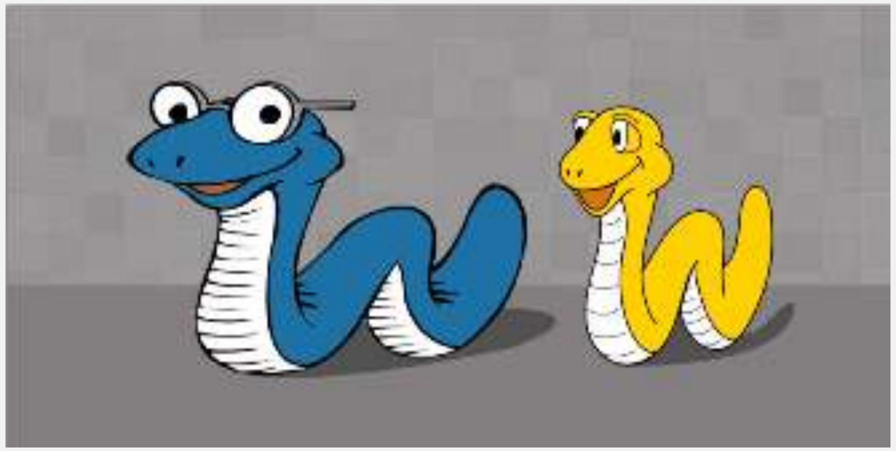
WEB PROGRAMMING 06016322

BY DR BUNDIT THANASOPON

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.”



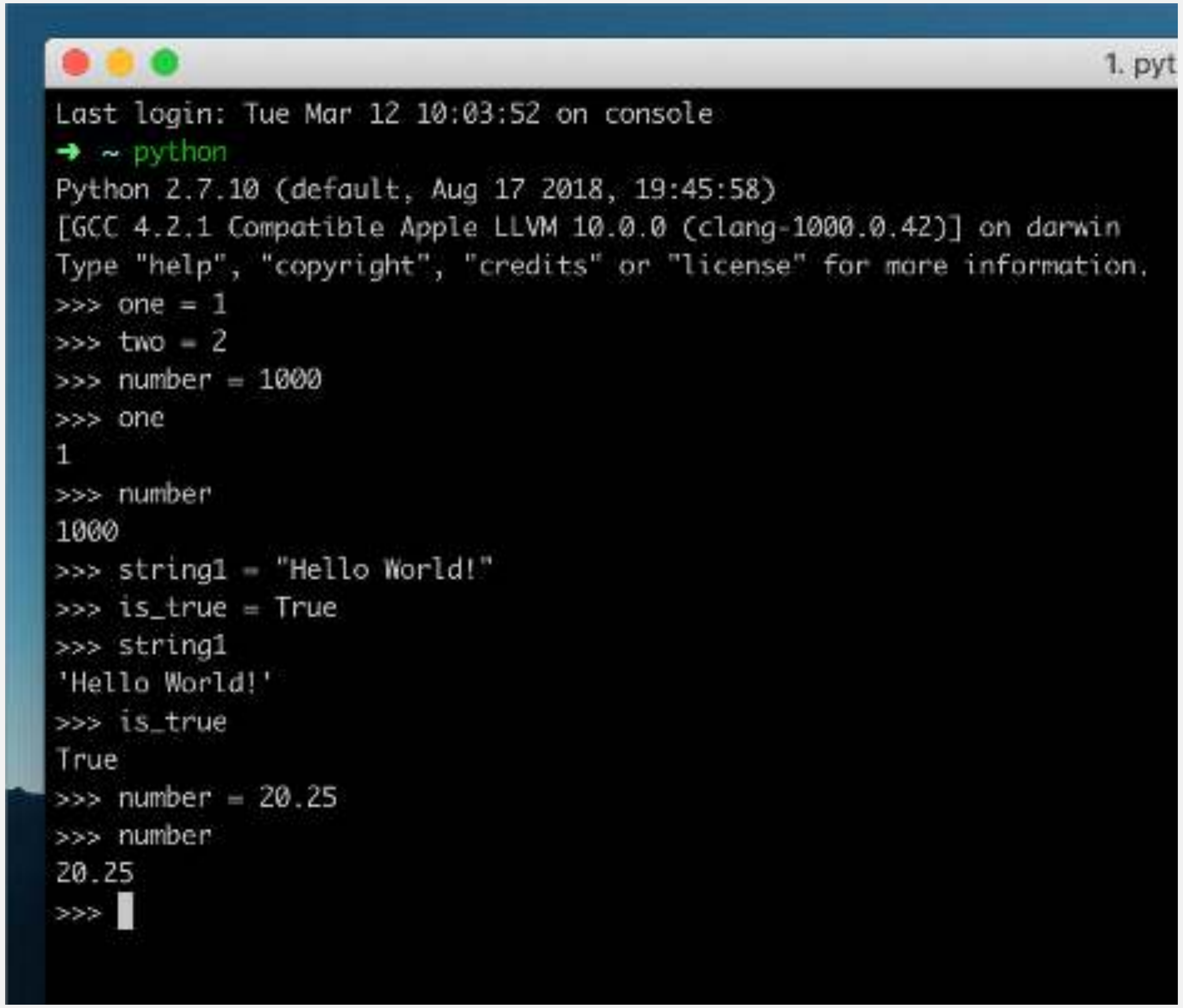
THE BASICS

HTTPS://MEDIUM.COM/THE-RENAISSANCE-DEVELOPER/LEARNING-PYTHON-FROM-ZERO-TO-HERO-8CEED48486D5

VARIABLES

- In Python, it is really easy to define a variable and set a value to it.

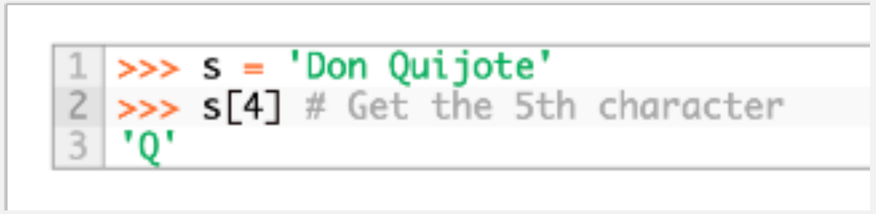
- Besides integers, we can also use booleans (True / False), strings, float, and so many other data types.



```
Last login: Tue Mar 12 10:03:52 on console
➜ ~ python
Python 2.7.10 (default, Aug 17 2018, 19:45:58)
[GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> one = 1
>>> two = 2
>>> number = 1000
>>> one
1
>>> number
1000
>>> string1 = "Hello World!"
>>> is_true = True
>>> string1
'Hello World!'
>>> is_true
True
>>> number = 20.25
>>> number
20.25
>>>
```

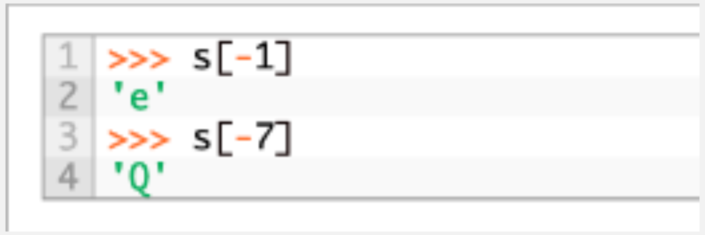
ACCESSING CHARACTERS IN STRINGS BY INDEX

- Typically it's more useful to access the individual characters of a string by using Python's array-like indexing syntax.
- **Important!!!** The first item in the sequence is number 0.



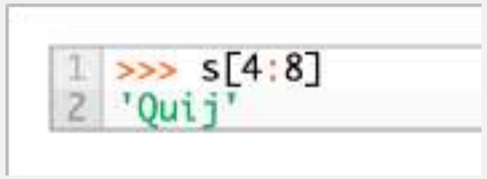
```
>>> s = 'Don Quijote'
>>> s[4] # Get the 5th character
'Q'
```

- If you want to start counting from the end of the string, use a negative index. For example, an index of -1 refers to the right-most character of the string.



