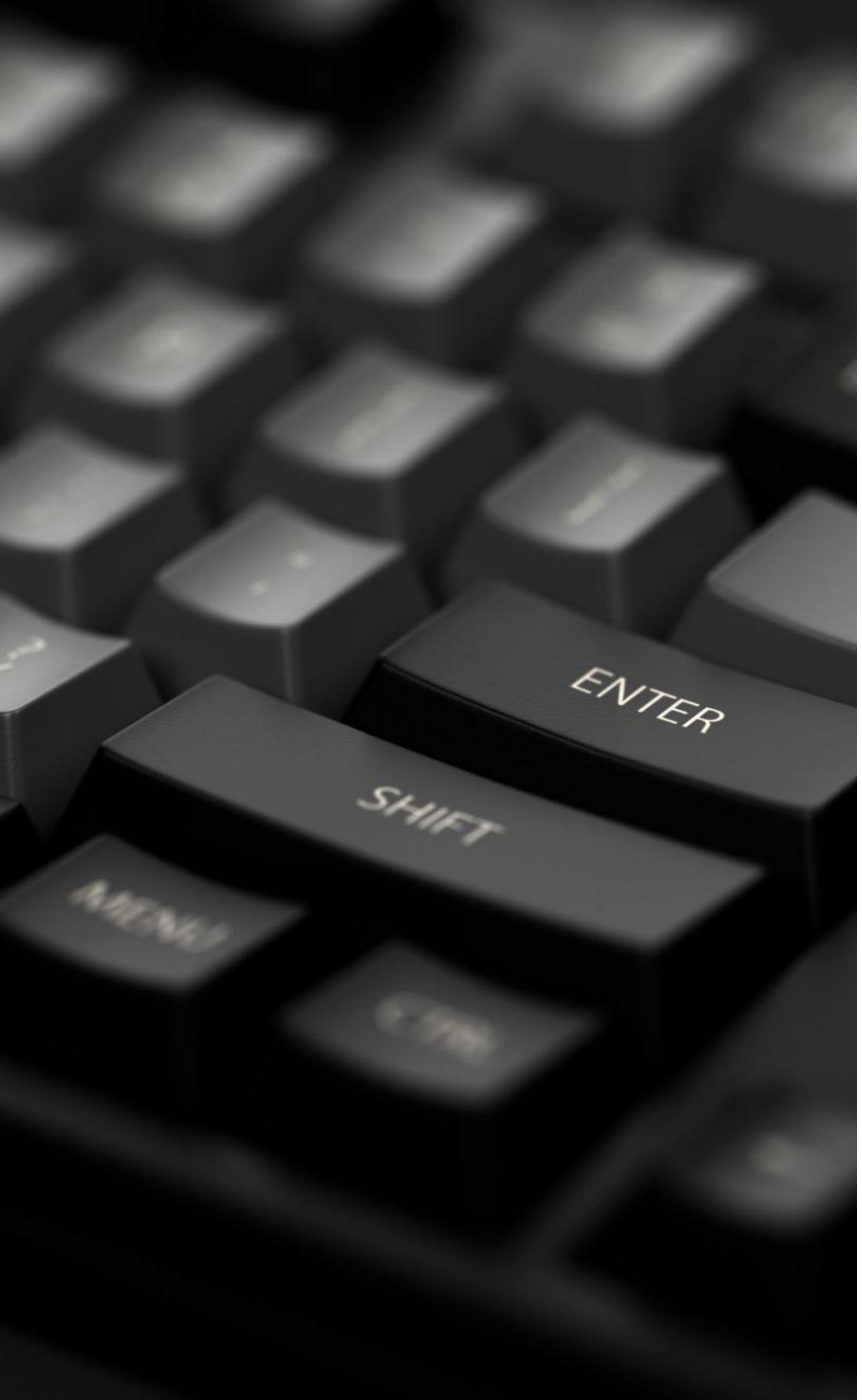
# A gamified introduction to Python Programming

# Lecture 15
# Introduction to Objects and OOP

Matthieu DE MARI – Singapore University of Technology and Design

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN



# Outline (Chapter 15)

- What are **objects** and **classes**? How to they relate to **dictionaries**?
- What are **attributes** and **methods** in a class?
- What is the **__init__** constructor method?
- What is the **__dict__** special attribute?
- What are **special methods** in Python?
- What is the **“has-a” relationship** in OOP?
- *(What is the **“is-a” relationship** in OOP?)*
- *(What are the different **attributes privacies**?)*
- *(What are **setters**, **getters** and **properties**?)*

# What is object-oriented programming?

**Definition (Object-Oriented Programming):**

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

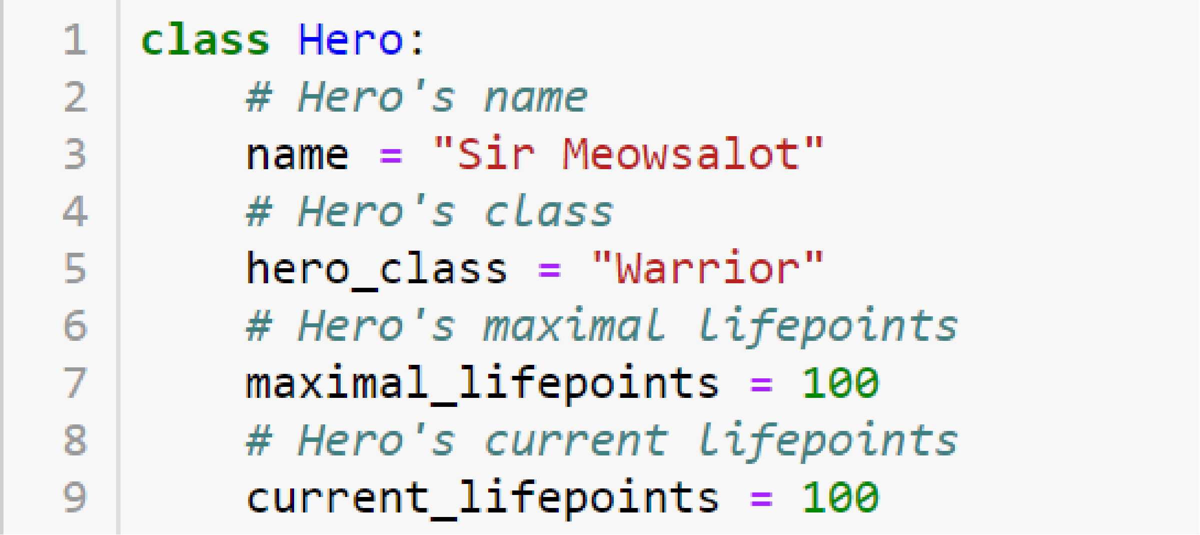
- Sometimes, the **concepts** we need for our programs cannot be described using only the basic int/float/str/list/dict types.
- We could assemble variables that relate to the same concept into dictionnaries (what we described as object-oriented thinking), we often prefer to **create our own custom types/objects**.
- These types are created using the **class keyword**, and might come with their own list of **attributes** and **methods**.

# Our toy example: an RPG main protagonist

Let us say we would like to code a video game and design our main character/protagonist.

- Our **main character** can be represented as a **custom** class **object.**
- It would have several **attributes**, e.g.
    - A name (string type, ‘Sir Meowsalot’)
    - A class (string type, ‘Warrior’)
    - Some lifepoints (int types, 100)
    - And many other attributes (intelligence, strength, speed, armor, etc.)

# Class attributes



```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
```

Let us say we would like to code a video game and design our main character/protagonist.

- Our **main character** can be represented as a **custom** class **object.**
- It would have several **attributes**, e.g.
  - A name (string type, ‘Sir Meowsalot’)
  - A class (string type, ‘Warrior’)
  - Some lifepoints (int types, 100)
  - And many other attributes (intelligence, strength, speed, armor, etc.)

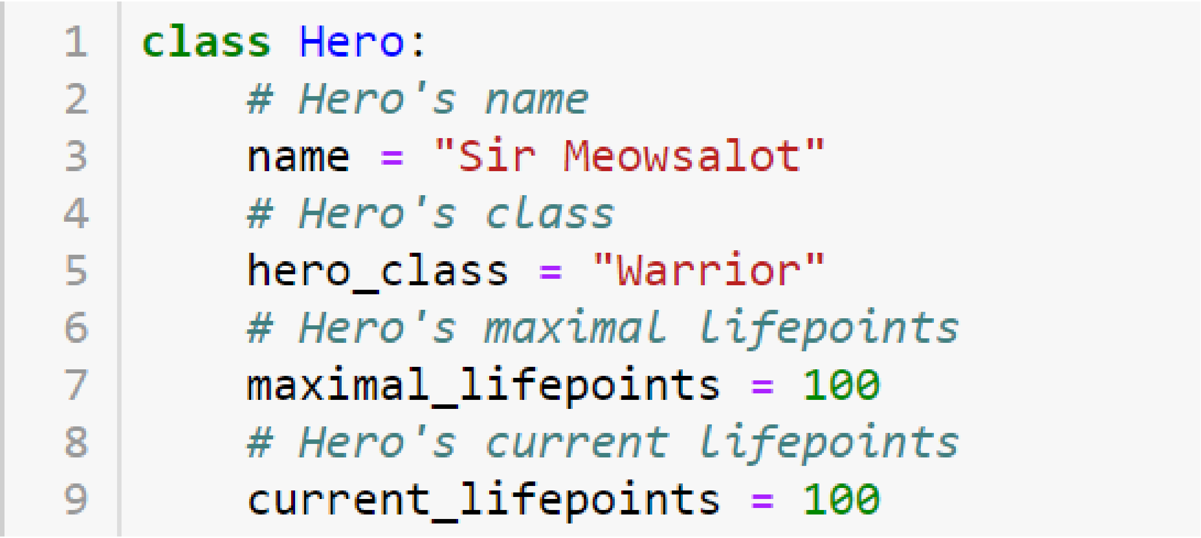
# Class attributes

To give you an analogy:

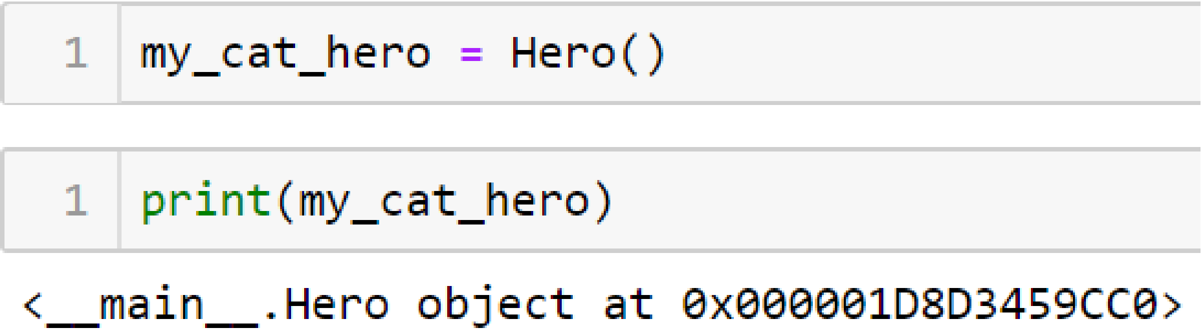
- The **def** keyword was used to introduce new functions.
- The **class** keyword is used to introduce a **new type of variable** to Python.

Here we defined the blueprint for a new class *"Hero"*.

Later, we created a variable *my_cat_hero*, which is a variable of type *"Hero"*.



```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
my_cat_hero = Hero()
```

```python
print(my_cat_hero)
```

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

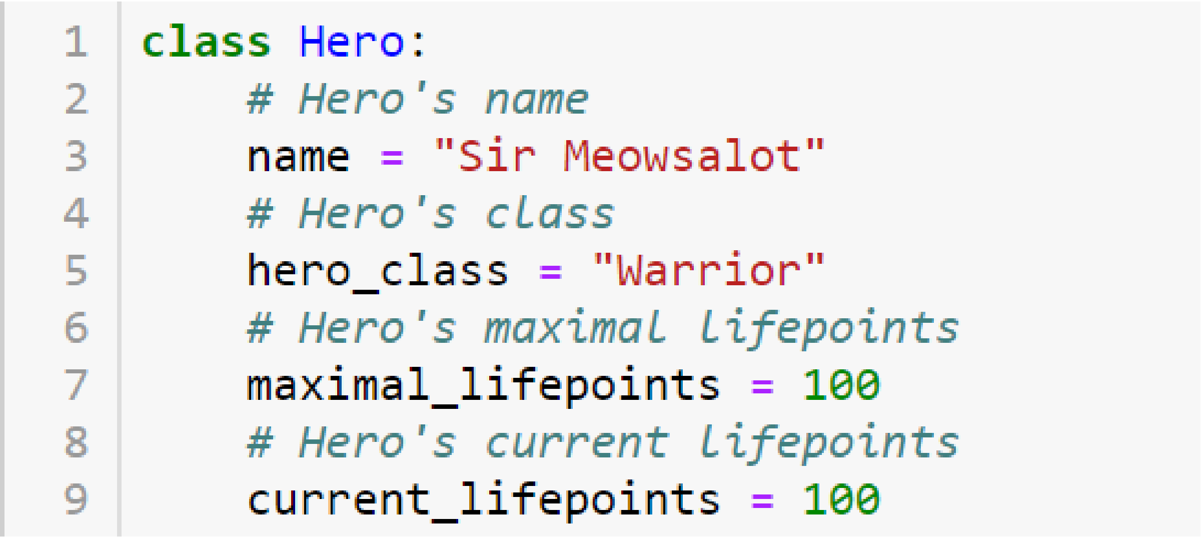
# Class attributes

**Definition (object attributes):**

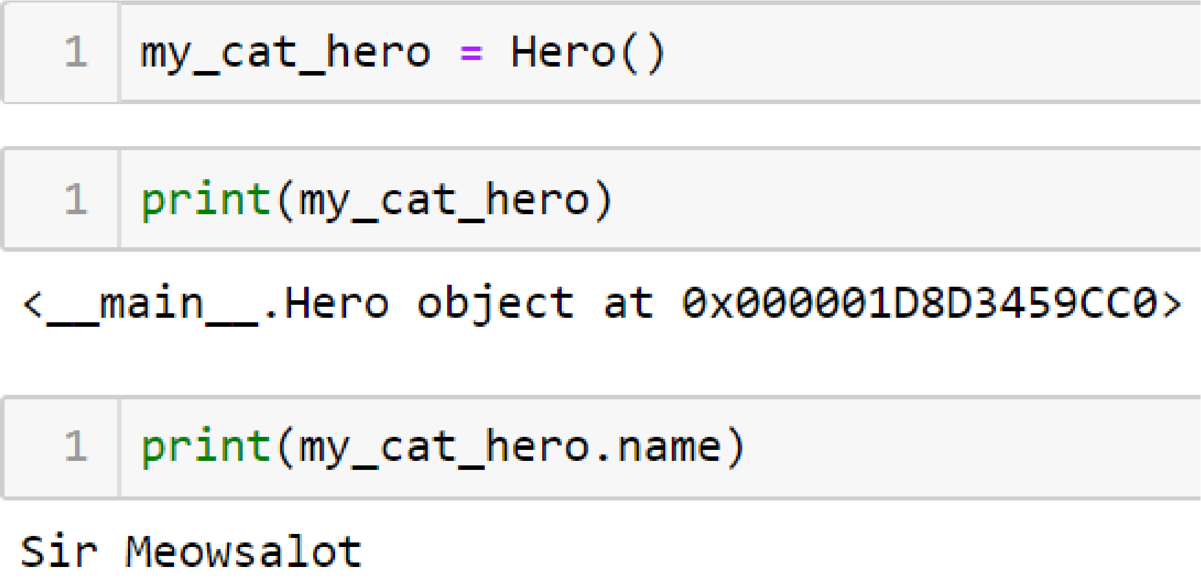
In Python, **object attributes** are variables that are associated with an object and belong to a class.

These attributes store values and data that are relevant to the object.

They are defined within the class block, as variables and can later be accessed using **the . notation** on **our object variable** (like when we did a.shape in Numpy).



```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
my_cat_hero = Hero()
```

```python
print(my_cat_hero)
```

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

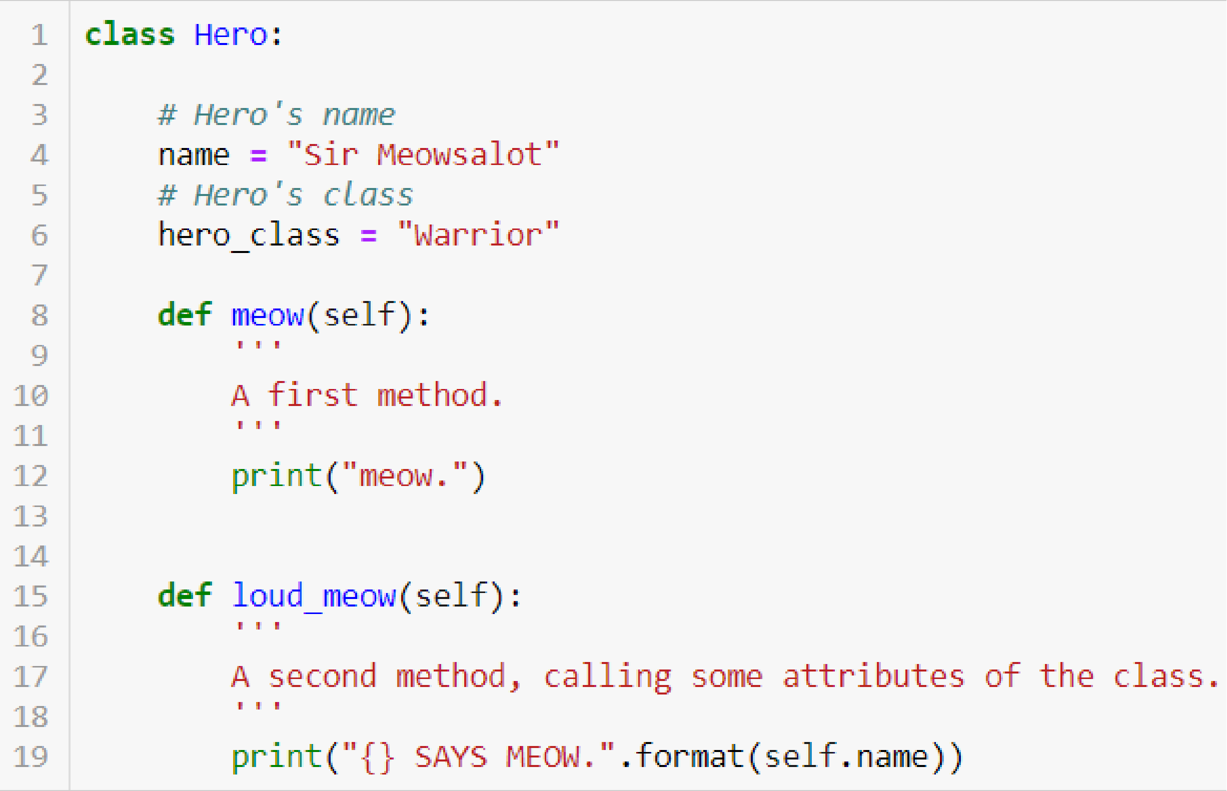
```python
print(my_cat_hero.name)
```

```
Sir Meowsalot
```

# Class methods

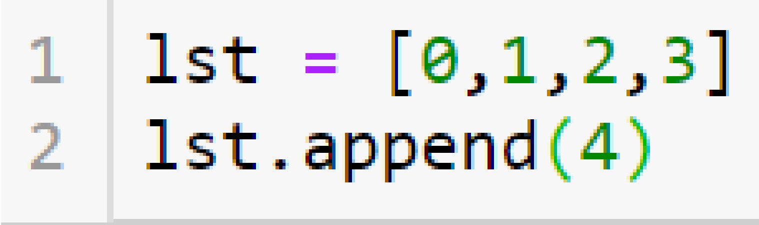
On top of a class **attributes**, we can also define **class methods.**

