ILP 2023 – W6S2-3
Object-oriented programming

Matthieu DE MARI
Singapore University of Technology and Design

Outline (Week6, Session2-3 – W6S2-3)

- From object dictionaries to proper objects
- Attributes, methods and constructor method
- The __dict__ special attribute
- About special methods
- Has-a relationship

If time allows,

- (Is-a relationship and inheritance)
- (Attributes privacy)
- (Setters, getters and properties)

What is object-oriented programming?

- **Object-oriented programming (OOP)** is a programming language **paradigm** organized around **custom objects**.

- Sometimes, the **objects** we need for our programs cannot be described using only the basic int/float/str/list/dict types.

- While it is true that we can assemble variables that relate to the same concept into dictionnaries (what we described as object-oriented thinking)…
- We often prefer to **create our own custom types/objects**.

Our toy example for this class: a video game hero

- Let us say we would like to code a video game and design our main character.
- Our **main character** can be represented as a **custom class object.**
- It has several **attributes**, such as
  - A name (string type, ‘Sir Meowsalot’)
  - A class (string type, ‘Warrior’)
  - Some lifepoints (int types, 100)
  - Other attributes (intelligence, strength, speed, armor, etc.)

Class attributes

- Let us say we would like to code a video game and design our main character.
- Our **main character** can be represented as a **custom class object.**
- It has several **attributes**, such as
  - A name (string type, ‘Sir Meowsalot’)
  - A class (string type, ‘Warrior’)
  - Some lifepoints (int types, 100)
  - Other attributes (intelligence, strength, speed, armor, etc.)

```
1  class Hero:
2      # Hero's name
3      name = "Sir Meowsalot"
4      # Hero's class
5      hero_class = "Warrior"
6      # Hero's maximal lifepoints
7      maximal_lifepoints = 100
8      # Hero's current lifepoints
9      current_lifepoints = 100
```

Class attributes

- Let us say we would like to code a video game and design our main character.
- Our **main character** can be represented as a **custom class object.**
- It has several **attributes**, such as
  - A name (string type, ‘Sir Meowsalot’)
  - A class (string type, ‘Warrior’)
  - Some lifepoints (int types, 100)
  - Other attributes (intelligence, strength, speed, armor, etc.)

```
1  class Hero:
2      # Hero's name
3      name = "Sir Meowsalot"
4      # Hero's class
5      hero_class = "Warrior"
6      # Hero's maximal lifepoints
7      maximal_lifepoints = 100
8      # Hero's current lifepoints
9      current_lifepoints = 100
```

```
1  my_cat_hero = Hero()
```

```
1  print(my_cat_hero)
```

```
<__main__.Hero object at 0x000001D8D3459CC0>
```

```
1  print(my_cat_hero.name)
```

```
Sir Meowsalot
```

Class methods

- On top of a class **attributes**, we can also define **methods.**
- **Methods:** functions which apply to our custom object.
- **Methods** may use some of the objects **attributes**.

```
1  class Hero:
2
3      # Hero's name
4      name = "Sir Meowsalot"
5      # Hero's class
6      hero_class = "Warrior"
7
8      def meow(self):
9          '''
10         A first method.
11         '''
12         print("meow.")
13
14
15     def loud_meow(self):
16         '''
17         A second method, calling some attributes of the class.
18         '''
19         print("{} SAYS MEOW.".format(self.name))
```

```
1  my_cat_hero = Hero()
```

```
1  my_cat_hero.meow()
```

```
meow.
```

```
1  my_cat_hero.loud_meow()
```

```
Sir Meowsalot SAYS MEOW.
```

The **self** keyword

- In all the methods we used a keyword, called **self**.
- **Self** simply refers to the object on which the method applies.
- It can also be used to fecth some attributes values of our object!

```
1  lst = [0,1,2,3]
2  lst.append(4)
```

```
1  my_cat_hero = Hero()
```

```
1  my_cat_hero.meow()
```

```
meow.
```

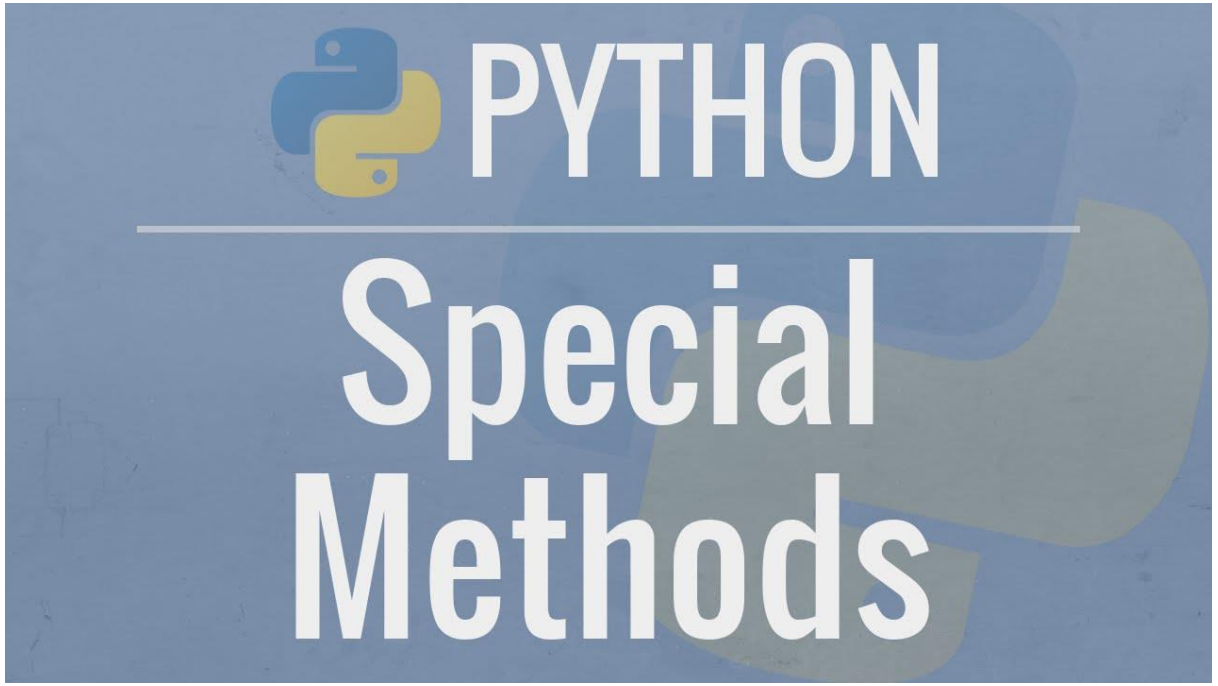
```
1  my_cat_hero.loud_meow()
```

```
Sir Meowsalot SAYS MEOW.
```

```
20      def loud_meow(self):
21          '''
22          A second method, calling some attributes of the class.
23          '''
24          print("{} SAYS MEOW.".format(self.name))
```