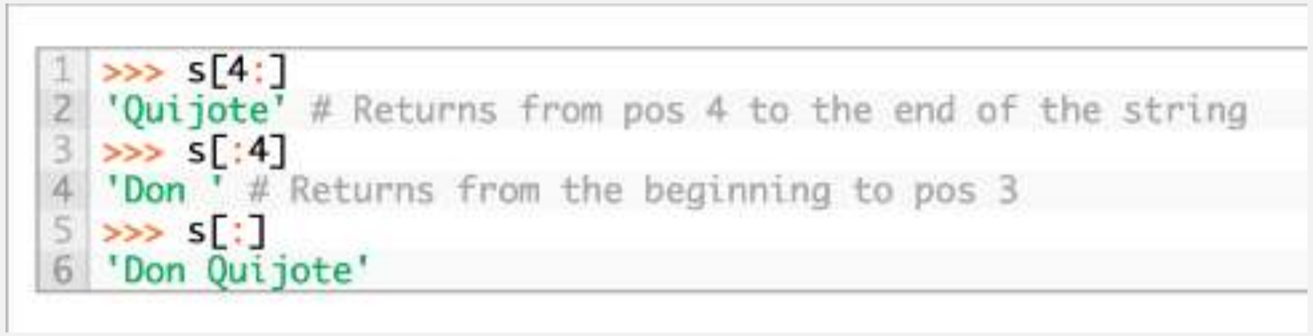
```
>>> s[-1]
'e'
>>> s[-7]
'Q'
```

SLICING STRINGS



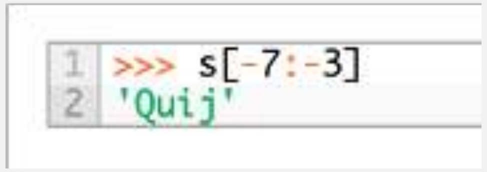
```
>>> s[4:8]
'Quij'
```

- As in the code above, we're specifying that we want to start at position 4 (zero-based) in the string. And, we're saying that we want more characters, up to **but not including** the character at position 8.



```
>>> s[4:]
'Quijote' # Returns from pos 4 to the end of the string
>>> s[:4]
'Don ' # Returns from the beginning to pos 3
>>> s[:]
'Don Quijote'
```

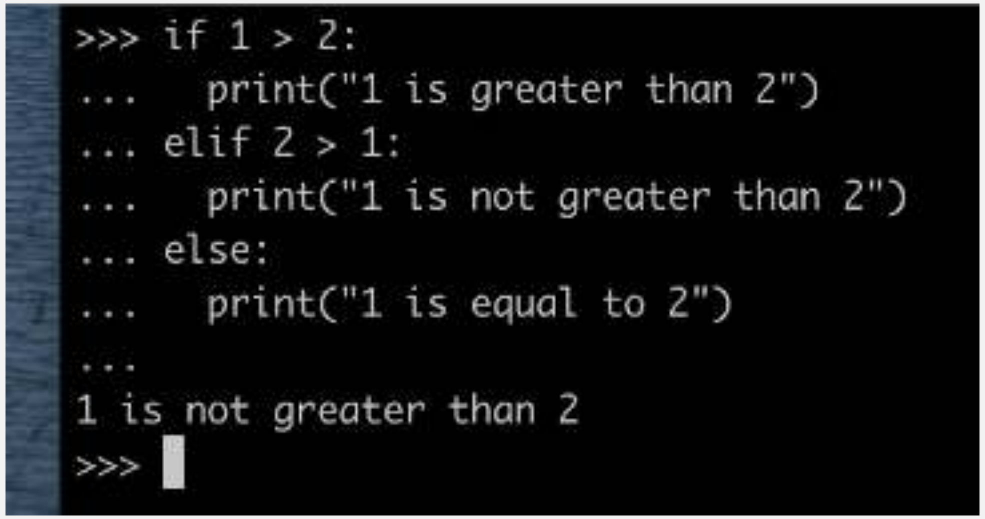
- Just as before, you can use negative numbers as indices, in which case the counting starts at the end of the string (with an index of -1) instead of at the beginning.



```
>>> s[-7:-3]
'Quij'
```

CONTROL FLOW: CONDITIONAL STATEMENTS

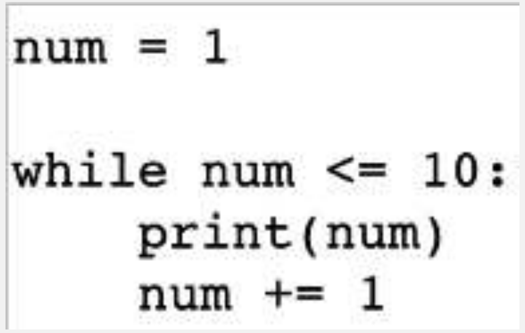
- **“if”** uses an expression to evaluate whether a statement is True or False.
- If it is True, it executes what is inside the “if” statement.
- The “**else**” statement will be executed if the “**if**” expression is **false**.
- You can also use **“elif”** or else if



```
>>> 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")
...
1 is not greater than 2
>>> 
```

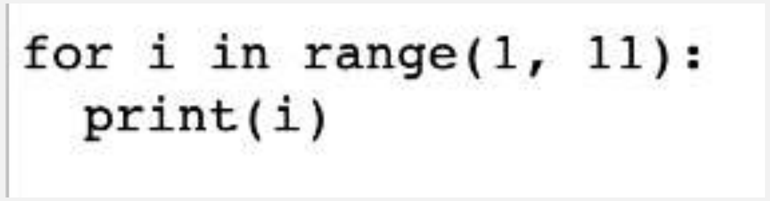
LOOPING / ITERATOR

- 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
```

- **For Looping:** This code will print the same as **while** code: from **1** to **10**.

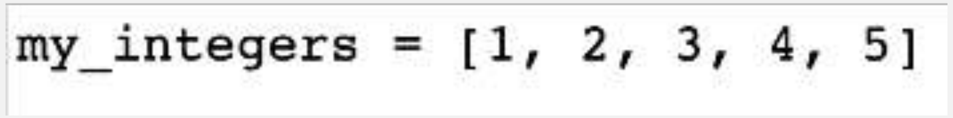


```
for i in range(1, 11):
  print(i)
```

LIST

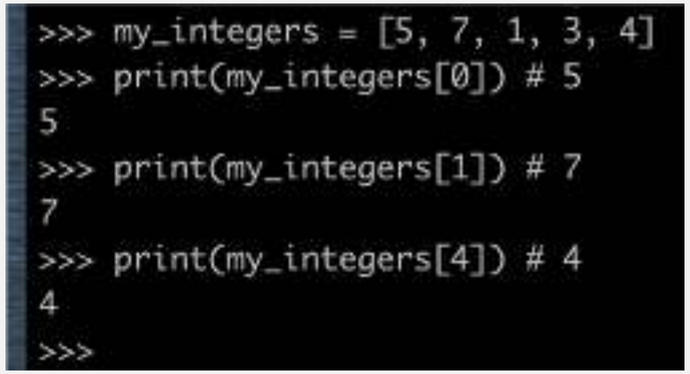
- Imagine you want to store the integer 1 in a variable. But maybe now you want to store 2. And 3, 4, 5 …

- 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]
```

- List has a concept called **index**.
  - The first element gets the index 0 (zero).
  - The second gets 1, and so on.



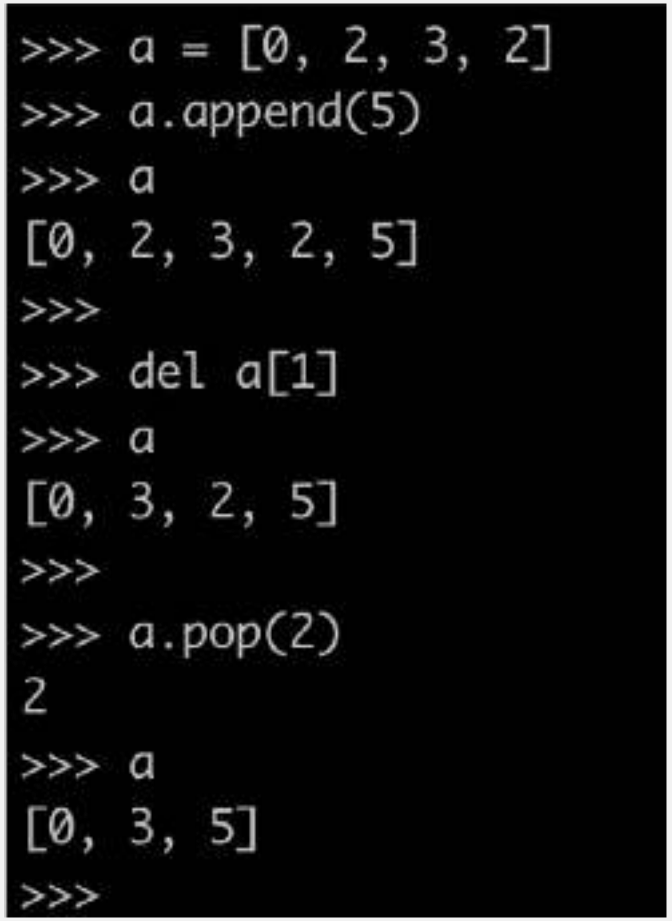
```
>>> my_integers = [5, 7, 1, 3, 4]
>>> print(my_integers[0]) # 5
5
>>> print(my_integers[1]) # 7
7
>>> print(my_integers[4]) # 4
4
>>>
```

ADDING AND REMOVING ELEMENT

- **"append"** adds a new element at the end of list

- **"del"** removes the item at a specific index.