- **Methods:** functions which apply on our custom object.
- **Methods** may use/modify some of the objects **attributes**.



```python
class Hero:

    # Hero's name
    name = "Sir Meowsalot"
    # Hero's class
    hero_class = "Warrior"

    def meow(self):
        '''
        A first method.
        '''
        print("meow.")


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

# The **self** keyword



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

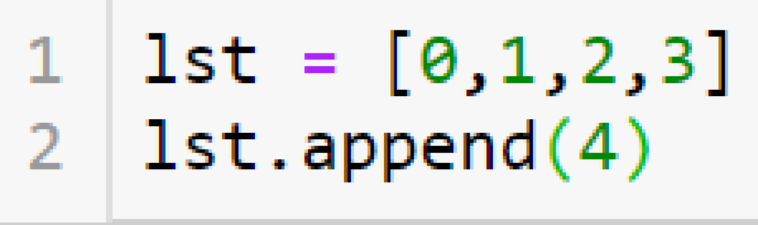
In all the methods we used a keyword, called **self**.

- **Self** simply refers to the object on which the method applies.
- In the case of *lst.append(4)*, **self** designated the list *lst* on which *append()* was applied.

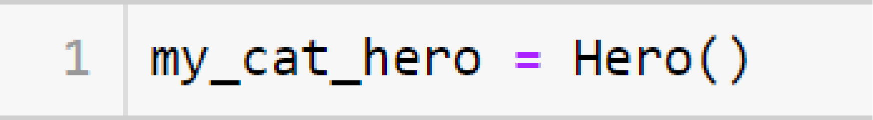
# The **self** keyword

In all the methods we used a keyword, called **self**.

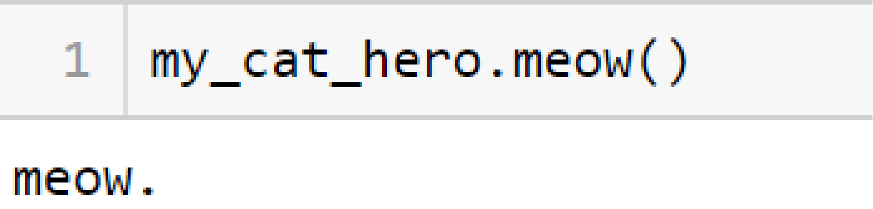
- **Self** simply refers to the object on which the method applies.
- In the case of *my_cat_hero.meow()*, **self** will refer to our newly typed variable *my_cat_hero.*



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

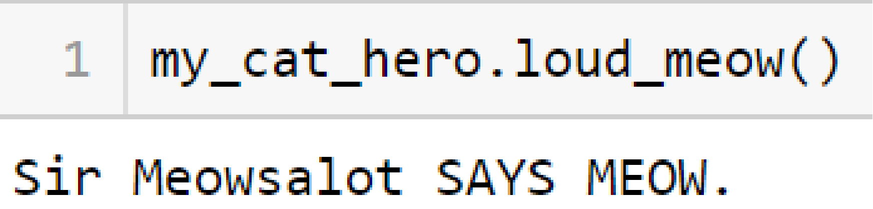


```
my_cat_hero = Hero()
```



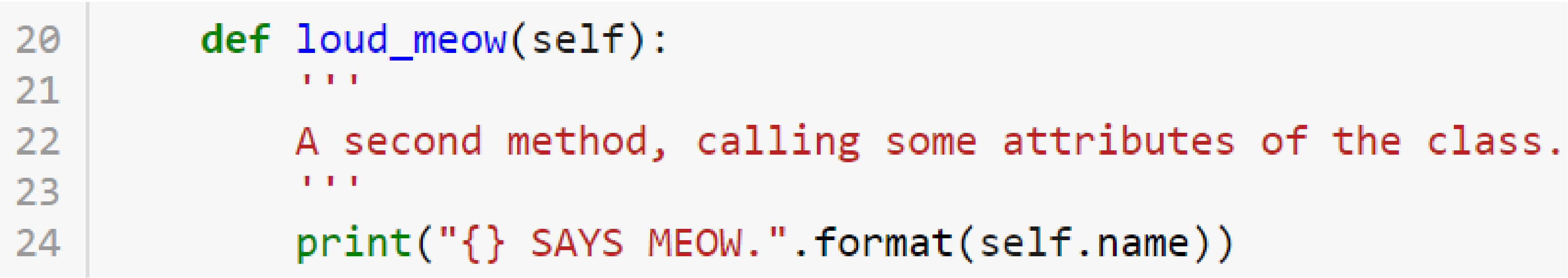
```
my_cat_hero.meow()
```

```
meow.
```



```
my_cat_hero.loud_meow()
```

```
Sir Meowsalot SAYS MEOW.
```



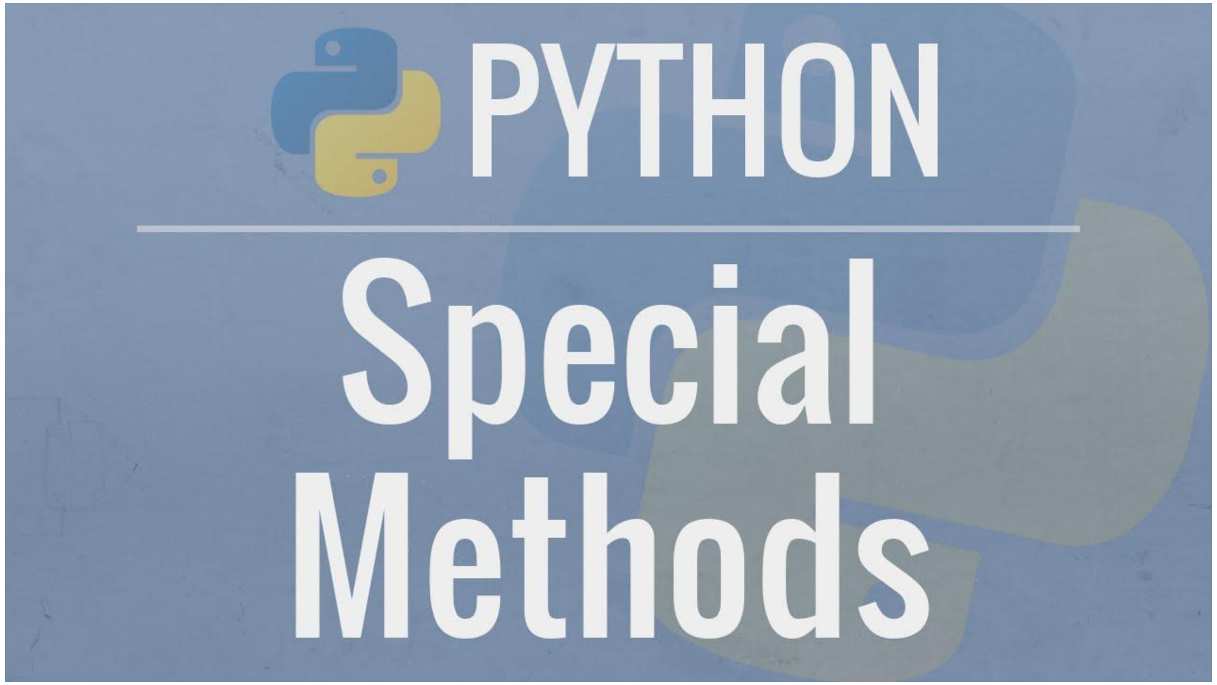
```
    def loud_meow(self):
        '''
        A second method, calling some attributes of the class.
        '''
        print("{} SAYS MEOW.".format(self.name))
```

# Class special methods

**Definition (special methods):**

A custom class may also have **special methods**.

- These methods have fixed names and are written with **double underscores (__)** before and after their names.
- These methods do something special when some basic operations (+, *, len(), etc.) are applied to our object.

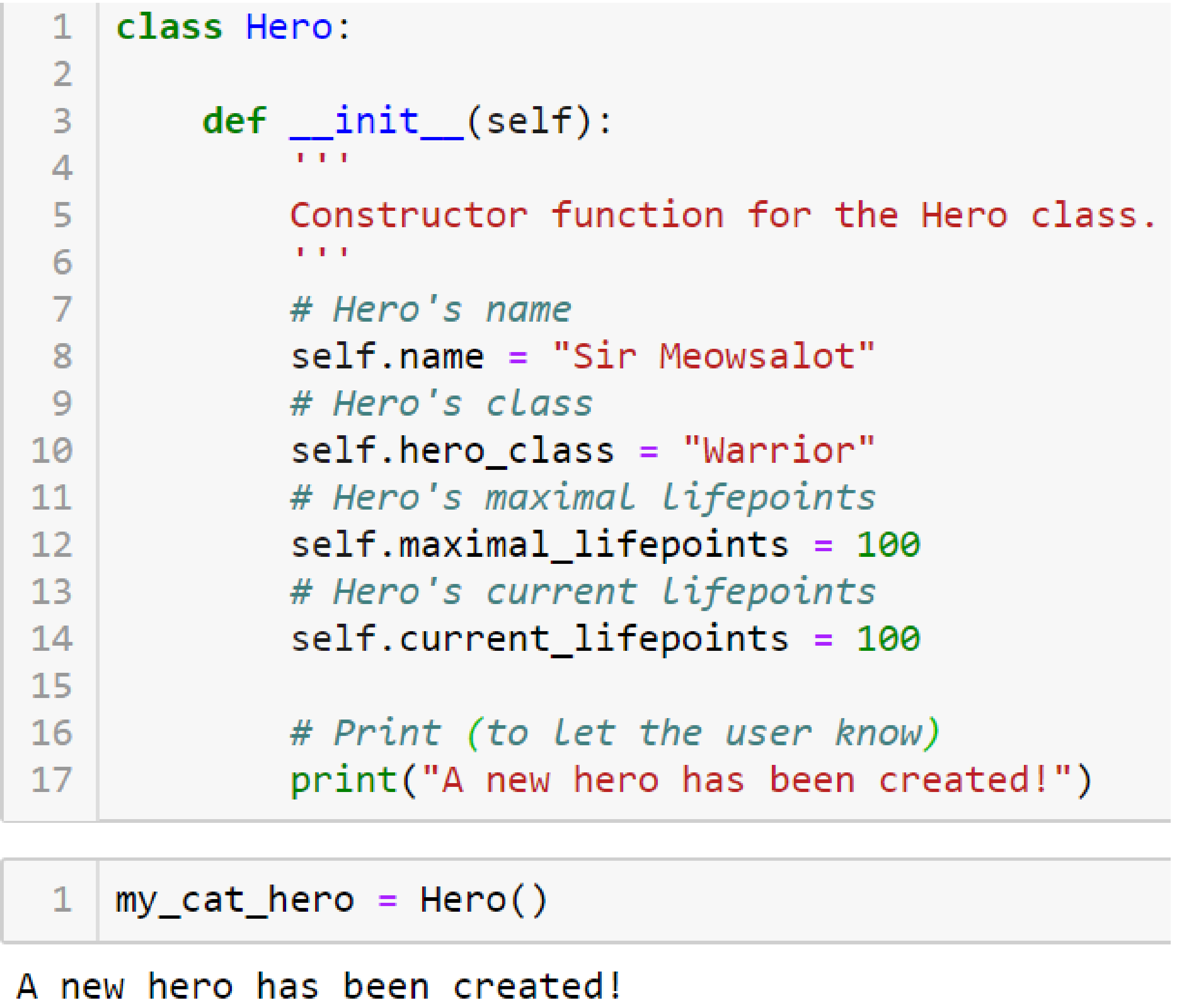


# The most important special method: the __init__ constructor method

**Definition (the __init__ method):**

The most important special method is **__init__**, which is called behind the scenes every time an object is created.

- This is called the **constructor** method of the class.
- When we run the operation *my_cat_hero = Hero()*, Python runs whatever is in the **__init__** method of the *Hero* class.



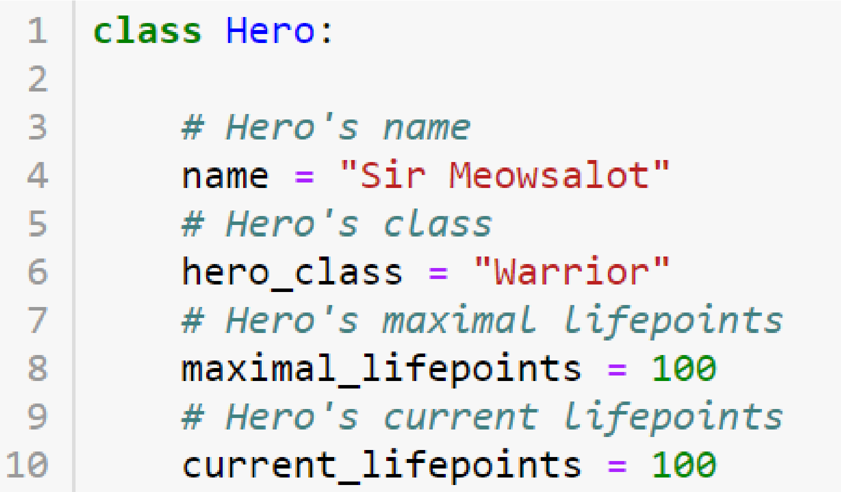
```python
class Hero:

    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()
```

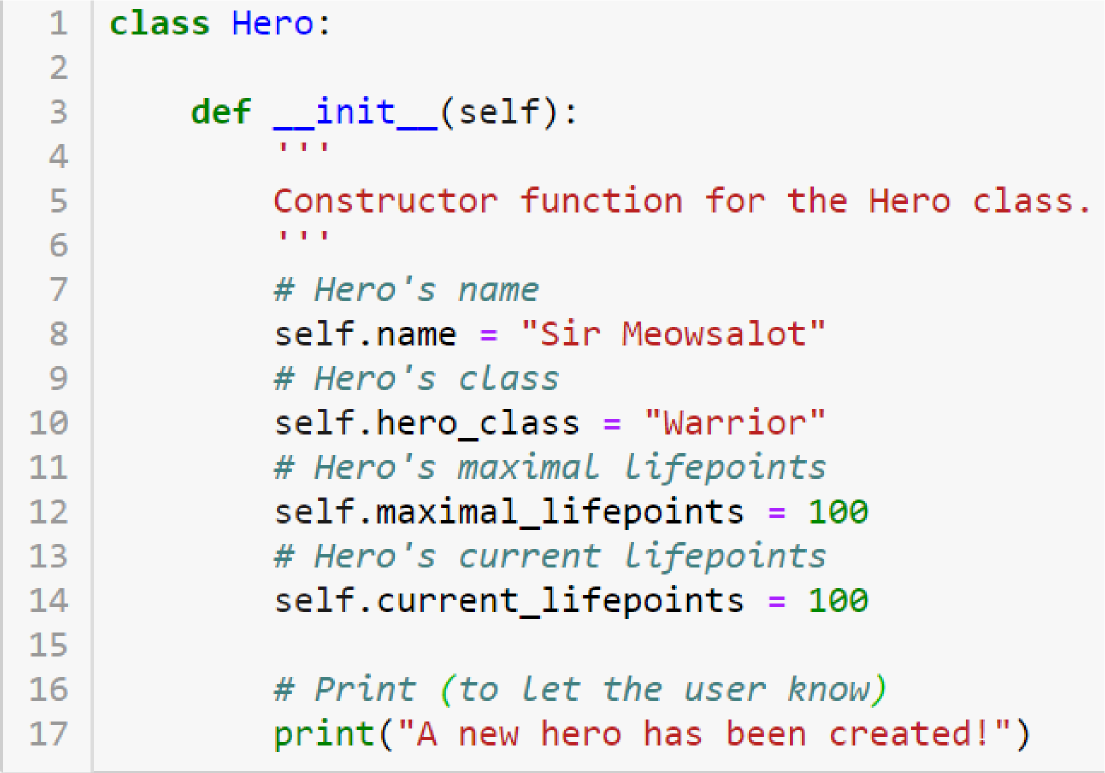
```
A new hero has been created!
```

# The __init__ constructor vs. « **trash** » initialization

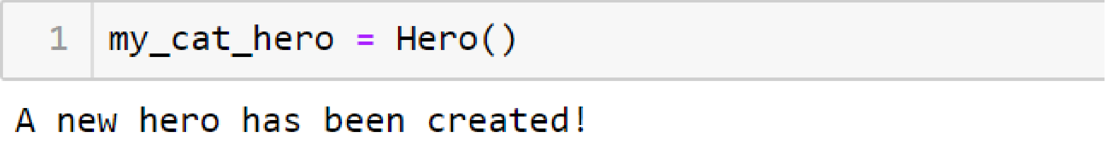
- It is often preferable to define and use the __init__ method!
- It is considered **good practice.**
- It also allows for more **actions** on **initialization.**



```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!")
```

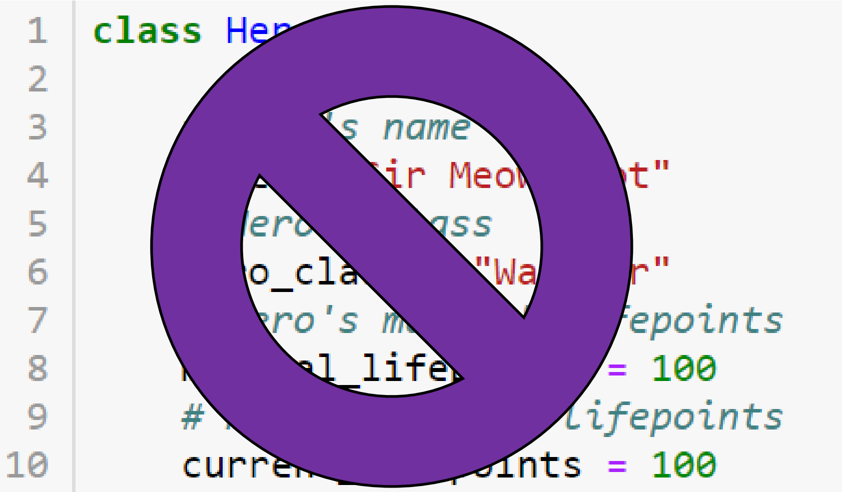


```python
my_cat_hero = Hero()
```

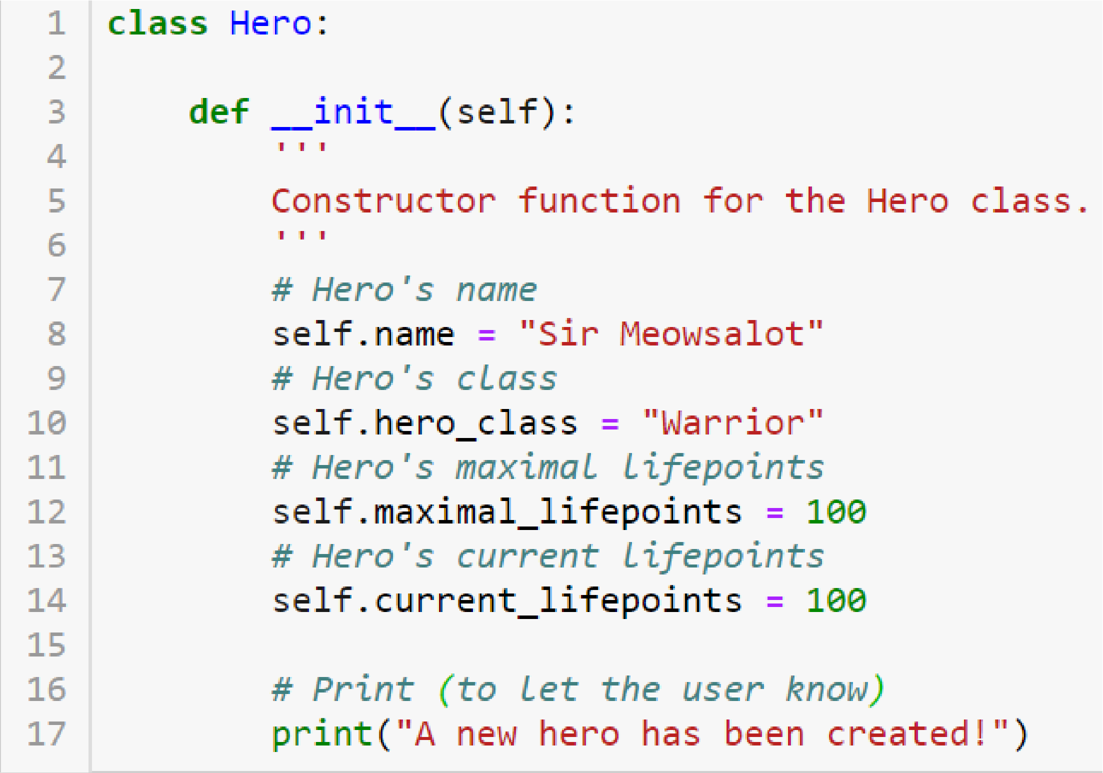
```
A new hero has been created!
```

# The __init__ constructor vs. « trash » initialization

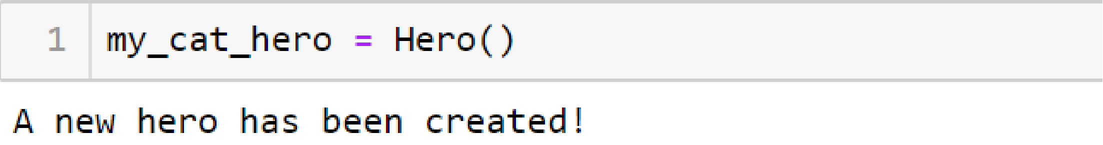
- It is often preferable to define and use the __init__ method!
- It is considered **good practice.**
- It also allows for more **actions** on **initialization.**



