DataScientest • com

Object Oriented Programming

Classes

Introduction and prerequisites

> In Python and many other programming languages, *object-oriented programming* consists in creating *classes* of *objects* that contain specific information and tools suitable for their handling.
>
> All the tools we use for data science (*DataFrames*, *scikit-learn models*, *matplotlib*, ...) are built in this way. Understanding the mechanics of Python objects and knowing how to use them is essential to exploit all the features of these very useful tools.
>
> Furthermore, object-oriented programming gives the developer the flexibility to readapt an object to his needs thanks to the *inheritance* that we will see in the second part. Indeed, this technique is widely used to develop packages such as **scikit-learn** which allow a user to easily develop and evaluate the models he needs.
>
> To approach these modules in the best possible conditions, it is important to have completed the *Introduction to Python Programming* module.

Introduction to Classes

In Python, a class is defined as follows:

~~~~
 class Vehicle: # definition of the class Vehicle
    def __init__ (self, a, b = []):
        self.seats = a                 # number of seats in the vehicle
        self.passengers = b            # list containing passenger names

    def print_passengers (self):
        for i in range (len (self.passengers)):
            print (self.passengers [i])
            ```
``` py
car1 = Vehicle(4, ['Pierre', 'Adrian']) # instantiation of an object from the class Vehicle
~~~~

The code above corresponds to the definition of a class named `Vehicle` which contains 2 pieces of information: the number of seats of the `Vehicle` in the variable **`seats`** and the names of the passengers onboard the `Vehicle` in the variable **`passengers`** .

This class contains a `print_passengers` *method* which displays the names of the passengers onboard in the console.

The instruction `car1 = Vehicle (4, ['Pierre', 'Adrian'])` corresponds to the *instantiation* of the class `Vehicle` .

Important Notes and Definitions

- `Vehicle` is a *class* of objects.
- `car1` is an *instance* of the `Vehicle` class.
- `seats` and `passengers` are called the **attributes** of the class `Vehicle` .
- The functions defined in the `Vehicle` class like `print_passengers` and `__init__` are called the **methods** of the `Vehicle` class.
- The `__init__` method takes as arguments the variables that will define the attributes of an instance when it is created.

> ℹ The `__init__` method is automatically called when instantiating any class.

> ⚠ All the methods defined within a class have the `self` argument as their first parameter. This parameter is used to specify the instance which called the method.

Based on syntax from the `Vehicle` class defined above:

- **(a)** Define a new `Complex` class with 2 attributes:
  - **`part_re`** which contains the real part of a complex number.
  - **`part_im`** which contains the imaginary part of a complex number.
- **(b)** Define in the `Complex` class a `display` method which prints a `Complex` in its algebraic form *a* ± *bi*. This method should adapt to the sign of the imaginary part (The method should be able to display `4 - 2i` , `6 + 2i` , `5` , ...).
- **(c)** Instantiate two `Complex` objects corresponding to the complex numbers $4+5i$ and $3-2i$, then print them on the console.

In [20]:

```
### Insert your code here

class Complex:

    def __init__(self, part_re, part_im):
        self.part_re = part_re
        self.part_im = part_im
        self.complex_number = complex(self.part_re, self.part_im)

    def display(self):
        print(self.complex_number)

c1 = Complex(4,5)
c2 = Complex(3,-2)

# Call the display method
Complex.display(c1)
c1.display()
c2.display()

print('\npart_re :', c1.part_re)
print('part_im :', c1.part_im)
```

```
(4+5j)
(4+5j)
(3-2j)

part_re : 4
part_im : 5
```

Show solution

Once an object of a class is instantiated, it is possible to access its attributes and methods using the `.attribute` and `.method()` commands as shown below:

In [21]:

```
class Vehicle:
    def __init__(self, a, b=[]):
        self.seats = a
        self.passengers = b
    def print_passengers(self):
        for i in range(len(self.passengers)):
            print(self.passengers[i])



# Run the cell. You can modify the instantiation so that the changes are reflected.
car2 = Vehicle(4,['Dimitri', 'Charles', 'Yohan'])

print(car2.seats)          # Display of the 'seats' attribute
car2.print_passengers()    # Calling of the print_passengers method
```

```
4
Dimitri
Charles
Yohan
```

The flexibility of classes in object-oriented programming allows the developer to broaden a class by adding new attributes and methods to it. All instances of this class will then be able to call these methods.

For example, we can define in the `Vehicle` class a new `add` method which will add an individual to the passenger list:

```
class Vehicle:
    def __init__(self, a, b = []):
        self.seats = a
        self.passengers = b

    def print_passengers(self):
        for i in range (len(self.passengers)):
            print (self.passengers[i])

    def add(self, name):            #New method
        self.passengers.append(name)
```

❓ In Python, a list is an instance of the built-in `list` class. Thus, calling the `append` method is done in the same way as calling a method from the `Vehicle` or `Complex` classes.

In [22]:

```
class Vehicle:
    def __init__(self, a, b=[]):
        self.seats = a
        self.passengers = b

    def print_passengers(self):
        for i in range(len(self.passengers)):
            print(self.passengers[i])

    def add(self,name): #New methd
        self.passengers.append(name)


car1 = Vehicle(4, ['Charles', 'Paul'])  # Instantiation  of car1
car1.add('Raphaël')                     # 'Raphaël' is added to the list of passengers

car1.print_passengers()             # Display of the list of passengers
```

```
Charles
Paul
Raphaël
```

- **(d)** Define in the `Complex` class an `add` method which takes as argument a `Complex` object and adds it to the instance calling the method. The result of this sum will be stored in the attributes of the `Complex` calling the method.
- **(e)** Test the new `add` method on two instances of the `Complex` class and display their sum.

In [1]:

```
6+2j
```

Out[1]:

```
(6+2j)
```

In [37]:

```
#  Solution 1
class Complex:

    def __init__(self, part_re, part_im):
        self.part_re = part_re
        self.part_im = part_im
        self.complex_number = complex(self.part_re, self.part_im)

    def display(self):
        print(self.complex_number)

    def add(self , c):
        self.c = complex(c)
        c_sum = self.c + self.complex_number
        print(c_sum)

c1 = Complex(4,5)
c2 = Complex(3,-2)

c1.add(10+20j)
c1.add(c2.complex_number)
```

```
(14+25j)
(7+3j)
```

In [59]:

```
#  Solution 2
class Complex:
    def __init__(self, a, b):
        self.part_re = a
        self.part_im = b

    def display(self):
        if(self.part_im < 0):
            print(self.part_re,'-', -self.part_im,'i')
        if(self.part_im == 0):
            print(self.part_re)
        if(self.part_im > 0):
            print(self.part_re, '+',self.part_im,'i')

    def add(self , c):
        if (type(c) == complex or type(c) == int or type(c) == float):
            self.c1 = complex(self.part_re, self.part_im)
            c_sum = self.c1 + c
            print(c_sum)
        else:
            self.part_re = self.part_re + c.part_re
            self.part_im = self.part_im + c.part_im
            print('Total = ', complex(self.part_re, self.part_im))

c1 = Complex(4,5)
c2 = Complex(3,-2)

c1.add(10+10j)
c1.add(c2)
```

```
(14+15j)
Total =  (7+3j)
```

In [25]:

```
class Complex:
    def __init__(self, a, b):
        self.part_re = a
        self.part_im = b

    def display(self):
        if(self.part_im < 0):
            print(self.part_re,'-', -self.part_im,'i')
        if(self.part_im == 0):
            print(self.part_re)
        if(self.part_im > 0):
            print(self.part_re, '+',self.part_im,'i')

    def add(self , c):
        raise NotImplementedError  #### Insert your code here
```

Show solution

✖ Unvalidate

DataScientest • com

Object Oriented Programming

Inheritance

Inheritance

> *Inheritance* is used to create a *subclass* from an existing class. We say that this new class *inherits* from the first one because it will automatically have the same attributes and methods.
>
> Furthermore, it is possible to add attributes or methods that will be specific to this subclass.
>
> In the first part of this module, we introduced the `Vehicle` class defined as follows:
>
> ```
> class Vehicle:
>     def __init__(self, a, b = []):
>         self.seats = a
>         self.passengers = b
>     def print_passengers(self):
>         for i in range (len (self.passengers)):
>             print (self.passengers [i])
>     def add(self, name):
>             self.passengers.append (name)
> ```
>
> We can define a `Motorcycle` class which inherits from the `Vehicle` class as follows:
>
> ```
>     class Motorcycle(Vehicle):
>         def__init__(self, b, c):
>             self.seats = 2
>             self.passengers = b
>             self.brand = c
>
> Motorcycle = Motorcycle(['Pierre', 'Dimitri'], 'Yamaha')
> ```
>
> By rewriting the `__init__` method, any `Motorcycle` object will automatically have 2 seats and a new `brand` attribute.
>
> Thanks to inheritance, we can call the `print_passengers` method defined in the `Vehicle` class from an instance of the `Motorcycle` class.

- **(a)** Run the following cell to convince yourself.

In [4]:

```
class Vehicle: # Definition of the Vehicle class
    def __init__(self, a, b = []):
        self.seats = a       # number of seats in the vehicle
        self.passengers = b  # list containing the names of the passengers

    def print_passengers(self): # Prints the names of the passengers in the vehicle
        for i in range(len(self.passengers)):
            print(self.passengers[i])

    def add(self,name): # Adds a new passenger to the list of passengers.
            self.passengers.append(name)

class Motorcycle(Vehicle):
    def __init__(self, b, c):
        self.seats = 2      # The number of seats is automatically set to 2 and is not modified by the arguments
        self.passengers = b
        self.brand = c

moto1 = Motorcycle(['Pierre','Dimitri'], 'Yamaha')
moto1.add('Yohann')
moto1.print_passengers()
```

```
Pierre
Dimitri
Yohann
```

In [5]:

```
moto1.print_passengers()
```

```
Pierre
Dimitri
Yohann
```

In [6]:

```
moto1.add("Abdullah")
```

In [7]:

```
moto1.print_passengers()
```

```
Pierre
Dimitri
Yohann
Abdullah
```

- **(b)** Define in the `Motorcycle` class an `add` method which will add a name passed as an argument to the list of passengers while checking that there are still seats available. If there are no seats left on the `Motorcycle` , it should display *The vehicle is full*. If there are any remaining, the method should add the name to the list and display the number of seats remaining

In [ ]:

```
class Motorcycle(Vehicle):
    def __init__(self, b, c):
        self.seats = 2
        self.passengers = b
        self.brand = c

    def add(self, name):
        raise NotImplementedError  #### Insert your code here
```

In [8]:

```
class Motorcycle(Vehicle):
    def __init__(self,  b, c, seats = 2):
        self.seats = seats
        self.passengers = b
        self.brand = c

    def add(self, name):
        if len(self.passengers) < self.seats :
            self.passengers.append(name)
            print("Number of seats remaining :", len(self.passengers)-self.seats)

        else:
            print("The vechicle is full.")

moto1 = Motorcycle(['Pierre','Dimitri'], 'Yamaha')
moto2 = Motorcycle(['Pierre'], 'Yamaha')

# add passenger to moto1
moto1.add("Abdullah")
print(moto1.seats)
# add passenger to moto2
moto2.add("Abdullah")
# try to add again passenger to moto2
moto2.add("Nurullah")
```

```
The vechicle is full.
2
Number of seats remaining : 0
The vechicle is full.
```

In [9]:

```
class Motorcycle(Vehicle):
    def __init__(self, a, b, c):
        self.seats = a
        self.passengers = b
        self.brand = c

    def add(self, name):
        if len(self.passengers) >= 2:
            print("The vechicle is full.")
        else:
            self.passengers.append(name)
            print("Number of seats remaining :", len(self.passengers)-self.seats)
```

In [10]:

```
moto1.__class__.__name__
```

Out[10]:

```
'Motorcycle'
```

Hide solution

In [14]:

```
class Motorcycle(Vehicle):
    def __init__(self, b, c):
        self.seats = 2
        self.passengers = b
        self.brand = c

    def add(self, name):
        if(len(self.passengers) <  self.seats):
            self.passengers.append(name)
            print('There are', self.seats - len(self.passengers), 'seats left.')
        else:
            print("The vehicle is full.")
```

We run the following instructions:

```
car2 = Vehicle(3, ['Antoine', 'Thomas', 'Raphaël'])
moto2 = Motorcycle(['Guillaume', 'Charles'], 'Honda')
car2.add('Benjamin')
moto2.add('Dimitri')
```

In addition, we recall that the classes `Vehicle` and `Motorcycle` are defined as follows:

```
class Vehicle:
        def __init__(self, a, b = []):
            self.seats = a
            self.passengers = b

        def print_passengers(self):
            for i in range(len(self.passengers)):
                print(self.passengers [i])

        def add(self, name):
            self.passengers.append(name)
```

```
class Motorcycle(Vehicle):
    def __init__(self, b, c):
        self.seats = 2
        self.passengers = b
        self.brand = c

    def add(self, name):
        if(len(self.passengers) <  self.seats):
            self.passengers.append(name)
            print('There are', self.seats - len(self.passengers), 'seats left.')
        else:
            print("The vehicle is full.")
```

```
VBox(children=(ToggleButtons(button_style='success', options=('Answer A', 'Answer B', 'Answer C'), tooltips=('…
```

- What is the output of the `print(car2.seats)` instruction?
  - A: `Antoine Thomas Raphael Benjamin`
  - B: `4`
  - C: `The vehicle is full.`
  - D: `3`

```
VBox(children=(ToggleButtons(button_style='success', options=('Answer A', 'Answer B', 'Answer C', 'Answer D'),…
```

- Why is the instruction `car3 = Vehicle(4)` well written but the instruction `moto3 = Motorcycle(6)` returns an error?
  - A: A `Motorcycle` object cannot have 6 seats.
  - B: The constructor of the `Vehicle` class takes only one argument.
  - C: An argument is missing when initializing the `moto3` instance.