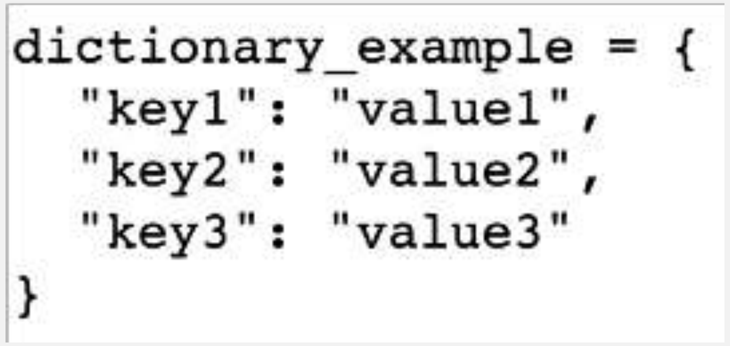
- **"pop"** removes the item at a specific index and returns it.



```
>>> a = [0, 2, 3, 2]
>>> a.append(5)
>>> a
[0, 2, 3, 2, 5]
>>>
>>> del a[1]
>>> a
[0, 3, 2, 5]
>>>
>>> a.pop(2)
2
>>> a
[0, 3, 5]
>>>
```

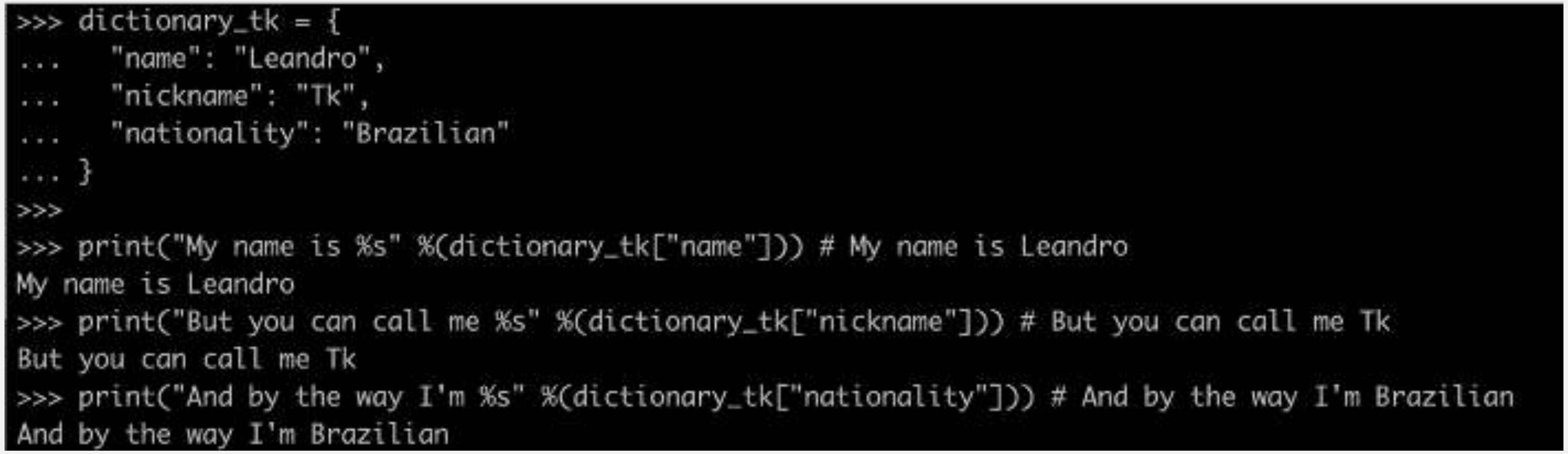
DICTIONARY: KEY-VALUE

- Dictionary is a collection of **key-value pairs**.
- Here's what it looks like:



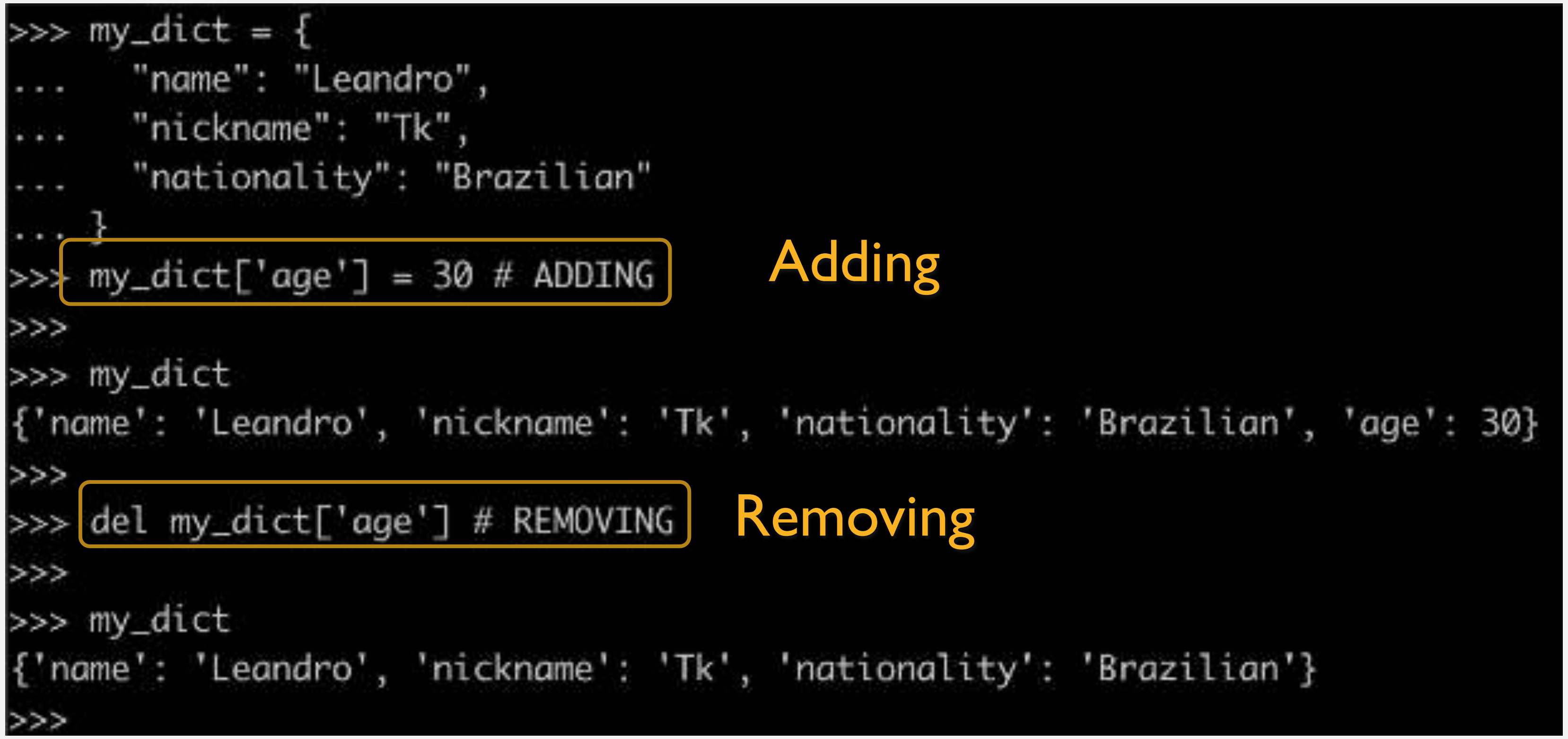
```
dictionary_example = {
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

- 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
My name is Leandro
>>> print("But you can call me %s" %(dictionary_tk["nickname"])) # But you can call me Tk
But you can call me Tk
>>> print("And by the way I'm %s" %(dictionary_tk["nationality"])) # And by the way I'm Brazilian
And by the way I'm Brazilian
```

