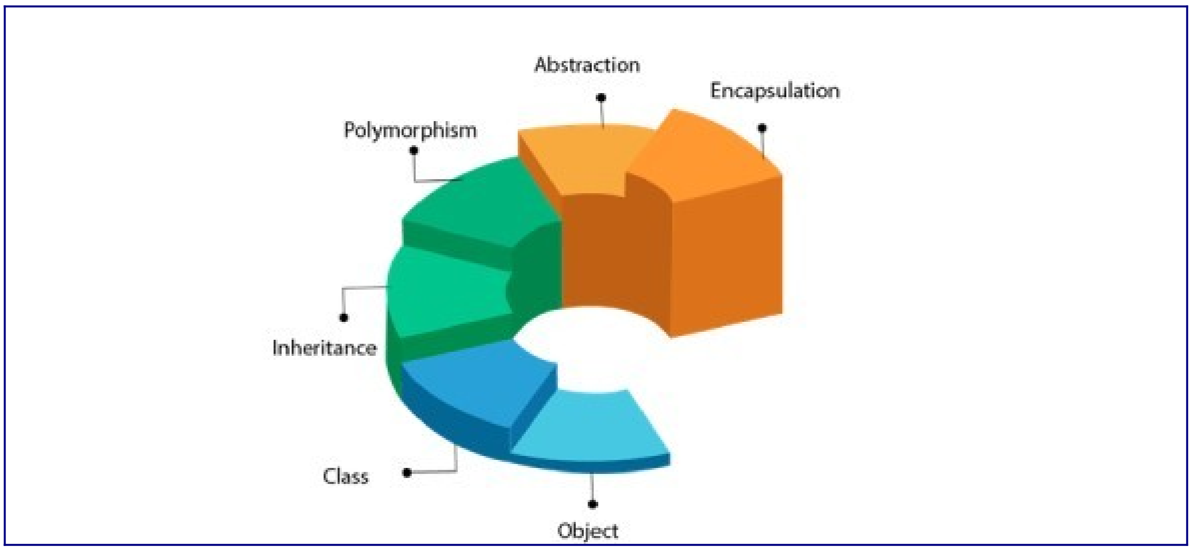
Object-Oriented Programming in Python



Intro

Once I've heard a phrase: if you want to comprehend something, teach it...so here I am almost finishing this tutorial about Object-Oriented Programming and put my hands on the next one. Data Science and machine learning fields are undoubtedly extensive and I am like you always learn something new day after day. From my perspective, OOP in Data Science isn't obligatory and you can benefit a lot with many different libraries (e.g. Scikit Learn / Keras and so forth). However, knowing OOP will give you more opportunities and flexibility in your feature projects. With the help of OOP, you will better understand relationships between objects and classes, concepts such as inheritance, encapsulation and polymorphism. Besides, frameworks such as PyTorch or Tensorflow will force you to face with OOP anyway.

I know, it may sound terrifying but only at first sight. This tutorial isn't going to be short because I like details. They are really important. For this purpose, don't be lazy, experiment with your code and test yourself. These are probably secrete ingredients for successful learning. Alright, enough talking and let's have a look at what we are going to cover.

Content

I do hope you'll find it both useful and not boring because learning must be interesting. Everything is ready to start. Let's set off. For better understanding, I'll be proving only key points of a chapter in a block called **Main Points of the Chapter**

1. What Is Object-Oriented Programming in Python

So what is OOP?

Object-Oriented Programming (OOP) is a paradigm where key elements are **objects** and **classes**.

- **Class**: simply an abstraction of something (e.g. a desk on which your laptop is laying is an object whereas a representation of all desks is a class)
- **Object**: is an object (e.g. my laptop, my phone or my bottle of water are objects)

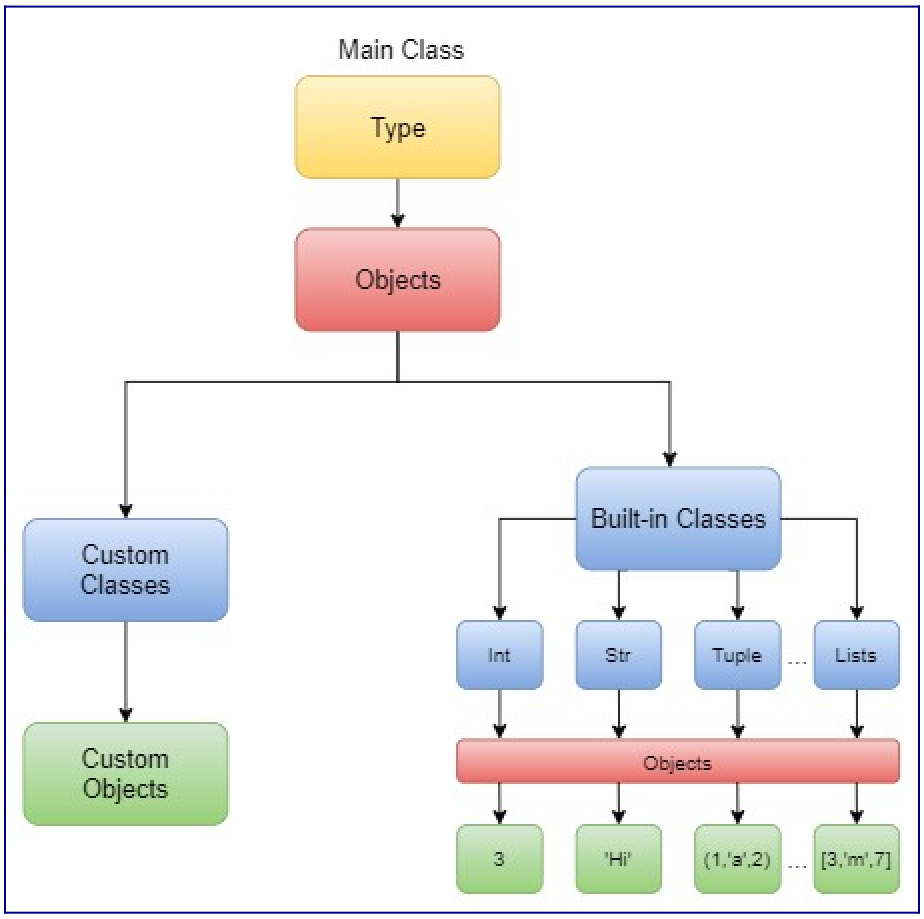
Cycles, conditions and functions are elements of structural programming that allows writing not complex programs. For advanced and complex systems using object-oriented programming is almost inevitable. Even not knowing OOP paradigm in Python we utilize objects and classes which hadn't been created by us.