Class special methods

- A custom class also has **special methods**.
- These methods are written with **double underscores (__)** before and after their names.
- These methods do something special when some basic operations are applied to our object.
- (More on this later!)



The most important special method: the __init__ constructor method

- The most important special method is **__init__**, which is called every time an object is created.
- This is called the **constructor** method of the class.

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_lifepoints = 100

        # Print (to let the user know)
        print("A new hero has been created!")
```

```python
my_cat_hero = Hero()
```

```
A new hero has been created!
```

The __init__ constructor vs. « trash » initialization

```python
class Hero:

    # Hero's name
    name = "Sir Meowsalot"
    # Hero's class
    hero_class = "Warrior"
    # Hero's maximal lifepoints
    maximal_lifepoints = 100
    # Hero's current lifepoints
    current_lifepoints = 100
```

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_lifepoints = 100

        # Print (to let the user know)
        print("A new hero has been created!")
```

The __init__ constructor vs. « trash » initialization

- It is often preferable to define and use the **__init__** method!
- It is considered **good practice**
- Allows for more **actions** on **initialization.**

```python
class Hero:

    # Hero's name
    name = "Sir Meowsalot"
    # Hero's class
    hero_class = "Warrior"
    # Hero's maximal lifepoints
    maximal_lifepoints = 100
    # Hero's current lifepoints
    current_lifepoints = 100
```

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_lifepoints = 100

        # Print (to let the user know)
        print("A new hero has been created!")
```

The __init__ constructor vs. « trash » initialization

- It is often preferable to define and use the **__init__** method!
- It is considered **good practice**
- Allows for more **actions** on **initialization.**

```python
class Hero:

    # Hero's name
    name = "Sir Meowsalot"
    # Hero's class
    hero_class = "Warrior"
    # Hero's maximal lifepoints
    maximal_lifepoints = 100
    # Hero's current lifepoints
    current_lifepoints = 100
```

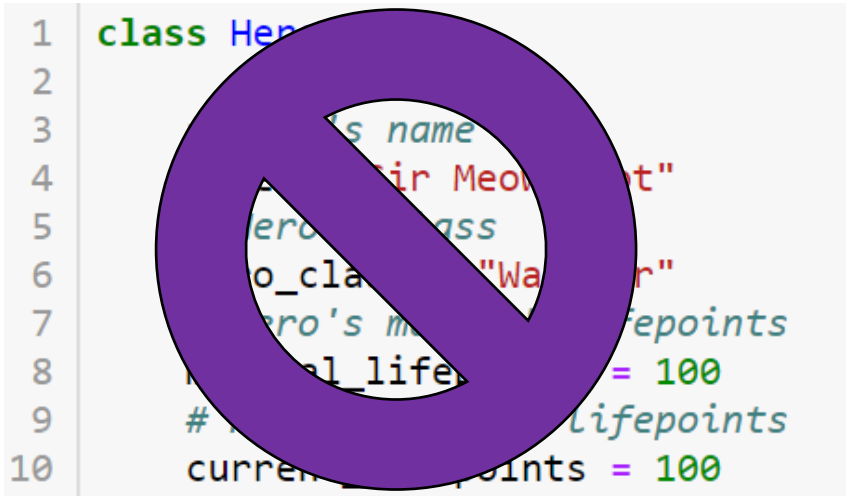
```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_lifepoints = 100

        # Print (to let the user know)
        print("A new hero has been created!")
```

```python
my_cat_hero = Hero()
```

```
A new hero has been created!
```

The __init__ constructor vs. « trash » initialization

- It is often preferable to define and use the **__init__** method!
- It is considered **good practice**
- Allows for more **actions** on **initialization.**



```
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_lifepoints = 100

        # Print (to let the user know)
        print("A new hero has been created!")
```

```
my_cat_hero = Hero()
```

```
A new hero has been created!
```

While we're at it, let us talk about default values in methods and functions

- We can also define **inputs** and **default values** in our **methods** and **special methods**.
- Here **__init__** expects a **name** and a **hero_class**, as **mandatory inputs**.

```python
class Hero:

    def __init__(self, name, hero_class, maximal_lifepoints = 100):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = name
        # Hero's class
        self.hero_class = hero_class

        # Hero's maximal lifepoints
        # (initialize as maximal_lifepoints)
        self.maximal_lifepoints = maximal_lifepoints

        # Hero's current lifepoints
        # (initialize as maximal_lifepoints)
        self.current_lifepoints = maximal_lifepoints
```

```python
my_first_hero = Hero(name = "Sir Meowsalot", hero_class = "Warrior")
print(my_first_hero.__dict__)
```

```
{'name': 'Sir Meowsalot', 'hero_class': 'Warrior', 'maximal_lifepoints': 100, 'current_lifepoints': 100}
```

```python
my_second_hero = Hero(name = "Lord Mustache", hero_class = "Mage", maximal_lifepoints = 50)
print(my_second_hero.__dict__)
```

```
{'name': 'Lord Mustache', 'hero_class': 'Mage', 'maximal_lifepoints': 50, 'current_lifepoints': 50}
```

While we're at it, let us talk about default values in methods and functions

- We can also pass an **optional input**, the **maximal_lifepoints** value.
- If no value is passed for **maximal_lifepoints**, we initialize it, **by default**, to **100**.

