]
2 print(my_integers[0]) # 5
3 print(my_integers[1]) # 7
```

```
4 print(my_integers[4]) # 4
```

list\_index.py hosted with ❤ by GitHub

[view raw](#)

Imagine that you don't want to store integers. You just want to store strings, like a list of your relatives' names. Mine would look something like this:

```
1 relatives_names = [  
2     "Toshiaki",  
3     "Juliana",  
4     "Yuji",  
5     "Bruno",  
6     "Kaio"  
7 ]  
8  
9 print(relatives_names[4]) # Kaio
```

family\_list.py hosted with ❤ by GitHub

[view raw](#)

It works the same way as integers. Nice.

We just learned how `Lists` indices work. But I still need to show you how we can add an element to the `List` data structure (an item to a list).

The most common method to add a new value to a `List` is `append`. Let's see how it works:

```
1 bookshelf = []  
2 bookshelf.append("The Effective Engineer")  
3 bookshelf.append("The 4 Hour Work Week")  
4 print(bookshelf[0]) # The Effective Engineer  
5 print(bookshelf[1]) # The 4 Hour Work Week
```

list\_append.py hosted with ❤ by GitHub

[view raw](#)

`append` is super simple. You just need to apply the element (eg. **"The Effective Engineer"**) as the `append` parameter.

Well, enough about `Lists`. Let's talk about another data structure.

## Dictionary: Key-Value Data Structure

Now we know that `Lists` are indexed with integer numbers. But what if we don't want to use integer numbers as indices? Some data structures that we can use are numeric, string, or other types of indices.

Let's learn about the `Dictionary` data structure. `Dictionary` is a collection of key-value pairs. Here's what it looks like:

```
1 dictionary_example = {  
2     "key1": "value1",  
3     "key2": "value2",  
4     "key3": "value3"  
5 }
```

dictionary\_1.py hosted with ❤ by GitHub

[view raw](#)

The **key** is the index pointing to the **value**. How do we access the `Dictionary` **value**? You guessed it — using the **key**. Let's try it:

```
1 dictionary_tk = {  
2     "name": "Leandro",  
3     "nickname": "Tk",  
4     "nationality": "Brazilian"  
5 }  
6  
7 print("My name is %s" %(dictionary_tk["name"])) # My name is Leandro  
8 print("But you can call me %s" %(dictionary_tk["nickname"])) # But you can call me Tk  
9 print("And by the way I'm %s" %(dictionary_tk["nationality"])) # And by the way I'm Brazilian
```

dictionary\_2.py hosted with ❤ by GitHub

[view raw](#)

I created a `Dictionary` about me. My name, nickname, and nationality. Those attributes are the `Dictionary` **keys**.

As we learned how to access the `List` using index, we also use indices (**keys** in the `Dictionary` context) to access the **value** stored in the `Dictionary`.

In the example, I printed a phrase about me using all the values stored in the `Dictionary`. Pretty simple, right?

Another cool thing about `Dictionary` is that we can use anything as the value. In the `Dictionary` I created, I want to add the **key** “age” and my real integer age in it:

```
1 dictionary_tk = {
2     "name": "Leandro",
3     "nickname": "Tk",
4     "nationality": "Brazilian",
5     "age": 24
6 }
7
8 print("My name is %s" %(dictionary_tk["name"])) # My name is Leandro
9 print("But you can call me %s" %(dictionary_tk["nickname"])) # But you can call me Tk
10 print("And by the way I'm %i and %s" %(dictionary_tk["age"], dictionary_tk["nationality"])) # I'm 24 and Brazilian
```

dictionary\_3.py hosted with  by GitHub

[view raw](#)

Here we have a **key** (age) **value** (24) pair using string as the **key** and integer as the **value**.

As we did with `Lists`, let's learn how to add elements to a `Dictionary`. The **key** pointing to a **value** is a big part of what `Dictionary` is. This is also true when we are talking about adding elements to it:

```
1 dictionary_tk = {
2     "name": "Leandro",
3     "nickname": "Tk",
4     "nationality": "Brazilian"
5 }
6
7 dictionary_tk['age'] = 24
8
9 print(dictionary_tk) # {'nationality': 'Brazilian', 'age': 24, 'nickname': 'Tk', 'name': 'Leandro'}
```

dictionary\_4.py hosted with  by GitHub

[view raw](#)

We just need to assign a **value** to a `Dictionary` **key**. Nothing complicated here, right?

## Iteration: Looping Through Data Structures

As we learned in the **Python Basics**, the `List` iteration is very simple. We `Python` developers commonly use `For` looping. Let's do it:

```
1 bookshelf = [
2     "The Effective Engineer",
3     "The 4 hours work week",
4     "Zero to One",
5     "Lean Startup"
```



```
5     Lean Startup ,
6     "Hooked"
7 ]
8
9 for book in bookshelf:
10     print(book)
```

list\_iteration.py hosted with ❤ by GitHub

[view raw](#)

So for each book in the bookshelf, we (**can do everything with it**) print it. Pretty simple and intuitive. That's Python.

For a hash data structure, we can also use the `for` loop, but we apply the `key` :

```
1 dictionary = { "some_key": "some_value" }
2
3 for key in dictionary:
4     print("%s --> %s" %(key, dictionary[key]))
5
6 # some_key --> some_value
```

dictionary\_iteration.py hosted with ❤ by GitHub

[view raw](#)

This is an example how to use it. For each `key` in the `dictionary` , we print the `key` and its corresponding `value` .

Another way to do it is to use the `iteritems` method.

```
1 dictionary = { "some_key": "some_value" }
2
3 for key, value in dictionary.items():
4     print("%s --> %s" %(key, value))
5
6 # some_key --> some_value
```

dictionary\_iteration\_1.py hosted with ❤ by GitHub

[view raw](#)

We did name the two parameters as `key` and `value` , but it is not necessary. We can name them anything. Let's see it:

```
1 dictionary_tk = {
2     "name": "Leandro",
3     "nickname": "Tk",
4     "nationality": "Brazilian",
5     "age": 24
```

```
6 }  
7  
8 for attribute, value in dictionary_tk.items():  
9     print("My %s is %s" %(attribute, value))  
10  
11 # My name is Leandro  
12 # My nickname is Tk  
13 # My nationality is Brazilian  
14 # My age is 24
```

dictionary iteration 2.py hosted with  by GitHub

[view raw](#)

We can see we used attribute as a parameter for the `Dictionary` key, and it works properly. Great!

## Classes & Objects

### A little bit of theory:

**Objects** are a representation of real world objects like cars, dogs, or bikes. The objects share two main characteristics: **data** and **behavior**.

Cars have **data**, like number of wheels, number of doors, and seating capacity. They also exhibit **behavior**: they can accelerate, stop, show how much fuel is left, and so many other things.

We identify **data** as **attributes** and **behavior** as **methods** in object-oriented programming. Again:

Data → Attributes and Behavior → Methods

And a **Class** is the blueprint from which individual objects are created. In the real world, we often find many objects with the same type. Like cars. All the same make and model (and all have an engine, wheels, doors, and so on). Each car was built from the same set of blueprints and has the same components.

### Python Object-Oriented Programming mode: ON

Python, as an Object-Oriented programming language, has these concepts: **class** and **object**.

A class is a blueprint, a model for its objects.

So again, a class is just a model, or a way to define **attributes** and **behavior** (as we talked about in the theory section). As an example, a vehicle **class** has its own **attributes** that define what **objects** are vehicles. The number of wheels, type of tank, seating capacity, and maximum velocity are all attributes of a vehicle.

With this in mind, let's look at Python syntax for **classes**:

```
1 class Vehicle:
2     pass
```

class.py hosted with ❤ by GitHub

[view raw](#)

We define classes with a **class statement** — and that's it. Easy, isn't it?

**Objects** are instances of a **class**. We create an instance by naming the class.

```
1 car = Vehicle()
2 print(car) # <__main__.Vehicle instance at 0x7fb1de6c2638>
```

object\_1.py hosted with ❤ by GitHub

[view raw](#)

Here `car` is an **object** (or instance) of the **class** `Vehicle`.

Remember that our vehicle **class** has four **attributes**: number of wheels, type of tank, seating capacity, and maximum velocity. We set all these **attributes** when creating a vehicle **object**. So here, we define our **class** to receive data when it initiates it:

```
1 class Vehicle:
2     def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
3         self.number_of_wheels = number_of_wheels
4         self.type_of_tank = type_of_tank
5         self.seating_capacity = seating_capacity
6         self.maximum_velocity = maximum_velocity
```

class\_2.py hosted with ❤ by GitHub

[view raw](#)

We use the `init` **method**. We call it a constructor method. So when we create the vehicle **object**, we can define these **attributes**. Imagine that we love the **Tesla Model S**, and we want to create this kind of **object**. It has four wheels, runs on electric energy, has space for five seats, and the maximum velocity is 250km/hour (155 mph). Let's create this **object**:

```
1 tesla_model_s = Vehicle(4, 'electric', 5, 250)
```

object\_2.py hosted with ❤ by GitHub

[view raw](#)

Four wheels + electric “tank type” + five seats + 250km/hour maximum speed.

All attributes are set. But how can we access these attributes’ values? We **send a message to the object asking about them**. We call it a **method**. It’s the **object’s behavior**. Let’s implement it:

```
1 class Vehicle:
2     def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
3         self.number_of_wheels = number_of_wheels
4         self.type_of_tank = type_of_tank
5         self.seating_capacity = seating_capacity
6         self.maximum_velocity = maximum_velocity
7
8     def number_of_wheels(self):
9         return self.number_of_wheels
10
11     def set_number_of_wheels(self, number):
12         self.number_of_wheels = number
```

method 1.py hosted with ❤ by GitHub

[view raw](#)

This is an implementation of two methods: **number\_of\_wheels** and **set\_number\_of\_wheels**. We call it `getter` & `setter`. Because the first gets the attribute value, and the second sets a new value for the attribute.

In Python, we can do that using `@property` ( `decorators` ) to define `getters` and `setters`. Let’s see it with code:

```
1 class Vehicle:
2     def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
3         self.number_of_wheels = number_of_wheels
4         self.type_of_tank = type_of_tank
5         self.seating_capacity = seating_capacity
6         self.maximum_velocity = maximum_velocity
7
8     @property
9     def number_of_wheels(self):
10         return self.__number_of_wheels
11
12     @number_of_wheels.setter
13     def number_of_wheels(self, number):
```

```
13         def number_of_wheels(self, number):
14             self.__number_of_wheels = number
```

getter and setter.py hosted with ❤ by GitHub

[view raw](#)

And we can use these methods as attributes:

```
1 tesla_model_s = Vehicle(4, 'electric', 5, 250)
2 print(tesla_model_s.number_of_wheels) # 4
3 tesla_model_s.number_of_wheels = 2 # setting number of wheels to 2
4 print(tesla_model_s.number_of_wheels) # 2
```

getter\_and\_setter\_1.py hosted with ❤ by GitHub

[view raw](#)

This is slightly different than defining methods. The methods work as attributes. For example, when we set the new number of wheels, we don't apply two as a parameter, but set the value 2 to `number_of_wheels`. This is one way to write `pythonic` `getter` and `setter` code.

But we can also use methods for other things, like the “**make\_noise**” method. Let's see it:

```
1 class Vehicle:
2     def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
3         self.number_of_wheels = number_of_wheels
4         self.type_of_tank = type_of_tank
5         self.seating_capacity = seating_capacity
6         self.maximum_velocity = maximum_velocity
7
8     def make_noise(self):
9         print('VRUUUUUUUM')
```

method\_2.py hosted with ❤ by GitHub

[view raw](#)

When we call this method, it just returns a string “**VRRRRRUUUUM.**”

```
1 tesla_model_s = Vehicle(4, 'electric', 5, 250)
2 tesla_model_s.make_noise() # VRUUUUUUUM
```

using\_make\_noise\_method.py hosted with ❤ by GitHub

[view raw](#)

## Encapsulation: Hiding Information

Encapsulation is a mechanism that restricts direct access to objects' data and methods. But at the same time, it facilitates operation on that data (objects' methods).

“Encapsulation can be used to hide data members and members function. Under this definition, encapsulation means that the internal representation of an object is generally hidden from view outside of the object’s definition.” —  
Wikipedia

All internal representation of an object is hidden from the outside. Only the object can interact with its internal data.

First, we need to understand how `public` and `non-public` instance variables and methods work.

## Public Instance Variables

For a Python class, we can initialize a `public instance variable` within our constructor method. Let’s see this:

Within the constructor method:

```
1 class Person:
2     def __init__(self, first_name):
3         self.first_name = first_name
```

public\_instance\_variable\_constructor.py hosted with ❤ by GitHub

[view raw](#)

Here we apply the `first_name` value as an argument to the `public instance variable`.