1.1 Why Instance and not Object?

You may have seen that there are several names: **Instance and Object.** So..What is the difference between these definitions? Here is what I've found.

A Nitty - Gritty Detail:

All classes in Python belong to **one class** that's called **class type**. Thus, lists, tuples, strings and others are objects of **Type class.** In order to avoid mess up a word **instance** is more appropriate for newly created classes. However, these names are interchangeable and both can be used. For better understanding, I've created the following picture:



Above picture depicts that even classes such as Int, Float..Tuple are objects of the **main metaclass Type.** If it's difficult to understand, come back to this chapter later. Just keep in mind that **everything in Python is an object.** For more info, check the link below:

- Additional Reading about Meta Classes: https://realpython.com/python-metaclasses/

1.2 Difference Between Structural Programming and OOP?

I think we have to understand the difference between these two different concepts though it might be clear. Anyway, let it be here.

- **Structural Programming:** logic and sequence of actions are key elements.
- **Object-Oriented Programming:** A program like a system of interactive objects.

2. Class and Object Creation

There is nothing difficult in creating a class and its object.

- **Classes** are being created with a keyword **Class** (e.g. class class_name:)
- **Objects** are being created according to **the following syntax:** object_name = class_name()

Unlike functions, when a **class** is being called (invoked), it **creates an object instead of code execution**. Of course, there is no much sense in only reading, you have to practice. let's create our first class and its object.

Important

All the following examples will be based on my favourite computer game/book the Witcher. I recommend finding **your favourite field** (e.g. games, sport, films...). You will better understand the topic by implementing OOP on own examples instead of mine. However, you can first go step by step through my examples and then reimplement with yours.

Geralt from Rivia is one of the main characters of the game/book The Witcher. In OOP paradigm, Geralt is simply an object of a class and of course, there are many other characters. To be able to effectively create new characters (objects) we need a special class. let's name it **GameCharacter:**

In [22]:

```
### Create an empty class GameCharacter. Class name must start with a capital
letter!
class GameCharacter:
    pass

### Let's create an object of GameCharacter class. Geralt is an object of
GameCharacter class
geralt = GameCharacter()

### let's find out to which class our created character belongs to
print(type(geralt))
```

```
<class '__main__.GameCharacter'>
```

Not surprisingly, Geralt belongs to just created class GameCharacter.

2.1 Class Attributes, Methods and Fields. How to Distinguish Them?

The next thing which is highly important for a class is **attributes/fields/properties and methods**. At first sight, it may sound confusing. Let's sort everything out:

Each class is unique and has to contain its own **attributes and methods**:

- **Methods** are just **Functions**
- **Fields** are just **Variables** (another name for fields is **properties or attributes**. These names are interchangeable)

When it comes to attributes of a class you may think of this as certain properties. For example, start with yourself and try figuring out what properties you have (e.g. age, gender, hair color and so forth). Easy isn't it?

You have to be sort of a future teller to define all attributes for a class in advance. Some of them might be obvious and some not...but don't worry you can easily add new ones later. Just don't include attributes that you know can't exist for a class (e.g. a person can't have a tail but who knows)

In order to **access class attributes** we need an object of that class first and then call an attribute with the help of the following syntax: object_name.attribute_name (it's called **the dot notation**)

let's create Vesimir character (another witcher) with attributes (e.g. weight, hair color, height and name) and a method which will be simply printing characters' names.

In [23]:

```
class GameCharacter:

    # Frist define class attributes (properties)
    weight = 90
    hair_color = 'Grey'
    height = 180
    name = 'Vesimir'

    # Define a method (function) for printing characters' names
    def say():
        print(f'Hello, My Name is {GameCharacter.name}')

# Object Creation
vesimir = GameCharacter()

# Access object attributes using the dot notation
print("Vesimir's Name: " , vesimir.name)
print("Vesimir's Hair Color: ", vesimir.hair_color )
print("Vesimir is Saying: ", vesimir.say() )
```

```
Vesimir's Name:  Vesimir
Vesimir's Hair Color:  Grey

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-4833ae967556> in <module>
     17 print("Vesimir's Name: " , vesimir.name)
     18 print("Vesimir's Hair Color: ", vesimir.hair_color )
---> 19 print("Vesimir is Saying: ", vesimir.say() )

TypeError: say() takes 0 positional arguments but 1 was given
```

From Errors to Success!

Don't afraid of making mistakes. Make them! The more you make, the better and always experiment with the code. Try to think what will happen if you rewrite the code or change the logic. In other words, experiments are the best way for successful coding.

Why We Got the Error?

The message says that the method takes 0 parameters but we've provided one. How come? Let's dive deeper. The reason is that when we define an object, its attributes and methods can be found only in its class from which can be inherited **many objects**. Thus, when a method is being called, it must **take a certain object** as an argument which then will be processed. The logic may be described like that:

1. Look for say method of vesimir object -> can't find;
2. Look for the method in class GameCharacter -> find it;
3. Give the current object to the methods, in other words, say(vesimir);
4. But say methods doesn't take any parameters, thus an error occurs

As a linkage between objects and methods, a keyword **self** is used

In [24]:

```
# Fixing the error
class GameCharacter:

    weight = 90
    hair_color = 'Grey'
    height = 180
    name = 'Vesimir'

    # This time we provide self argument
    def say(self):
        print(f'Hello, My Name is {self.name}')

vesimir = GameCharacter()

print("Vesimir's Name: " , vesimir.name)
print("Vesimir's Hair Color: ", vesimir.hair_color)
print("Vesimir is Saying: ", vesimir.say())
```

```
Vesimir's Name:  Vesimir
Vesimir's Hair Color:  Grey
Hello, My Name is Vesimir
Vesimir is Saying:  None
```

Now everything looks fine. Each new object will be linked with a method with the help of **self keyword** and the error won't be raised anymore. Awesome!!!

