I - Python Programming

I.4. Oriented Object Programming (OOP)

A `class` is a **formal description** of how an object is designed, i.e. which attributes and methods it has: you can think of it as **a pattern or a structure**.

Each specific object is then an **instance** of the `class` : they all work as described by the `class` definition, but each instance can have different characteristic values from another.

Examples:

- `Human` is a `class` and each person is an instance of the `class`
- `Car` is a `class` and each car is an instance of the `class`

Basically, **everything is an object in Python**.

I. How does a `class` work?

I.1. Define a class and create instances

Beware of conventions:

- a filename is written in `lower_snake_case` (ex: **magic_dice.py**)
- a class name is always written in `UpperCamelCase` (ex: `class MagicDice()``)

In [1]:
```
# Let's define a claas to represent the way dices work
class GotCharacter():
    """
    An empty class representing characters of GOT.
    """
    pass
```

In [9]:
```
# Now we can create an instance of this class
jon = GotCharacter()

# In fact, we can creat as many instances as we want
daenerys = GotCharacter()

print(jon == daenerys)
```

```
False
```

Ok, so now we have two people, but they don't do anything right ? Let's improve this !

II.2. Attributes

Attributes describe the **characteristics (or the state)** of the class objects.

In [3]:
```
# Let's add (dynamically) some attributes to our characters
jon.name = "John Snow"
jon.position = "The Wall"
jon.is_alive = True

daenerys.name = "Daenerys Targaryen"
daenerys.position = "Mereen"
daenerys.is_alive = True
```

> Be careful : this is **NOT** the proper way to create instance attributes, we'll see that just below.

In [4]:
```
# We can now check the values for each attributes of ours instances
print(jon.position)
print(daenerys.is_alive)
```

```
The Wall
True
```

In fact, each instance has a dictionnary **`__dict__`**, which is used to store their attributes and their corresponding values

In [5]:
```
# We can use this __dict__ to check all the attributes at once
jon.__dict__
```

Out[5]:
```
{'name': 'John Snow', 'position': 'The Wall', 'is_alive': True}
```

I.3. Methods

Methods represent the **actions (or the behavior)** of the class objects. A method differs from a function only in two aspects:

- it belongs to a class and it is defined **within** a class
- the **first parameter** in the definition of a method has to be **a reference to the instance**, which called the method: this parameter is usually called `self`.

In [6]:
```
class GotCharacter():
    """
    A class representing characters of GOT.
    """
    def say_hello(self):
        print("Hello")
```

In [7]:
```
jon.__dict__
```

Out[7]:
```
{'name': 'John Snow', 'position': 'The Wall', 'is_alive': True}
```

In [10]:
```
jon.say_hello()
```

```
Hello
```

II. Special methods

Special methods are a set of predefined methods you can use to enrich your classes. They let you **emulate the behavior of built-in types**.

They are easy to recognize because they start and end with double underscores, for example `__init__` or `__str__`. These "dunder methods" or "dunders" are also sometimes called "magic methods".

II.1. `__init__` : we can now properly define attributes with a `contructor`

In [51]:
```
class GotCharacter():
    """
    A class representing characters of GOT.
    """
    def __init__(self, name=None, position="Unknown", is_alive=True):
        self.name = name
        self.position = position
        self.is_alive = is_alive
```

- We can directly pass the attributes while creating the instances

In [58]:
```
jon = GotCharacter("John Snow", "The Wall")
```

In [53]:
```
daenerys = GotCharacter("Daenerys Targaryen", "Mereen")

daenerys.__dict__
```

Out[53]:
```
{'name': 'Daenerys Targaryen', 'position': 'Mereen', 'is_alive': True}
```

- We can also elaborate more interesting methods

In [13]:
```
class GotCharacter():
    """
    A class representing characters of GOT.
    """
    def __init__(self, name=None, position="Unknown", is_alive=True):
        self.name = name
        self.position = position
        self.is_alive = is_alive

    def die(self):
        self.is_alive = False

    def move_to(self, destination):
        self.position = destination
```

In [16]:
```
# Let's apply these methods to our characters
daenerys.move_to("Dragonstone")
daenerys.position
```

Out[16]:
```
'Dragonstone'
```

In [17]:
```
jon.die()
jon.is_alive
```

Out[17]:
```
False
```

II.2. __repr__ and __str__

__repr__ : what Python displays an object when you enter its name

In [18]:
```
class GotCharacter():
    """
    A class representing characters of GOT.
    """
    def __init__(self, name=None, position="Unknown", is_alive=True):
        self.name = name
        self.position = position
        self.is_alive = is_alive

    def __repr__(self):
        return self.name
```

In [21]:
```
jon
```

Out[21]:
```
John Snow
```

__str__ : what Python displays when you `print` an object

In [25]:
```
class GotCharacter():
    """
    A class representing characters of GOT.
    """
    def __init__(self, name=None, position="Unknown", is_alive=True):
        self.name = name
        self.position = position
        self.is_alive = is_alive

    def __str__(self):
        return self.name
```