```
class Her

        's name
          ir Meo      t"
    ero      ass
    o_cla      "Wa      r"
     ro's m         fepoints
       l_life           = 100
    #                lifepoints
    curre        oints = 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!")
```

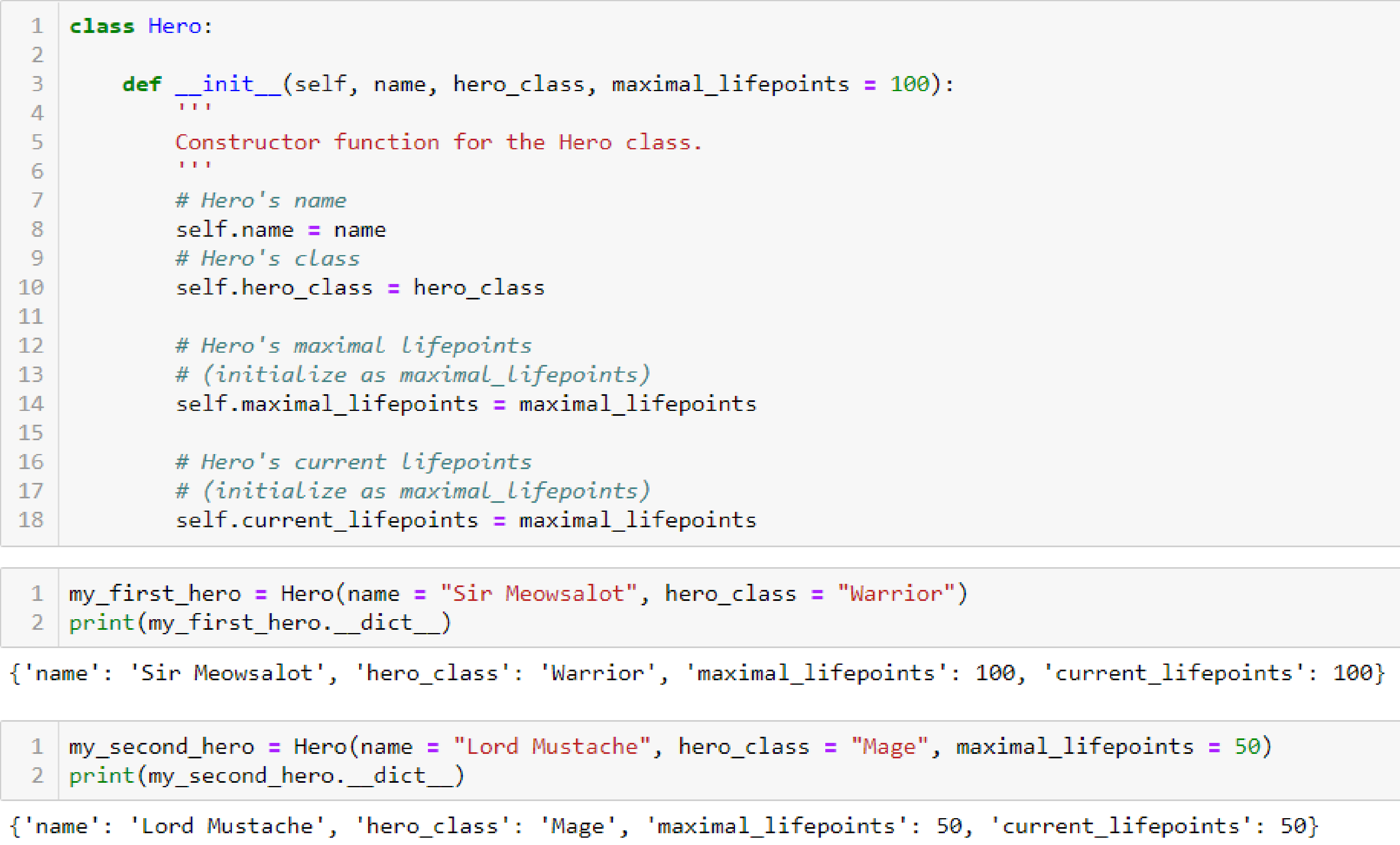


```python
my_cat_hero = Hero()
```

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

# 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__)
```

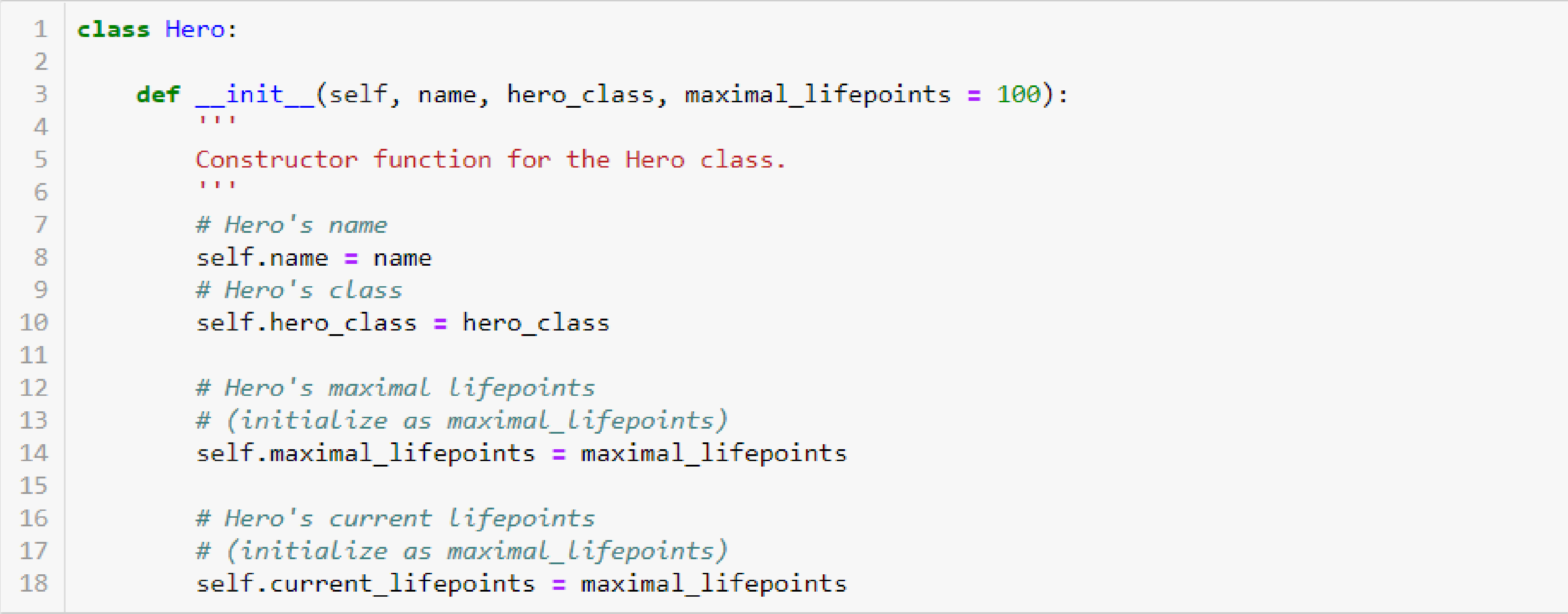
```
{'name': 'Sir Meowsalot', 'hero_class': 'Warrior', 'maximal_lifepoints': 100, 'current_lifepoints': 100}
```

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

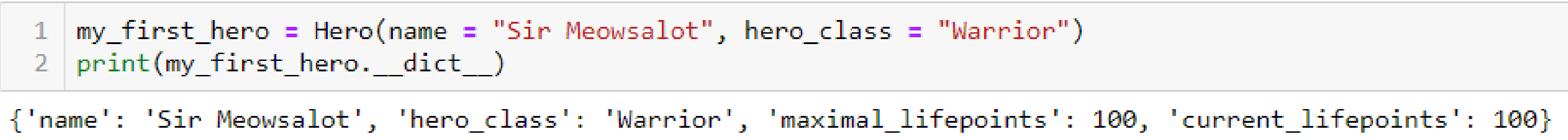
```
{'name': 'Lord Mustache', 'hero_class': 'Mage', 'maximal_lifepoints': 50, 'current_lifepoints': 50}
```

# 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**.

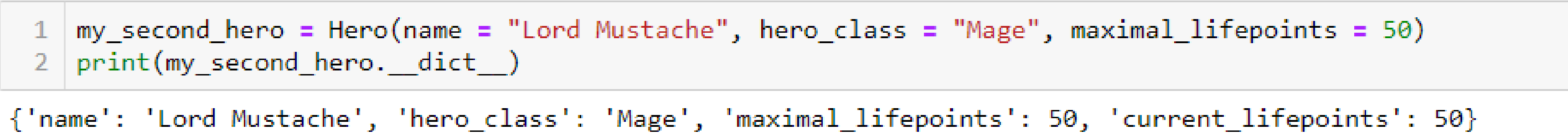


```python
class Hero:

    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}
```

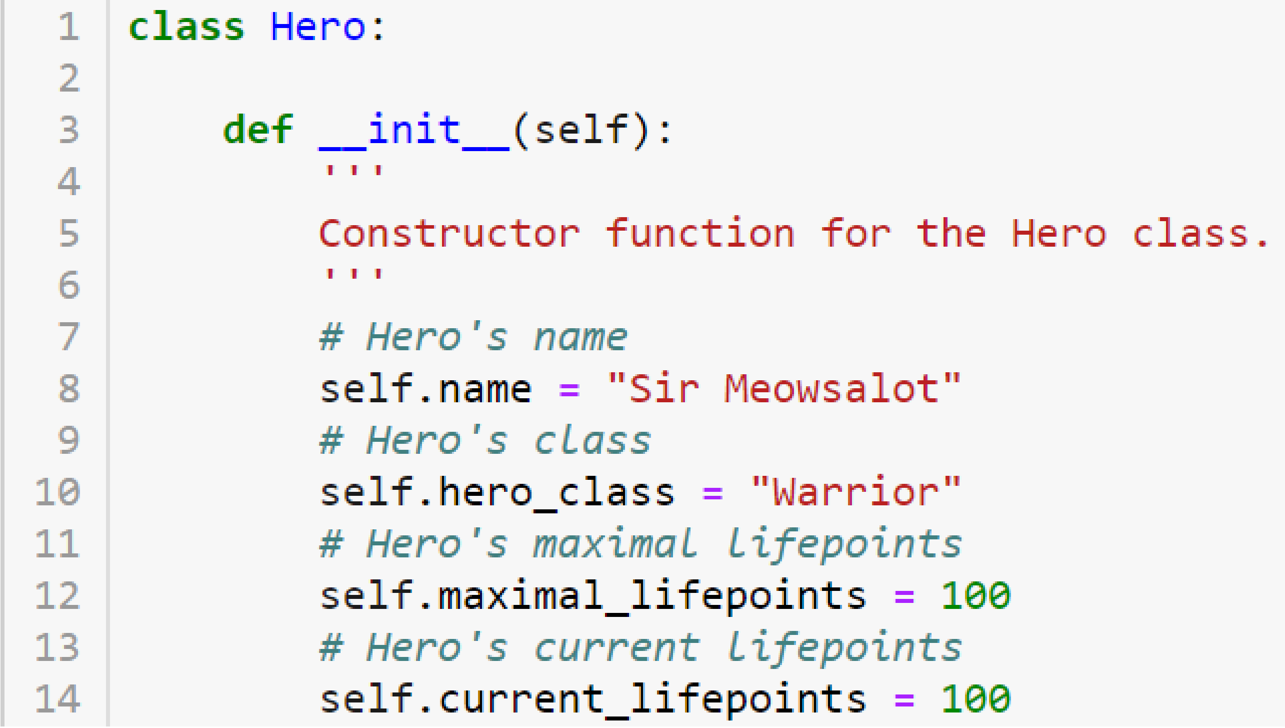
# The __dict__ special attribute

**Definition (__dict__):**

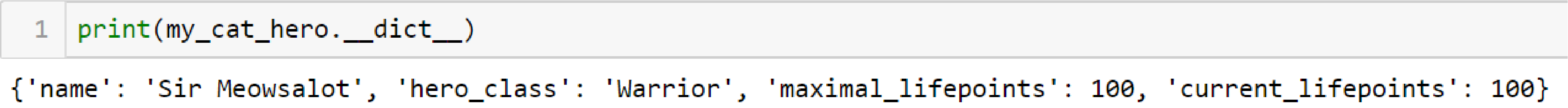
Another useful concept is the **special dictionnary attribute**, __dict__.

By default, it produces a **dictionnary**

- containing all the **attributes** of your object and their currently assigned **values**.
- (Looks familiar? → OOT!)



```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
```



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

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

# Practice 1

Let us put these concept in practice with a first Activity.

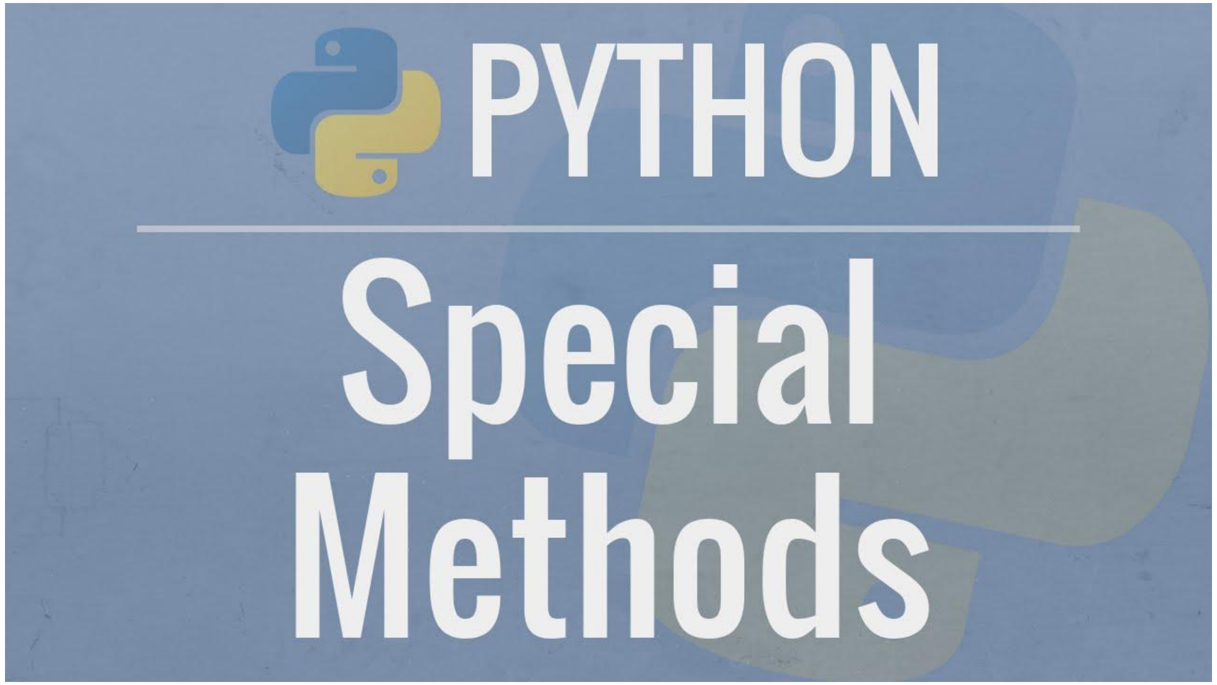
- You will have to design several **custom classes of objects**.
- Some of them will require **attributes** and **methods**.
- Some of them will require to write **__init__** methods.
- Remember to control the values of attributes in your objects using the **__dict__** operation!

# Class special methods

**Definition (special methods):**

A custom class may also have **special methods**.

- These methods have fixed names and are written with **double underscores (__)** before and after their names.
- These methods do something special when some basic operations (+, *, len(), etc.) are applied to our object.

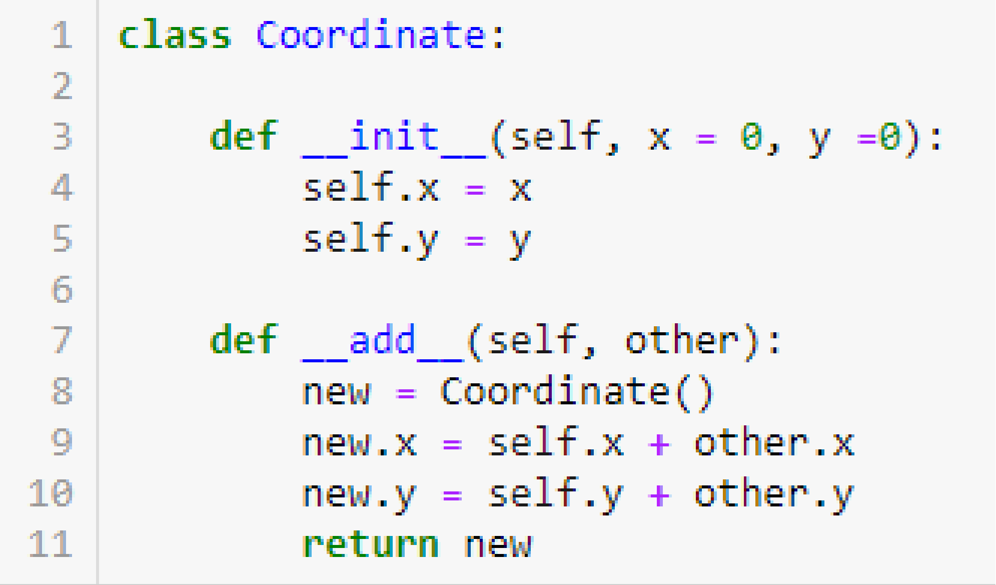


# About special methods

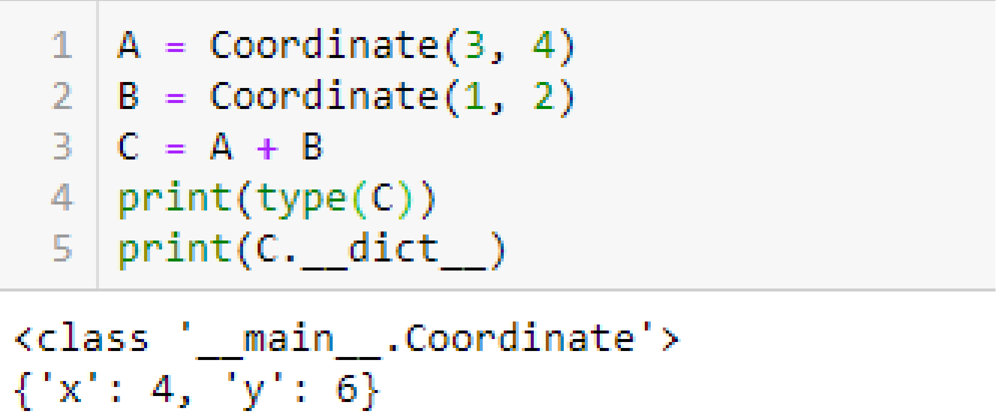
**Definition (__add__):**

There are more special methods, besides the __init__ one. The __add__ special method defines the behavior you want for your objects when two objects of our custom class are summed together with **+.**

- This allows us to control what the **+** behavior should be. This decided that **+** would be concatenation for strings/lists, math addition for int/float, etc.



```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
```