By the way, do you know why we got Vesimir is Saying: None? Well, this is because the **function say()** doesn't return anything, **thus it returns None.**

2.2 Class Built-in Attributes and Methods

For each class, there are attributes and methods that had been predefined (built-in)

- **Built-in Attributes:**
    - __name__ - returns a class name;
    - __doc__ - returns description of a class (documentation);
    - __dict__ - returns a dictionary of local variables (attributes) for an object/class;
- **Built-in Functions:**
    - getattr(obj, 'name') - returns an attribute value of an object;
    - setattr(obj, 'name', value) - set a new value for an attribute;
    - delattr(obj, 'name') - deletes an attribute;
    - hasattr(obj, 'name') - checks if an object has an attribute;
    - dir(obj or a class) - returns a complete set of attributes for an object or a class;
    - isinstance(obj, class) - checks whether an object is an instance of a certain class

In [25]:

```
# Let's demonstrate built-in attributes
print('Name of the Class: ', GameCharacter.__name__)
print('Description of the Class: ', GameCharacter.__doc__)
print('All Local Variables of the Class: ', GameCharacter.__dict__)
print('All Local Variables of the Object: ', vesimir.__dict__)
```

```
Name of the Class:  GameCharacter
Description of the Class:  None
All Local Variables of the Class:  {'__module__': '__main__', 'weight': 90,
'hair_color': 'Grey', 'height': 180, 'name': 'Vesimir', 'say': <function
GameCharacter.say at 0x7f845a6115f0>, '__dict__': <attribute '__dict__' of
'GameCharacter' objects>, '__weakref__': <attribute '__weakref__' of
'GameCharacter' objects>, '__doc__': None}
All Local Variables of the Object:  {}
```

Here we can notice that vesimir object doesn't have any local attributes and this is true. Only GameCharacter class has local attributes. To define local variables for an object we have to either define a constructor (will be covered later) or assign object attributes explicitly. For example, below we've set a new attribute value **weight** and can delete only this attribute because other attributes exist only for the class. However, they can be accessed via the object. Keep it in mind!

In [26]:

```
# Let's demonstrate built-in methods
print("Vesimir's Height: ", getattr(vesimir, 'height'))

print("Setting a New Value For Vesimir's Weight: ", setattr(vesimir, 'weight',
85))
print("New Vesimir's Weight: ", getattr(vesimir, 'weight'))

print('Does Vesimir Has hair_color Attribute: ', hasattr(vesimir, 'hair_color'))
print('Deleting Attribute hair_color: ', delattr(vesimir, 'weight'))

print('Does Vesimir belong to GameCharacter Class: ', isinstance(vesimir,
GameCharacter))
```

```
Vesimir's Height:  180
Setting a New Value For Vesimir's Weight:  None
New Vesimir's Weight:  85
Does Vesimir Has hair_color Attribute:  True
Deleting Attribute hair_color:  None
Does Vesimir belong to GameCharacter Class:  True
```

2.3 Class Attributes Changing

Each attribute of a class can be easily changed. All we need is just access a certain attribute and provide a new value. Let's demonstrate that by changing the attribute **weight:**

In [27]:

```
# Suppose Vesimir's weight a week ago
print("Vesimir's Weight a Week Ago: ", vesimir.weight)

# Let's change weight value
vesimir.weight = 100
print("Current Vesimir's Weight: ", vesimir.weight)
```

```
Vesimir's Weight a Week Ago:  90
Current Vesimir's Weight:  100
```

2.4 Creating Attributes Outside the Class. Is It Worth Doing?

We can not only change current attribute values but also create new ones. However, it isn't recommended as introduces chaos in the system (i.e. objects of the same class will be different in terms of attributes)

In [28]:

```
# Let's create a new bool attribute: has_a_hourse
vesimir.has_a_hourse = True

print('Does Vesimir Has a Hourse: ', vesimir.has_a_hourse)
# We've just defined the new attribute though it hasn't been defined in a class 
previously
```

```
Does Vesimir Has a Hourse:  True
```

2.5 Main points of the chapter

- **Attributes** are variables;
- **Methods** are functions;
- Attributes and methods are called by using **the dot notation;**
- **self** argument is a linkage between methods and objects;
- If a method doesn't take in an object, it's a **static method;**
- Static methods are called by using a special decorator (9. Decorators) **@static method;**
- New attributes that haven't been defined in a class can be created. However, it leads to inconsistency.

3. Constructor and Destructor. Who Are They?

3.1 Constructor. Let's Start Building!

We are already familiar with the attributes of a class. We can build as many new objects as we want but here we are facing a problem. Although objects will belong to one class, they will be different in terms of attribute values. If we look at our previous implementation, we can notice that all the attributes are predefined and we can't change them when initializing an object. It's inconvenient and creates many problems. Any ideas?

Well, why not to define a special method in a class for changing/initializing the attributes. Sounds like a good idea. Let's implement that!

In [29]:

```
class GameCharacter:

    # Method for setting/initializing attribute values
    def set_attributes(self, name, weight):
        self.name = name
        self.weight = weight
```

```
# Creation of Vesimir and Geralt characters without any attributes
vesimir = GameCharacter()
geralt = GameCharacter()

# Setting attributes for Vesimir and Geralt
vesimir.set_attributes(name = 'Vesimir', weight = 100)
geralt.set_attributes(name = 'Geralt', weight = 90)

# Let's find out the attributes of the objects
print(f"Vesimir's Name: {vesimir.name}\nVesimir's Weight: {vesimir.weight}")
print('\n')
print(f"Geralt's Name: {geralt.name}\nGeralt's Weight: {geralt.weight}")
```

```
Vesimir's Name: Vesimir
Vesimir's Weight: 100


Geralt's Name: Geralt
Geralt's Weight: 90
```

Method **set_attributes()** enables setting attribute values for objects. Unfortunately, we have to **call the method every time we create a new object.** Luckily, in Python there is a special method on this case it's called **constructor**.

Constructor is a method called automatically when an object is being created. Besides, it's a **special method** and has the following syntax: **__init__(self, params)**

In Python **methods with underscores** belong to a **special group of methods** called **overloading or magic methods**. We don't have to call these methods explicitly, **they are called automatically** when an object takes part in some action (e.g. when an object is being created **__init__()** method is called and builds an object with predefined attributes)

Let's define a constructor for the class GameCharacter:

In [30]:

```
class GameCharacter:

    # Create the constructor of the class. For simplicity,
    # let's leave only name and hair_color attributes
    def __init__(self, name, hair_color):
        self.name = name
        self.hair_color = hair_color

# Above we've created the constructor method.
# It will be called automatically when an object is being created.
# The last thing to do is only enumerate arguments when creating an object

# Let's create the main game characters with unique attributes
vesimir = GameCharacter(name = 'Vesimir', hair_color = 'Grey')
geralt = GameCharacter(name = 'Geralt', hair_color = 'White')
ciri = GameCharacter(name = 'Ciri', hair_color = 'White')

print("Ciri's Hair Color: ", ciri.hair_color)
print("Geralt's Name: ", geralt.name)
print("Vesimir's Hair Color: ", vesimir.hair_color)
```

```
Ciri's Hair Color:  White
```

```
Geralt's Name:  Geralt
Vesimir's Hair Color:  Grey
```

We've successfully created 3 main game characters: Ciri, Geralt and Vesimir. We not only created them but also defined their attributes. Rember that you can define **default arguments in a method.** In this case, you have to pass only obligatory arguments in a method, **default values can be omitted.**

In [31]:

```
# Let's experiment. Let's say on average all game characters have height = 179
# Set them by default
class GameCharacter:

    # Can notice that the argument height has a default value
    def __init__(self, name, hair_color, height = 179):
        self.name = name
        self.hair_color = hair_color
        self.height = height

# Now when creating an object we don't have to provide the height attribute.
# It will be set to its default value
vesimir = GameCharacter(name = 'Vesimir', hair_color = 'Grey')
print("Vesimir's Height: ", vesimir.height)
```

```
Vesimir's Height:  179
```

But here I must say a few words. In fact, **__init__()** is not a constructor of a class, it just initializes created objects.