```python
class Hero:

    def __init__(self, name, hero_class, maximal_lifepoints = 100):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = name
        # Hero's class
        self.hero_class = hero_class

        # Hero's maximal lifepoints
        # (initialize as maximal_lifepoints)
        self.maximal_lifepoints = maximal_lifepoints

        # Hero's current lifepoints
        # (initialize as maximal_lifepoints)
        self.current_lifepoints = maximal_lifepoints
```

```python
my_first_hero = Hero(name = "Sir Meowsalot", hero_class = "Warrior")
print(my_first_hero.__dict__)
```

```
{'name': 'Sir Meowsalot', 'hero_class': 'Warrior', 'maximal_lifepoints': 100, 'current_lifepoints': 100}
```

```python
my_second_hero = Hero(name = "Lord Mustache", hero_class = "Mage", maximal_lifepoints = 50)
print(my_second_hero.__dict__)
```

```
{'name': 'Lord Mustache', 'hero_class': 'Mage', 'maximal_lifepoints': 50, 'current_lifepoints': 50}
```

The __dict__ special attribute

- Another useful concept is the **special dictionnary attribute**, called **__dict__**.
- By default, it produces a **dictionnary** containing
  - all the **attributes** of your object,
  - and their currently assigned **values**.
  - (Looks familiar?)

```
1  class Hero:
2
3      def __init__(self):
4          '''
5          Constructor function for the Hero class.
6          '''
7          # Hero's name
8          self.name = "Sir Meowsalot"
9          # Hero's class
10         self.hero_class = "Warrior"
11         # Hero's maximal lifepoints
12         self.maximal_lifepoints = 100
13         # Hero's current lifepoints
14         self.current_lifepoints = 100
```

```
1  print(my_cat_hero.__dict__)
```

```
{'name': 'Sir Meowsalot', 'hero_class': 'Warrior', 'maximal_lifepoints': 100, 'current_lifepoints': 100}
```

About special methods

- We have seen some special methods, such as **__init__**.
- But there are many more methods, such as **__add__**, which defines the bahavior when two objects of our custom class are summed with **+.**
- Typically, we will have **special methods** for each **operator** (+, -, *, /, etc.), and **built-in function** (len, print, etc.) in Python.

```python
class Coordinate:

    def __init__(self, x = 0, y =0):
        self.x = x
        self.y = y

    def __add__(self, other):
        new = Coordinate()
        new.x = self.x + other.x
        new.y = self.y + other.y
        return new
```

```python
A = Coordinate(3, 4)
B = Coordinate(1, 2)
C = A + B
print(type(C))
print(C.__dict__)
```

```
<class '__main__.Coordinate'>
{'x': 4, 'y': 6}
```

About special methods

- We have seen some special methods, such as **__init__**.
- But there are many more methods, such as **__add__**, which defines the bahavior when two objects of our custom class are summed with **+.**
- Typically, we will have **special methods** for each **operator** (+, -, *, /, etc.), and **built-in function** (len, print, etc.) in Python.

Method	Result
`__add__(self, other)`	self + other
`__sub__(self, other)`	self - other
`__mul__(self, other)`	self * other
`__div__(self, other)`	self / other
`__truediv__(self, other)`	self / other (future)
`__floordiv__(self, other)`	self // other
`__mod__(self, other)`	self % other
`__divmod__(self, other)`	`divmod(self, other)`
`__pow__(self, other [, modulo])`	self ** other, `pow(self, other, modulo)`
`__lshift__(self, other)`	self << other
`__rshift__(self, other)`	self >> other
`__(self, other)`	self ... other

The str special method

- The **__str__** special method defines what happens when you attempt to **convert** your custom object into a **string** type object.

- It is typically useful to decide what should be **displayed** on screen when you attempt to **print()** your object!

```
1  class Coordinate:
2
3      def __init__(self, x = 0, y =0):
4          self.x = x
5          self.y = y
6
7      def __str__(self):
8          return "This is a Coordinate object with values x = {} and y = {}".format(self.x, self.y)
```

```
1  A = Coordinate(3, 4)
2  B = str(A)
3  print(type(B))
4  print(B)
```

```
<class 'str'>
This is a Coordinate object with values x = 3 and y = 4
```

```
1  print(A)
```

```
This is a Coordinate object with values x = 3 and y = 4
```

Call method

- The **__call__** special method defines what happens when you attempt to **call** your custom object as a **function** and **pass it some arguments**.

```python
class Coordinate:

    def __init__(self, x = 0, y =0):
        self.x = x
        self.y = y

    def __call__(self, a, b, c):
        val = a*self.x + b*self.y + c
        return val
```

```python
x = 3
y = 4
A = Coordinate(x, y)
a = 1
b = 2
c = 3
print(A(a, b, c))
print(a*x + b*y + c)
```

```
14
14
```

Call method

- The **__call__** special method defines what happens when you attempt to **call** your custom object as a **function** and **pass it some arguments**.

- **REVELATION:** This basically means that functions defined with **def** are nothing but variables with a **__call__** method containing your instructions.