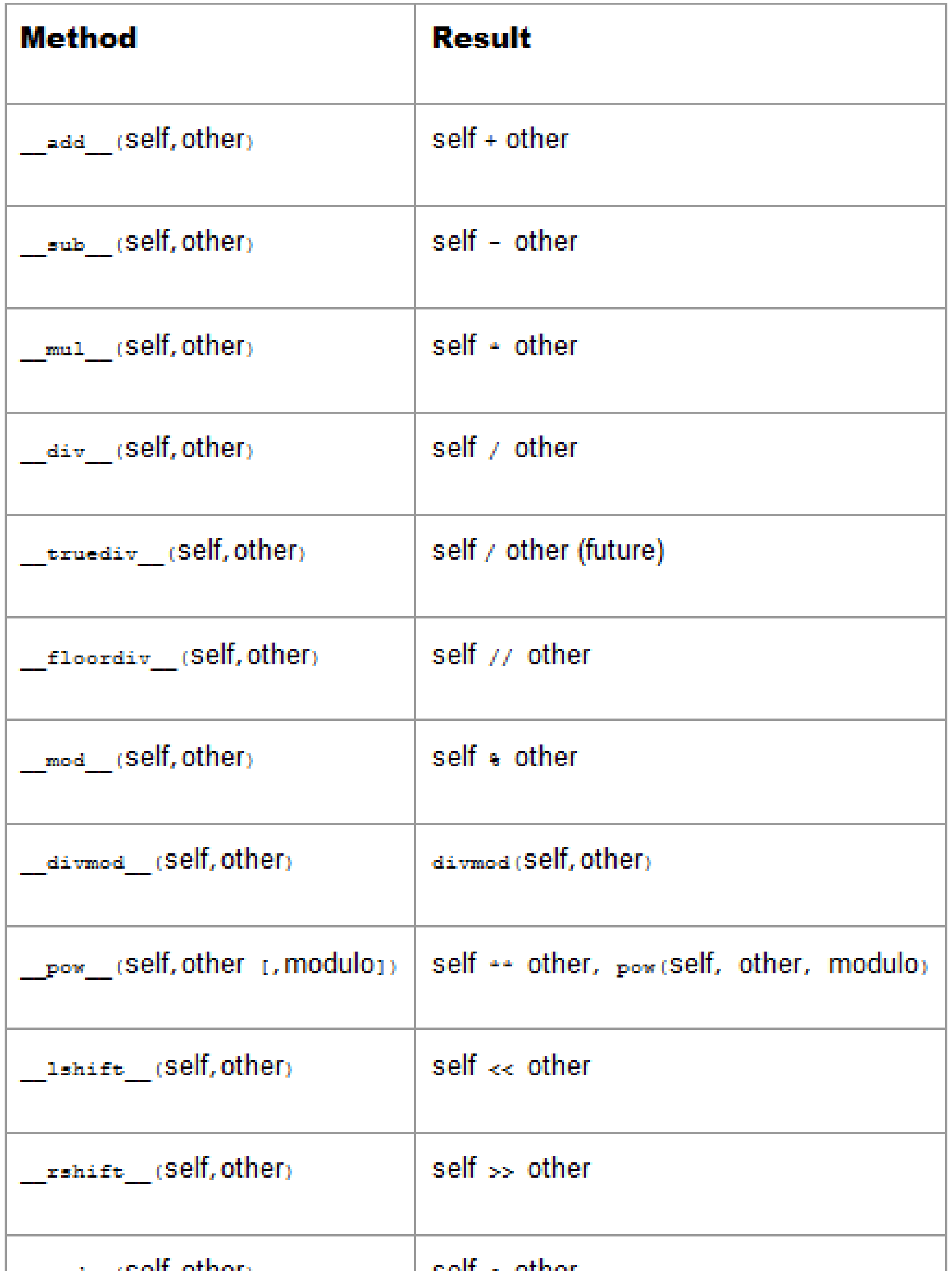
```python
A = Coordinate(3, 4)
B = Coordinate(1, 2)
C = A + B
print(type(C))
print(C.__dict__)
```

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

# About special methods

Typically, we will have **special methods** for each **operator** (+, -, *, /, etc.), and **built-in function** (len, print, etc.).

- While it would be impossible for me to describe and showcase all of them one by one, you can easily find them in the documentation, here: [https://docs.python.org/3/reference/datamodel.html#special-method-names](https://docs.python.org/3/reference/datamodel.html#special-method-names)



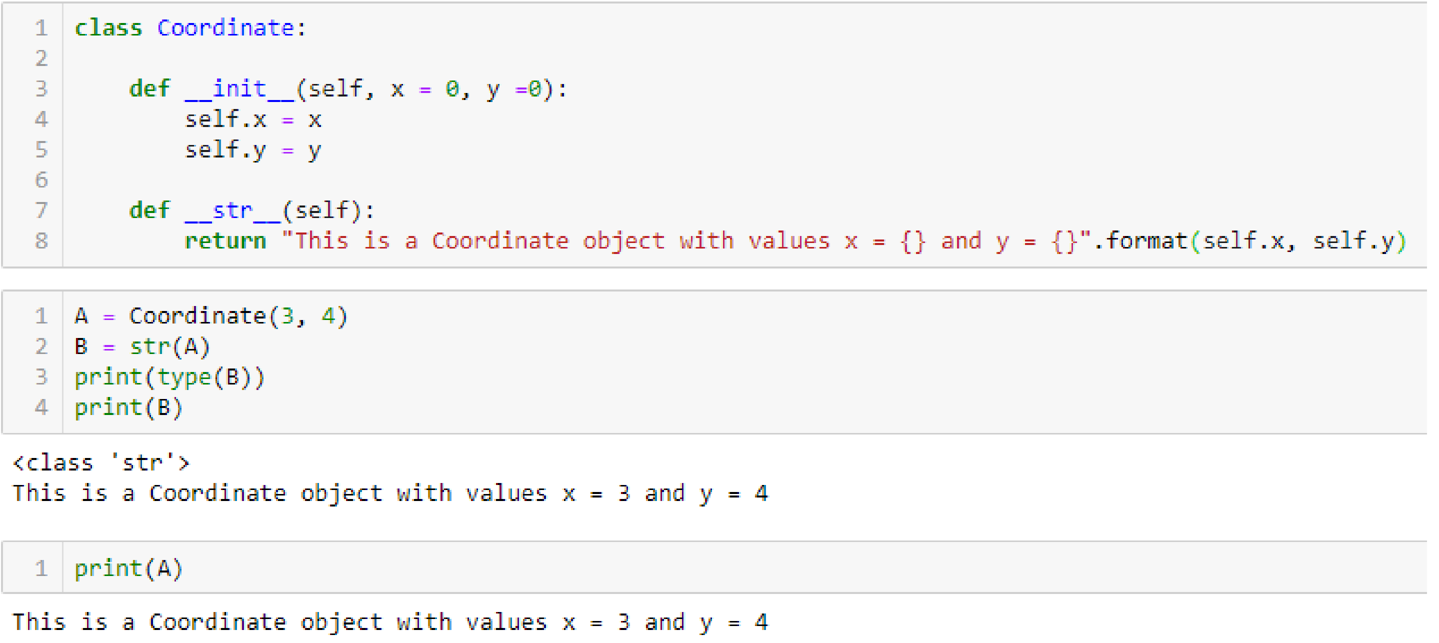
| Method | Result |
|---|---|
| `__add__`(self,other) | self + other |
| `__sub__`(self,other) | self - other |
| `__mul__`(self,other) | self * other |
| `__div__`(self,other) | self / other |
| `__truediv__`(self,other) | self / other (future) |
| `__floordiv__`(self,other) | self // other |
| `__mod__`(self,other) | 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 |
| [illegible](self,other) | self - other |

# The str special method

**Definition (__str__):**

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!



```
class Coordinate:

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

    def __str__(self):
        return "This is a Coordinate object with values x = {} and y = {}".format(self.x, self.y)
```

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

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

```
print(A)
```

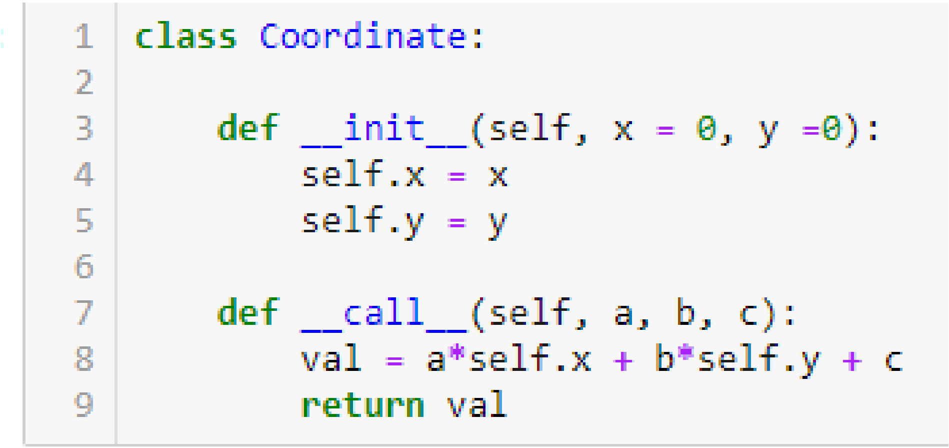
```
This is a Coordinate object with values x = 3 and y = 4
```

# Call method

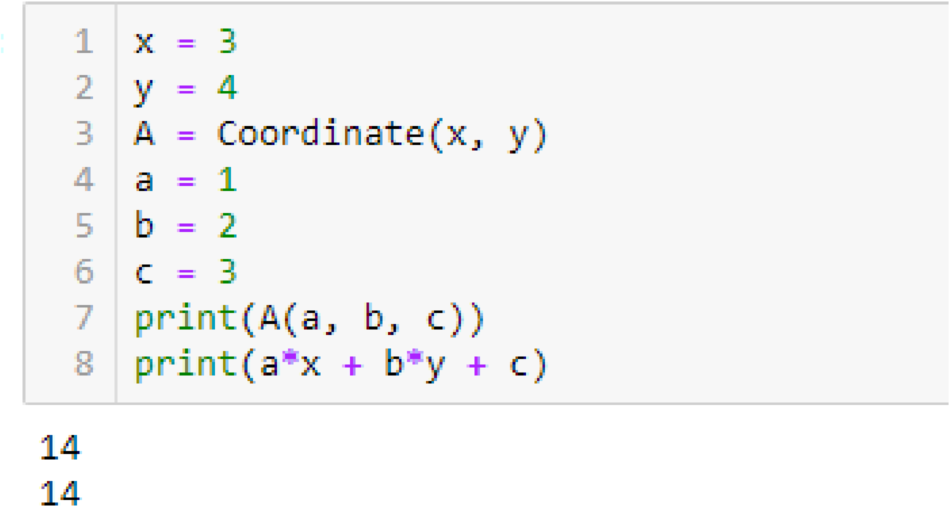
**Definition (__call__):**

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

- This also serves to show that functions are, technically, “just a special type of variable”.
- Functions defined with **def** could be seen as variables with a __call__ method containing your instructions. (Technically a gross oversimplification, but ok for now).



```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
```

# Practice 2

Let us put these concept in practice with a new Activity.

- You will reuse several **custom classes of objects from Activity 1**.
  - Some of them will require **some additional special methods**.
- Remember to control the values of attributes in your objects using the **__dict__** operation!

# Our toy example: an RPG main protagonist

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

- **Problem:** our Hero's attack capabilities probably depend on the weapon he has **equipped**.
- And this weapon should probably be an **object** as well!
- Maybe we should have variables of type *Weapon*, and have them interact with the *Hero* class?

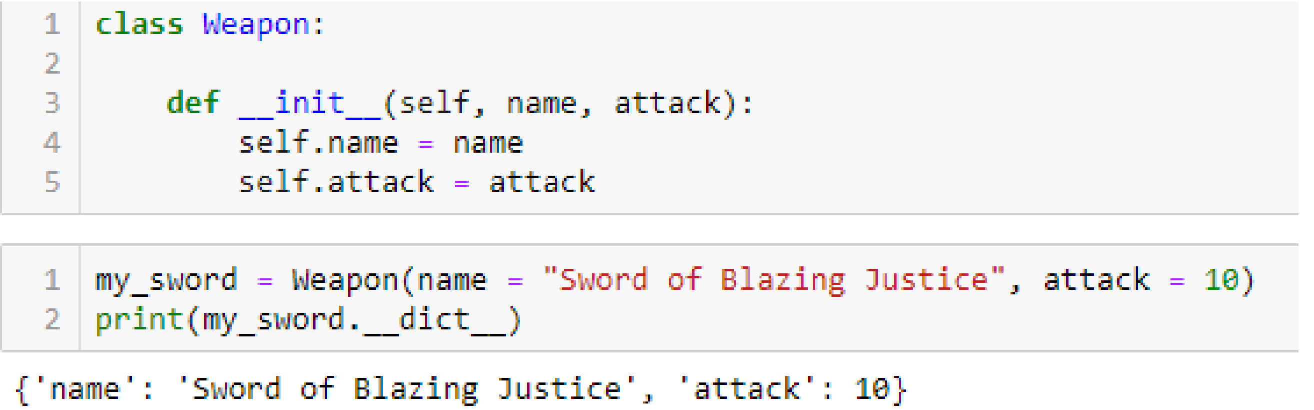
# Introducing a Weapon class object!

Reusing the previous concepts, we could define a **Weapon object**

It will have its own attributes, such as

- a name
- some attack values,
- and possibly more stuff.

*(For now, let us keep it simple.)*



```python
class Weapon:

    def __init__(self, name, attack):
        self.name = name
        self.attack = attack
```

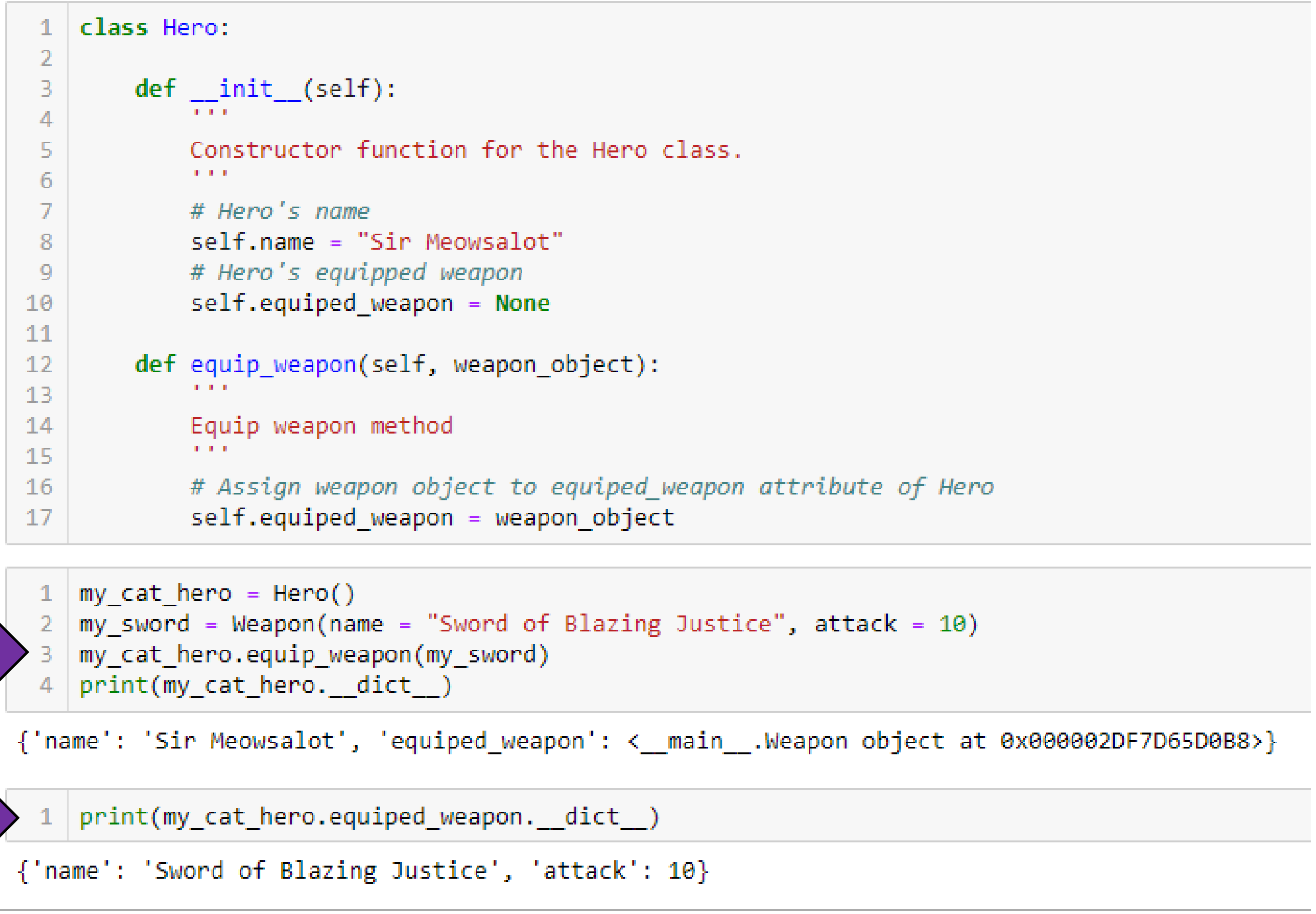
```python
my_sword = Weapon(name = "Sword of Blazing Justice", attack = 10)
print(my_sword.__dict__)
```

```
{'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!



```
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
```

```
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>}
```

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

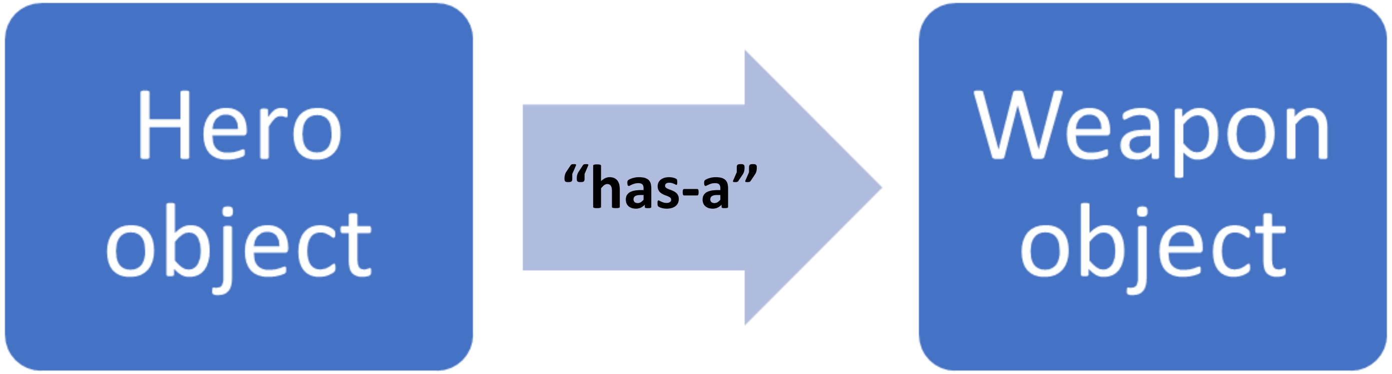
```
{'name': 'Sword of Blazing Justice', 'attack': 10}
```

# The « has-a » relationship

**Definition (the “has-a” relationship in OOP):**

In Object-Oriented Programming, this defines a **“has-a” relationship** between our Hero class and our Weapon class.

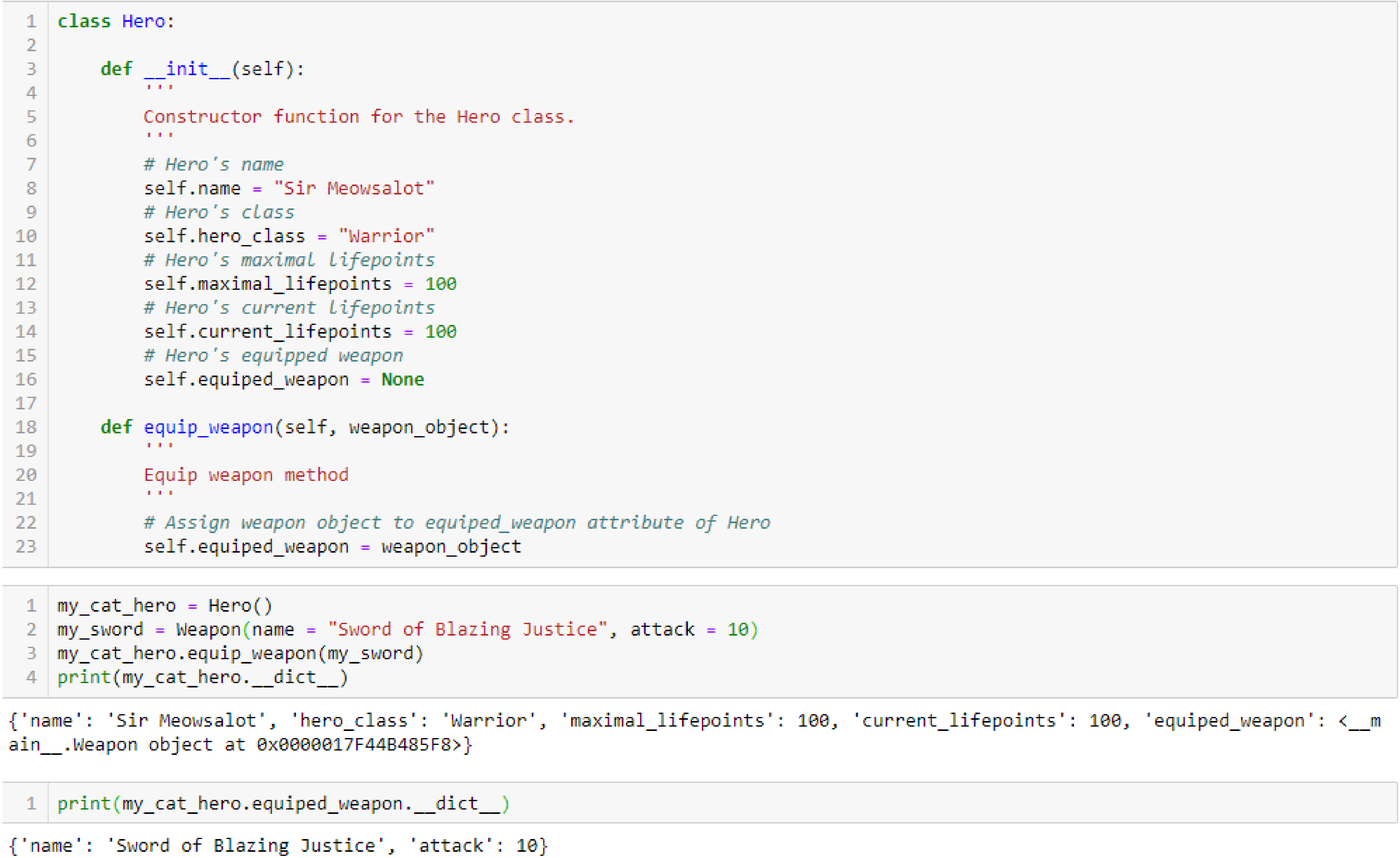
- We then say that our **Hero** object **“has-a” Weapon** object.
- Because one of our **Hero object’s attributes** is a **Weapon object.**



Hero object

“has-a”

Weapon object