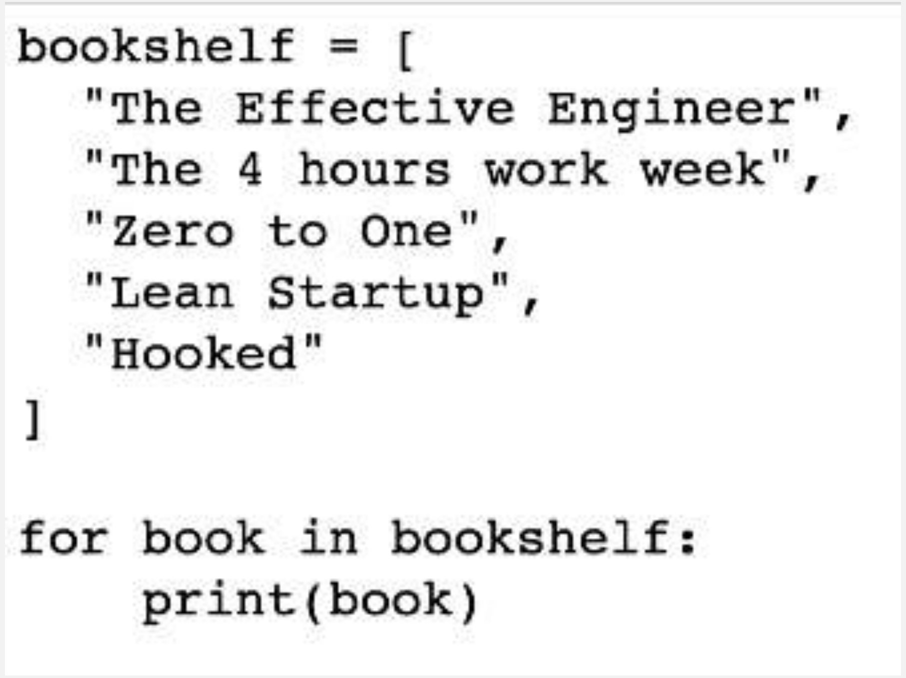
ADDING AND REMOVING KEY-VALUE



```
>>> my_dict = {
...    "name": "Leandro",
...    "nickname": "Tk",
...    "nationality": "Brazilian"
... }
>>> my_dict['age'] = 30 # ADDING
>>>
>>> my_dict
{'name': 'Leandro', 'nickname': 'Tk', 'nationality': 'Brazilian', 'age': 30}
>>>
>>> del my_dict['age'] # REMOVING
>>>
>>> my_dict
{'name': 'Leandro', 'nickname': 'Tk', 'nationality': 'Brazilian'}
>>>
```

ITERATION: LOOPING THROUGH LIST

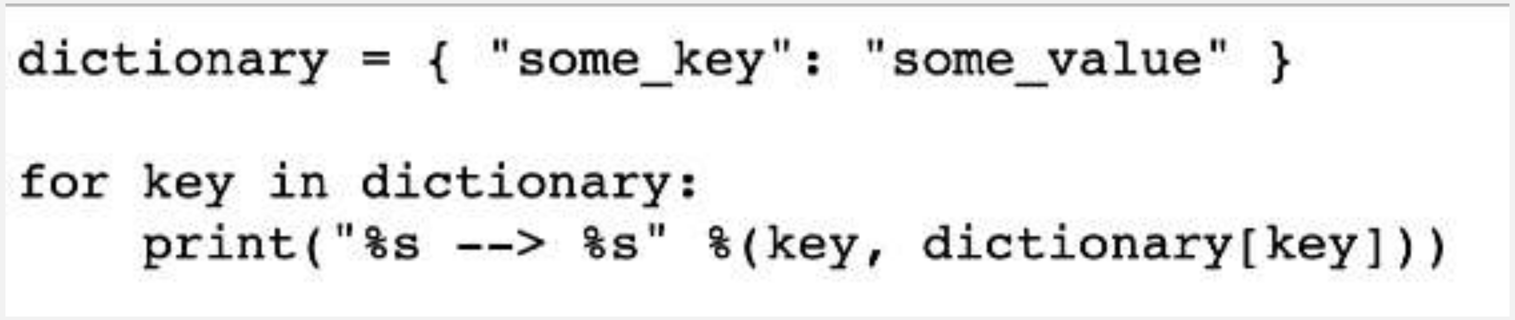
- The List iteration is very simple. We commonly use **”For”** looping. Let’s do it:



```
bookshelf = [
  "The Effective Engineer",
  "The 4 hours work week",
  "Zero to One",
  "Lean Startup",
  "Hooked"
]

for book in bookshelf:
    print(book)
```

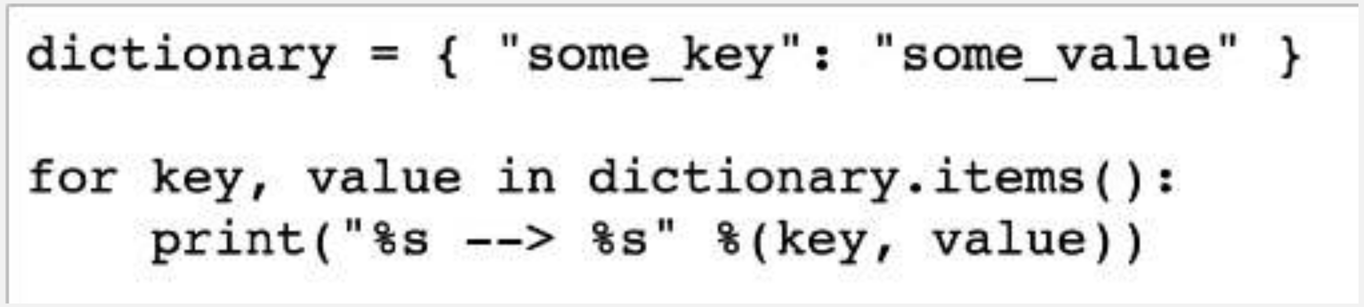
ITERATION: LOOPING THROUGH DICTIONARY

- For a dictionary 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]))
```

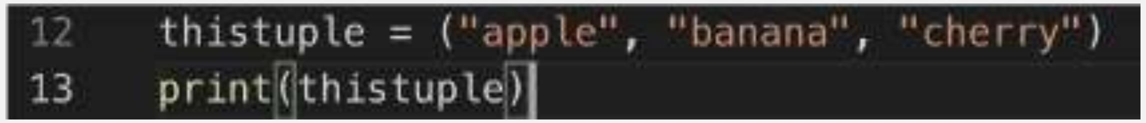
- Another way to do it is to use the iteritems method.



```
dictionary = { "some_key": "some_value" }

for key, value in dictionary.items():
    print("%s --> %s" %(key, value))
```

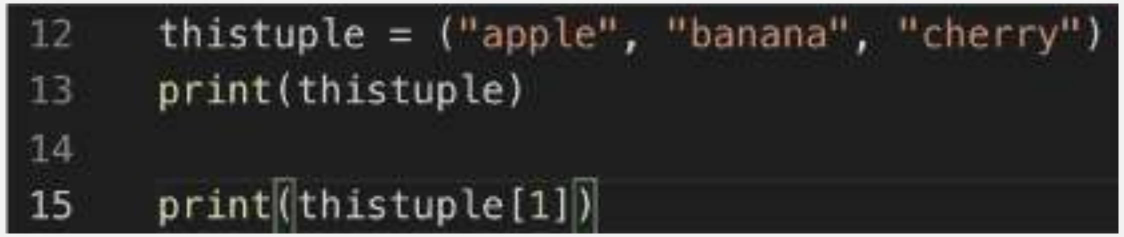
TUPLES

- A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.



```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

- You can access tuple items by referring to the index number, inside square brackets:



```
thistuple = ("apple", "banana", "cherry")
print(thistuple)

print(thistuple[1])
```

- Once a tuple is created, you cannot change its values. Tuples are **unchangeable**.
- You can loop through the tuple items by using a **"for"** loop.