```
VBox(children=(ToggleButtons(button_style='success', options=('Answer A', 'Answer B', 'Answer C'), tooltips=('…
```

- **(c)** Create a `Convoy` class which will have 2 attributes: The first attribute, named `vehicle_list` is a list of `Vehicle` objects and the second attribute `length` is the total number of vehicles in the `Convoy` . A convoy will be automatically initialized with a `Vehicle` that has 4 seats and no passengers.

- **(d)** Define in `Convoy` class an `add_vehicle` method which will add an object of type `Vehicle` at the end of the list of vehicles of the convoy. Do not forget to update the length of the convoy.

In [15]:

```
### Insert your code here

class Convoy():
    def __init__(self, length = 0, vehicle_list = [] ):
        self.length = length
        self.vehicle_list = vehicle_list


    def add_vehicle(self, new_vehicle):
        self.vehicle_list.append(new_vehicle)
        self.length += 1

car1 = Vehicle(4, [])
car2 = Vehicle(3, ["Abdullah", "Hatice"])

convoy1 = Convoy()
convoy1.add_vehicle("car2")
print(convoy1.vehicle_list)
```

```
['car2']
```

Hide solution

In [4]:

```
class Convoy:
    def __init__(self):
        self.vehicle_list = []
        self.vehicle_list.append(Vehicle(4)) # vehicule_list is initialized with a list containing 1 vehicle
        self.length = 1                      # the length attribute is initialized to 1

    def add_vehicle(self, vehicle):
        self.vehicle_list.append(vehicle)    # a Vehicle is added at the end of the list
        self.length = self.length + 1        # update the length of the convoy
```

- **(e)** Initialize a `convoy1` object of the `Convoy` class.

- **(f)** Add the passenger `"Albert"` to the first vehicle of `convoy1` .

- **(g)** Add a motorcycle from the brand `"Honda"` to `convoy1` which will be driven by `"Raphael"` .

In [17]:

```
### Insert your code here

convoy1 = Convoy()

convoy1.vehicle_list[0].add("Albert")

moto1 = Motorcycle(["Raphael"], "Honda")

convoy1.add_vehicle(moto1)
```

In [18]:

```
convoy1.vehicle_list
```

Out[18]:

```
[<__main__.Vehicle at 0x7f30f433f6a0>, <__main__.Motorcycle at 0x7f30f433f8e0>]
```

Hide solution

In [5]:

```
convoy1 = Convoy()                                        # Instanciation of the convoy

convoy1.vehicle_list[0].add('Albert')                     # "Albert" is added to the first vehicle in the convoy

convoy1.add_vehicle(Motorcycle(['Raphael'] , 'Honda')) # We have to remember that the first argument of the Motorcycle
                                                          # constructor is a list and not a string.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-070e9d4959c6> in <module>
----> 1 convoy1 = Convoy()                                        # Instanciation of the convoy
      2 
      3 convoy1.vehicle_list[0].add('Albert')                     # "Albert" is added to the first vehicle in the convoy
      4 
      5 convoy1.add_vehicle(Motorcycle(['Raphael'] , 'Honda')) # We have to remember that the first argument of the Motorcycle

<ipython-input-4-e8d75ab76f00> in __init__(self)
      2     def __init__(self):
      3         self.vehicle_list = []
----> 4         self.vehicle_list.append(Vehicle(4)) # vehicule_list is initialized with a list containing 1 vehicle
      5         self.length = 1                      # the length attribute is initialized to 1
      6 

NameError: name 'Vehicle' is not defined
```

In [20]:

```
convoy1.vehicle_list
```

Out[20]:

```
[<__main__.Vehicle at 0x7f30f433f610>, <__main__.Motorcycle at 0x7f30f433f7f0>]
```

- **(h)** Write a small script that will display all the passengers in `convoy1`.

In [21]:

```
### Insert your code here

class Convoy():
    def __init__(self):
        self.vehicle_list = []
        car1 = Vehicle(0, [])
        self.vehicle_list.append(car1)
        self.length = 1


    def add_vehicle(self, new_vehicle):
        self.vehicle_list.append(new_vehicle)
        self.length += 1

    def display(self):
        num = 0
        for i in self.vehicle_list:
            print(i.passengers)
            for j in i.passengers:
                num +=1
        print("Number of vehicle :", len(self.vehicle_list))
        print("Total number of passengers : ", num)



convoy1 = Convoy()

convoy1.vehicle_list[0].add('Albert')

convoy1.add_vehicle(Motorcycle(['Raphael', 'Ali'], 'Honda'))
```

In [22]:

```
convoy1.display()
```

```
['Albert']
['Raphael', 'Ali']
Number of vehicle : 2
Total number of passengers :  3
```

Hide solution

In [168]:

```
for vehicle in convoy1.vehicle_list: # We go through the list of vehicles in the convoy
    vehicle.print_passengers()       # We use the print_passengers method of the Vehicle class
```

```
Albert
Raphael
```

In [ ]:

✖ Unvalidate

DataScientest • com

Object Oriented Programming

Predefined classes

> In Python, many predefined classes such as the `list` , `tuple` or `str` classes are regularly used to facilitate the developer's tasks. Like all other classes, they have their own attributes and methods that are available to the user.
>
> One of the great interests of object oriented programming is to be able to create classes and share them with other developers. This is done through packages such as `numpy` , `pandas` or `scikit-learn` . All of these packages are actually classes created by other developers in the Python community to give us tools that will make easier to develop our own algorithms.
>
> We will first discuss one of the most important predefined object classes, the `list` class, in order to learn how to use it to its full potential.
> Next, we will briefly introduce the `DataFrame` class of the `pandas` package and learn to identify and manipulate its methods.

1. The list class

- **(a)** Use the `dir(list)` command to display all attributes and methods of the `list` class.

In [1]:

```
dir(list)
```

Out[1]:

```
['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

- **(b)** Use the `help(list)` command to display the *documentation* of the `list` class. This documentation is useful to understand how to use the methods of a class.

In [1]:

```
help(list)
```

```
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
```

> ℹ The `dir` and `help` commands are the first commands to run when you don't understand how to use a method of a class or when you can't remember the name of a method.

- **(c)** Using the `dir` or `help` commands, find a method that will reverse the order of the elements of the list `list_1` .

In [2]:

```
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]

### Insert your code here

list_1.reverse()
list_1
```

Out[2]:

```
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Hide solution

In [4]:

```
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]

list_1.reverse() # reverses the order of the elements of the list calling it.
                 # this method WILL MODIFY the list that calls it.

list_1
```

Out[4]:

```
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

- **(d)** Using the `dir` and `help` commands, find a method that will insert the value `10` in the fifth position of the list `list_2` .

In [5]:

```
list_2 = [1, 2, 3, 4, 5, 6, 7, 8, 9]

### Insert your code here


list_2.insert(4, 10)
list_2
```

Out[5]:

```
[1, 2, 3, 4, 10, 5, 6, 7, 8, 9]
```

Hide solution

In [6]:

```
list_2 = [1, 2, 3, 4, 5, 6, 7, 8, 9]

list_2.insert(4, 10) # inserts the value 10 at the index 4 (fifth position in Python) of the list.

list_2
```

Out[6]:

```
[1, 2, 3, 4, 10, 5, 6, 7, 8, 9]
```

- **(e)** Using the `dir` and `help` commands, find a method that will sort the list `list_3` .

In [7]:

```
list_3 = [5, 2, 4, 9, 6, 7, 8, 3, 10, 1]

### Insert your code here

list_3.sort()
list_3
```

Out[7]:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Show solution

2. The DataFrame class

> The `pandas` package contains a class named `DataFrame` whose usefulness makes it the most used package by data scientists to manipulate data.
>
> To use the `pandas` package, you must first import it. Then, to instantiate a `DataFrame` , you must call its constructor defined in the `pandas` package.

- **(a)** Import the `pandas` package under the alias `pd` .

In [8]:

```
### Insert your code here

import pandas as pd

df = pd.DataFrame()
df
```

Out[8]:

—

Hide solution

In [ ]:

```
import pandas as pd

df = pd.DataFrame()
```

> If you run the `dir(df)` or `dir(pd.DataFrame)` instructions, you will see that the `DataFrame` class has a lot of methods and attributes. It is very difficult to remember them all, hence the usefulness of the commands `dir` and `help` .
>
> However, given the length of the documentation, it is not practical to directly use the `dir(df)` or `help (df)` commands. To have direct access to the documentation of a specific method, you can instead use the `help` function with the argument `object.method` .