```
1  class Coordinate:
2
3      def __init__(self, x = 0, y =0):
4          self.x = x
5          self.y = y
6
7      def __call__(self, a, b, c):
8          val = a*self.x + b*self.y + c
9          return val
```

```
1  x = 3
2  y = 4
3  A = Coordinate(x, y)
4  a = 1
5  b = 2
6  c = 3
7  print(A(a, b, c))
8  print(a*x + b*y + c)
```

```
14
14
```

Our toy example for this class: a RPG hero

- Let us get back to our **Hero** object.

- **<u>Problem:</u>** our Hero's attack capabilities probably depend on the weapon he has **equipped**.
- And that is probably an **object** as well!

Introducing a Weapon class object!

- Reusing the previous concepts, we could define a **Weapon object**
- It will have its own attributes, such as
  - a name
  - and some attack values.

```
1  class Weapon:
2
3      def __init__(self, name, attack):
4          self.name = name
5          self.attack = attack
```

```
1  my_sword = Weapon(name = "Sword of Blazing Justice", attack = 10)
2  print(my_sword.__dict__)
```

```
{'name': 'Sword of Blazing Justice', 'attack': 10}
```

Equip a weapon!
(a.k.a the **« has-a » relationship)**

- We can then have our Hero equip a Weapon object, with our own **equip_weapon()** method.
- It **assigns** a custom **Weapon object** (which we created earlier)
  - to the **attribute equiped_weapon**
  - of our **Hero object.**
- Our Hero can then equip a Weapon object, with our **equip_weapon()** method.

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's equipped weapon
        self.equiped_weapon = None

    def equip_weapon(self, weapon_object):
        '''
        Equip weapon method
        '''
        # Assign weapon object to equiped_weapon attribute of Hero
        self.equiped_weapon = weapon_object
```

```python
my_cat_hero = Hero()
my_sword = Weapon(name = "Sword of Blazing Justice", attack = 10)
my_cat_hero.equip_weapon(my_sword)
print(my_cat_hero.__dict__)
```

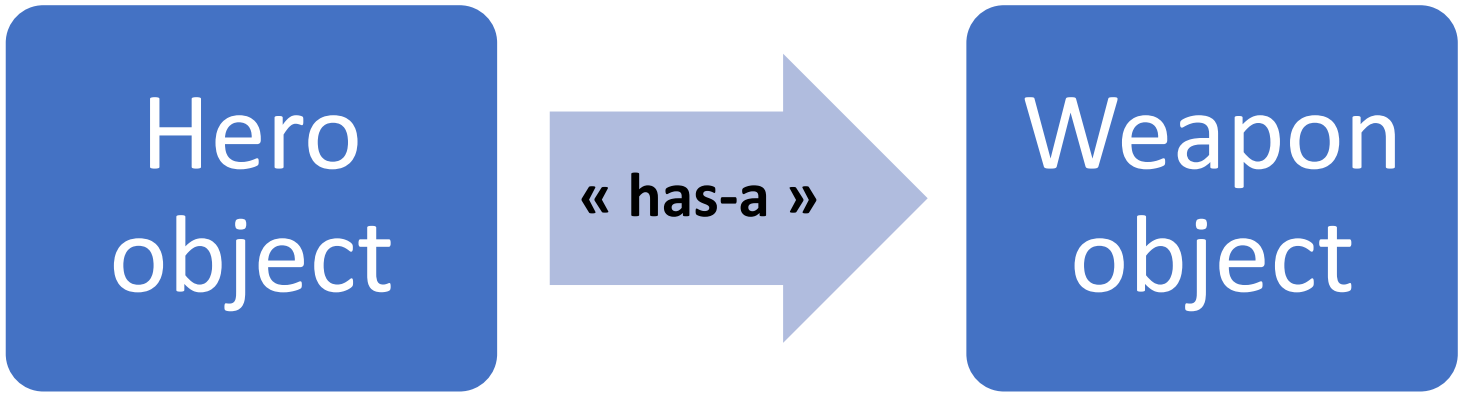
```
{'name': 'Sir Meowsalot', 'equiped_weapon': <__main__.Weapon object at 0x000002DF7D65D0B8>}
```

```python
print(my_cat_hero.equiped_weapon.__dict__)
```

```
{'name': 'Sword of Blazing Justice', 'attack': 10}
```

The « has-a » relationship

- We can then force our Hero to equip a Weapon object, with our own equip_weapon method.
- We then say that our **Hero** object **« has-a »** **Weapon** object.
- Because one of our **Hero object's attributes** is a **Weapon object.**



```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_lifepoints = 100
        # Hero's equipped weapon
        self.equiped_weapon = None

    def equip_weapon(self, weapon_object):
        '''
        Equip weapon method
        '''
        # Assign weapon object to equiped_weapon attribute of Hero
        self.equiped_weapon = weapon_object
```

```python
my_cat_hero = Hero()
my_sword = Weapon(name = "Sword of Blazing Justice", attack = 10)
my_cat_hero.equip_weapon(my_sword)
print(my_cat_hero.__dict__)
```

```
{'name': 'Sir Meowsalot', 'hero_class': 'Warrior', 'maximal_lifepoints': 100, 'current_lifepoints': 100, 'equiped_weapon': <__main__.Weapon object at 0x0000017F44B485F8>}
```

```python
print(my_cat_hero.equiped_weapon.__dict__)
```

```
{'name': 'Sword of Blazing Justice', 'attack': 10}
```

Conclusion

- From object dictionaries to proper objects
- Attributes, methods and constructor method
- The __dict__ special attribute
- About special methods
- Has-a relationship

If time allows,

- (Is-a relationship and inheritance)
- (Attributes privacy)
- (Setters, getters and properties)

Modifying our Weapon class object

- Earlier, we defined a Weapon object, with attributes such as name and attack values.
- **Problem:** how do we efficiently take into account multiple weapons possibilities?