Objects are being created with the method __new__()

For more info, check the following article: https://spyhce.com/blog/understanding-new-and-init

3.2 Destructor. Who is Going to be Destructed?

It's obvious that if we can create then we can destruct as well. In Python, there is a special method for this purpose.

__del__() is a special method and it's responsible for deleting the objects (i.e. it's a destructor of a class)

In [32]:

```
# Let's demonstrate constructor and destructor methods in action
class GameCharacter:

    # Define the constructor of the class
    def __init__(self, name, hair_color, height = 179):
        self.name = name
        self.hair_color = hair_color
        self.height = height

    # Define the destructor of the class
    def __del__(self):
        print(f'Game Character {self.name} Was Deleted')

geralt = GameCharacter(name = 'Geralt', hair_color = 'White', height = 180)
vesimir = GameCharacter(name = 'Vesimir', hair_color = 'Grey', height = 180)

# Let's delete Vesimir
del vesimir
```

```
Game Character Vesimir Was Deleted
```

We've defined the constructor and the destructor methods of the class. As we already know, methods with underscores are special and they are called automatically when an object takes part in a certain operation (e.g. object creation - **__init__()**, object deletion - **__del__()**).

Above we've deleted game character Vesimir and can't access the object any more. Have a look:

In [33]:

```
# Geralt still exists
geralt.hair_color
```

Out[33]:

```
'White'
```

In [34]:

```
# However, Vesimir doesn't exist any more.
# If we try to call vesimir object, an error will be raised
vesimir
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-34-eefa797665c4> in <module>
      1 # However, Vesimir doesn't exist any more.
      2 # If we try to call vesimir object, an error will be raised
----> 3 vesimir

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

3.3 Attributes Creation Control

We already know that new attributes can be created though we haven't defined them explicitly in a class.

Is it possible to control which exactly attributes can be created in a class?

Yes! This is where **__slots__** attribute comes into play. It checks attributes and if some attributes haven't been defined in **__slots__**, an error is raised and new attributes aren't created. As a result, attributes in a class will be more consistent.

Let's have a look at the following example:

In [35]:

```
# Let's deprecate creating new attributes that aren't defined in a class
class GameCharacter:

    # Here we allow only certain attributes to be created
    __slots__ = ('name', 'hair_color', 'height')

    # Constructor
    def __init__(self, name, hair_color, height):
        self.name = name
        self.hair_color = hair_color
        self.height = height

# Let's call allowed attribute
geralt = GameCharacter(name = 'Geralt', hair_color = 'White', height = 180)
geralt.hair_color
```

```
Game Character Geralt Was Deleted
```

Out[35]:

```
'White'
```

__slots__ checks whether new attributes are allowed or not. If a certain attribute is allowed, an attribute is created, otherwise not. Have a look:

In [36]:

```
# New attribute creation
geralt.has_sword = True
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-36-fb5e8cdb0244> in <module>
      1 # New attribute creation
----> 2 geralt.has_sword = True

AttributeError: 'GameCharacter' object has no attribute 'has_sword'
```

Besides, I'd like to point out one point. Sometimes you may see the following constructor:

In [37]:

```
class GameCharacter:

    __slots__ = ('name', 'hair_color', 'height')

    # We can define expected values in the constructor (the notations)
    def __init__(self, name:str = 'geralt', hair_color:str = 'white', height:int
= 180):
        self.name = name
        self.hair_color = hair_color
        self.height = height
```

When you see something like that then it's called the **notations**. Notations tell which types and values a constructor is expected to get. It doesn't mean that it's impossible to provide other types, not at all. It just tells what types we should provide.

3.4 Main points of the chapter

- **Constructor** is a method which is called automatically when an object is being created (e.g. **__init__()**);
- **Destructor** is a method which is called automatically when an object is being deleted (e.g. **__del__()**);
- self.name, self.weight... are **attributes/fields** whereas name, weight, heigh... are **parameters;**
- Having **default parameters** in any methods, make sure you follow the order: **non - default parameters first, then default parameters;**
- **__init()__** has to take in **self parameter;**
- Methods with underscores are special. They are called **methods of operator overloading or magic methods;**
- **Magic methods** are called automatically when an object takes part in a certain operation (e.g. addition __add__());
- **Methods with the same name** override each other;
- **Constructor** allows predefining attributes of objects;
- **__slots__ field** allows only certain attributes to be created

4. Class and Object Attributes. Scope of Variables

Before we dive deeper we have to grasp that there is a difference between the class and object attributes. Moreover, attributes have a **different access type and can be local and global.**

4.1 Accessing Class and Object Attributes

First of all, you may ask: **How to distinguish class and object attributes?**

Well, the answer is simple. All variables that are defined **inside the methods are object attributes** and all the rest **(outside the methods) is class attributes.**

Usually, class attributes are placed right after the class name (at the top) and shared by all objects.

In [38]:

```
class GameCharacter:

    # This attribute will belong to the class.
    class_name = 'Game Character'

    # All attributes inside this method will belong to objects
    def __init__(self, name, hair_color, height):
        self.name = name
        self.hair_color = hair_color
        self.height = height

geralt = GameCharacter(name = 'Geralt', hair_color = 'White', height = 180)
```

- **Class attributes** can be accessed **via an object or a class** (when there aren't any objects yet)
- **Object attributes** can be accessed **only via an object**

In [39]:

```
# Accessing the class attribute via the class and the object
print(GameCharacter.class_name, geralt.class_name)
```

```
Game Character Game Character
```

In [40]:

```
# Accessing object attributes via the class is impossible, only via the object
print(GameCharacter.height)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-40-c4965cb4e398> in <module>
      1 # Accessing object attributes via the class is impossible, only via the
object
----> 2 print(GameCharacter.height)

AttributeError: type object 'GameCharacter' has no attribute 'height'
```

4.2 Local Variables

Local variables in a class are variables that defined inside methods. They exist only there and can't be used outside those methods. In the above code, variables such as **name, hair_color and height are local.** We can't access them using a class name.

In [41]:

```
# Accesssing the local variable
GameCharacter.hair_color
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-41-2cd2414baa63> in <module>
      1 # Accesssing the local variable
----> 2 GameCharacter.hair_color

AttributeError: type object 'GameCharacter' has no attribute 'hair_color'
```

4.3 Global Variables

Global variables aren't defined in code blocks (e.g. functions, statements and so forth) and can be accessed by using a class or an object.

In [42]:

```
# Accesssing the global variable by using a class and an object
print(GameCharacter.class_name, geralt.class_name)
```

```
Game Character Game Character
```

Important

Although **Global/Local variables** and **class/object attributes** are looking similar, they **differ in the way they are accessed.**

For **global/local variables** the most important thing is the **place where they can be accessed**, whereas for **class/object attribute** the most important is **how they are accessed (by using a class/object name).** Hope you understand the difference.

4.4 Main Points of the Chapter

- **Class attributes** are defined **outside methods** and can be accessed **via class or object;**
- **Object attributes** are defined **inside methods** and can be accessed only **via an object;**
- **Local variables/attributes** are defined **in methods or code blocks;**
- **Global variables/attributes** are defined **outside methods or code blocks**

5. Inheritance, Polymorphism and Encapsulation

Let's deal with this, at first sight, spooky definitions one by one starting from inheritance.

5.1 Inheritance

This is just an abstraction and the sense is pretty straightforward. We've already faced with inheritance when were dealing with objects creation. An object inherits attributes and methods of its class when is being created. Taking this fact into account we can say that a new class can also inherit methods and attributes of an already existing class. This is inheritance in OOP.

Alright, but why do we need that?

Imagine that we have to create thousands or even millions of different game characters. Common sense guides us that all they will have something in common, plus something unique. Rewriting the same code millions of times is tedious and not a good idea, we need a better solution...and here inheritance comes into play. All we have to do is just define the main class (it's called **parental class**) with the main attributes and inherit as many classes as we want (they are called **child classes**). Child classes will have not only own unique attributes and methods but also attributes and methods from parental class or classes (yes, a child class may have not only one parental class but many). In other words, inheritance provides code flexibility and consistency.

To inherit a class or several classes, provide the following syntax:

new_class_name (parental_class_1, parental_class_2, ..., parental_class_n)

Now, let's practice on examples. I'm coming back to my favourite computer game.

In the game Geralt, Ciri and Vesimir are the Witchers. There are many other different game characters (e.g. monsters, villagers, knights and so forth). Although they all are game characters, they have some unique attributes (e.g. the witchers and monsters have extraordinary abilities whereas villagers certain features of a game location (clothes, type of voice and so on))

I hope you got the idea.

In [43]:

```
# Again define the main GameCharacter Class (Parental Class)
class GameCharacter:

    # Define the constructor
    def __init__(self, name, hair_color, height):
        self.name = name
        self.hair_color = hair_color
        self.height = height

    # Define 2 main methods for greeting and saying good bye
    def greeting(self):
        return f'I am glad to see you, my name is {self.name}'

    def farewell(self):
        return 'Farewell'

# Define Witcher Class (child class)
# Let's create a method signs_ability that prints which signs the Witcher has
class Witcher(GameCharacter):

    # No need for constructor because it is being inherited from the parental
class.
    # Define a unique method. Only the witchers have sign abilities
    def signs_ability(self):
        signs = ['Axii','Qven','Ignii','Aard','Yrden']
        return signs

# Create a new object
geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180)

# Now Geralt belongs to Witcher Class as well as GameCharacter class
# Let's get info about signs
print(geralt.signs_ability())

# Accessing other attributes from the parental class
print("Geralt's Name: ", geralt.name)
print(geralt.greeting())
```

```
['Axii', 'Qven', 'Ignii', 'Aard', 'Yrden']
Geralt's Name:  Geralt
I am glad to see you, my name is Geralt
```

From the above code, we can see that **Witcher class** has been inherited from **GameCharacter class** and we didn't have to rewrite all the previous attributes. We just added several lines of code and initialized a new object with the unique method **signs_ability()** because in the game only the Witchers have extraordinary abilities with signs. To make sure that a class is a subclass of another class, use the function **issubclass()**

In [44]:

```
print('Is Withcer Class Subclass of GameCharacter Class: ', issubclass(Witcher,
GameCharacter))

Is Withcer Class Subclass of GameCharacter Class:  True
```

5.1.1 Constructor Extension

Previously, we haven't provided **__init__()** method for **Witcher class.** In this case, Witcher class will be looking for the constructor in the parental class and will ultimately find it. We can create **__init__()** for Witcher class as well. In this case, **__init__()** from Witcher class will override GameCharacter **__init__()**. In the game, the Witchers belong to different schools (e.g. Cat School, Wolf School, Bear School and others). Let's add this unique attribute in Witcher class.

In [45]:

```
# Let's create the constructor for Witcher class
class Witcher(GameCharacter):

    # Geralt belongs to Wolf School
    def __init__(self, witcher_school = 'Wolf'):
        self.witcher_school = witcher_school

# Access the new attribute value
geralt = Witcher()
geralt.witcher_school
```

Out[45]:

```
'Wolf'
```

In the above code **__init__()** from GameCharacter class has been overridden and we can't access attributes from GameCharacter class anymore.

In [46]:

```
geralt.name

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-46-85dac99ee49a> in <module>
----> 1 geralt.name

AttributeError: 'Witcher' object has no attribute 'name'
```

However, most of the time we don't want to rewrite the parental constructor, we want to extend it!

For this purpose, we just need to **call the parental constructor and then extend it** with new fields:

In [47]:

```
class Witcher(GameCharacter):
    # As this constructor will be extended we have to provide all previous
parameters
    # Extend constructor for Witcher class. Make __init__() and enumerate all
fields
    def __init__(self, name, hair_color, height, witcher_school = 'Wolf'):
        # Calling the parental constructor
        GameCharacter.__init__(self, name, hair_color, height)
        # Extend it with a new attribute
        self.witcher_school = witcher_school

# Now we have to provide not only one but all arguments because the constructor
of Witcher class has been extended by GameCharacter constructor
geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180)

# Can access not only unique but also attributes from parental constructor.
# Thanks to the constructor extension!
print("Witcher School : ", geralt.witcher_school)
print("Geralt's Hair Color: ", geralt.hair_color)
```