```
1 tk = Person('TK')
2 print(tk.first_name) # => TK
```

public\_instance\_variable\_test.py hosted with ❤ by GitHub

[view raw](#)

Within the class:

```
1 class Person:
2     first_name = 'TK'
```

public\_instance\_variable\_class.py hosted with ❤ by GitHub

[view raw](#)

Here, we do not need to apply the `first_name` as an argument, and all instance objects will have a `class attribute` initialized with `TK`.

```
1 tk = Person()
2 print(tk.first_name) # => TK
```

public\_instance\_variable\_test2.py hosted with ❤ by GitHub

[view raw](#)

Cool. We have now learned that we can use `public instance variables` and `class attributes`. Another interesting thing about the `public` part is that we can manage the variable value. What do I mean by that? Our `object` can manage its variable value: `Get` and `Set` variable values.

Keeping the `Person` class in mind, we want to set another value to its `first_name` variable:

```
1 tk = Person('TK')
2 tk.first_name = 'Kaio'
3 print(tk.first_name) # => Kaio
```

set\_public\_instance\_variable.py hosted with ❤ by GitHub

[view raw](#)

There we go. We just set another value ( `kaio` ) to the `first_name` instance variable and it updated the value. Simple as that. Since it's a `public variable`, we can do that.

## Non-public Instance Variable

We don't use the term “private” here, since no attribute is really private in Python (without a generally unnecessary amount of work). — PEP 8

As the `public instance variable`, we can define the `non-public instance variable` both within the constructor method or within the class. The syntax difference is: for `non-public instance variables`, use an underscore ( `_` ) before the `variable name`.

“‘Private’ instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member)” — Python Software Foundation

Here’s an example:

```
1 class Person:
2     def __init__(self, first_name, email):
3         self.first_name = first_name
4         self._email = email
```

private\_instance\_variable.py hosted with ❤ by GitHub

[view raw](#)

Did you see the `email` variable? This is how we define a non-public variable :

```
1 tk = Person('TK', 'tk@mail.com')
2 print(tk._email) # tk@mail.com
```

private\_instance\_variable\_get.py hosted with ❤ by GitHub

[view raw](#)

*We can access and update it. Non-public variables are just a convention and should be treated as a non-public part of the API.*

So we use a method that allows us to do it inside our class definition. Let’s implement two methods (`email` and `update_email`) to understand it:

```
1 class Person:
2     def __init__(self, first_name, email):
3         self.first_name = first_name
4         self._email = email
5
```



```
6     def update_email(self, new_email):
7         self._email = new_email
8
9     def email(self):
10         return self._email
```

manage\_private\_variables.py hosted with ❤ by GitHub

[view raw](#)

Now we can update and access `non-public variables` using those methods. Let's see:

```
1 tk = Person('TK', 'tk@mail.com')
2 print(tk.email()) # => tk@mail.com
3 # tk._email = 'new_tk@mail.com' -- treat as a non-public part of the class API
4 print(tk.email()) # => tk@mail.com
5 tk.update_email('new_tk@mail.com')
6 print(tk.email()) # => new_tk@mail.com
```

manage\_private\_variables\_test.py hosted with ❤ by GitHub

[view raw](#)

1. We initiated a new object with `first_name` TK and `email` tk@mail.com
2. Printed the email by accessing the `non-public variable` with a method
3. Tried to set a new `email` out of our class
4. We need to treat `non-public variable` as `non-public` part of the API
5. Updated the `non-public variable` with our instance method
6. Success! We can update it inside our class with the helper method

## Public Method

With `public methods`, we can also use them out of our class:

```
1 class Person:
2     def __init__(self, first_name, age):
3         self.first_name = first_name
4         self._age = age
5
6     def show_age(self):
7         return self._age
```

public\_method.py hosted with ❤ by GitHub

[view raw](#)

Let's test it:

```
1 tk = Person('TK', 25)
2 print(tk.show_age()) # => 25
```

public\_method\_test.py hosted with ❤ by GitHub

[view raw](#)

Great — we can use it without any problem.

## Non-public Method

But with non-public methods we aren't able to do it. Let's implement the same `Person` class, but now with a `show_age` non-public method using an underscore (`_`).