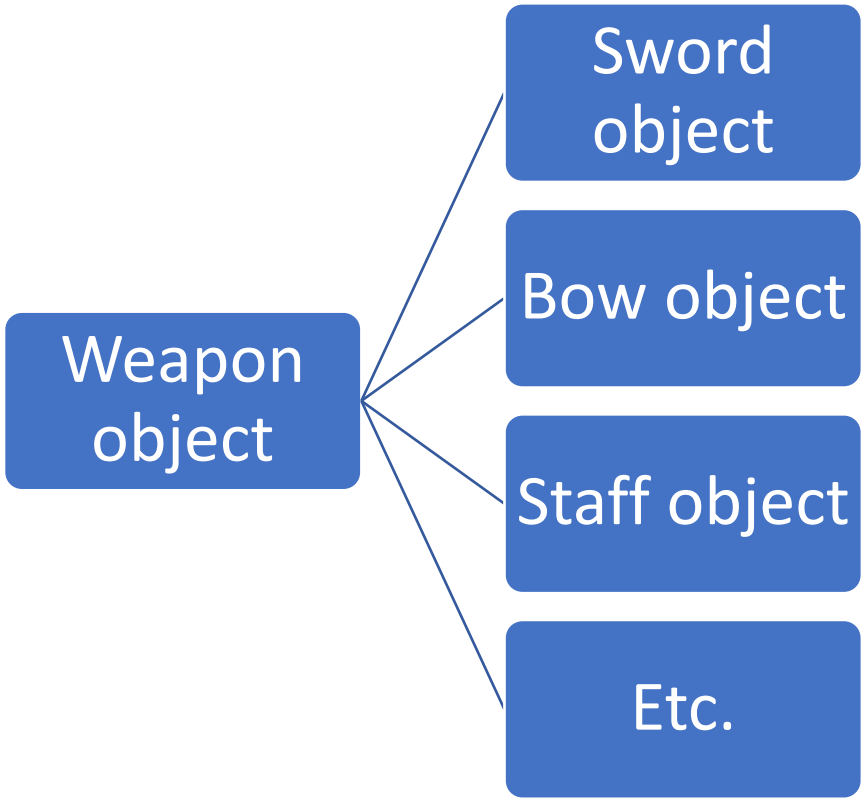
```
1  class Weapon:
2
3      def __init__(self, name, attack):
4          self.name = name
5          self.attack = attack
```

```
1  my_sword = Weapon(name = "Sword of Blazing Justice", attack = 10)
2  print(my_sword.__dict__)
```

```
{'name': 'Sword of Blazing Justice', 'attack': 10}
```

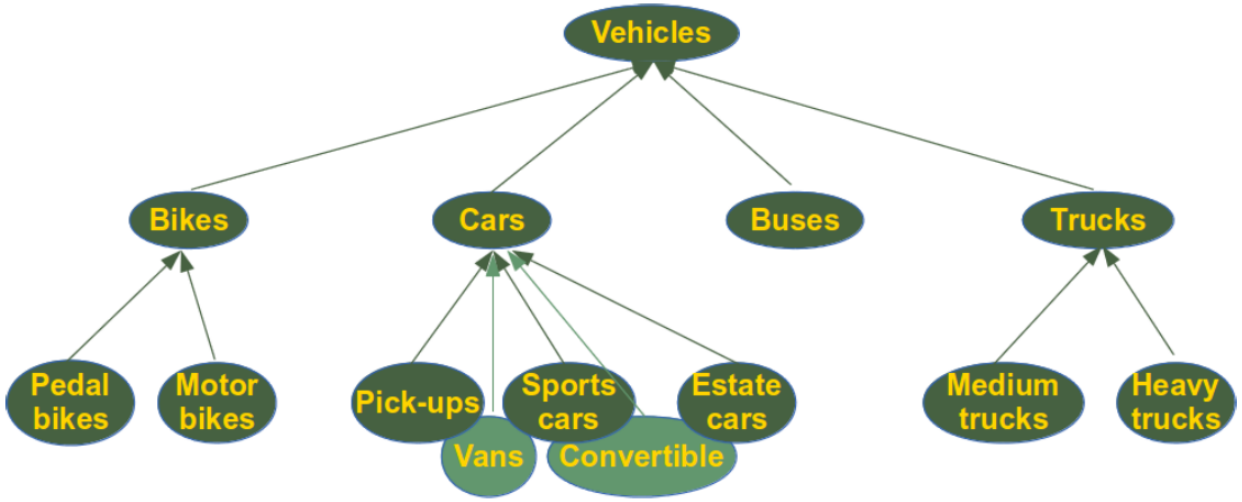
Objects and sub-classes of objects

- Typically, our Hero could equip a Weapon object...
- But it could be a **sword**, a **bow**, a **staff**, etc.
- These weapons will probably have attributes and methods in common…
- But they might also have different attributes and methods, as our Hero will probably operate those weapons differently.



Objects and sub-classes of objects

- In fact, it is common in life to have **objects** and **sub-classes** of objects.
- Cars, buses and trucks probably have some **attributes and methods** in common, because they are **vehicles objects**.
- But they probably have **attributes** and **methods** that are **specific** to them.



Introducting inheritance! (a.k.a. the « is-a » relationship)

- **Inheritance:** it is possible to **create** a class, which **reuses methods** and **attributes** from another class!
- We simply mention the name of the previous class in the class **definition**!

```python
class Sword(Weapon):

    def __init__(self, name, attack):
        # Reuse the Weapon object __init__ method!
        Weapon.__init__(self, name, attack)
        # Add extra attributes, specific to this weapon
        self.weapon_type = 'Sword'
        self.weapon_range = '3'

    def slash(self):
        print("A big slash to the face!")
```

```python
my_sword = Sword(name = "Sword of Blazing Justice", attack = 10)
print(my_sword.__dict__)
```

```
{'name': 'Sword of Blazing Justice', 'attack': 10, 'weapon_type': 'Sword', 'weapon_range': '3'}
```

```python
class Staff(Weapon):

    def __init__(self, name, attack):
        # Reuse the Weapon object __init__ method!
        Weapon.__init__(self, name, attack)
        # Add extra attributes, specific to this weapon
        self.weapon_type = 'Staff'
        self.weapon_range = '25'

    def cast_fireball(self):
        print("PEW PEW PEW!")
```

```python
my_staff = Staff(name = "Staff of Impeccable Fireworks", attack = 5)
```