- **(c)** Using the `help(pd.DataFrame)` command, build a `DataFrame` named `df1` using the list `list_4` .

In [9]:

```
help(pd.DataFrame)
```

```
Help on class DataFrame in module pandas.core.frame:

class DataFrame(pandas.core.generic.NDFrame, pandas.core.arraylike.OpsMixin)
 |  DataFrame(data=None, index: 'Axes | None' = None, columns: 'Axes | None' = None, dtype: 'Dtype | None' = None, copy: 'bool | None' = None)
 |
 |  Two-dimensional, size-mutable, potentially heterogeneous tabular data.
 |
 |  Data structure also contains labeled axes (rows and columns).
 |  Arithmetic operations align on both row and column labels. Can be
 |  thought of as a dict-like container for Series objects. The primary
 |  pandas data structure.
 |
 |  Parameters
 |  ----------
 |  data : ndarray (structured or homogeneous), Iterable, dict, or DataFrame
 |      Dict can contain Series, arrays, constants, dataclass or list-like objects. If
 |      data is a dict, column order follows insertion-order.
 |
 |      .. versionchanged:: 0.25.0
 |         If data is a list of dicts, column order follows insertion-order.
```

In [10]:

```
list_4 = [1, 5, 45, 42, None, 123, 4213 , None, 213]

### Insert your code here

df1 = pd.DataFrame({"col1" : list_4})
df1 = pd.DataFrame(list_4)
df1.head(2)
```

Out[10]:

	0
0	1.0
1	5.0

Hide solution

In [11]:

```
list_4 = [1, 5, 45, 42, None, 123, 4213, None, 213]

df1 = pd.DataFrame(data = list_4)

df1
```

Out[11]:

	0
0	1.0
1	5.0
2	45.0
3	42.0
4	NaN
5	123.0
6	4213.0
7	NaN
8	213.0

In [12]:

```
list_4.__contains__(1)
```

Out[12]:

```
True
```

> By displaying the `DataFrame` `df1`, you can see that some of its values are assigned to `NaN`, which stands for *Not a Number*. In practice, this happens very often when we import a database that is unprocessed. The `DataFrame` class contains a very simple method to get rid of these missing values: the `dropna` method.

In [13]:

```
### Insert your code here

df2 = df1.dropna()
df2
```

Out[13]:

	0
0	1.0
1	5.0
2	45.0
3	42.0
5	123.0
6	4213.0
8	213.0

Hide solution

In [14]:

```
df2 = df1.dropna()

df2
```

Out[14]:

	0
0	1.0
1	5.0
2	45.0
3	42.0
5	123.0
6	4213.0
8	213.0

> Another method of the `DataFrame` class which is widely used is the `apply` method. This method allows you to apply a function passed as an argument to all the entries of the `DataFrame` calling the method.

- **(e)** Define a function named `divide2` which returns the division by 2 of a number passed as argument.
- **(f)** Create a `DataFrame` named `df3` which will contain the values of `df2` divided by 2.

In [15]:

```
# Insert your code here

def divide2(x):
    return x / 2

df3 = df2.apply(divide2)
df3
```

Out[15]:

	0
0	0.5
1	2.5
2	22.5
3	21.0
5	61.5
6	2106.5
8	106.5

Hide solution

In [16]:

```
def divide2(x):
    return x/2

df3 = df2.apply(divide2)  # applies the function divide2 to all entries of the DataFrame

df3
```

Out[16]:

	0
0	0.5
1	2.5
2	22.5
3	21.0
5	61.5
6	2106.5
8	106.5

> The `DataFrame` class has many methods like `apply` or `dropna` that you will explore in more depth during your learning journey. The `list` class being too basic for the needs of data scientists, these methods make the `DataFrame` class the standard to manipulate data.
>
> All the packages that you will be invited to use in your training will be handled as objects, i.e. you will first have to initialize an object of the class (`DataFrame` , `Scikit Model` ,

Validate

DataScientest • com

Object Oriented Programming

Built-in methods

1. Built-in methods

> All classes defined in Python have methods whose name is already defined. The first example of such a method we have seen is the `__init__` method which allows us to initialize an object, but it is not the only one.
>
> Built-in methods give the class the ability to interact with predefined Python functions such as `print`, `len`, `help` and basic operators. These methods usually have the affixes `__` at the beginning and end of their names, which allows us to easily identify them.
>
> Thanks to the `dir(object)` command, we can get an overview of some predefined methods common to all Python objects.