```
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
```

```
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': <__m
ain__.Weapon object at 0x0000017F44B485F8>}
```

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

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

# Practice 3

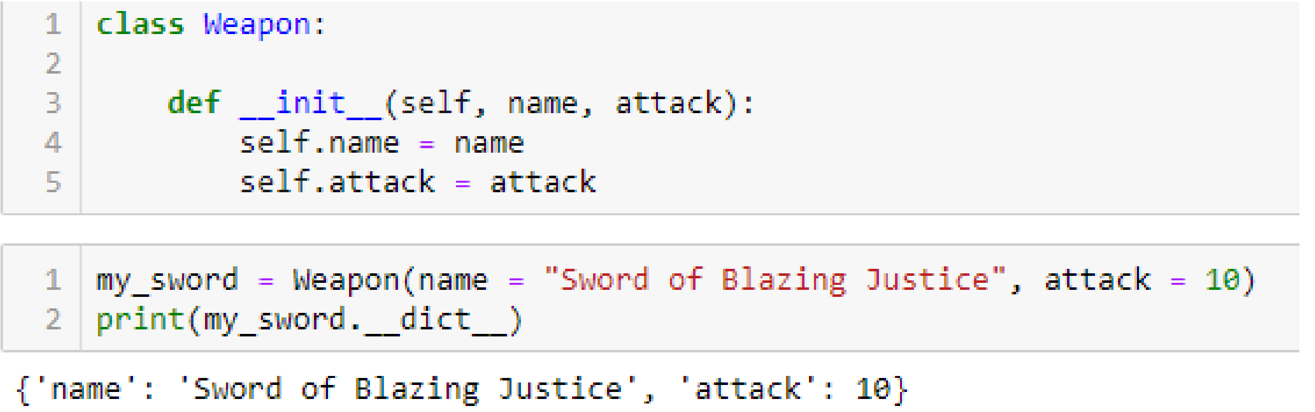
Let us put these concept in practice with a new Activity.

- You will reuse several **custom classes of objects from Activity 1&2**.
- You will have to implement new additional classes and establish some **“has-a” relationships** between the different classes.
- Remember to control the values of attributes in your objects using the **__dict__** operation!

# 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 types of weapons possibilities? (sword, bow, axe, magic staff, etc.



```
class Weapon:

    def __init__(self, name, attack):
        self.name = name
        self.attack = attack
```

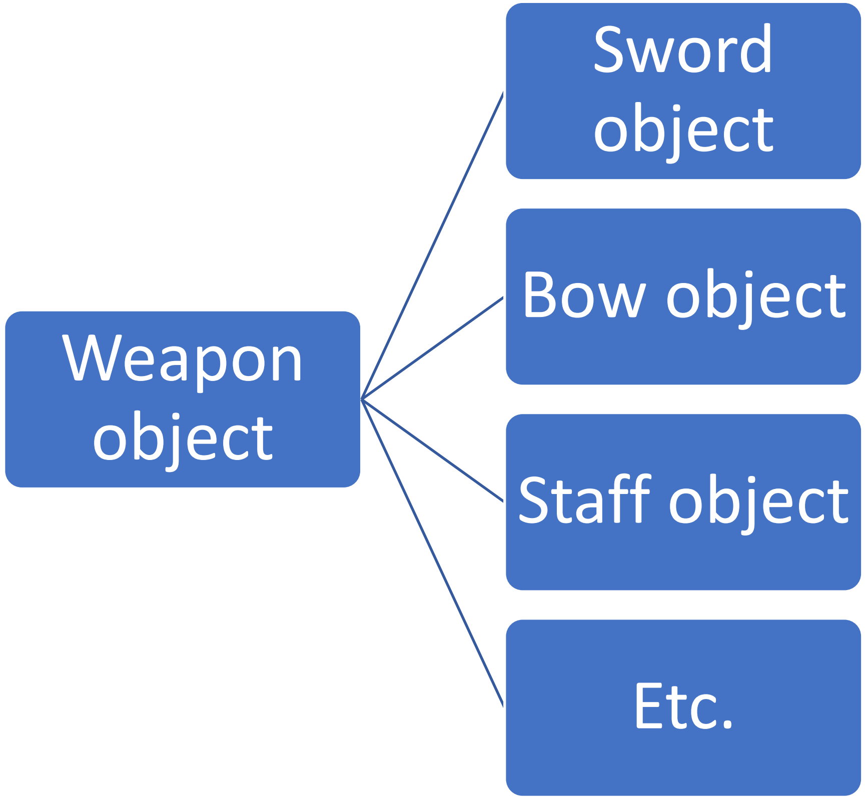
```
my_sword = Weapon(name = "Sword of Blazing Justice", attack = 10)
print(my_sword.__dict__)
```

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

# Objects and sub-classes of objects

Typically, our Hero could equip a Weapon object...

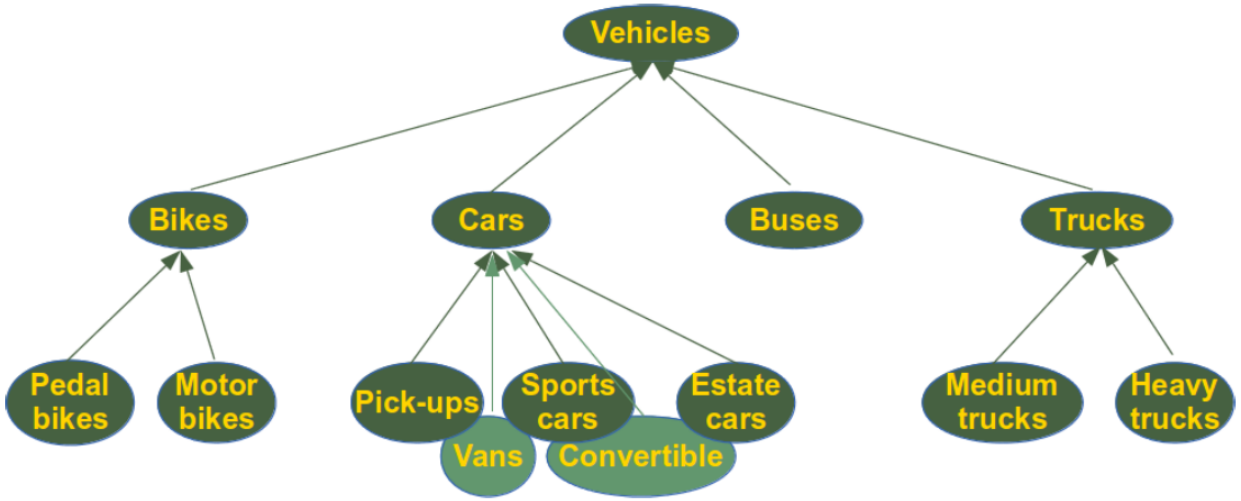
- But it could be a **sword**, a **bow**, a **staff**, etc.
- These weapons will probably have attributes and methods in common.
- But they might also have different attributes and methods, as our Hero will probably operate those weapons differently.



# 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.

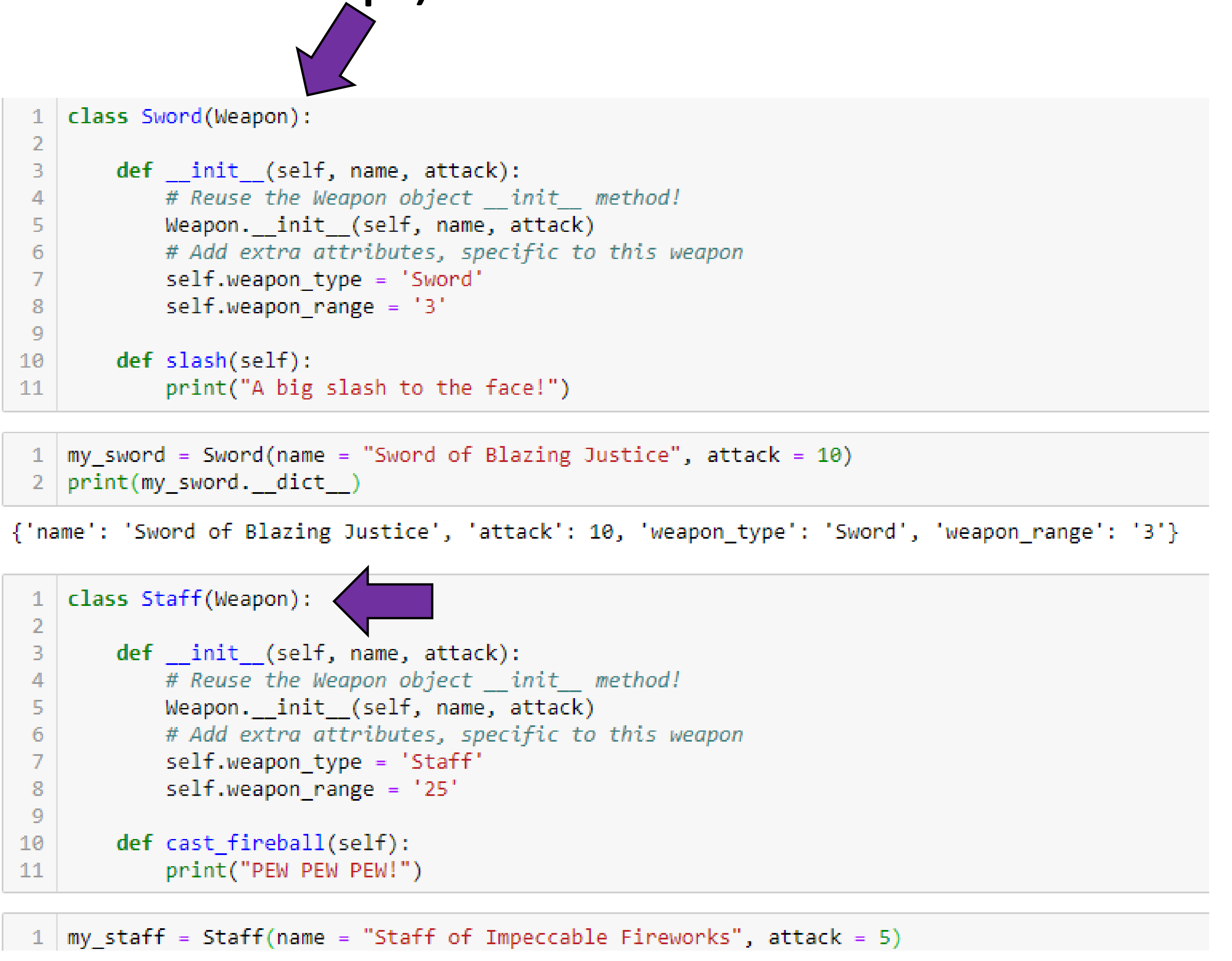


# Introducting inheritance!
# (a.k.a. the « is-a » relationship)

**Definition (Inheritance):**

It is possible to **create** a class, which **reuses methods** and **attributes** from another class!

- In OOP, this is called **inheritance**.
- 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)
```

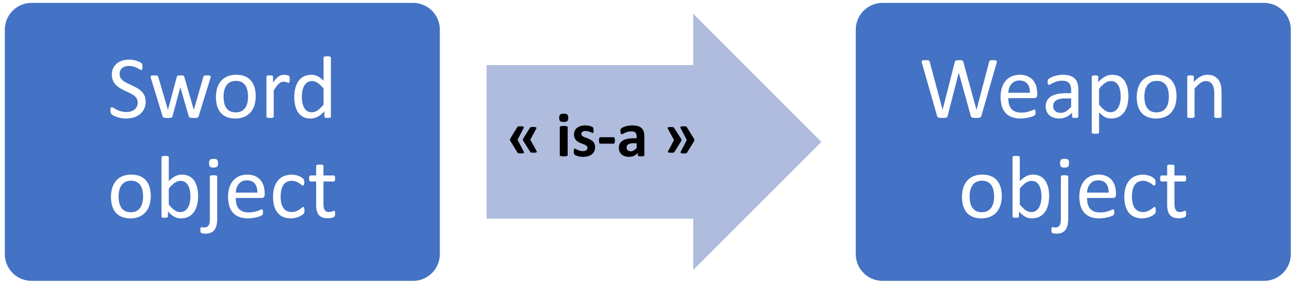
# Inheritance and « is-a » relationship

**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.

We say that Sword is the **child class** and Weapon is the **mother class** in that case. We might also refer to them as **derived class** and **base class**.



Sword object

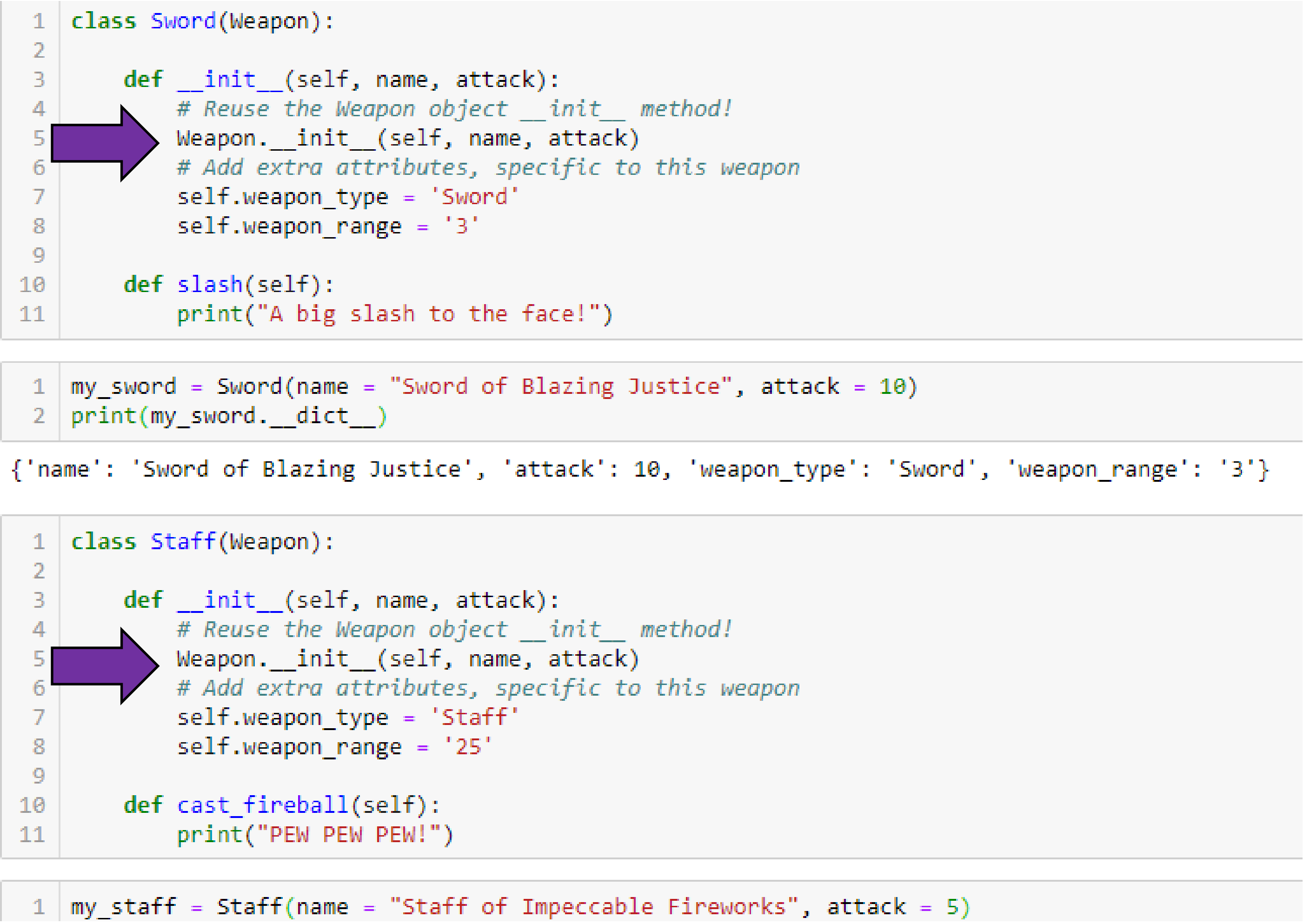
« is-a »

Weapon object

# Inheritance in practice

With **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'}
```

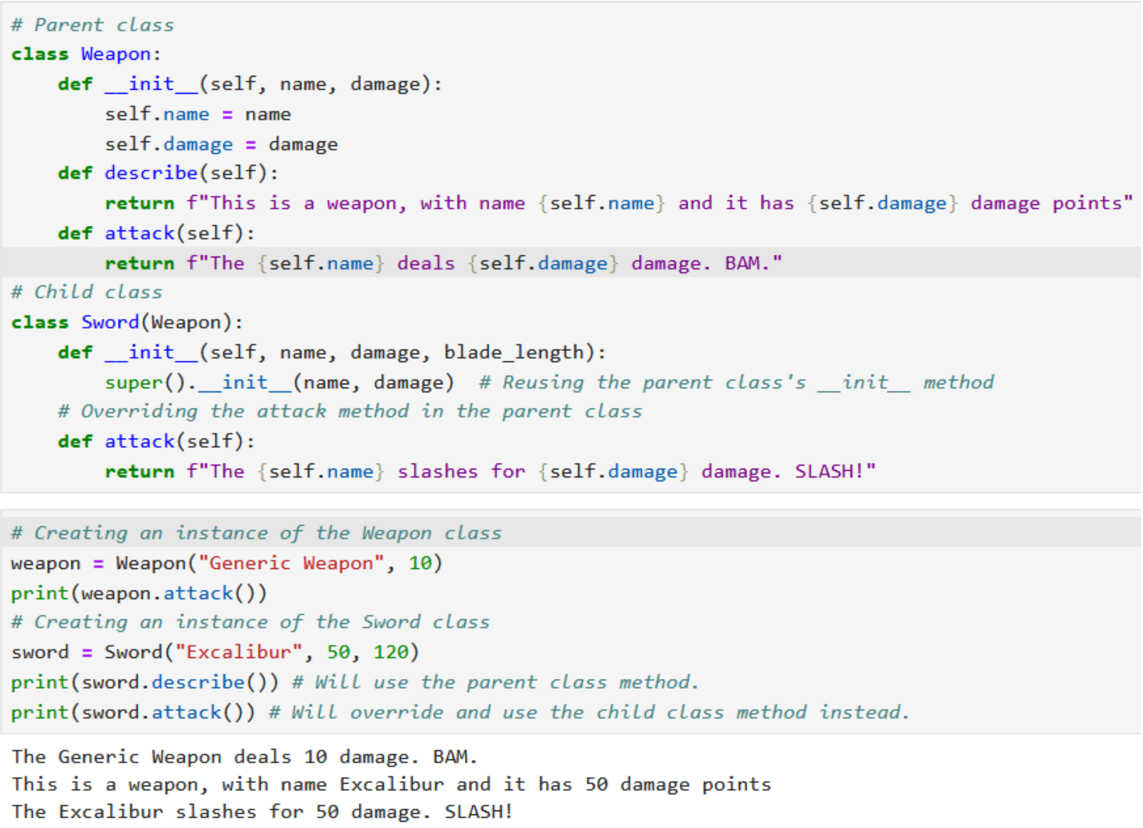
```python
class Staff(Weapon):

    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 in practice

On top of attributes, inheritance allows for methods reuse.