Introducting inheritance! (a.k.a. the « is-a » relationship)

- **Inheritance:** it is possible to **create** a class, which **reuses methods** and **attributes** from another class!
- We simply mention the name of the previous class in the class **definition**!
- And reuse the **__init__ constructor** from the previous class in the **__init__ constructor** of new class!

```python
class Sword(Weapon):

    def __init__(self, name, attack):
        # Reuse the Weapon object __init__ method!
        Weapon.__init__(self, name, attack)
        # Add extra attributes, specific to this weapon
        self.weapon_type = 'Sword'
        self.weapon_range = '3'

    def slash(self):
        print("A big slash to the face!")
```

```python
my_sword = Sword(name = "Sword of Blazing Justice", attack = 10)
print(my_sword.__dict__)
```

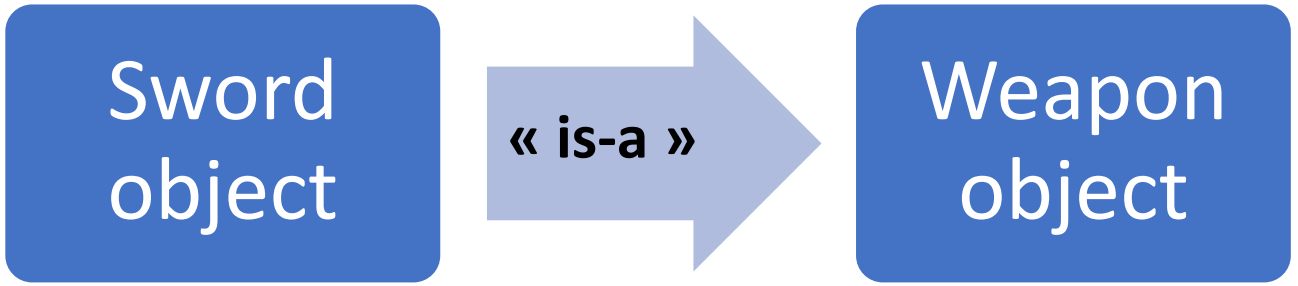
```
{'name': 'Sword of Blazing Justice', 'attack': 10, 'weapon_type': 'Sword', 'weapon_range': '3'}
```

```python
class Staff(Weapon):

    def __init__(self, name, attack):
        # Reuse the Weapon object __init__ method!
        Weapon.__init__(self, name, attack)
        # Add extra attributes, specific to this weapon
        self.weapon_type = 'Staff'
        self.weapon_range = '25'

    def cast_fireball(self):
        print("PEW PEW PEW!")
```

```python
my_staff = Staff(name = "Staff of Impeccable Fireworks", attack = 5)
```

Inheritance and **« is-a »** relationship

- **Inheritance** is very useful
  - it allows for **code reuse**
  - and **better architecture** of the objects in your code!
- **Inheritance** defines a **« is-a »** relationship between objects



Let us then equip a sword and/or a staff on our beloved cat Hero!

Let us then equip a sword on our beloved cat Hero!

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_lifepoints = 100
        # Hero's equipped weapon
        self.equiped_weapon = None
        # Hero's attack points
        self.attack = 0

    def equip_weapon(self, weapon_object):
        '''
        Equip weapon method and define hero's attack points
        '''
        # Assign weapon object to equiped_weapon attribute of Hero
        self.equiped_weapon = weapon_object
        # Set attack points of Hero, by retrieving the value of the weapon_object's attack attribute
        self.attack = weapon_object.attack
```

```python
my_cat_hero = Hero()
print(my_cat_hero.__dict__)
```

```
{'name': 'Sir Meowsalot', 'hero_class': 'Warrior', 'maximal_lifepoints': 100, 'current_lifepoints': 100, 'equiped_weapon': None, 'attack': 0}
```

```python
my_sword = Sword(name = "Sword of Blazing Justice", attack = 10)
my_cat_hero.equip_weapon(my_sword)
print(my_cat_hero.__dict__)
```

```
{'name': 'Sir Meowsalot', 'hero_class': 'Warrior', 'maximal_lifepoints': 100, 'current_lifepoints': 100, 'equiped_weapon': <__main__.Sword object at 0x0000015FC3764C50>, 'attack': 10}
```

Good practice in Object-Oriented Programming

- As mentionned earlier, using **__init__ constructors** for your class is considered **good** practice.

- In addition, it could also be interesting to use **setters** and **getters methods** for your class **attributes**.
- And also, to define if your class **attributes** should be **public** or **private**.

Display lifepoints method

- Let us say we want to design a method that prints the current lifepoints of our Hero on screen.

- A possible way to do it is this →
  - And it works just fine!
  - However, this is considered **bad practice**.

```
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_lifepoints = 100


    def display_current_life(self):
        '''
        A method calling some attributes of the class.
        '''
        # Print to let the player know about its current life total
        print("Your hero has {} lifepoints.".format(self.current_lifepoints))
```

```
my_cat_hero = Hero()
```

```
my_cat_hero.display_current_life()
```

```
Your hero has 100 lifepoints.
```

Setters and getters

- **Good practice:** design methods for getting and setting attributes of a class
- **Getter** method: fetches the current value stored in attribute.

- Here, we demonstrate an example of a **getter** method.
  - And apply it in our display_current_life method.