```
Witcher School :  Wolf
Geralt's Hair Color:  White
```

Important note

Above **child constructor** has been extended by parental constructor with the help of the following syntax: **class_name.__init__()**

However, this option isn't reliable because when it comes to **multiple inheritance** the order of which parental classes are called is highly important. Wrong order introduces chaos and errors. To avoid this, use reliable construction: **super().__init__()**. The **function super()** will go through all parental classes in the right order (the function uses special algorithm inside)

5.1.2 Method Extension

Well, a constructor is a method like any other methods...just a bit special due to underscores (magic method). Previously, we've already extended it so that we could get access to parental attributes. Thus, we can make extensions on other functions as well. In other words, we can call parental methods in child methods and then process their results (i.e. call function inside a function)

For example, I'd like to change the greeting method in Witcher class. I don't want to change it much, just add a phrase that current character is the Witcher.

In [48]:

```
class Witcher(GameCharacter):

    def __init__(self, name, hair_color, height, witcher_school = 'Wolf'):
        # use super() this time
        super().__init__(name, hair_color, height)
        self.witcher_school = witcher_school

    # This method will be extended.
    # It means that firstly it executes greeting method from GameCharacter class
    # Then the rest code
    def greeting(self):
        # Call greeting method form Parental Class (Method Extension)
        parental_res = super().greeting()
        # Continue executing the rest of the code
        print(f'{parental_res}\nI am the Withcer')

geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180)
geralt.greeting()
```

```
I am glad to see you, my name is Geralt
I am the Withcer
```

Splendidly, the method **greeting()** from Witcher class has been extended!

5.2 Polymorphism or Method Overriding

The first thing which comes up in your mind when you hear the word polymorphism is probably something that has many different forms or shapes. **Polymorphism is just an abstract term.** Basically, **polymorphism in OOP is just methods with the same names but absolutely different logic.**

Surprisingly but we've already seen polymorphism in action (e.g. **__init__()** method).

Be careful, **polymorphism of a function and class methods are different terms** (polymorphism of a function isn't related to polymorphism in OOP)

Let's implement polymorphism. For this purpose, simply create the method **greeting()** (the same method name but different logic). We know that the method already exists in **GameCharacter class** and defining the same method name in **Witcher class** will override the method, this is exactly what we need.

In [49]:

```
# Write familiar code once again
class Witcher(GameCharacter):

    # Constructor of the class
    def __init__(self, name, hair_color, height, witcher_school = 'Wolf'):
        super().__init__(name, hair_color, height)
        self.witcher_school = witcher_school

    # Polymorphism in action.
    def greeting(self):
        return 'I am the Witcher and I am Looking for the Monsters!'

geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180)
geralt.greeting()
```

Out[49]:

```
'I am the Witcher and I am Looking for the Monsters!'
```

From the above code, we can notice that the execution of the method **greeting()** leads to a different result. This is due to the fact that the **parental method has been overridden** by the child method (polymorphism)

5.3 Encapsulation

We are facing one more abstraction which is called **encapsulation.** As you may guess encapsulation means something that we want to **hide and protect.** Classes can be huge and complex and inside can be many auxiliary attributes and methods which must not be used outside a certain class. They are sort of small elements that provide stability for a class. Thus, we may want to protect stability elements of a class and not allow changing them in a usual way. But before we have to grasp **access modifiers** and how they differ because it's the **core of the encapsulation mechanism.**

5.3.1 Access Modifiers

Access modifiers are used to modify the scope of attributes in a class. There are **3 main types:**

- **Public:** attribute_name (can be accessed anywhere);
- **Protected:** _attribute_name (can be accessed only in the class as well as from all child classes);
- **Private:** __attribute_name (can be accessed only in that class in which has been defined)

Why do we need them?

Some attributes might be valuable for a class and we may want to prevent them from changing or accessing outside the class. Have a look:

In [50]:

```
# Let's access the attribute name from GameCharacter class and change it
geralt.name = 100
```

We definitely don't want that. **Numbers are inappropriate for names.** To prevent this situation we have to apply **access modifiers.**

In [51]:

```
# For demonstration, define all three type of access modifiers in the class
class GameCharacter:

    def __init__(self, name, hair_color, height):
        self.__name = name # private
        self._hair_color = hair_color # protected
        self.height = height # public

geralt = GameCharacter(name = 'Geralt', hair_color = 'White', height = 180)

# We can only access public and protected attributes via the object
# They are actually idenctical, read below why
print(geralt._hair_color, geralt.height)

# Accessing the private attributes raises an error
geralt.__name
```

```
White 180

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-51-19401113736e> in <module>
```

```
     14 
     15 # Accessing the private attributes raises an error
---> 16 geralt.__name

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

The above error tells us that the attribute **__name** doesn't exist. However, it does..we just hid it and it isn't available outside the class.

You may say: if we can access **attributes height and _hair_color** and change both of them what is the difference then? Well, this is my answer: the access modifier with a single underscore only **indicates or tells** that it's protected and can be used only in a certain class and its child classes. In other words, **protected attributes just don't exist in Python.**

In fact, **protected and public attributes are identical.** Single underscore only tells that the attribute protected and we shouldn't use it outside the class, otherwise it may lead to unpredictable errors because it's assumed not to be touched. If it's still difficult to grasp, here is a sort of a rule of thumb: if you see attributes with a single underscore, don't call them directly because they are inner service variables.

Keep in mind that **not only attributes can have different access modifiers but also methods!**

Private/protected methods are used to provide functionality inside the class. Don't touch them and don't try to call outside the class!!!

To better understand the topic let's cover one more example. For example, we may want to keep track of how many game characters we've already created. It may be really important because we may want to create an only certain number of characters for a certain location of the game. A wrong counter will introduce chaos and uncertainty. Let's create a private (encapsulated) class attribute **__counter.**

In [52]:

```
class GameCharacter:

    # New private class attribute
    __counter = 0

    # Make all object attribute private
    def __init__(self, name, hair_color, height):
        self.__name = name
        self.__hair_color = hair_color
        self.__height = height

        # New game character creation will be leading to an increase in counter
        # We use a specila syntax to access class attribute via an object
        self.__class__.__counter += 1

    # Decrease the counter if delete a game character
    def __del__(self):
        self.__class__.__counter -= 1

geralt = GameCharacter(name = 'Geralt', hair_color = 'White', height = 180)
ciri = GameCharacter(name = 'Ciri', hair_color = 'White', height = 180)
```

Everything seems to be working! However, we **can't access private attributes** not to mention changing them.

How can we solve this problem?