- In addition, methods from the parent class can be reused freely by the child class.
- If both the child class and the parent class have a method with the same name, the child class will override the method definition for objects of the child class.



```python
# Parent class
class Weapon:
    def __init__(self, name, damage):
        self.name = name
        self.damage = damage
    def describe(self):
        return f"This is a weapon, with name {self.name} and it has {self.damage} damage points"
    def attack(self):
        return f"The {self.name} deals {self.damage} damage. BAM."
# Child class
class Sword(Weapon):
    def __init__(self, name, damage, blade_length):
        super().__init__(name, damage)  # Reusing the parent class's __init__ method
    # Overriding the attack method in the parent class
    def attack(self):
        return f"The {self.name} slashes for {self.damage} damage. SLASH!"
```

```python
# Creating an instance of the Weapon class
weapon = Weapon("Generic Weapon", 10)
print(weapon.attack())
# Creating an instance of the Sword class
sword = Sword("Excalibur", 50, 120)
print(sword.describe()) # Will use the parent class method.
print(sword.attack()) # Will override and use the child class method instead.
```

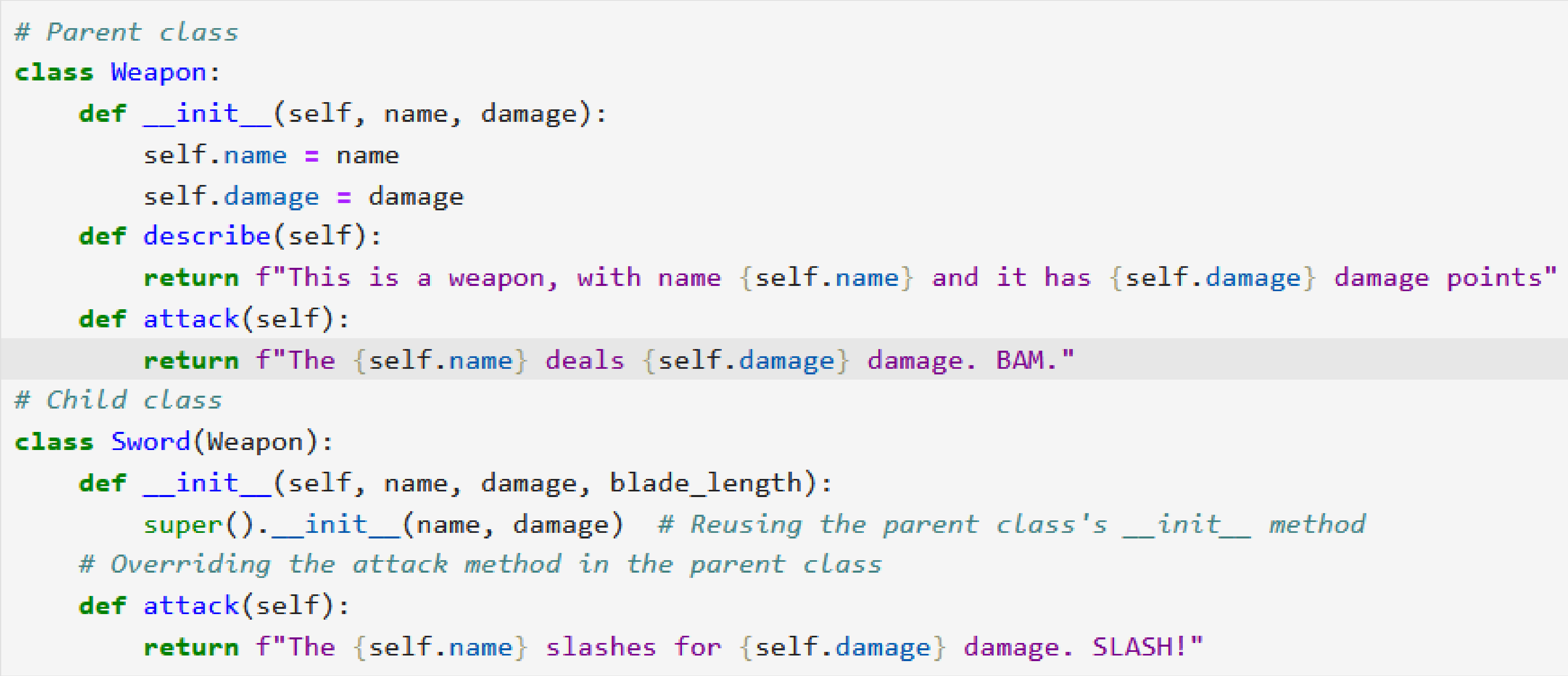
```
The Generic Weapon deals 10 damage. BAM.
This is a weapon, with name Excalibur and it has 50 damage points
The Excalibur slashes for 50 damage. SLASH!
```

*(Will enlarge in the next slide.)*

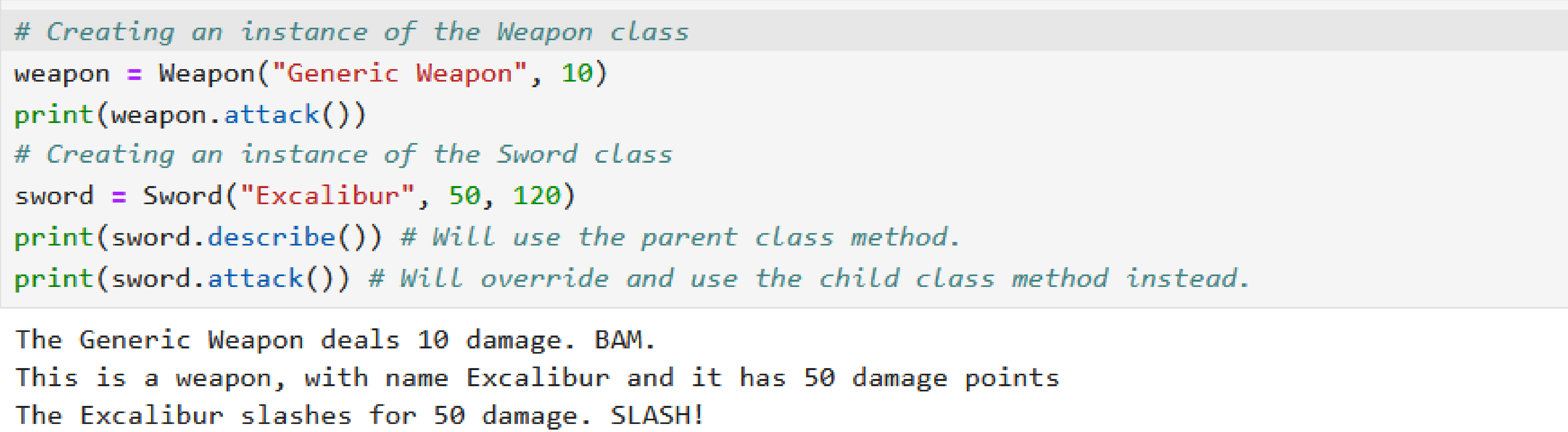
# In

Or
all

- I
  p
  b
- I
  p
  t
  c
  c



```python
# Parent class
class Weapon:
    def __init__(self, name, damage):
        self.name = name
        self.damage = damage
    def describe(self):
        return f"This is a weapon, with name {self.name} and it has {self.damage} damage points"
    def attack(self):
        return f"The {self.name} deals {self.damage} damage. BAM."
# Child class
class Sword(Weapon):
    def __init__(self, name, damage, blade_length):
        super().__init__(name, damage)  # Reusing the parent class's __init__ method
    # Overriding the attack method in the parent class
    def attack(self):
        return f"The {self.name} slashes for {self.damage} damage. SLASH!"
```



```python
# Creating an instance of the Weapon class
weapon = Weapon("Generic Weapon", 10)
print(weapon.attack())
# Creating an instance of the Sword class
sword = Sword("Excalibur", 50, 120)
print(sword.describe()) # Will use the parent class method.
print(sword.attack()) # Will override and use the child class method instead.
```

```
The Generic Weapon deals 10 damage. BAM.
This is a weapon, with name Excalibur and it has 50 damage points
The Excalibur slashes for 50 damage. SLASH!
```

# Practice 4

Let us put these concept in practice with a new Activity.

- You will reuse several **custom classes of objects from Activity 1&2**.
- You will have to implement new additional classes and establish some **“is-a” relationships** between the different classes.
- Remember to control the values of attributes in your objects using the **__dict__** operation!

# Some good practice in OOP

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

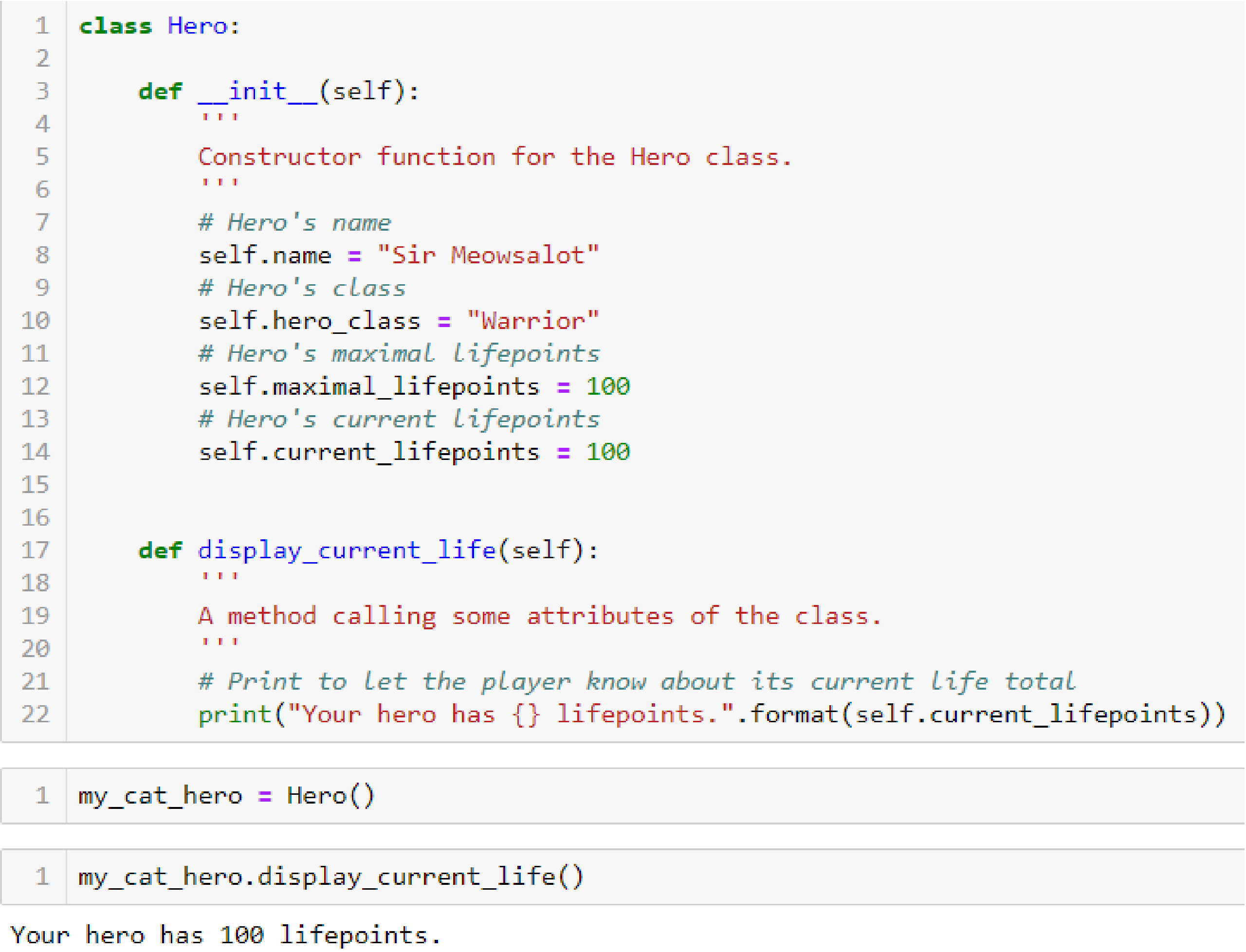
- In addition, it could also be interesting to use **setters** and **getters methods** for your class **attributes**.
- And also, to define if your class **attributes** should be **public** or **private**.

# 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 shown on the right.

- And it works just fine!
- However, this is considered **bad 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_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))
```

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

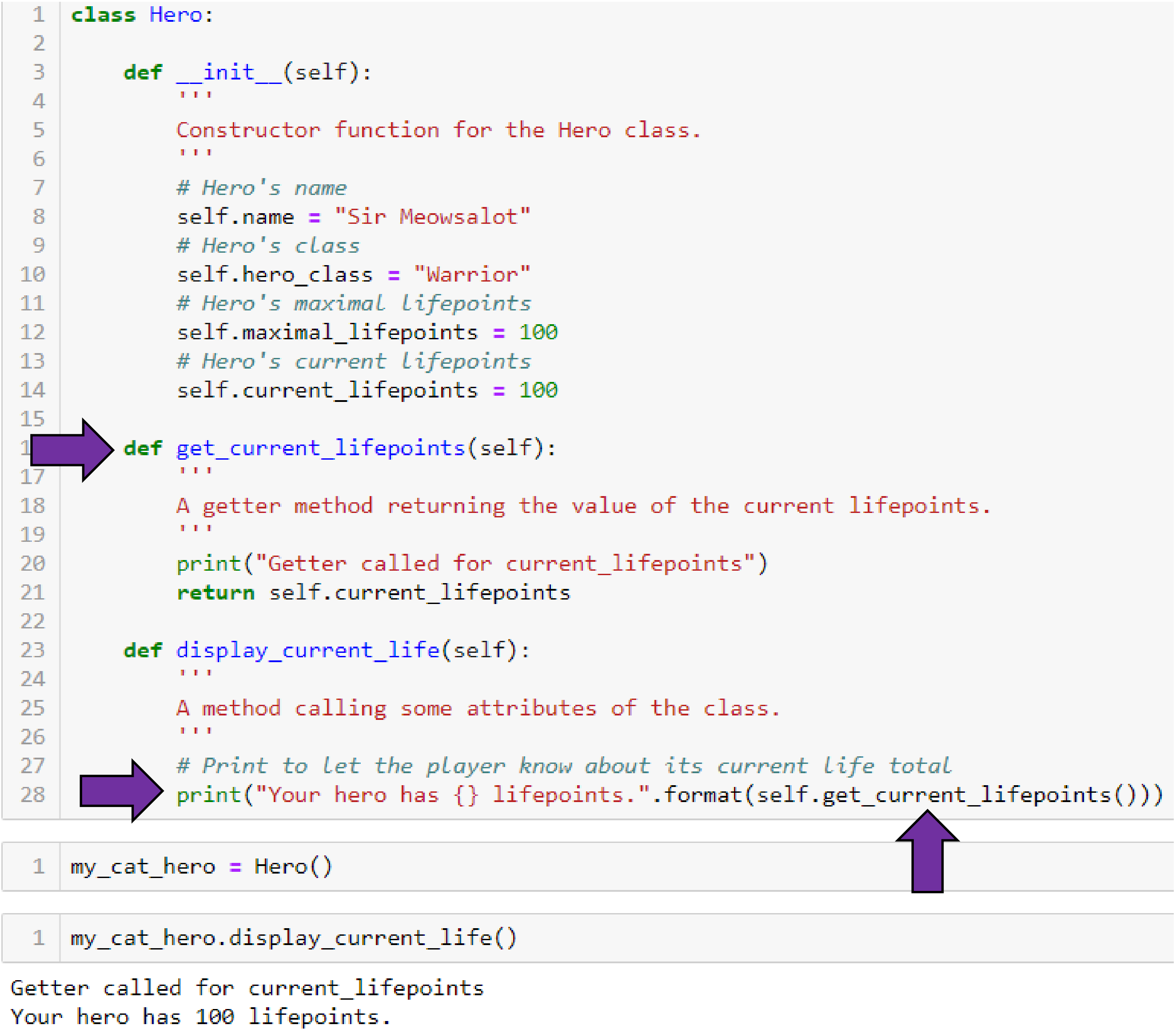
```python
my_cat_hero.display_current_life()
```

```
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 method *display_current_life()*.



```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()
```

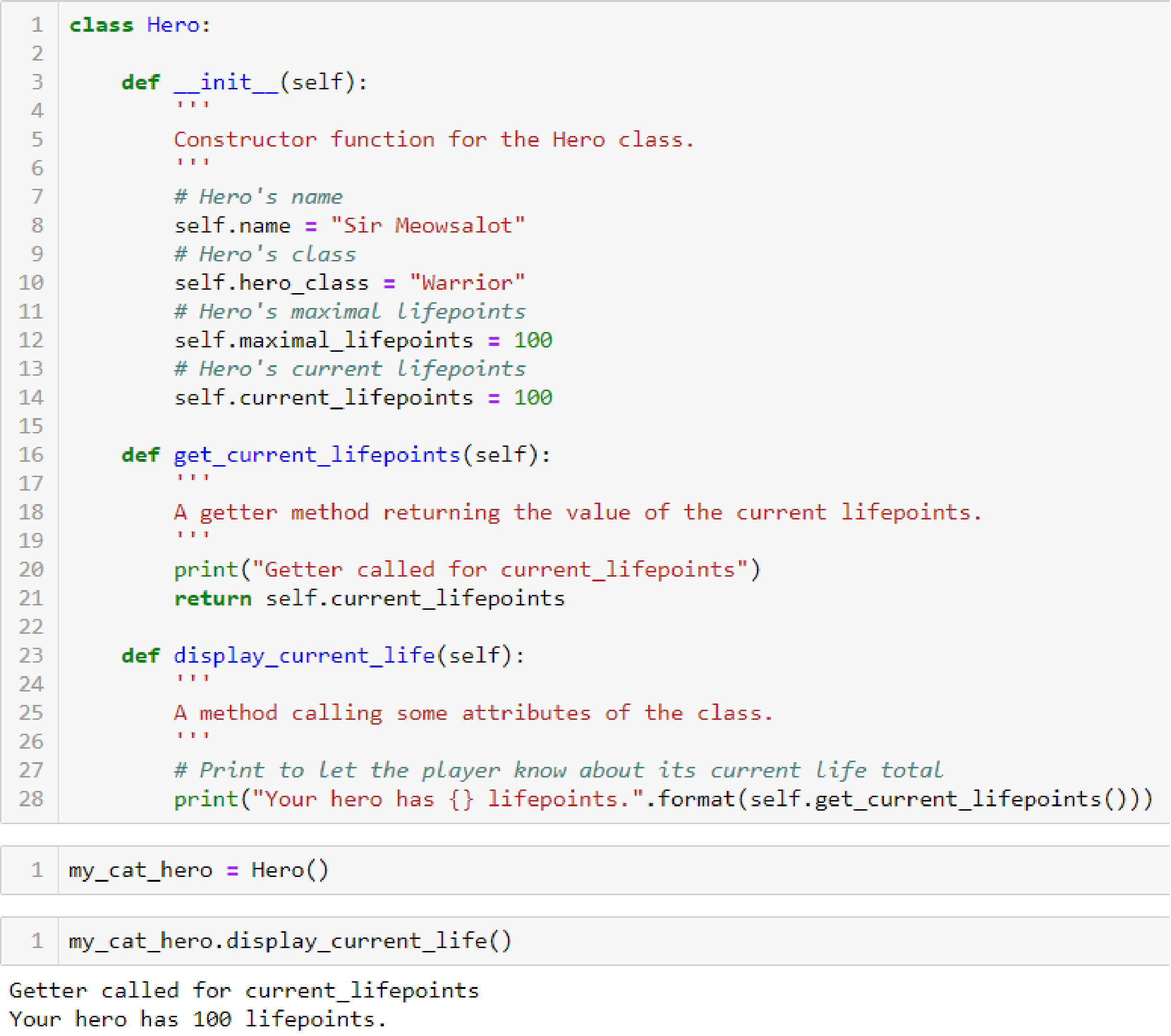
```python
my_cat_hero.display_current_life()
```

```
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!



