30 SEPTEMBER 2017 / #PYTHON

Learning Python: From Zero to Hero

by TK

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."

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:

```
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 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
```

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 `11` th element (`10` is the `10` th 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 …

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:

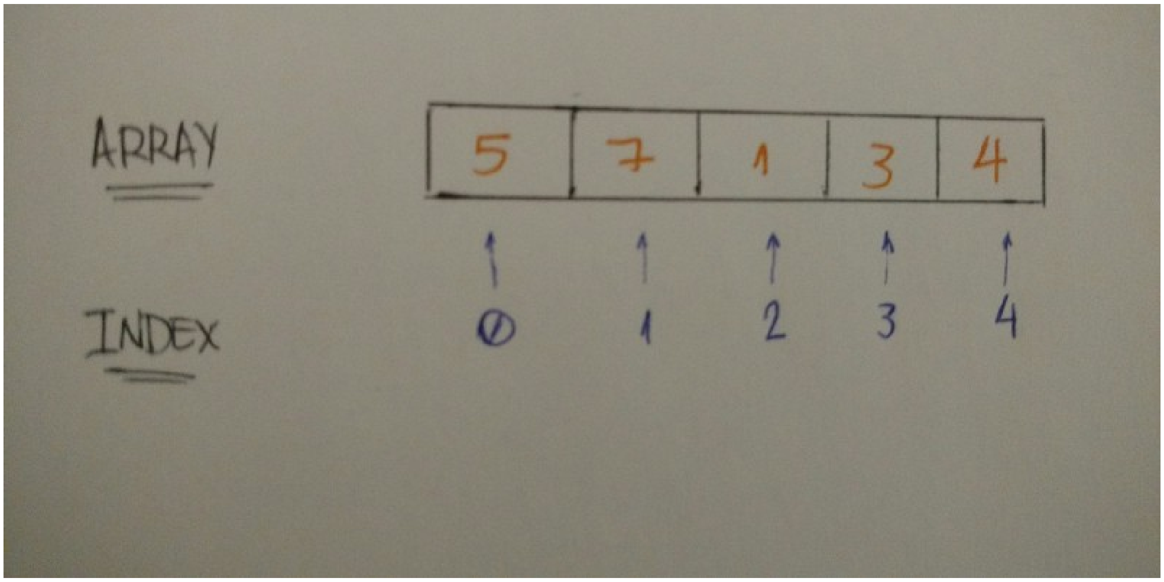
```
my_integers = [1, 2, 3, 4, 5]
```

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"
]
```

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

```
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
print("And by the way I'm %s" %(dictionary_tk["nationality"])) # And b
```

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 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
print("And by the way I'm %i and %s" %(dictionary_tk["age"], dictionar
```

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, 'nickna
```

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:

```
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

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.

```
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_capacit
        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 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_capacit
        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 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_capacit
        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**"

```
class Vehicle:
    def __init__(self, number_of_wheels, type_of_tank, seating_capacit
        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 is generally hidden from view outside of the object's definition."
> — Wikipedia

All internal representation of an object is hidden from the outside.

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

```
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

As the `public instance variable`, we can define the `non-public instance variable` both within the constructor method or within the 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:

```
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

```
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 c
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

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:
```

```
        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
```

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_vel
        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_vel
        Car.__init__(self, number_of_wheels, seating_capacity, maximum
```

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.

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.

If you want a complete Python course, learn more real-world coding skills and build projects, try ***One Month Python Bootcamp***. See you there ☺

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

Have fun, keep learning, and always keep coding.

My Twitter & Github. ☺

If this article was helpful, tweet it.

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

Countinue reading about

Python

Truthy and Falsy Values in Python: A Detailed Introduction

How to Replace Bash with Python as Your Go-To Command Line Language

NodeJS vs Python: How to Choose the Best Technology to Develop Your Web App's Back End

See all 268 posts →

#IOS

MobileSpace — A story about my passions

2 YEARS AGO

#SPORTS

How to code sports games for Amazon Alexa, plus some fun games I built

2 YEARS AGO

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can make a tax-deductible donation here.