```
1 class Person:
2     def __init__(self, first_name, age):
3         self.first_name = first_name
4         self._age = age
5
6     def _show_age(self):
7         return self._age
```

private\_method.py hosted with ❤ by GitHub

[view raw](#)

And now, we'll try to call this non-public method with our object:

```
1 tk = Person('TK', 25)
2 print(tk._show_age()) # => 25
```

private\_method\_test.py hosted with ❤ by GitHub

[view raw](#)

*We can access and update it. Non-public methods are just a convention and should be treated as a non-public part of the API.*

Here's an example for how we can use it:

```
1 class Person:
2     def __init__(self, first_name, age):
3         self.first_name = first_name
4         self._age = age
5
6     def show_age(self):
7         return self._get_age()
8
9     def _get_age(self):
10        return self._age
11
```

```
12 tk = Person('TK', 25)
13 print(tk.show_age()) # => 25
```

using private method.py hosted with ❤ by GitHub

[view raw](#)

Here we have a `_get_age` non-public method and a `show_age` public method. The `show_age` can be used by our object (out of our class) and the `_get_age` only used inside our class definition (inside `show_age` method). But again: as a matter of convention.

## Encapsulation Summary

With encapsulation we can ensure that the internal representation of the object is hidden from the outside.

## Inheritance: behaviors and characteristics

Certain objects have some things in common: their behavior and characteristics.

For example, I inherited some characteristics and behaviors from my father. I inherited his eyes and hair as characteristics, and his impatience and introversion as behaviors.

In object-oriented programming, classes can inherit common characteristics (data) and behavior (methods) from another class.

Let's see another example and implement it in Python.

Imagine a car. Number of wheels, seating capacity and maximum velocity are all attributes of a car. We can say that an **ElectricCar** class inherits these same attributes from the regular **Car** class.

```
1 class Car:
2     def __init__(self, number_of_wheels, seating_capacity, maximum_velocity):
3         self.number_of_wheels = number_of_wheels
4         self.seating_capacity = seating_capacity
5         self.maximum_velocity = maximum_velocity
```

car\_class.py hosted with ❤ by GitHub

[view raw](#)

Our **Car** class implemented:

```
1 my_car = Car(4, 5, 250)
2 print(my_car.number_of_wheels)
3 print(my_car.seating_capacity)
```

```
4 print(my_car.maximum_velocity)
```

car\_instance.py hosted with ❤ by GitHub

[view raw](#)

Once initiated, we can use all instance variables created. Nice.

In Python, we apply a parent class to the child class as a parameter. An **ElectricCar** class can inherit from our **Car** class.

```
1 class ElectricCar(Car):
2     def __init__(self, number_of_wheels, seating_capacity, maximum_velocity):
3         Car.__init__(self, number_of_wheels, seating_capacity, maximum_velocity)
```

python\_inheritance.py hosted with ❤ by GitHub

[view raw](#)

Simple as that. We don't need to implement any other method, because this class already has it (inherited from **Car** class). Let's prove it:

```
1 my_electric_car = ElectricCar(4, 5, 250)
2 print(my_electric_car.number_of_wheels) # => 4
3 print(my_electric_car.seating_capacity) # => 5
4 print(my_electric_car.maximum_velocity) # => 250
```

inheritance\_subclass.py hosted with ❤ by GitHub

[view raw](#)

Beautiful.

## That's it!

We learned a lot of things about Python basics:

- How Python variables work
- How Python conditional statements work
- How Python looping (while & for) works
- How to use Lists: Collection | Array
- Dictionary Key-Value Collection
- How we can iterate through these data structures
- Objects and Classes

- Attributes as objects' data
- Methods as objects' behavior
- Using Python getters and setters & property decorator
- Encapsulation: hiding information
- Inheritance: behaviors and characteristics

## Resources

- Big-O Notation For Coding Interviews and Beyond
- Learn Python from Scratch
- Learn Object-Oriented Programming in Python
- Data Structures in Python: An Interview Refresher
- Data Structures and Algorithms in Python
- Data Structures for Coding Interviews in Python
- One Month Python Course

Congrats! You completed this dense piece of content about Python.

From more stories and posts about my journey learning & mastering programming, follow my publication **The Renaissance Developer**.

Have fun, keep learning, and always keep coding.

# I hope you liked this content. Support my work on Ko-Fi

My Twitter & Github. ☺

Programming   Coding   Python   Software Development   Software Engineering

Get the Medium app