```
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()
```

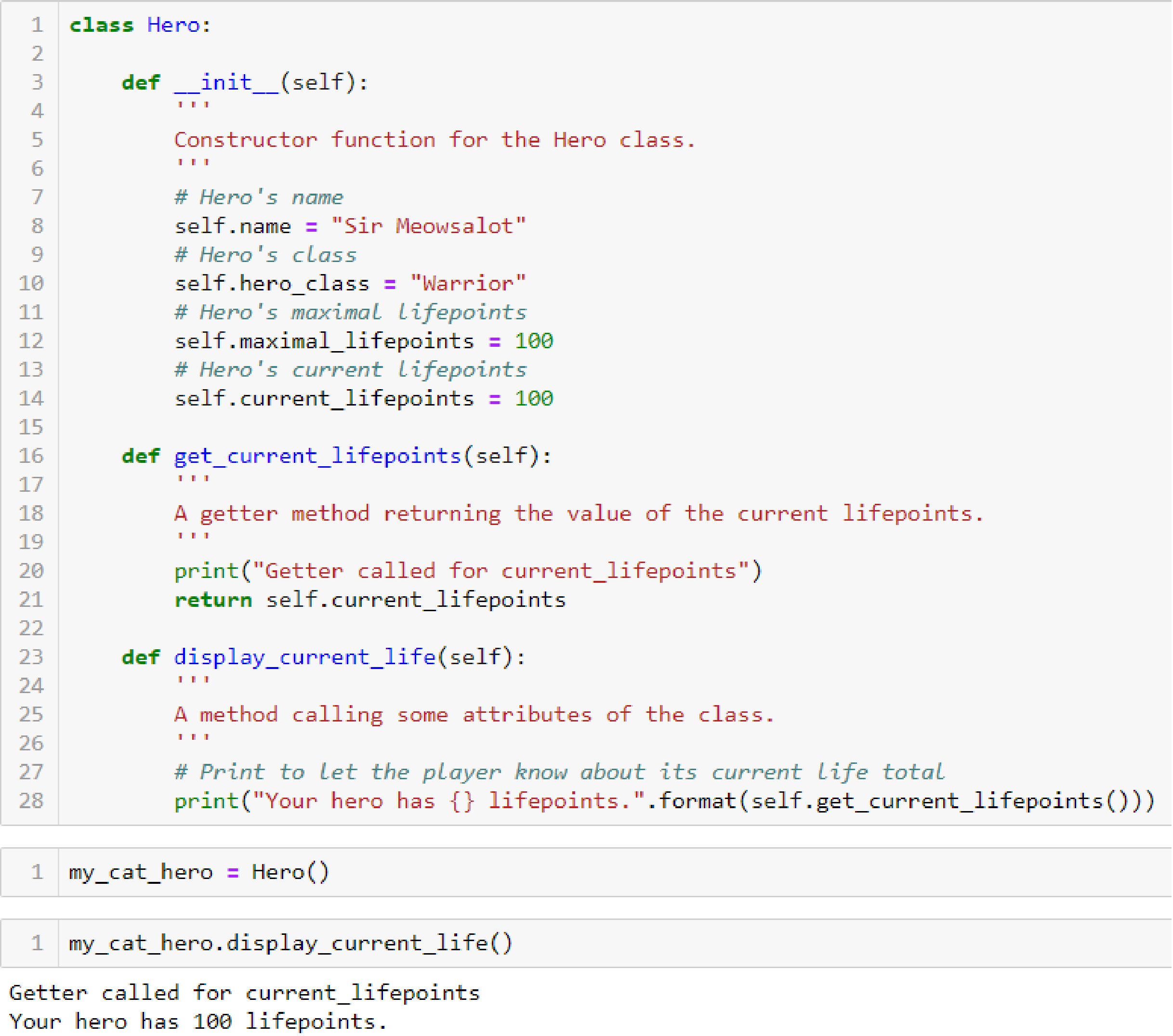
```
Getter called for current_lifepoints
Your hero has 100 lifepoints.
```

# Why is it good practice to use setters and getters?

**Question:** 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()))
```

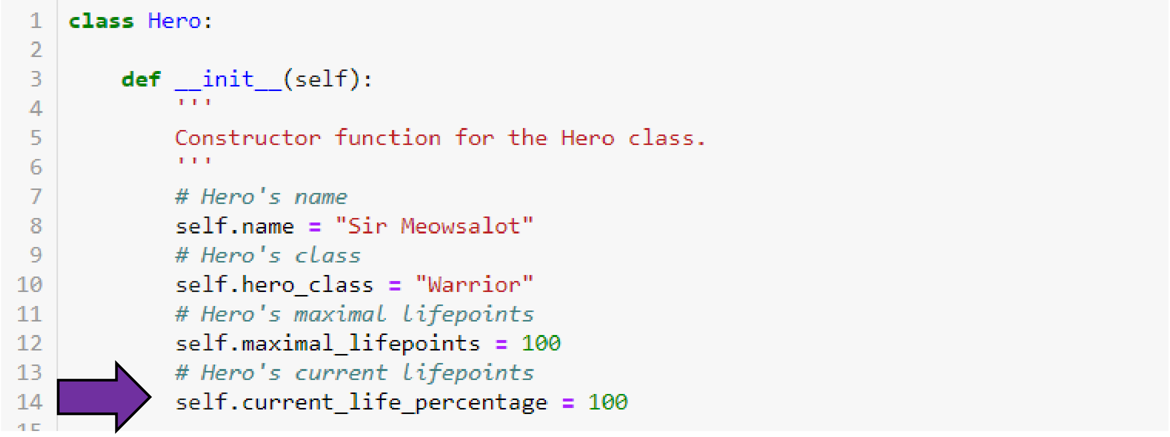
```python
my_cat_hero = Hero()
```

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

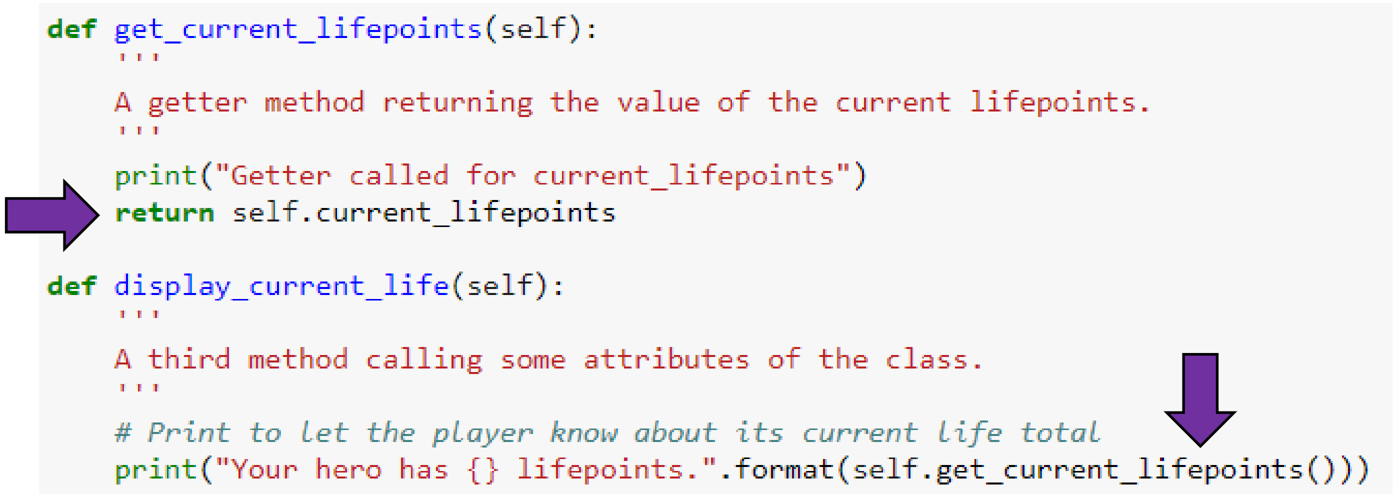
```
Getter called for current_lifepoints
Your hero has 100 lifepoints.
```

# 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.
- Our methods no longer work!



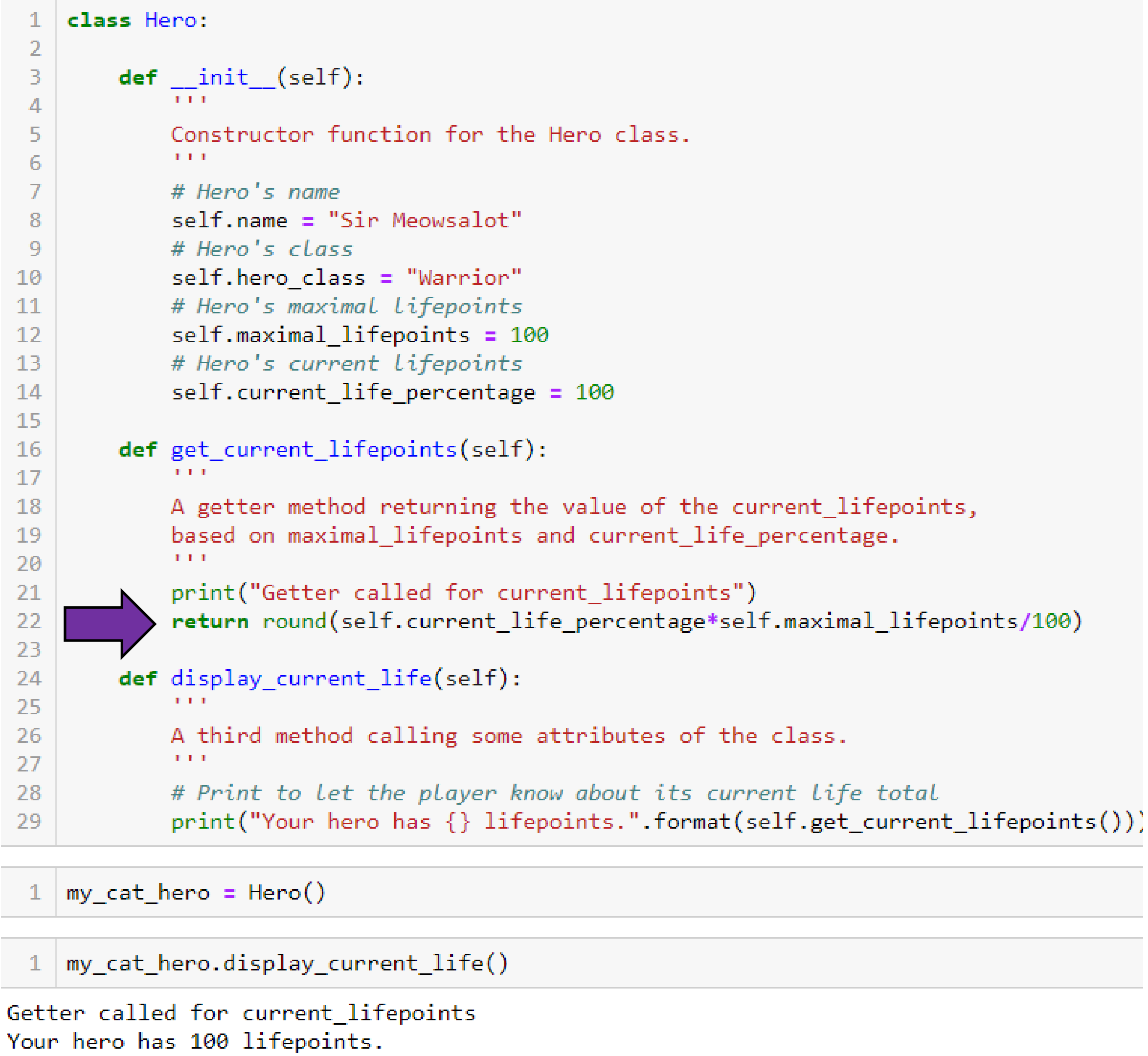
```python
class Hero:

    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.
- Our methods no longer work!
- But, only **one line** to change to make it work again!



```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()
```

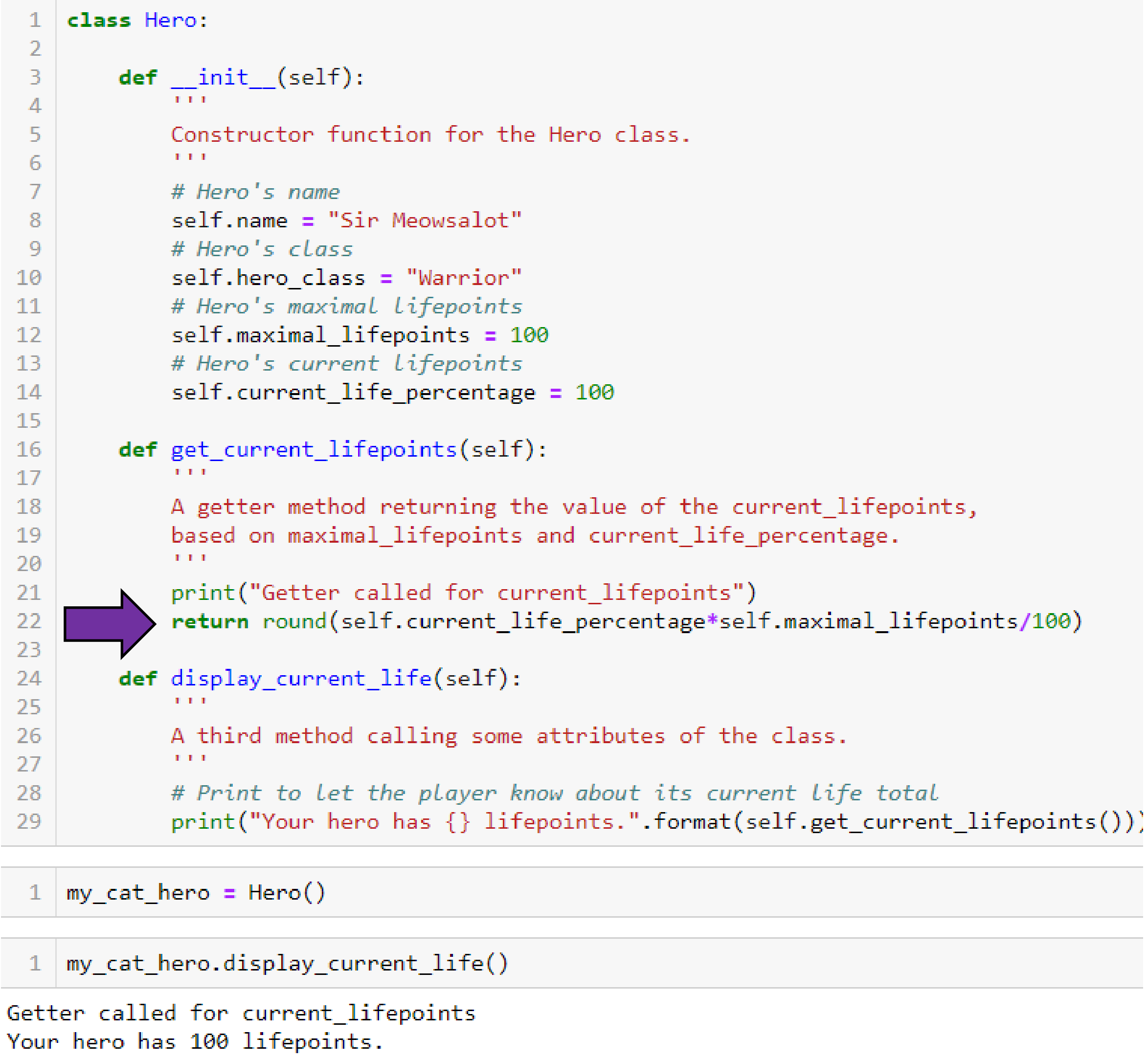
```python
my_cat_hero.display_current_life()
```

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

# Why is it good practice to use setters and getters?

**Lesson:** 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**.

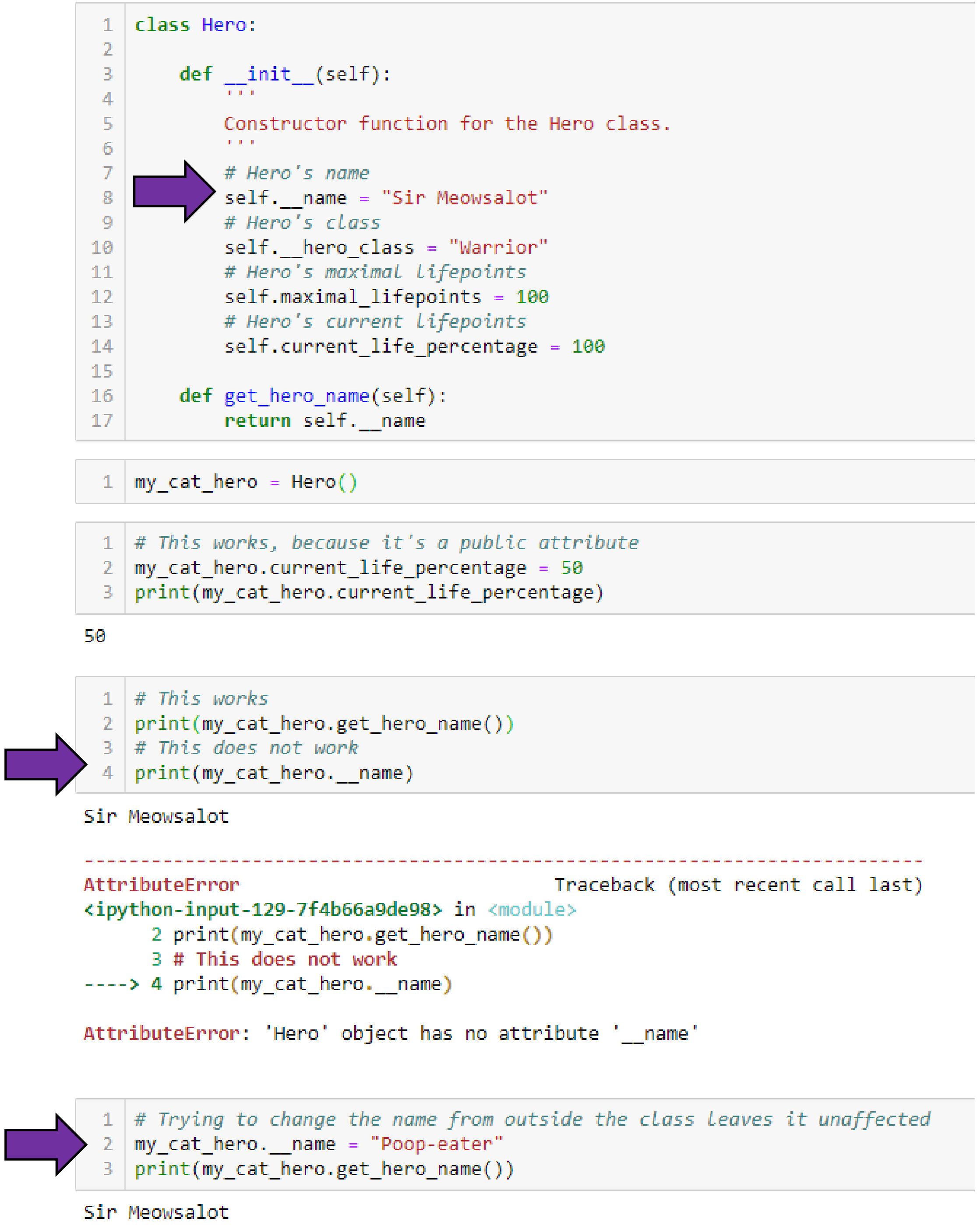
- **Public**: everyone can modify the attribute.
- **Protected:** public, but refrain from modifying it from outside the class.
- **Private**: only the functions called within my object can modify the attribute.

| 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 |

# 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'
```