In [1]:

```
dir(object)
```

Out[1]:

```
['__class__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__']
```

2. The `str` method

> One of the most convenient methods is the `__str__` method which is called automatically when the user calls the `print` command on an object. This method returns a character string that represents the object passed to it.
>
> All the classes in Python on which we can apply the `print` function have this method in their definition.

In [2]:

```
# The int class

i = 10
i.__str__()
```

Out[2]:

```
'10'
```

Hide solution

In [3]:

```
# The list class

tab = [1, 2 , 3, 4, 5, 6]
tab.__str__()
```

Out[3]:

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

When we define our own classes, it is better to define a `__str__` method rather than a method like `display` as we did previously. This will allow all future users to directly use the `print` function to display the object on the console.

We are going to use the `Complex` class that we defined in the first module of introduction to object-oriented programming:

```
class Complex:
    def __init__(self, a, b):
       self.part_re = a
       self.part_im = b

    def display(self):
       if(self.part_im < 0):
            print(self.part_re,'-', -self.part_im,'i')
       if(self.part_im == 0):
            print(self.part_re)
       if(self.part_im > 0):
            print(self.part_re, '+',self.part_im,'i')
```

- **(a)** Define in the `Complex` class the `__str__` method which **must return a character string** corresponding to the algebraic representation $a + bi$ of a complex number. This method will replace the `display` method.

ℹ To get the string representation of a number, you can call its `__str__` method.

- **(b)** Instantiate a `Complex` object corresponding to the number $6 - 3i$ then display it on the console using the `print` function.

In [ ]:

```
class Complex:
    def __init__(self, a = 0, b = 0):
        self.part_re = a
        self.part_im = b
    def __str__(self):
        raise NotImplemented  #### Insert your code here
```

In [4]:

```
class Complex:
    def __init__(self, a = 0, b = 0):
        self.part_re = a
        self.part_im = b
    def __str__(self):
        if(self.part_im < 0):
            return f"{self.part_re} -  {-self.part_im} i "
        if(self.part_im == 0):
            return f"{self.part_re}"
        if(self.part_im > 0):
            return f"{self.part_re} + {self.part_im} i"

c = Complex(5 , 10)
print(c)
```

```
5 + 10 i
```

In [5]:

```
# Output gives an TypeError !!!!!!

class Complex:
    def __init__(self, a = 0, b = 0):
        self.part_re = a
        self.part_im = b

    def __str__(self):
        if(self.part_im < 0):
            print(str(self.part_re) + str(self.part_im) + 'i')

        if(self.part_im == 0):
            print(str(self.part_re))

        if(self.part_im > 0):
            print(str(self.part_re) + '+' +str(self.part_im) + 'i')

c = Complex(5, 10)
print(c)
```

```
5+10i

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-4762435f2fd6> in <module>
     17 
     18 c = Complex(5, 10)
---> 19 print(c)

TypeError: __str__ returned non-string (type NoneType)
```

In [6]:

```
class Car():
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        print("Init Func executed")
    def __str__(self):
        return f"Make : {self.make} \nModel : {self.model} \nYear : {self.year}"


car = Car('Verso', 'Toyota', 2008)
print(car)
```

```
Init Func executed
Make : Verso 
Model : Toyota 
Year : 2008
```

Hide solution

In [7]:

```
class Complex:
    def __init__(self, a = 0, b = 0):
        self.part_re = a
        self.part_im = b

    def __str__(self):
        if(self.part_im < 0):
            return self.part_re.__str__() + self.part_im.__str__() + 'i'  # returns 'a' '-b' 'i'

        if(self.part_im == 0):
            return self.part_re.__str__()    # returns 'a'

        if(self.part_im > 0):
            return self.part_re.__str__() + '+' + self.part_im.__str__() + 'i' # returns 'a' '+' 'b' + 'i'

z = Complex(6, -3)
print(z)
```

```
6-3i
```

3. Comparison methods

> As for the `int` or `float` classes, we would like to be able to compare the objects of the `Complex` class with each other, i.e. to be able to use the comparison operators (`>` , `<` , `==` , `!=` , ...).
>
> To this end, the Python developers have provided the following methods:
>
> - `__le__` / `__ge__` : *lesser or equal / greater or equal*
> - `__lt__` / `__gt__` : *lesser than / greater than*
> - `__eq__` / `__ne__` : *equals / not equal*
>
> These methods are automatically called when the comparison operators are used and return a Boolean value (`True` or `False`).

