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.

> 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
```

In [7]:

Hide solution

```
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]:

```
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]:

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

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

In [ ]:

```
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