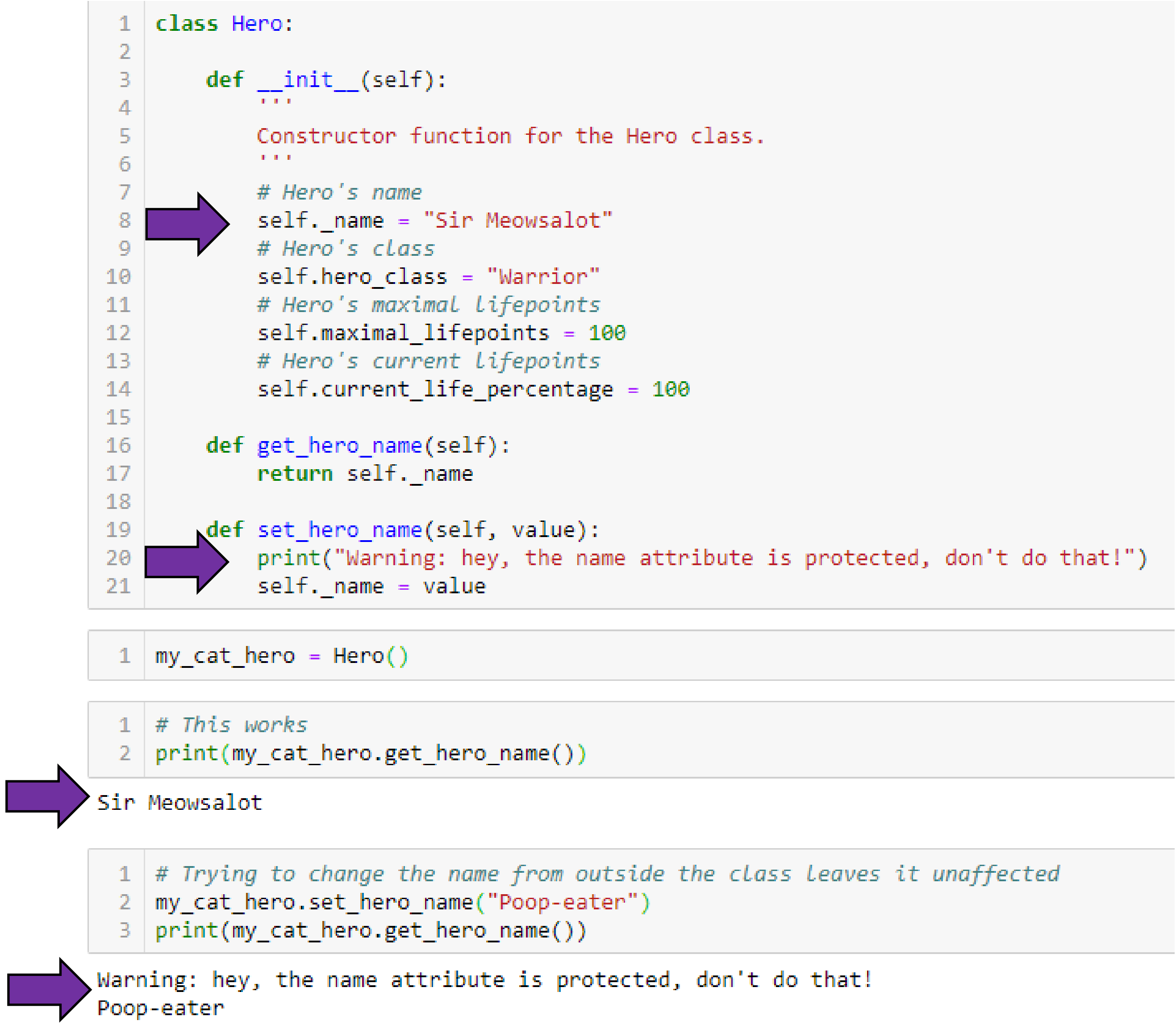
```python
# Trying to change the name from outside the class leaves it unaffected
my_cat_hero.__name = "Poop-eater"
print(my_cat_hero.get_hero_name())
```

```
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.
- More of an indication to other devs working in your team!



```
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
```

```
my_cat_hero = Hero()
```

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

```
Sir Meowsalot
```

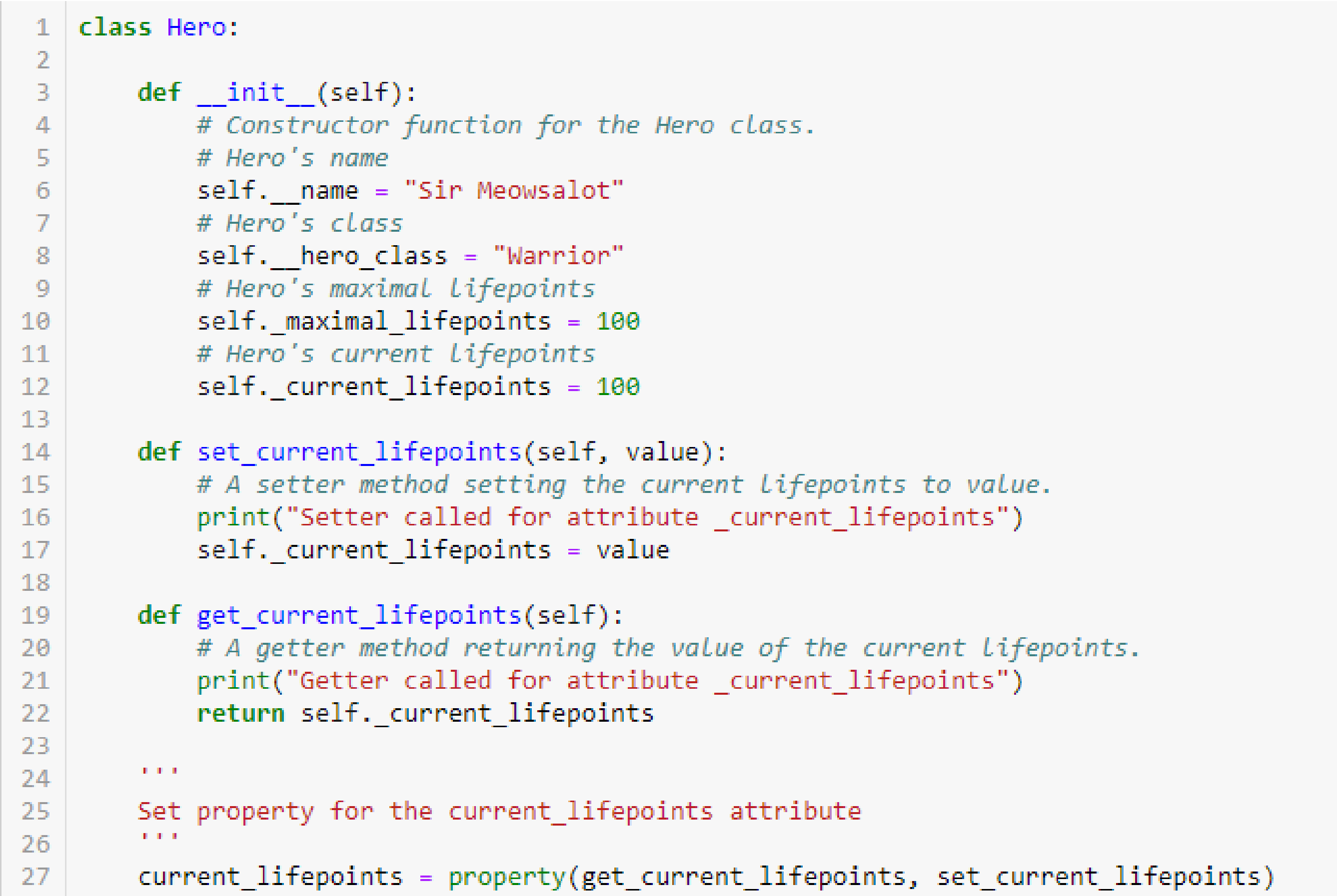
```
# Trying to change the name from outside the class leaves it unaffected
my_cat_hero.set_hero_name("Poop-eater")
print(my_cat_hero.get_hero_name())
```

```
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.

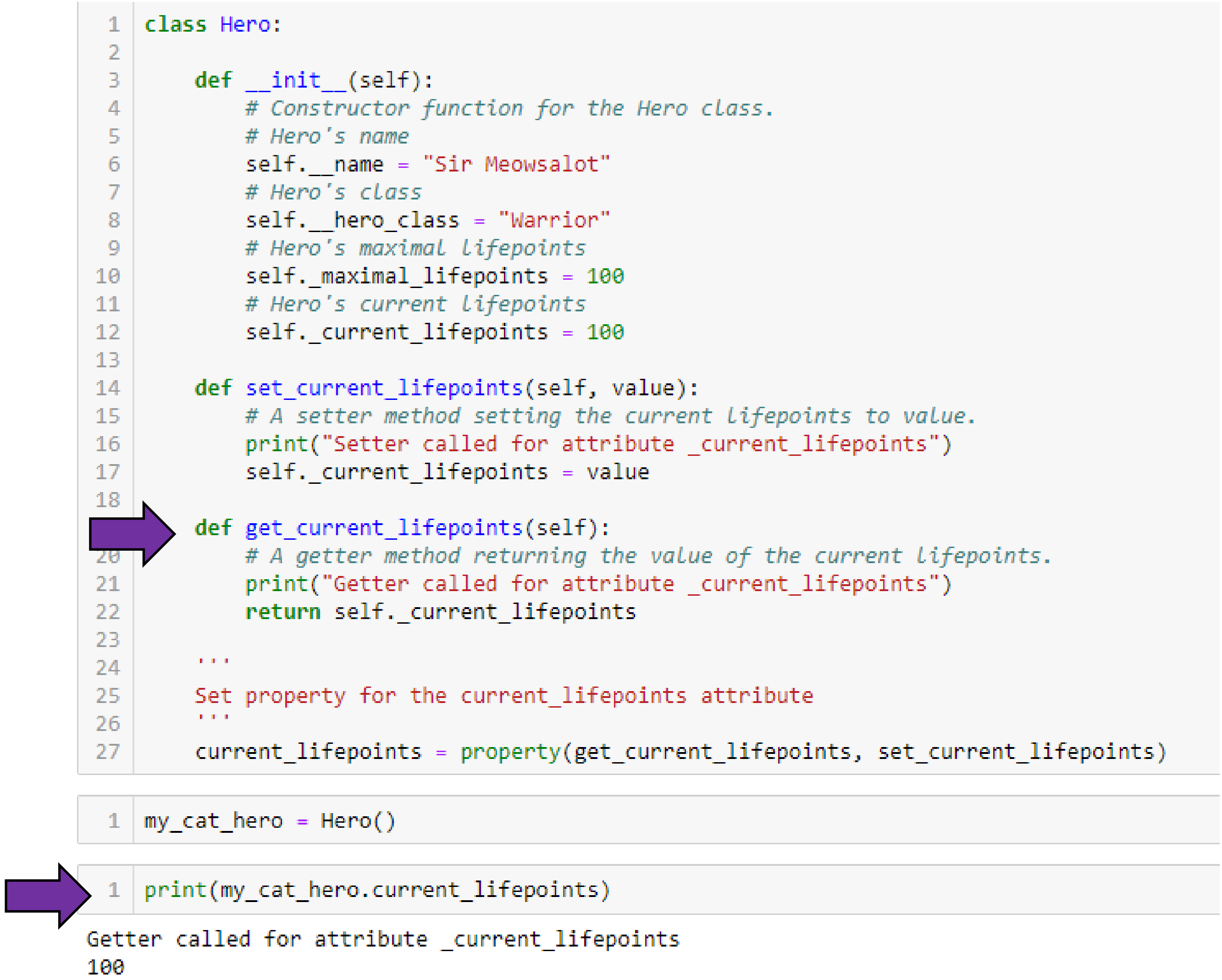


```python
class Hero:

    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 now automatically call the **getter** method, i.e.

***my_cat_hero.get_current_lifepoints()***



```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)
```

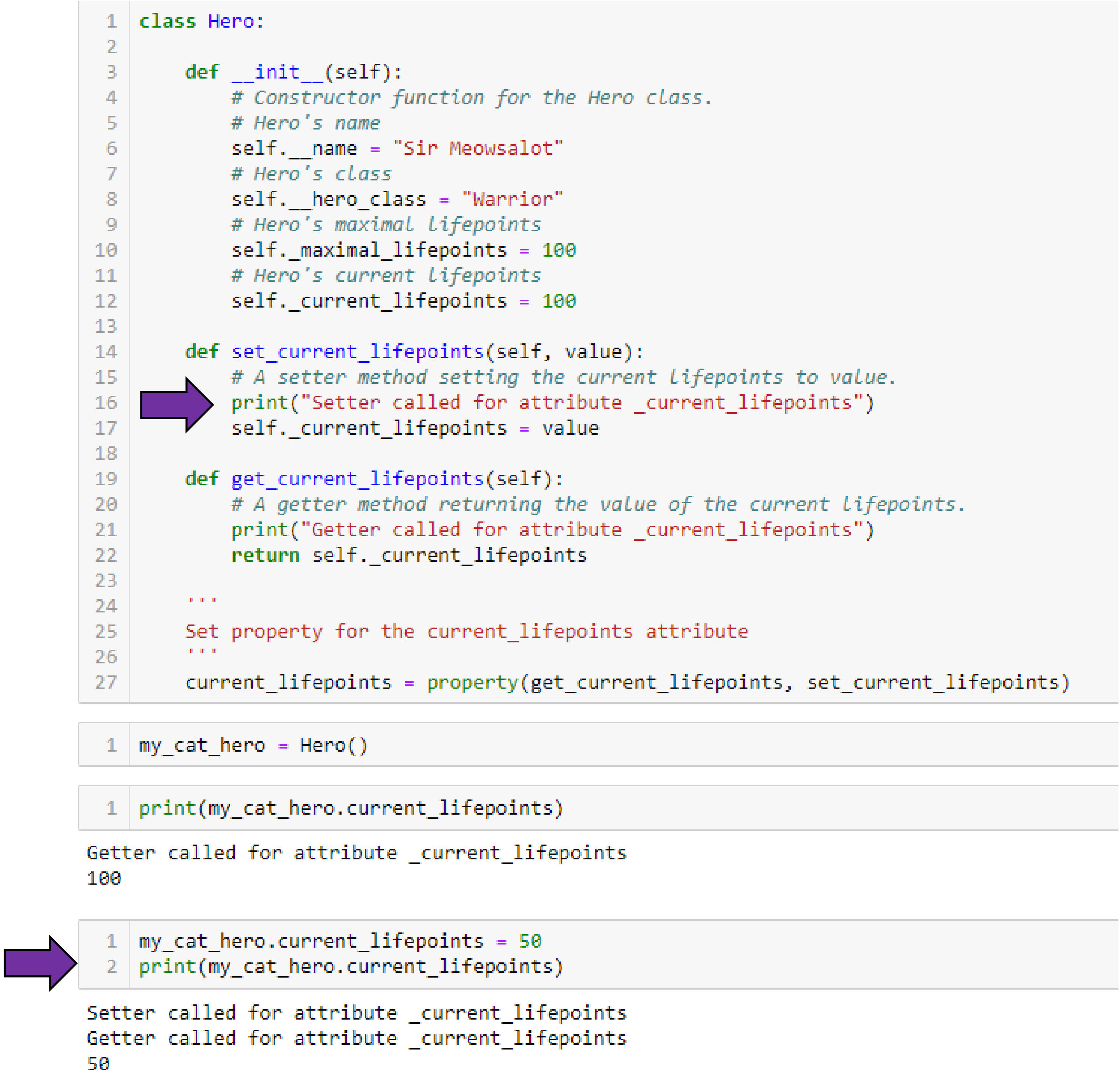
```python
my_cat_hero = Hero()
```

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

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

# 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.



```
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
```

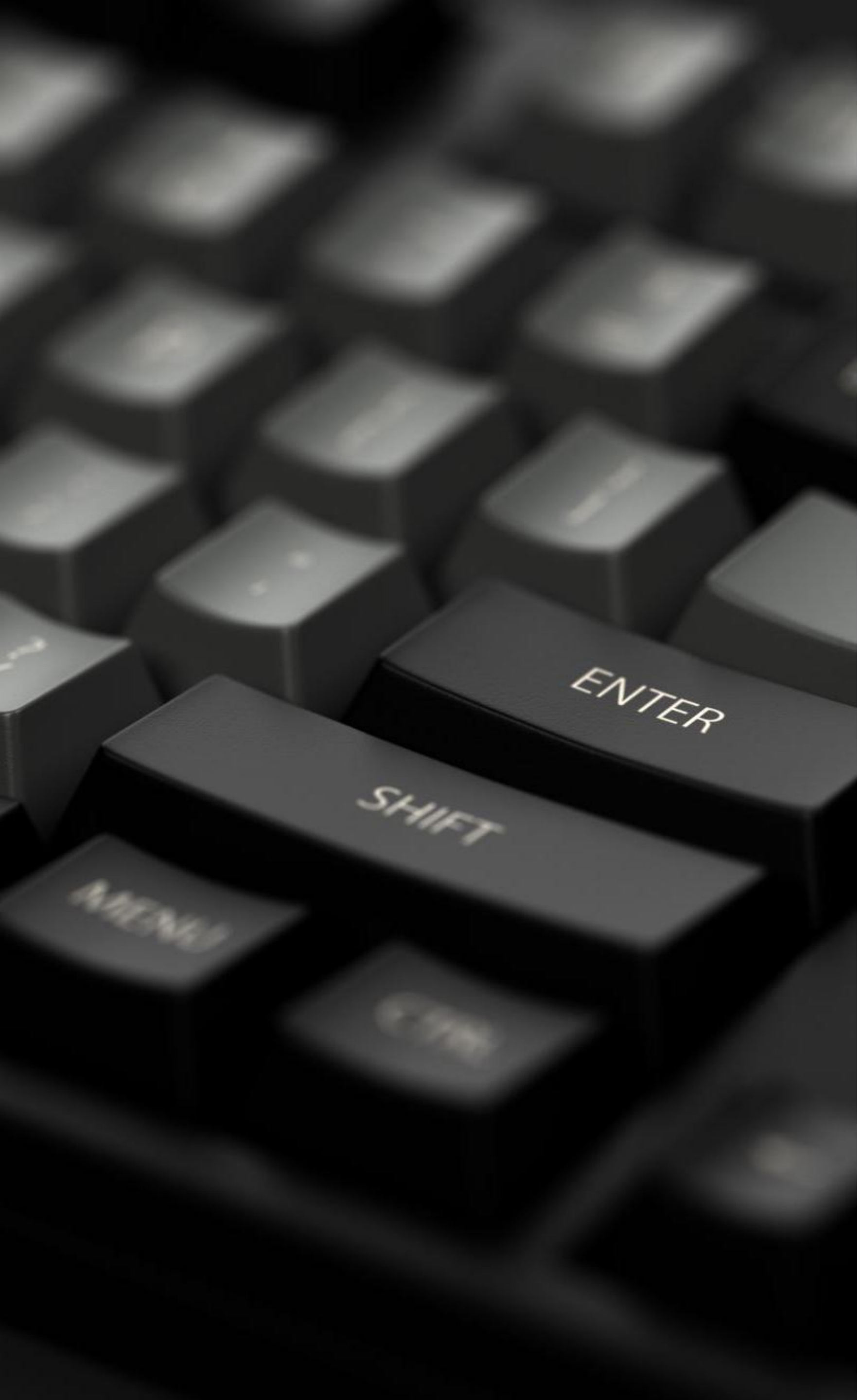
```
my_cat_hero.current_lifepoints = 50
print(my_cat_hero.current_lifepoints)
```

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

# Feel free to explore and play with OOP concepts!

They will bring your coding capabilities to great heights!

*(At this point, you should also start to recognize how your favorite games/apps have probably been coded!)*



# Conclusion (Chapter 15)

- What are **objects** and **classes**? How to they relate to **dictionaries**?
- What are **attributes** and **methods** in a class?
- What is the **__init__** constructor method?
- What is the **__dict__** special attribute?
- What are **special methods** in Python?
- What is the **“has-a” relationship** in OOP?
- *(What is the **“is-a” relationship** in OOP?)*
- *(What are the different **attributes privacies**?)*
- *(What are **setters**, **getters** and **properties**?)*