```
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_lifepoints = 100

    def get_current_lifepoints(self):
        '''
        A getter method returning the value of the current lifepoints.
        '''
        print("Getter called for current_lifepoints")
        return self.current_lifepoints

    def display_current_life(self):
        '''
        A method calling some attributes of the class.
        '''
        # Print to let the player know about its current life total
        print("Your hero has {} lifepoints.".format(self.get_current_lifepoints()))
```

```
my_cat_hero = Hero()
```

```
my_cat_hero.display_current_life()
```

```
Getter called for current_lifepoints
Your hero has 100 lifepoints.
```

Why is it good practice to use setters and getters?

- But, why would I write a getter method?!
  - It seems cumbersome for no reason!

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_lifepoints = 100

    def get_current_lifepoints(self):
        '''
        A getter method returning the value of the current lifepoints.
        '''
        print("Getter called for current_lifepoints")
        return self.current_lifepoints

    def display_current_life(self):
        '''
        A method calling some attributes of the class.
        '''
        # Print to let the player know about its current life total
        print("Your hero has {} lifepoints.".format(self.get_current_lifepoints()))
```

```python
my_cat_hero = Hero()
```

```python
my_cat_hero.display_current_life()
```

```
Getter called for current_lifepoints
Your hero has 100 lifepoints.
```

Why is it good practice to use setters and getters?

- But, why would I write a getter method?!
  - It seems cumbersome for no reason!
- Actually, it makes your code **more modular and stable.**

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_lifepoints = 100

    def get_current_lifepoints(self):
        '''
        A getter method returning the value of the current lifepoints.
        '''
        print("Getter called for current_lifepoints")
        return self.current_lifepoints

    def display_current_life(self):
        '''
        A method calling some attributes of the class.
        '''
        # Print to let the player know about its current life total
        print("Your hero has {} lifepoints.".format(self.get_current_lifepoints()))
```

```python
my_cat_hero = Hero()
```

```python
my_cat_hero.display_current_life()
```

```
Getter called for current_lifepoints
Your hero has 100 lifepoints.
```

Why is it good practice to use setters and getters?

- Let us pretend that – for some reason – the dev team in charge of the lifepoints, decided to change the way it is stored in memory.
- No longer stored as lifepoints numbers, but as a lifepoint percentage.
- These methods no longer work!

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_life_percentage = 100
```

```python
def get_current_lifepoints(self):
    '''
    A getter method returning the value of the current lifepoints.
    '''
    print("Getter called for current_lifepoints")
    return self.current_lifepoints

def display_current_life(self):
    '''
    A third method calling some attributes of the class.
    '''
    # Print to let the player know about its current life total
    print("Your hero has {} lifepoints.".format(self.get_current_lifepoints()))
```

Why is it good practice to use setters and getters?

- Let us pretend that – for some reason – the dev team in charge of the lifepoints, decided to change the way it is stored in memory.
- No longer stored as lifepoints numbers, but as a lifepoint percentage.
- But, only **one line** to change to make it work again!

```
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_life_percentage = 100

    def get_current_lifepoints(self):
        '''
        A getter method returning the value of the current_lifepoints,
        based on maximal_lifepoints and current_life_percentage.
        '''
        print("Getter called for current_lifepoints")
        return round(self.current_life_percentage*self.maximal_lifepoints/100)

    def display_current_life(self):
        '''
        A third method calling some attributes of the class.
        '''
        # Print to let the player know about its current life total
        print("Your hero has {} lifepoints.".format(self.get_current_lifepoints()))
```

```
my_cat_hero = Hero()
```

```
my_cat_hero.display_current_life()
```

```
Getter called for current_lifepoints
Your hero has 100 lifepoints.
```

Why is it good practice to use setters and getters?

- Using setters and getters for attributes will make your code more modular and robust to change.
- It is therefore considered **good practice**.

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_life_percentage = 100

    def get_current_lifepoints(self):
        '''
        A getter method returning the value of the current_lifepoints,
        based on maximal_lifepoints and current_life_percentage.
        '''
        print("Getter called for current_lifepoints")
        return round(self.current_life_percentage*self.maximal_lifepoints/100)

    def display_current_life(self):
        '''
        A third method calling some attributes of the class.
        '''
        # Print to let the player know about its current life total
        print("Your hero has {} lifepoints.".format(self.get_current_lifepoints()))
```

```python
my_cat_hero = Hero()
```

```python
my_cat_hero.display_current_life()
```

```
Getter called for current_lifepoints
Your hero has 100 lifepoints.
```

Defining public, protected and private attributes

- Another good practice consists of defining **public, protected** and **private attributes**.
- **Public**: everyone can modify the attribute.
- **Protected:** public, but should refrain from modifying it from outside the class.
- **Private**: only the functions called within my object can modify the attribute.

Type of attribute	Public	Protected	Private
Can be modified within the class methods.	Yes	Yes	Yes
Can be modified by function and methods outside of the class.	Yes	Yes	No
It is acceptable for this attribute to be modified by functions and methods outside the class.	Yes	No	Does not apply

Why is it good practice to use public, protected and private attributes?

- Another good practice consists of defining **public, protected** and **private attributes**.
- **Public**: everyone can modify the attribute.
- **Protected:** public, but should refrain from modifying it from outside the class.
- **Private**: only the functions called within my object can modify the attribute.

- For instance,
- Your lifepoints total can be changed by other players
  - (this typically happens when they attack you and shall be allowed in the game!)
- However, other players should not be allowed to change your Hero's name!

Private attributes

- **Private**: only the functions called within my object can modify the attribute.

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.__name = "Sir Meowsalot"
        # Hero's class
        self.__hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_life_percentage = 100

    def get_hero_name(self):
        return self.__name
```

```python
my_cat_hero = Hero()
```

```python
# This works, because it's a public attribute
my_cat_hero.current_life_percentage = 50
print(my_cat_hero.current_life_percentage)
```

```
50
```

```python
# This works
print(my_cat_hero.get_hero_name())
# This does not work
print(my_cat_hero.__name)
```

```
Sir Meowsalot
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-129-7f4b66a9de98> in <module>
      2 print(my_cat_hero.get_hero_name())
      3 # This does not work
----> 4 print(my_cat_hero.__name)

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

```python
# Trying to change the name from outside the class leaves it unaffected
my_cat_hero.__name = "Poop-eater"
print(my_cat_hero.get_hero_name())
```

