**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:

one = 1

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

two = 2

some\_number = 10000

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.

# booleans

true\_boolean = True

false\_boolean = False

# string

my\_name = "Leandro Tk"

# float

book\_price = 15.80

**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:

if True:

print("Hello Python If")

if 2 > 1:

print("2 is greater than 1")

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

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

if 1 > 2:

print("1 is greater than 2")

else:

print("1 is not greater than 2")

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

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

if 1 > 2:

print("1 is greater than 2")

elif 2 > 1:

print("1 is not greater than 2")

else:

print("1 is equal to 2")

**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**.

num = 1

while num <= 10:

print(num)

num += 1

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:

loop\_condition = True

while loop\_condition:

print("Loop Condition keeps: %s" %(loop\_condition))

loop\_condition = False

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**.

for i in range(1, 11):

print(i)

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:

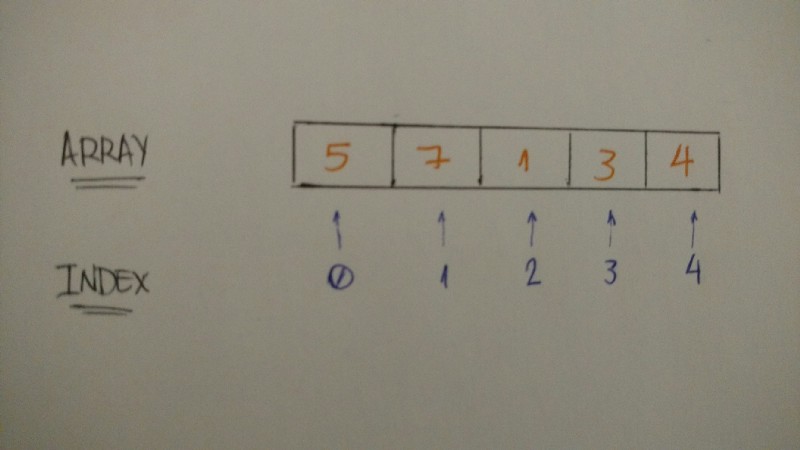
my\_integers = [1, 2, 3, 4, 5]

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:

my\_integers = [5, 7, 1, 3, 4]

print(my\_integers[0]) # 5

print(my\_integers[1]) # 7

print(my\_integers[4]) # 4

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:

relatives\_names = [

"Toshiaki",

"Juliana",

"Yuji",

"Bruno",

"Kaio"

]

print(relatives\_names[4]) # Kaio

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:

bookshelf = []

bookshelf.append("The Effective Engineer")

bookshelf.append("The 4 Hour Work Week")

print(bookshelf[0]) # The Effective Engineer

print(bookshelf[1]) # The 4 Hour Work Week

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:

dictionary\_example = {

"key1": "value1",

"key2": "value2",

"key3": "value3"

}

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:

dictionary\_tk = {

"name": "Leandro",

"nickname": "Tk",

"nationality": "Brazilian"

}

print("My name is %s" %(dictionary\_tk["name"])) # My name is Leandro

print("But you can call me %s" %(dictionary\_tk["nickname"])) # But you can call me Tk

print("And by the way I'm %s" %(dictionary\_tk["nationality"])) # And by the way I'm Brazilian

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 DictionaryI created, I want to add the **key** “age” and my real integer age in it:

dictionary\_tk = {

"name": "Leandro",

"nickname": "Tk",

"nationality": "Brazilian",

"age": 24

}

print("My name is %s" %(dictionary\_tk["name"])) # My name is Leandro

print("But you can call me %s" %(dictionary\_tk["nickname"])) # But you can call me Tk

print("And by the way I'm %i and %s" %(dictionary\_tk["age"], dictionary\_tk["nationality"])) # And by the way I'm Brazilian

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:

dictionary\_tk = {

"name": "Leandro",

"nickname": "Tk",

"nationality": "Brazilian"

}

dictionary\_tk['age'] = 24

print(dictionary\_tk) # {'nationality': 'Brazilian', 'age': 24, 'nickname': 'Tk', 'name': 'Leandro'}

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**](https://medium.com/the-renaissance-developer/python-101-the-basics-441136fb7cc3), the List iteration is very simple. We Pythondevelopers commonly use For looping. Let’s do it:

bookshelf = [

"The Effective Engineer",

"The 4-hour Workweek",

"Zero to One",

"Lean Startup",

"Hooked"

]

for book in bookshelf:

print(book)

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 :

dictionary = { "some\_key": "some\_value" }

for key in dictionary:

print("%s --> %s" %(key, dictionary[key]))

# some\_key --> some\_value

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.

dictionary = { "some\_key": "some\_value" }

for key, value in dictionary.items():

print("%s --> %s" %(key, value))

# some\_key --> some\_value

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

dictionary\_tk = {

"name": "Leandro",

"nickname": "Tk",

"nationality": "Brazilian",

"age": 24

}

for attribute, value in dictionary\_tk.items():

print("My %s is %s" %(attribute, value))

# My name is Leandro

# My nickname is Tk

# My nationality is Brazilian

# My age is 24

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 it 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**:

class Vehicle:

pass

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.

car = Vehicle()

print(car) # <\_\_main\_\_.Vehicle instance at 0x7fb1de6c2638>

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:

class Vehicle:

def \_\_init\_\_(self, number\_of\_wheels, type\_of\_tank, seating\_capacity, maximum\_velocity):

self.number\_of\_wheels = number\_of\_wheels

self.type\_of\_tank = type\_of\_tank

self.seating\_capacity = seating\_capacity

self.maximum\_velocity = maximum\_velocity

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:**

tesla\_model\_s = Vehicle(4, 'electric', 5, 250)