I must say that there are several solutions. One of them is to use a **special syntax:**

- obj.__class__.attribute/method_name;
- obj._class_name__attribute/method name;
- class_name._class_name__counter

In [53]:

```
# Special syntax allows not only getting but also changing private attributes
geralt._GameCharacter__name
```

Out[53]:

```
'Geralt'
```

However, it's not recommended and a **double underscore** must indicate the developers that they must work with this attribute only using **special methods** called **getters, setters and deleters**. We must remember what we can do with object attributes:

- Getting an attribute value: obj.field_name;
- Setting an attribute value: obj.field_name = new_value;
- Attribute deleting: del obj.field_name

5.3.2 Calling Private Attributes From Child Classes

Let's figure out how behave private attributes during inheritance. In GameCharacter class we've defined a private attribute **__counter.** To be able to access this attribute let's create a special method called **get_counter().** Then create a child class Witcher and try calling the private attribute

In [54]:

```
class GameCharacter:

    __counter = 0

    def __init__(self, name, hair_color, height):
        self.__name = name
        self.__hair_color = hair_color
        self.__height = height
        self.__class__.__counter += 1

    def __del__(self):
        self.__class__.__counter -= 1

    # For accessing the private attribute
    def get_counter(self):
        return self.__counter
```

```
class Witcher(GameCharacter):

    def __init__(self, name, hair_color, height, witcher_school = 'Wolf'):
        super().__init__(name, hair_color, height)
        self.witcher_school = witcher_school

    def greeting(self):
        return 'I am the Witcher and I am Looking for the Monsters!'

geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180)

# let's try calling __counter via the child object
geralt.__counter
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-54-fc879ce4e30e> in <module>
     28 
     29 # let's try calling __counter via the child object
---> 30 geralt.__counter

AttributeError: 'Witcher' object has no attribute '__counter'
```

Above error says that private attribute **__counter** has become unavailable and it's impossible to call the attribute from a child class. It's happening because once a private attribute has been defined, its name changes and has a prefix of a class to which belong. In our case, it belongs to the class GameCharacter, thus its name must have a prefix of that class.

Let's find out local attributes of Witcher class.

In [ ]:

```
Witcher.__dict__
```

It can be clearly seen that the name of private attribute **__counter** has changed to **_GameCharacter__counter.** Because of this, we can't access this attribute from Witcher class, the attribute has another prefix!

We can access this attribute only using the method which has been inherited **get_counter()**

In [55]:

```
geralt.get_counter()
```

Out[55]:

```
1
```

Everything is working. Please remember, when it comes to private attributes they can be called only via **get()** methods.

5.3.3 Getting, Setting and Deleting Encapsulated Attributes</a></a>

To call encapsulated attributes we have to just define methods such as **get/set/delete** in the class. In addition, with the help of the method **set** we can control which types are allowed as attribute values. First, let's implement these methods for the **__counter** attribute.

In [56]:

```
class GameCharacter:

    __counter = 0

    def __init__(self, name, hair_color, height):
        self.__name = name
        self.__hair_color = hair_color
        self.__height = height

        self.__class__.__counter += 1

    def __del__(self):
        self.__class__.__counter -= 1

    # Define the getter
    def get_counter(self):
        return self.__class__.__counter

    # Define the setter
    def set_counter(self, new_value):
        # Only int is allowed
        if isinstance(new_value, int):
            self.__class__.__counter = new_value
        else:
            raise TypeError('Counter Must Be Int!')

    # Define the deleter
    def drop_counter(self):
        print('Counter Has Been Deleted')
        del self.__class__.__counter

# Awesome, now we can keep track of created characters. Let's check that:
geralt = GameCharacter(name = 'Geralt', hair_color = 'White', height = 180)
ciri = GameCharacter(name = 'Ciri', hair_color = 'White', height = 180)

# Let's call the getter
print('Initial Number of Created Game Characters: ', ciri.get_counter() )

# let's delete an object and make sure that __counter is working properly
del geralt
print('Current Number of Created Game Characters: ', ciri.get_counter() )

# let's call the setter
ciri.set_counter(42)
print('New Counter Value: ', ciri.get_counter())

# Let's call the deleter
ciri.drop_counter()
```

```
Initial Number of Created Game Characters:  2
Current Number of Created Game Characters:  1
New Counter Value:  42
Counter Has Been Deleted
```

Perfect, the getters, setters and deleters are seemed to be working. However, these methods work only for the **__counter**.

They aren't going to work for the rest attributes (e.g. **__name, __hair_color and __height**). As a solution, we can create individual setters, getters and deleters for every private attribute but it **contradicts DRY conception** (Don't Repeat Yourself).

Luckily, exists a solution which is called **descriptors.**

Descriptors allow writing getters, setters and deleters only once and prevent repeating (10. Descriptors). Moreover, the way how we've defined the getters, setters and deleters isn't the only one and not the best. There are more "convenient" ways based on the class/decorator **property.**

Class property can be found here: (9.7.1 Property Class Implementation)

5.4 Main Points of the Chapter

- **Child class** inherits parental methods and attributes;
- **Child methods** can be extended by parental ones;
- **Polymorphism** - the same methods name but different logic;
- Attributes and methods can be encapsulated (public, protected and private);
- Protected and Public attributes are **identical;**
- For calling private attributes define **getters and setters**

6. Instance, Class and Static Methods. What Is the Difference?

I think that it's important to know that not all methods can have access to all attributes of a class. Some methods can change only the state of a class whereas others the state of objects. These **methods have a different scope of variables** and understanding the difference between them is crucial!

6.1 Class Methods

Let's start with the class method. The name of the method stands on its own. They take in **class as an argument** and has the following decorator: **@classmethod**. These methods can change only the state of a class because objects attributes aren't available for them (these methods don't take in self as an argument). For simplicity, let's come back to encapsulated class attribute **__counter.**

Previously we've already defined two special methods **set_counter()** and **get_counter().** Now let's make them class methods.

In [57]:

```
# The same class
class GameCharacter:

    __counter = 0

    # The same constructor
    def __init__(self, name, hair_color, height):
        self.name = name
        self.hair_color = hair_color
        self.height = height

        self.__class__.__counter += 1

    # Make the following methods class methods
    @classmethod
    def get_counter(cls):
        return cls.__counter

    @classmethod
    def set_counter(cls, new_value):
        cls.__counter = new_value
        return cls.__counter

geralt = GameCharacter(name = 'Geralt', hair_color = 'White', height = 180)
ciri = GameCharacter(name = 'Ciri', hair_color = 'White', height = 180)

# Let's call these methods
print('Total Number of Created Objects: ', GameCharacter.get_counter())
print("Changed Number of Created Objects: ", GameCharacter.set_counter(10))
```

```
Total Number of Created Objects:  2
Changed Number of Created Objects:  10

Exception ignored in: <function GameCharacter.__del__ at 0x7f845a5a49e0>
Traceback (most recent call last):
  File "<ipython-input-56-a43d31b9ea64>", line 13, in __del__
AttributeError: type object 'GameCharacter' has no attribute
'_GameCharacter__counter'
```

Previously these methods took in neither an object nor a class what isn't right.
Methods **get_counter()** and **set_counter()** are responsible for getting and changing the class attribute. Thus, it must be a **class method.** The current implementation is more appropriate and the first sight at the code gives more details (decorators indicate that these methods belong to the class)

6.2 Static Methods

These methods **take in neither objects nor classes** as arguments. Thus, their scope of attributes is strongly bounded. Attributes of classes and objects **aren't available** for them and they **can't change** the state of classes and objects. They can operate only those arguments which are defined in them. The main advantage of static methods that they **don't depend on classes and objects.** In addition, they can be called directly through a class without any objects creation. Calling these methods through objects is possible as well.