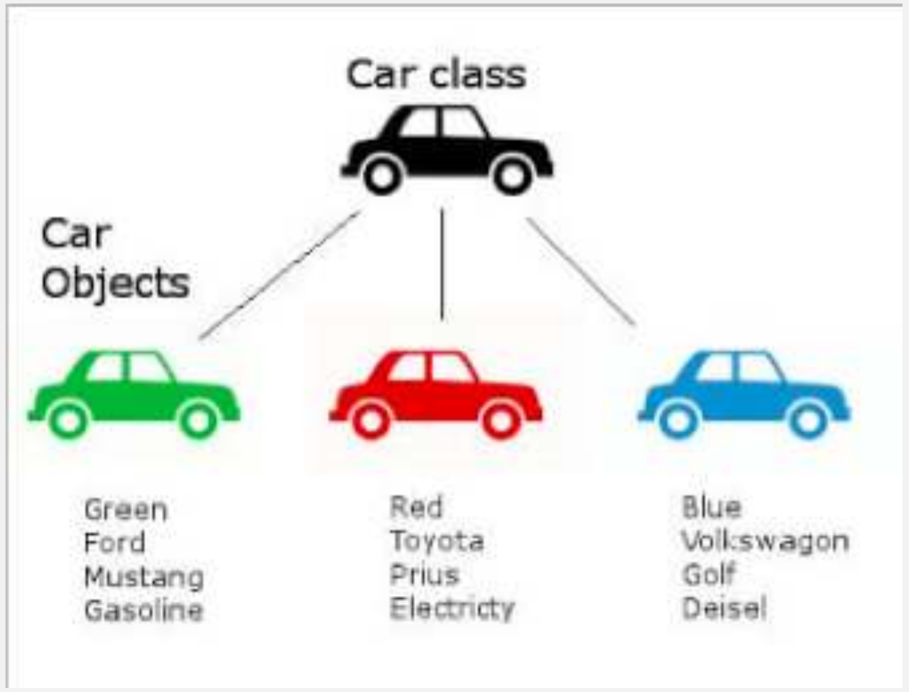
LET’S DO SOME EXERCISES

COMPLETE EXERCISES IN EXERCISE1.PY

CLASSES & OBJECTS

CLASSES AND OBJECTS: BASIC CONCEPTS

- **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.


Car class
Car Objects
Green Ford Mustang Gasoline
Red Toyota Prius Electricty
Blue Volkswagon Golf Deisel

CLASSES AND OBJECTS: BASIC CONCEPTS

Data → Attributes

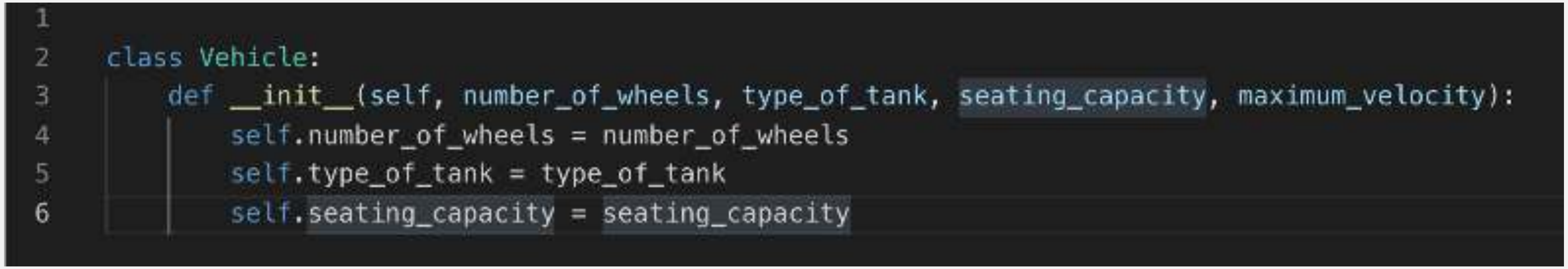
Behavior → Methods

CLASSES AND OBJECTS: BASIC CONCEPTS

- 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.

CLASSES AND OBJECTS

- 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:

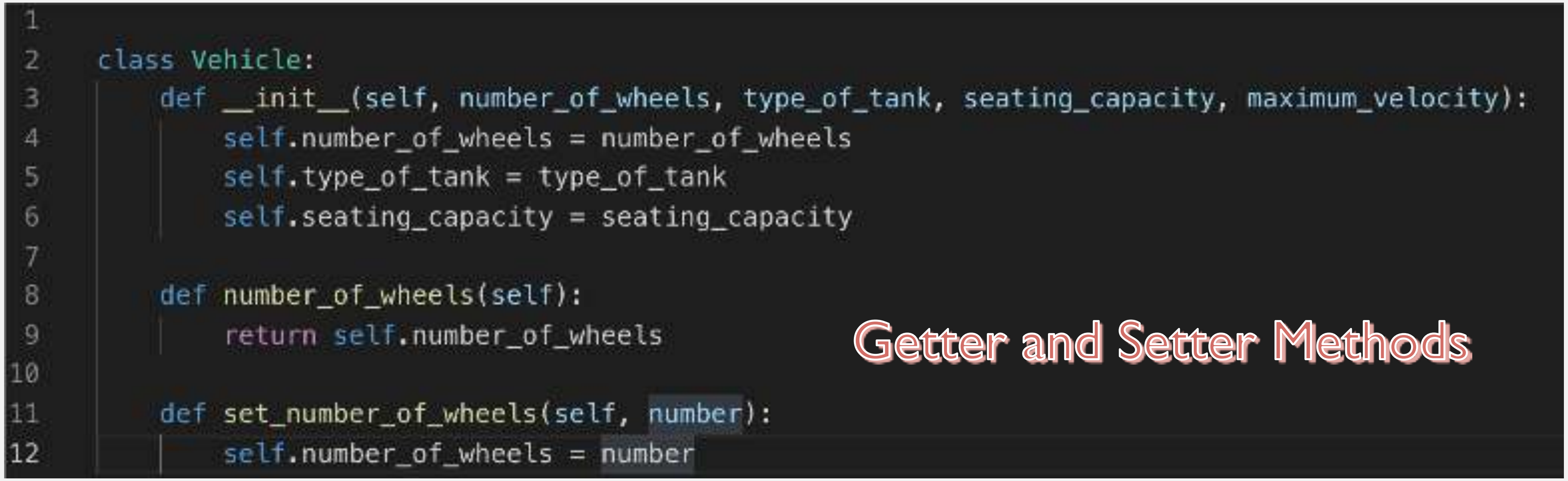


```python

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
```

- We use the init **method**. We call it a constructor method.

CLASSES AND OBJECTS

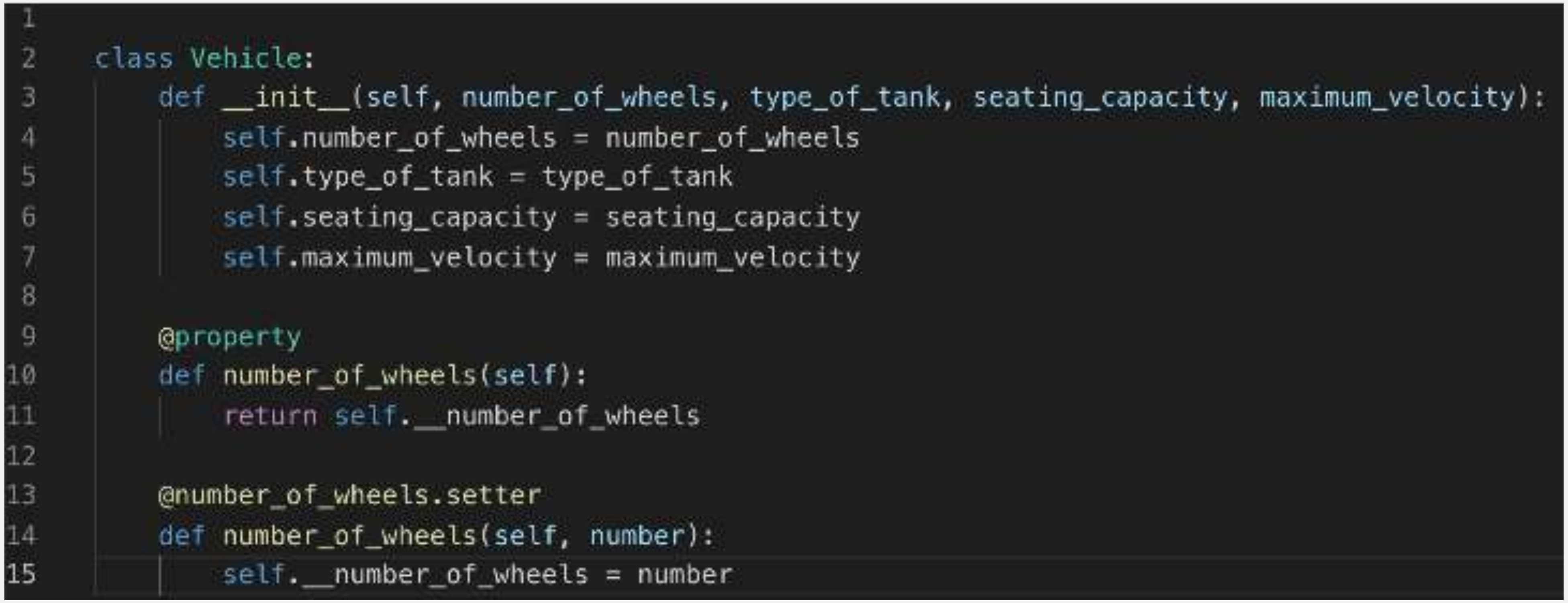
- All attributes are set. But how can we access these attributes' values?
- We **send a message to the object asking about them**. We call it a **method**. It's the **object's behavior**.