In [8]:

```
x = 5

print(x > 3)  # True

print(x.__gt__(3)) # True
                                    # These two types of syntax are strictly equivalent
print(x < 3) # False

print(x.__lt__(3)) # False
```

```
True
True
False
False
```

> For the `Complex` class, we will make the comparison thanks to the modulus calculated by the formula $|a + bi| = \sqrt{a^2 + b^2}$

- **(a)** Define in the `Complex` class a `mod` method which returns the modulus of the `Complex` calling the method. You can use the `sqrt` function of the `numpy` package to calculate a square root.
- **(b)** Define in the `Complex` class the methods `__lt__` and `__gt__` (strictly lower and strictly higher). These methods must return a boolean.
- **(c)** Perform the two comparisons defined above on the complex numbers $3 + 4i$ and $2 - 5i$

In [9]:

```python
import numpy as np

class Complex:
    def __init__(self, a = 0, b = 0):
        self.part_re = a
        self.part_im = b

    def __str__(self):
        if(self.part_im < 0):
            return self.part_re.__str__() + self.part_im.__str__() + 'i'  # returns 'a' '-b' 'i'

        if(self.part_im == 0):
            return self.part_re.__str__()    # returns 'a'

        if(self.part_im > 0):
            return self.part_re.__str__() + '+' + self.part_im.__str__() + 'i' # returns 'a' '+' 'b' + 'i'

    def mod(self):
        a = self.part_re**2
        b = self.part_im**2
        return np.sqrt(a + b)

# The 'other' argument in the following methods corresponds to the object
# of type Complex that we wish to compare to

    def __lt__(self, other):
        other_modus = np.sqrt(other.real**2 + other.imag**2)

        if self.mod() < other_modus:
            return True
        else:
            return False


    def __gt__(self, other):
        other_modus = np.sqrt(other.real**2 + other.imag**2)

        if self.mod() > other_modus:
            return True
        else:
            return False


c = Complex(3, 4)
c.mod()

print(c.__lt__(1-4j))
print(f"{c} < 1-4j = ", c.__lt__(1-4j))
print(f"{c} > 1-4j = ", c.__gt__(1-4j))
```

```
False
3+4i < 1-4j =  False
3+4i > 1-4j =  True
```

In [10]:

```python
c = 1-4j
print("Real = ", c.real, "\nImaginary = ", c.imag)
```

```
Real =  1.0
Imaginary =  -4.0
```

In [ ]:

```python
import numpy as np

class Complex:
    def __init__(self, a = 0, b = 0):
        self.part_re = a
        self.part_im = b

    def __str__(self):
        if(self.part_im < 0):
            return self.part_re.__str__() + self.part_im.__str__() + 'i'  # returns 'a' '-b' 'i'

        if(self.part_im == 0):
            return self.part_re.__str__()    # returns 'a'

        if(self.part_im > 0):
            return self.part_re.__str__() + '+' + self.part_im.__str__() + 'i' # returns 'a' '+' 'b' + 'i'

    def mod(self):
        raise NotImplemented ### Insert your code here

# The 'other' argument in the following methods corresponds to the object
# of type Complex that we wish to compare to

    def __lt__(self, other):
        raise NotImplemented ### Insert your code here

    def __gt__(self, other):
        raise NotImplemented ### Insert your code here
```

Hide solution

In [11]:

```
import numpy as np

class Complex:
    def __init__(self, a = 0, b = 0):
        self.part_re = a
        self.part_im = b

    def __str__(self):
        if(self.part_im < 0):
            return self.part_re.__str__() + self.part_im.__str__() + 'i'  # returns 'a' '-b' 'i'

        if(self.part_im == 0):
            return self.part_re.__str__()    # returns 'a'

        if(self.part_im > 0):
            return self.part_re.__str__() + '+' + self.part_im.__str__() + 'i' # returns 'a' '+' 'b' + 'i'

    def mod(self):
        return np.sqrt( self.part_re ** 2 + self.part_im ** 2)  # returns (sqrt(a² + b²))

    def __lt__(self, other):
        if(self.mod() < other.mod()):   # returns True if |self| < |other|
            return True
        else:
            return False

    def __gt__(self, other):
        if(self.mod() > other.mod()):   # returns True if |self| > |other|
            return True
        else:
            return False


z1 = Complex(3, 4)
z2 = Complex(2, 5)
print(z1 > z2)
print(z1 < z2)
```

```
False
True
```

In [ ]:

Validate