In [27]:
```
jon
```

Out[27]:
```
<__main__.GotCharacter at 0x105c26358>
```

In [28]:
```
print(jon)
```

```
John Snow
```

II.3. __add__

In [29]:
```
class GotCharacter():
    """
    A class representing characters of GOT.
    """
    def __init__(self, name=None, position="Unknown", is_alive=True):
        self.name = name
        self.position = position
        self.is_alive = is_alive

    def __add__(self, other):
        return f"{self.name} is in a relationship with {other.name}"
```

In [31]:
```
jon + daenerys
```

Out[31]:
```
'John Snow is in a relationship with Daenerys Targaryen'
```

There are a lot of special methods : don't forget to check the documentation (https://docs.python.org/3/reference/datamodel.html)

III. Inheritance and overriding

- A class can inherit attributes and behaviour methods from another class, called the **superclass** or parent class.
- A class which inherits from a superclass is called a **subclass** or child class.

In [32]:
```
# Definition of a subclass
class Stark(GotCharacter):
    pass
```

In [43]:
```
arya = Stark("Arya Stark", "King's Landing")
```

`isinstance()`

In [34]:
```
isinstance(jon, Stark)
```

Out[34]:
```
False
```

In [35]:
```
isinstance(jon, GotCharacter)
```

Out[35]:
```
True
```

In [36]:
```
isinstance(arya, Stark)
```

Out[36]:
```
True
```

In [37]:
```
isinstance(arya, GotCharacter)
```

Out[37]:
```
True
```

Overriding

Method overriding allows a subclass to provide **a different implementation** of a method that is already defined by its superclass or by one of its superclasses.

The implementation in the subclass overrides the implementation of the superclass by providing a method with the same name, same parameters or signature, and same return type as the method of the parent class.

In [38]:
```
class Stark(GotCharacter):
    def die(self):
        print("Stark house never really dies!")
```

In [41]:
```
arya.die()
arya.is_alive
```

```
Stark house never really dies!
```

Out[41]:
```
True
```

`super()`

In [42]:
```
# We can override `__init__` by using `super()`to retrive the parent `__init__()` and adding new attribute
s.
class Stark(GotCharacter):
    def __init__(self, name=None, position="Unknown", is_alive=True):
        super().__init__(name=None, position="Unknown", is_alive=True)
```

```
        super().__init__(name=None, position="Unknown", is_alive=True)
        self.house = "Stark"
```

In [44]:
```
arya.__dict__
```

Out[44]:
```
{'name': None, 'position': 'Unknown', 'is_alive': True, 'house': 'Stark'}
```

In [45]:
```
# We could also write:
class Stark(GotCharacter):
    def __init__(self, name=None, position="Unknown", is_alive=True):
        GotCharacter.__init__(self, name=None, position="Unknown", is_alive=True)
        self.house = "Stark"

# But in that case, we loose a precious information : the dependency relation between the Stark class
# and its parent class.
```

IV. Encapsulation

We can **restrict access to methods and variables** in order to prevent the data from being modified by accident. This is known as encapsulation.

Private variables and private methods

They are accessible only in their own class, but not from outside. Their name starts with a double underscore __.

In [57]:
```
class GotCharacter():
    """
    A class representing characters of GOT.
    """
    def __init__(self, name=None, position="Unknown", is_alive=True):
        self.__name = name
        self.__position = position
        self.__is_alive = is_alive

    def __pretend_to_die(self):
        print("I'm dead")
```

In [60]:
```
jon.__name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-60-bccee1730930> in <module>()
----> 1 jon.__name

AttributeError: 'GotCharacter' object has no attribute '__name'
```

In [61]:
```
jon.__pretend_to_die()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-61-26c6066c8156> in <module>()
----> 1 jon.__pretend_to_die()

AttributeError: 'GotCharacter' object has no attribute '__pretend_to_die'
```

Properties

We write properties to control the access to private variables.

In [64]:
```
class GotCharacter():
    """
    A class representing characters of GOT.
    """
    def __init__(self, name=None, position="Unknown", is_alive=True):
        self.__name = name
        self.__position = position
        self.__is_alive = is_alive

    @property
    def position(self):
        return self.__position

    @position.setter
    def position(self, position):
        self.__position = position

    @property
    def name(self):
        return self.__name

jon = GotCharacter("Jon Snow", "The Wall")
```

In [65]:
```
# We cannot directly access the private attribute `__position`
jon.__position
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-65-4e49c5108b19> in <module>()
      1 # We cannot directly access the private attribute `__position`
----> 2 jon.__position

AttributeError: 'GotCharacter' object has no attribute '__position'
```

In [66]:
```
# We access it thanks to the `@property`
jon.position
```

Out[66]:
```
'The Wall'
```

In [67]:
```
# We can also change it thanks to the `setter`
jon.position = "Beyond the Wall"
jon.position
```

Out[67]:
```
'Beyond the Wall'
```

What's the point in doing this ? We have a better control access. See in the following example:

In [68]:
```
# We cannont directly access the private attribute `__name`
jon.__name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-68-985b9cf8c5fd> in <module>()
      1 # We cannont directly access the private attribute `__name`
----> 2 jon.__name
```