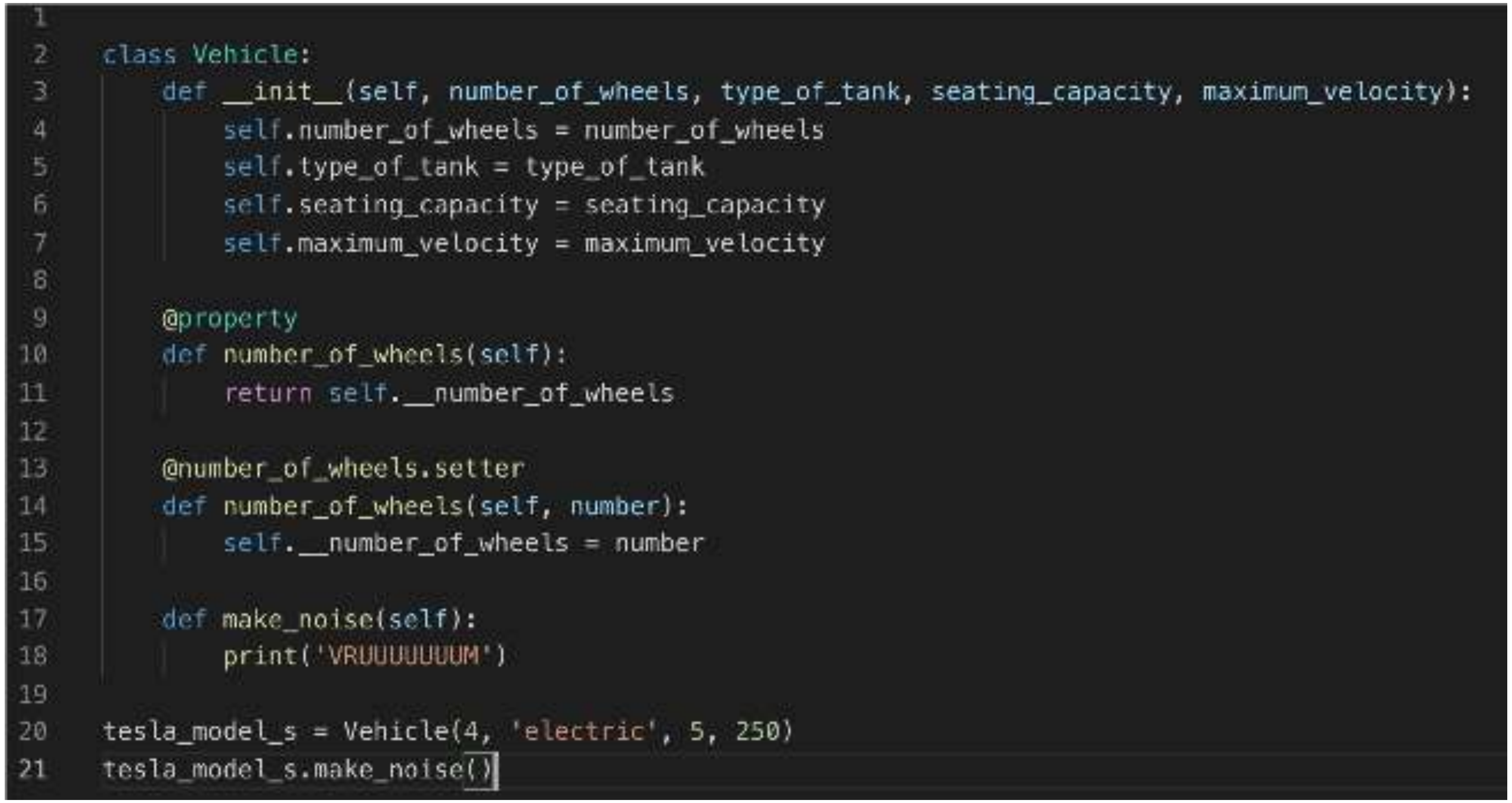
```python

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

    def number_of_wheels(self):
        return self.number_of_wheels

    def set_number_of_wheels(self, number):
        self.number_of_wheels = number
```

GETTERS AND SETTERS

- In Python, we can define "getters and setters" by using @property (decorators). Let's see it with code:



```python

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
```



```python

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

    def make_noise(self):
        print('VRUUUUUUUM')

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

CLASS METHODS

LET’S ADD A MAKE_NOISE METHOD.

STANDARD PYTHON CLASS METHODS

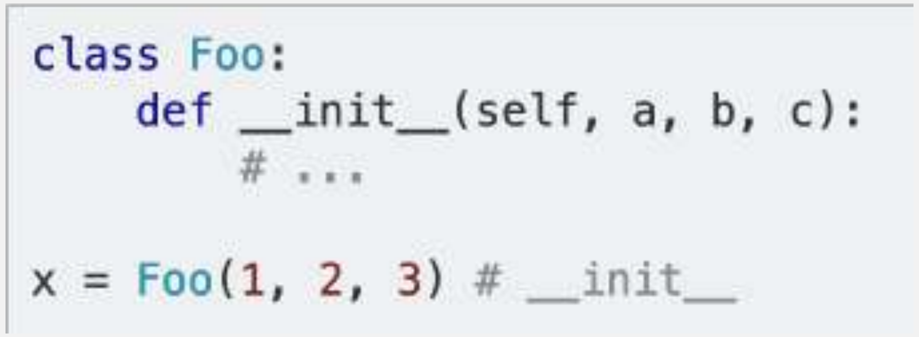
- Python classes have many standard methods. These are some of the more commonly-used ones:
  - __del__: Called when an instance is about to be destroyed, which lets you do any clean-up e.g. closing file handles or database connections.
  - __repr__ and __str__: Both return a string representation of the object.
  - __cmp__: Called to compare the object with another object. Note that this is only used with Python 2.x. In Python 3.x, only *rich comparison methods* are used. Such as __lt__.

STANDARD PYTHON CLASS METHODS

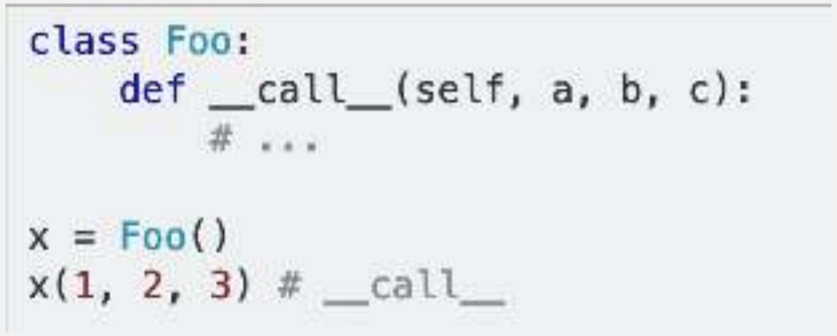
- __lt__, __le__, __eq__, __ne__, __gt__ and __ge__: Called to compare the object with another object. These will be called if defined, otherwise Python will fall-back to using __cmp__.

- __hash__: Called to calculate a hash for the object, which is used for placing objects in data structures such as sets and dictionaries.

- __call__: Lets an object be "called”.



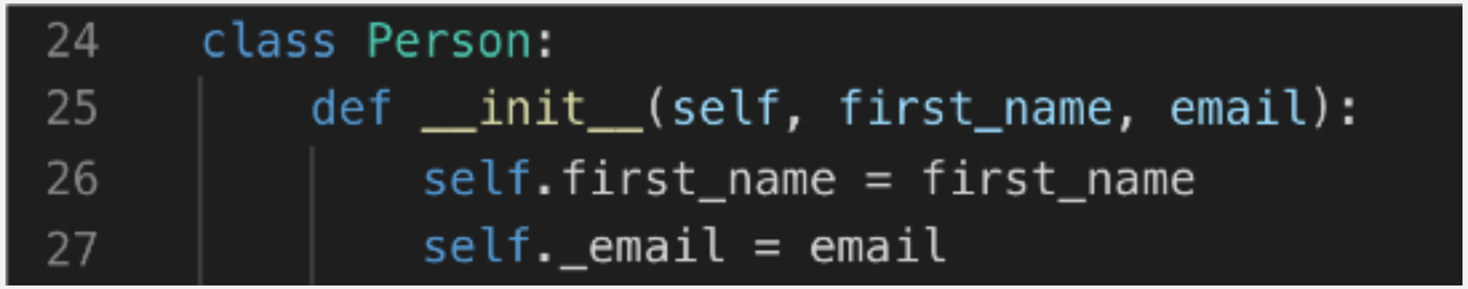
```
class Foo:
    def __init__(self, a, b, c):
        # ...