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:

class Vehicle:

def \_\_init\_\_(self, number\_of\_wheels, type\_of\_tank, seating\_capacity, maximum\_velocity):

self.number\_of\_wheels = number\_of\_wheels

self.type\_of\_tank = type\_of\_tank

self.seating\_capacity = seating\_capacity

self.maximum\_velocity = maximum\_velocity

def number\_of\_wheels(self):

return self.number\_of\_wheels

def set\_number\_of\_wheels(self, number):

self.number\_of\_wheels = number

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:

class Vehicle:

def \_\_init\_\_(self, number\_of\_wheels, type\_of\_tank, seating\_capacity, maximum\_velocity):

self.number\_of\_wheels = number\_of\_wheels

self.type\_of\_tank = type\_of\_tank

self.seating\_capacity = seating\_capacity

self.maximum\_velocity = maximum\_velocity

@property

def number\_of\_wheels(self):

return self.\_\_number\_of\_wheels

@number\_of\_wheels.setter

def number\_of\_wheels(self, number):

self.\_\_number\_of\_wheels = number

And we can use these methods as attributes:

tesla\_model\_s = Vehicle(4, 'electric', 5, 250)

print(tesla\_model\_s.number\_of\_wheels) # 4

tesla\_model\_s.number\_of\_wheels = 2 # setting number of wheels to 2

print(tesla\_model\_s.number\_of\_wheels) # 2

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:

class Vehicle:

def \_\_init\_\_(self, number\_of\_wheels, type\_of\_tank, seating\_capacity, maximum\_velocity):

self.number\_of\_wheels = number\_of\_wheels

self.type\_of\_tank = type\_of\_tank

self.seating\_capacity = seating\_capacity

self.maximum\_velocity = maximum\_velocity

def make\_noise(self):

print('VRUUUUUUUM')

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

tesla\_model\_s = Vehicle(4, 'electric', 5, 250)

tesla\_model\_s.make\_noise() # VRUUUUUUUM

**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](https://en.wikipedia.org/wiki/Object_(computer_science)) 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:

class Person:

def \_\_init\_\_(self, first\_name):

self.first\_name = first\_name

Here we apply the first\_name value as an argument to the public instance variable.

tk = Person('TK')

print(tk.first\_name) # => TK

Within the class:

class Person:

first\_name = 'TK'

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.

tk = Person()

print(tk.first\_name) # => TK

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:

tk = Person('TK')

tk.first\_name = 'Kaio'

print(tk.first\_name) # => Kaio

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](https://www.python.org/dev/peps/pep-0008/#designing-for-inheritance)

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](https://docs.python.org/2/tutorial/classes.html#private-variables-and-class-local-references)

Here’s an example:

class Person:

def \_\_init\_\_(self, first\_name, email):

self.first\_name = first\_name

self.\_email = email

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

tk = Person('TK', 'tk@mail.com')

print(tk.\_email) # tk@mail.com

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:

class Person:

def \_\_init\_\_(self, first\_name, email):

self.first\_name = first\_name

self.\_email = email

def update\_email(self, new\_email):

self.\_email = new\_email

def email(self):

return self.\_email

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

tk = Person('TK', 'tk@mail.com')

print(tk.email()) # => tk@mail.com

# tk.\_email = 'new\_tk@mail.com' -- treat as a non-public part of the class API

print(tk.email()) # => tk@mail.com

tk.update\_email('new\_tk@mail.com')

print(tk.email()) # => new\_tk@mail.com

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:

class Person:

def \_\_init\_\_(self, first\_name, age):

self.first\_name = first\_name

self.\_age = age

def show\_age(self):

return self.\_age

Let’s test it:

tk = Person('TK', 25)

print(tk.show\_age()) # => 25

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 (\_).

class Person:

def \_\_init\_\_(self, first\_name, age):

self.first\_name = first\_name

self.\_age = age

def \_show\_age(self):

return self.\_age

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

tk = Person('TK', 25)

print(tk.\_show\_age()) # => 25

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:

class Person:

def \_\_init\_\_(self, first\_name, age):

self.first\_name = first\_name

self.\_age = age

def show\_age(self):

return self.\_get\_age()

def \_get\_age(self):

return self.\_age

tk = Person('TK', 25)

print(tk.show\_age()) # => 25

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.

class Car:

def \_\_init\_\_(self, number\_of\_wheels, seating\_capacity, maximum\_velocity):

self.number\_of\_wheels = number\_of\_wheels

self.seating\_capacity = seating\_capacity

self.maximum\_velocity = maximum\_velocity

Our **Car** class implemented:

my\_car = Car(4, 5, 250)

print(my\_car.number\_of\_wheels)

print(my\_car.seating\_capacity)

print(my\_car.maximum\_velocity)

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.

class ElectricCar(Car):

def \_\_init\_\_(self, number\_of\_wheels, seating\_capacity, maximum\_velocity):

Car.\_\_init\_\_(self, number\_of\_wheels, seating\_capacity, maximum\_velocity)

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:

my\_electric\_car = ElectricCar(4, 5, 250)

print(my\_electric\_car.number\_of\_wheels) # => 4

print(my\_electric\_car.seating\_capacity) # => 5

print(my\_electric\_car.maximum\_velocity) # => 250

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

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