```
Sir Meowsalot
```

Private attributes

- **Private**: only the functions called within my object can modify the attribute.

- We can **make** an attribute **private** by adding a double **underscore (__)** in front of its name.
- Calling it or modifying it has no effect.

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self.__name = "Sir Meowsalot"
        # Hero's class
        self.__hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_life_percentage = 100

    def get_hero_name(self):
        return self.__name
```

```python
my_cat_hero = Hero()
```

```python
# This works, because it's a public attribute
my_cat_hero.current_life_percentage = 50
print(my_cat_hero.current_life_percentage)
```

```
50
```

```python
# This works
print(my_cat_hero.get_hero_name())
# This does not work
print(my_cat_hero.__name)
```

```
Sir Meowsalot

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-129-7f4b66a9de98> in <module>
      2 print(my_cat_hero.get_hero_name())
      3 # This does not work
----> 4 print(my_cat_hero.__name)

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

```python
# Trying to change the name from outside the class leaves it unaffected
my_cat_hero.__name = "Poop-eater"
print(my_cat_hero.get_hero_name())
```

```
Sir Meowsalot
```

Protected attributes

- **Protected:** public, but should refrain from modifying it from outside the class.
- We can **make** an attribute **protected** by adding a single **underscore (_)** in front of its name.

```python
class Hero:

    def __init__(self):
        '''
        Constructor function for the Hero class.
        '''
        # Hero's name
        self._name = "Sir Meowsalot"
        # Hero's class
        self.hero_class = "Warrior"
        # Hero's maximal lifepoints
        self.maximal_lifepoints = 100
        # Hero's current lifepoints
        self.current_life_percentage = 100

    def get_hero_name(self):
        return self._name

    def set_hero_name(self, value):
        print("Warning: hey, the name attribute is protected, don't do that!")
        self._name = value
```

```python
my_cat_hero = Hero()
```

```python
# This works
print(my_cat_hero.get_hero_name())
```

```
Sir Meowsalot
```

```python
# Trying to change the name from outside the class leaves it unaffected
my_cat_hero.set_hero_name("Poop-eater")
print(my_cat_hero.get_hero_name())
```

```
Warning: hey, the name attribute is protected, don't do that!
Poop-eater
```

Merging everything, by setting attributes properties using the **property** keyword

- Last but not least, it is also considered good practice to create properties.
  - i.e default setter/getter methods for each attribute.
- It is done with the **property** keyword.

```python
class Hero:

    def __init__(self):
        # Constructor function for the Hero class.
        # Hero's name
        self.__name = "Sir Meowsalot"
        # Hero's class
        self.__hero_class = "Warrior"
        # Hero's maximal lifepoints
        self._maximal_lifepoints = 100
        # Hero's current lifepoints
        self._current_lifepoints = 100

    def set_current_lifepoints(self, value):
        # A setter method setting the current lifepoints to value.
        print("Setter called for attribute _current_lifepoints")
        self._current_lifepoints = value

    def get_current_lifepoints(self):
        # A getter method returning the value of the current lifepoints.
        print("Getter called for attribute _current_lifepoints")
        return self._current_lifepoints

    '''
    Set property for the current_lifepoints attribute
    '''
    current_lifepoints = property(get_current_lifepoints, set_current_lifepoints)
```

Merging everything, by setting attributes properties using the **property** keyword

- Thanks to the **property** keyword…
- Whenever we execute ***my_cat_hero.current_lifepoints***, we automatically call the **getter** method, i.e.

my_cat_hero.get_current_lifepoints()

```
class Hero:

    def __init__(self):
        # Constructor function for the Hero class.
        # Hero's name
        self.__name = "Sir Meowsalot"
        # Hero's class
        self.__hero_class = "Warrior"
        # Hero's maximal lifepoints
        self._maximal_lifepoints = 100
        # Hero's current lifepoints
        self._current_lifepoints = 100

    def set_current_lifepoints(self, value):
        # A setter method setting the current lifepoints to value.
        print("Setter called for attribute _current_lifepoints")
        self._current_lifepoints = value

    def get_current_lifepoints(self):
        # A getter method returning the value of the current lifepoints.
        print("Getter called for attribute _current_lifepoints")
        return self._current_lifepoints

    '''
    Set property for the current_lifepoints attribute
    '''
    current_lifepoints = property(get_current_lifepoints, set_current_lifepoints)
```

```
my_cat_hero = Hero()
```

```
print(my_cat_hero.current_lifepoints)
```

```
Getter called for attribute _current_lifepoints
100
```

Merging everything, by setting attributes properties using the **property** keyword

- And similarly, with the **setter** method, when we try to assign a value to our attribute.

```python
class Hero:

    def __init__(self):
        # Constructor function for the Hero class.
        # Hero's name
        self.__name = "Sir Meowsalot"
        # Hero's class
        self.__hero_class = "Warrior"
        # Hero's maximal lifepoints
        self._maximal_lifepoints = 100
        # Hero's current lifepoints
        self._current_lifepoints = 100

    def set_current_lifepoints(self, value):
        # A setter method setting the current lifepoints to value.
        print("Setter called for attribute _current_lifepoints")
        self._current_lifepoints = value

    def get_current_lifepoints(self):
        # A getter method returning the value of the current lifepoints.
        print("Getter called for attribute _current_lifepoints")
        return self._current_lifepoints

    '''
    Set property for the current_lifepoints attribute
    '''
    current_lifepoints = property(get_current_lifepoints, set_current_lifepoints)
```

```python
my_cat_hero = Hero()
```

```python
print(my_cat_hero.current_lifepoints)
```

```
Getter called for attribute _current_lifepoints
100
```

```python
my_cat_hero.current_lifepoints = 50
print(my_cat_hero.current_lifepoints)
```

```
Setter called for attribute _current_lifepoints
Getter called for attribute _current_lifepoints
50
```