x = Foo(1, 2, 3) # __init__
```



```
class Foo:
    def __call__(self, a, b, c):
        # ...

x = Foo()
x(1, 2, 3) # __call__
```

PRIVATE PYTHON CLASS METHODS AND MEMBER VARIABLES

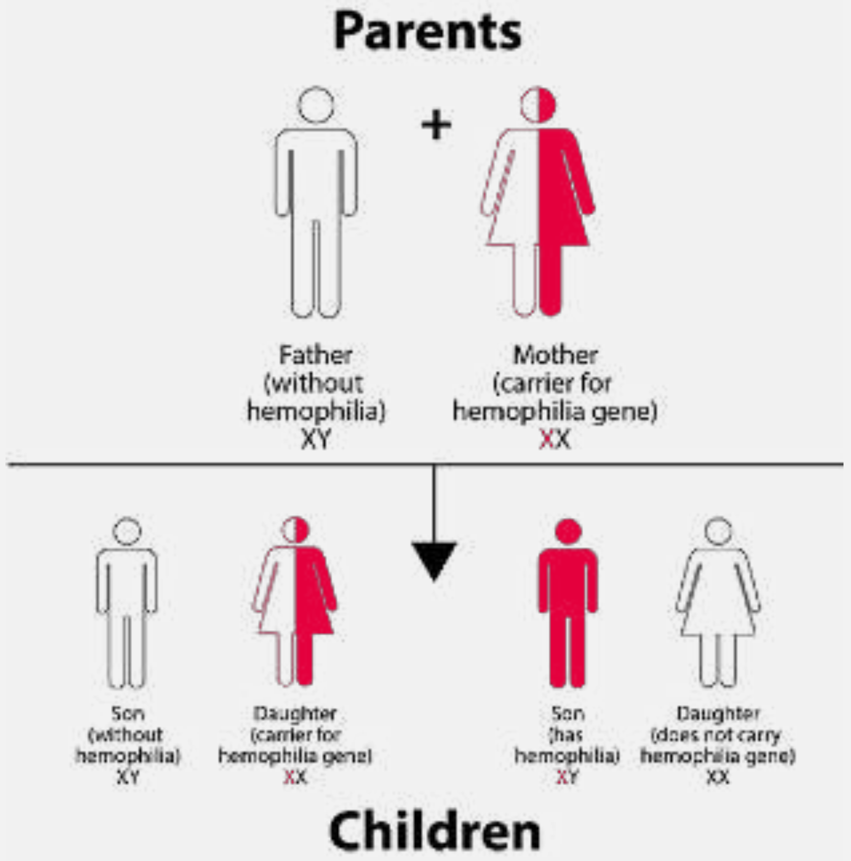
- In Python, methods and member variables are always public i.e. anyone can access them.
- This is not very good for ***encapsulation.***
- Class methods and variable names that start with ***an underscore*** are considered to be private *to that class,* i.e. a derived class can define a similarly-named method or variable and it won't interfere with any other definitions.



```python
class Person:
    def __init__(self, first_name, email):
        self.first_name = first_name
        self._email = email
```

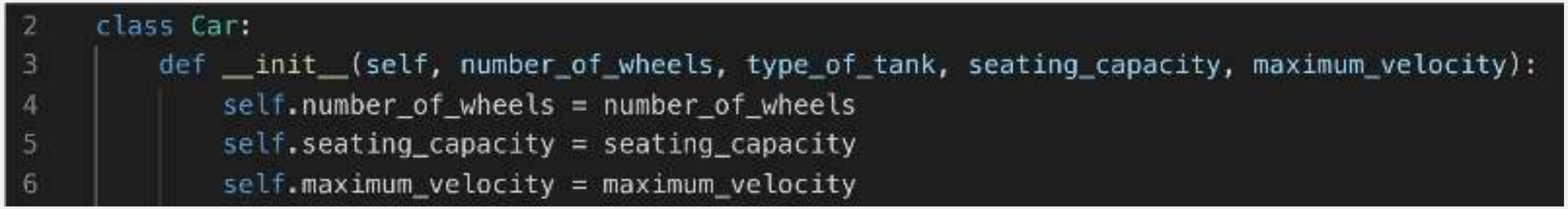
PYTHON CLASS INHERITANCE

- 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 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.


Parents
+
Father (without hemophilia) XY
Mother (carrier for hemophilia gene) XX
Son (without hemophilia) XY
Daughter (carrier for hemophilia gene) XX
Son (has hemophilia) XY
Daughter (does not carry hemophilia gene) XX
Children

PYTHON CLASS INHERITANCE

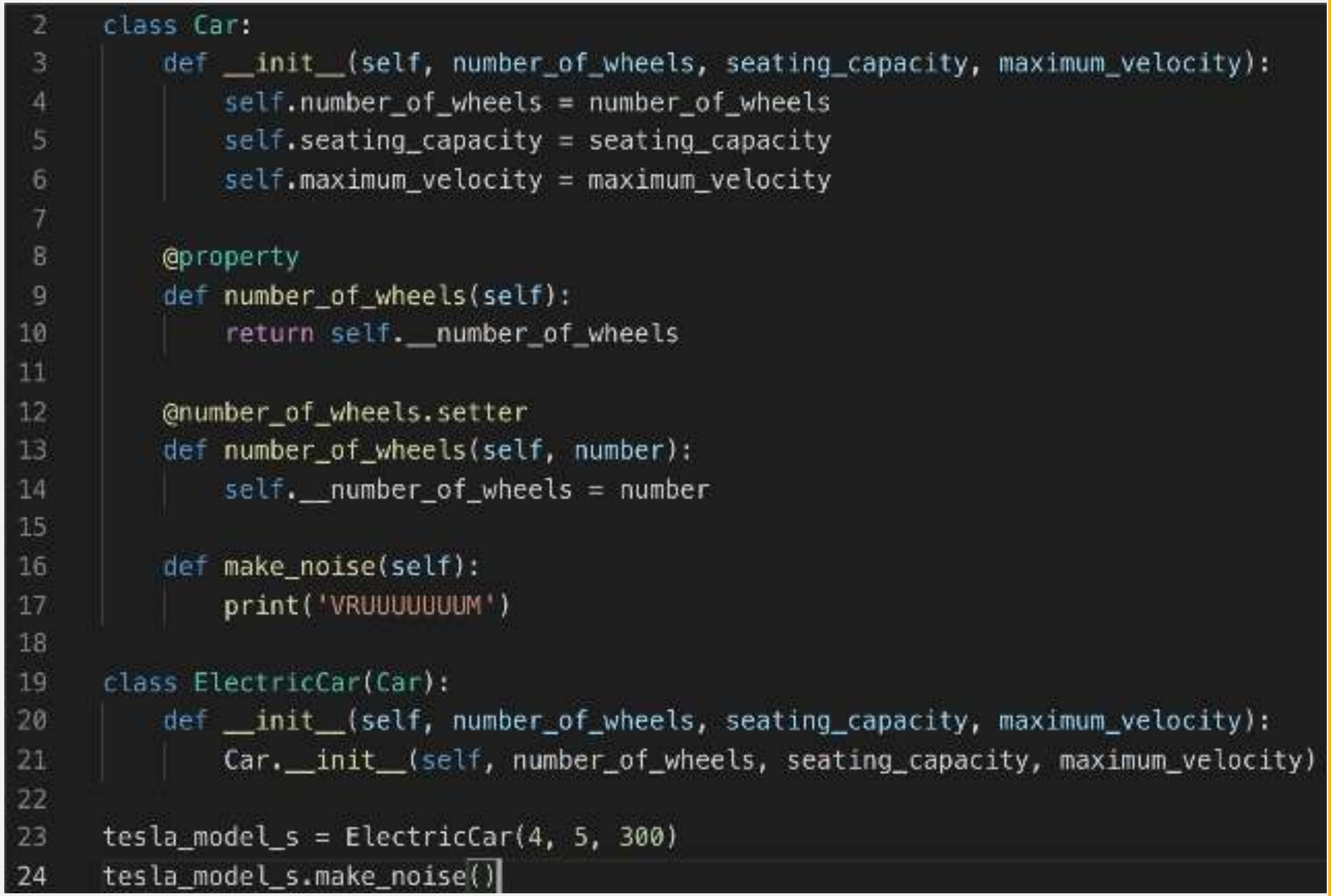
- Imagine a car. Number of wheels, seating capacity and maximum velocity are all attributes of a car. We can say that an **ElectricCar** class inherits these same attributes from the regular **Car** class.



```python
class Car:
    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        self.number_of_wheels = number_of_wheels
        self.seating_capacity = seating_capacity
        self.maximum_velocity = maximum_velocity
```

PYTHON CLASS INHERITANCE

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



```python
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

    @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

    def make_noise(self):
        print('VRUUUUUUUM')

class ElectricCar(Car):
    def __init__(self, number_of_wheels, seating_capacity, maximum_velocity):
        Car.__init__(self, number_of_wheels, seating_capacity, maximum_velocity)

tesla_model_s = ElectricCar(4, 5, 300)
tesla_model_s.make_noise()
```

DECORATORS

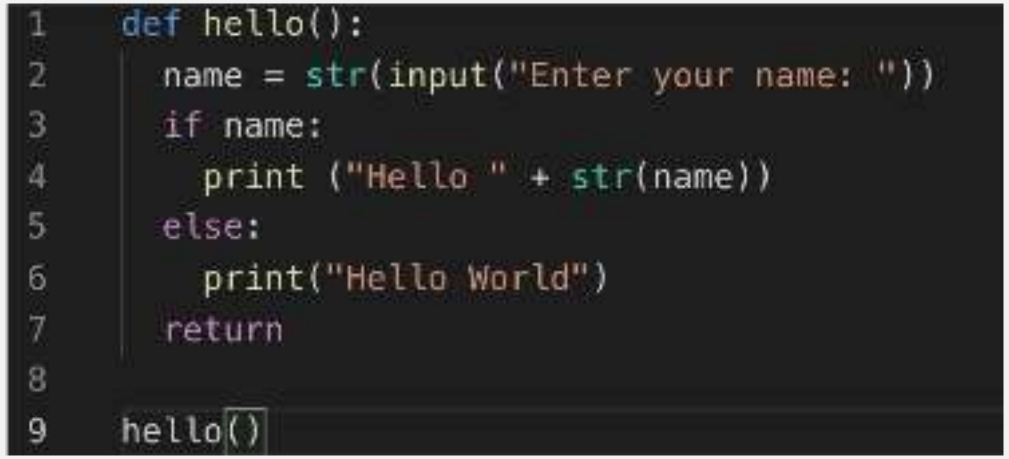
FUNCTIONS

- A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

- There are three types of functions in Python:
  - **Built-in functions**, such as help() to ask for help, min() to get the minimum value, print() to print an object to the terminal.
  - **User-Defined Functions (UDFs).**
  - Anonymous functions, which are also called **lambda functions** because they are not declared with the standard def keyword.

USER-DEFINED FUNCTIONS (UDFS)

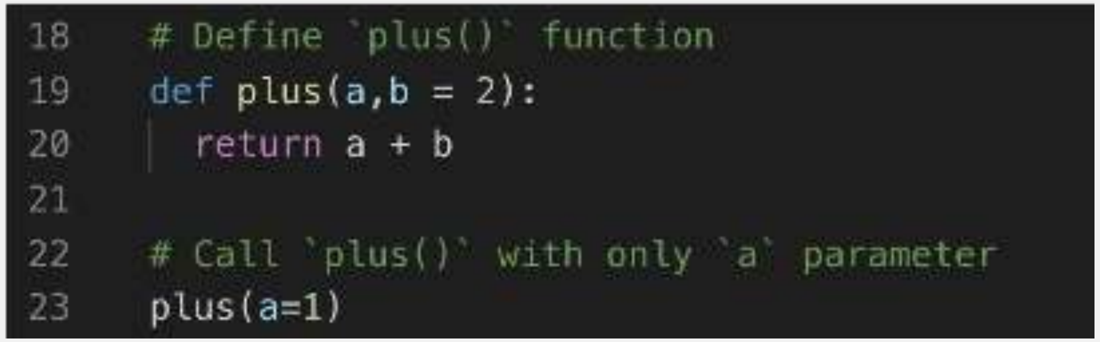
- The four steps to defining a function in Python are the following:
  - Use the keyword **def** to declare the function and follow this up with the function name.
  - Add parameters to the function: they should be within the parentheses of the function. End your line with a colon.
  - Add statements that the functions should execute.
  - End your function with a **return** statement if the function should output something. Without the **return** statement, your function will return an object **None**.



```python
def hello():
  name = str(input("Enter your name: "))
  if name:
    print ("Hello " + str(name))
  else:
    print("Hello World")
  return

hello()
```

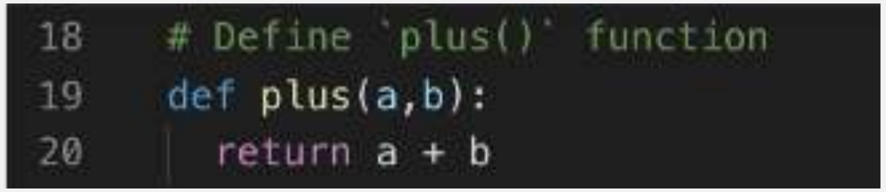
FUNCTION ARGUMENTS IN PYTHON

- There are four types of arguments that Python UDFs can take:
- Default arguments
  - Default arguments are those that take a default value if no argument value is passed during the function call.



```python
# Define `plus()` function
def plus(a,b = 2):
  return a + b

# Call `plus()` with only `a` parameter
plus(a=1)
```

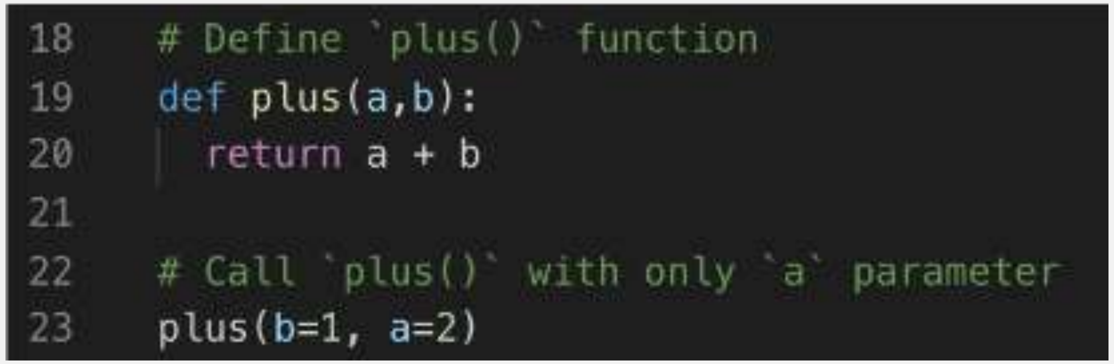
- Required arguments
  - These arguments need to be passed during the function call and in precisely the right order.



```python
# Define `plus()` function
def plus(a,b):
  return a + b
```

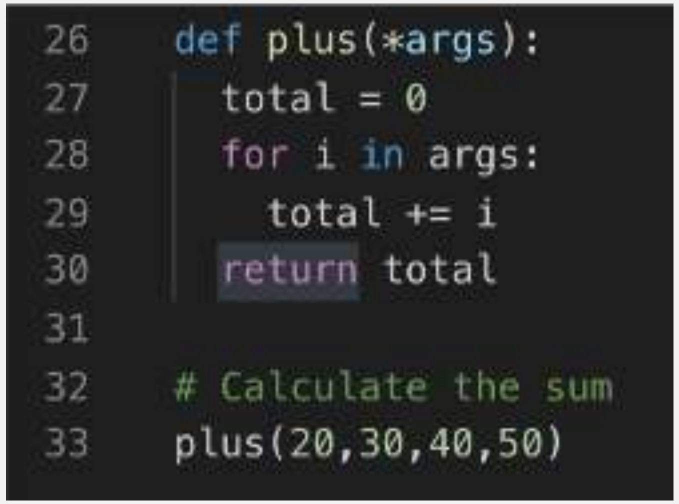
FUNCTION ARGUMENTS IN PYTHON

- Keyword arguments
  - by using the keyword arguments, you can also switch around the order of the parameters and still get the same result when you execute your function.



```python
# Define `plus()` function
def plus(a,b):
  return a + b

# Call `plus()` with only `a` parameter
plus(b=1, a=2)
```

- Variable number of arguments
  - In cases where you don’t know the exact number of arguments that you want to pass to a function, you can use the following syntax with *args.



```python
def plus(*args):
  total = 0
  for i in args:
    total += i
  return total

# Calculate the sum
plus(20,30,40,50)
```

LET'S DO SOME EXERCISES

COMPLETE EXERCISES IN EXERCISE1.PY