We can define static methods with the help of a special decorator **@staticmethod**.

If you see a method without **self** parameter, it's a good idea to make it either static or class method.

Let's create **@staticmethod** for a randomly generating age of a game character.

In [58]:

```
import numpy as np
np.random.seed(42)

class GameCharacter:
    # Define the static method first
    @staticmethod
    def generate_random_number():
        return np.random.randint(1,100,1)[0]

    # Define the constructor where age will be randomly generated by the static
method
    def __init__(self, name, hair_color, height):
        self.name = name
        self.hair_color = hair_color
        self.height = height
        # static methods available for both objects and classes.
        # Let's call the method via the class
        self.age = self.__class__.generate_random_number()

random_character = GameCharacter(name = 'Bob', hair_color = 'Grey', height =
188)
print('Age of a game character: ', random_character.age)

Age of a game character:  52
```

The method **generate_random_number()** takes in neither an object nor a class as an argument. In addition, we were able to pass this method into the constructor to randomly create age values. Static methods might be a good choice if you need functions that will be responsible for some calculations and won't be changing states of objects or classes. These facts make static methods **more stable and reliable.**

6.3 Instance Methods

We're already familiar with them. They take in the **self argument** (i.e. they take in an object explicitly). It allows them getting access and control attributes of objects. In addition, with the syntax: **self.__class__** they can get access to class attributes and manipulate them as well. All these make them the most powerful methods in a class. The following example will be rather lengthy because this is how I want to demonstrate the flexibility and power of instance methods.

In [59]:

```
import random

class GameCharacter:

    # Define class attributes assuming that the age of a game character
    # is in the range from 1 to 100 and make them private
    __counter = 0
    __min_age = 1
    __max_age = 100

    # Add new the attribute age in the costructor.
    # Now let's demonstrate how powerfult the self is
    def __init__(self, name, hair_color, height, age):

        # Don't change
        self.name = name
        self.hair_color = hair_color
        self.height = height
        # Let's allow to create a game character if the age is in the range
        # attributes  __min_age and __max_age are private and belong to the
class
        # but we can call them anyway
        if age >= self.__class__.__min_age and age <= self.__class__.__max_age:
            print(f'{self.name} has been successfully created')
            self.__class__.__counter += 1

        # Raise Value Error for inappropriate age
        else:
            raise ValueError

class Witcher(GameCharacter):

    # Previously it was a method. Let's make it a private attribute now
    __signs = ('Axii', 'Qven', 'Ignii', 'Aard', 'Yrden')

    # Define the constructor of the class
    def __init__(self, name, hair_color, height, age, witcher_school = 'Wolf'):
        super().__init__(name, hair_color, height, age)
        self.witcher_school = witcher_school
        # Create a new attribute of an object
        self.signs = self.__class__.__signs

    # Randomy picks some sign
    def pick_sign(self):
        sign_number = random.sample(range(0,len(self.signs)),1)[0]
        return self.signs[sign_number]

    # Attack method
    def attack(self):
        print('The Sword Was Bared!')
        print(f'Sign {self.pick_sign()} Is Ready')

geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180, age = 48)

# Let's call a new attribute
print('Available Signs: ', geralt.signs)

# Let's call new methods
print('Randomly picked sign: ', geralt.pick_sign())
print('\n')
geralt.attack()
```

```
Geralt has been successfully created
Available Signs:  ('Axii', 'Qven', 'Ignii', 'Aard', 'Yrden')
Randomly picked sign:  Axii


The Sword Was Bared!
Sign Qven Is Ready
```

Let me explain what was going on in the code. First, I've decided to allow creating a new game character on condition that a **proper age is provided.** For this purpose, I've created **private attributes __min_age and __max_age.**

Because **__init__()** method is an instance method it has access to attributes of the class, thus we can call **__min_age and __max_age** by using a special syntax.

This time class Witcher had its own private attribute called **__signs** (a tuple with signs names). The constructor of Witcher class was extended by parental class and had a **new attribute called signs.** It was a new object attribute. Two new methods were defined. The most interesting here was that these two methods used attributes from the classes as well as object attributes (i.e. instance methods had access to both object and class attributes). I do hope that you understood why instance methods are so powerful and flexible.

6.4 Main Points of the Chapter

- Class methods take in a **class as an argument** and they have **access to all class attributes;**
- Provide **@classmethod** to define a class method;
- Class methods are used to **change the state of a class (its attributes);**
- Static methods take in **neither a class nor an object as an argument** and they don't have access to class or object attributes;
- Provide **@staticmethod** to define a static method;
- Instance methods take in an object as an argument and have access to both class and object attributes

7. Methods and Operators Overloading

Key elements here will be **operators** (e.g +, -, / and so forth) and **functions.** For now, all we need to know is that overloading helps us change the logic or behaviour of a certain operator or a function.

Let's have a look at methods overloading.

7.1 Method Overloading

This is my favourite topic because it makes your functions/code flexible. I've encountered dozens of times with situations when I wanted to control the behaviour of a function depending on incoming parameters. Imagine that you are developing a plot function and you want to control whether or not to display a legend, confidence intervals of a graph or a plot type. For this purpose, you define parameters or let's name them triggers depending on which you switch the behaviour of the function (e.g. we can define the following parameters: show legend = True, show_conf_intervals = True. Switching these parameters to True/False will lead to changing the function behaviour, exactly what we want). That's probably all we need to know about methods overloading.

To overload a method just define a set of parameters with default values. I personally call them **states.**

As default states values: **True/False or None** can be used.

Let's define a method **attack()** for Witcher class and overload it. In the game, we use a silver sword for monsters and a usual one for people.

In [60]:

```
class Witcher(GameCharacter):

    __signs = ('Axii', 'Qven', 'Ignii', 'Aard', 'Yrden')

    def __init__(self, name, hair_color, height, age, witcher_school = 'Wolf'):
        super().__init__(name, hair_color, height, age)
        self.witcher_school = witcher_school
        self.signs = self.__class__.__signs

    def pick_sign(self):
        sign_number = random.sample(range(0,len(self.signs)),1)[0]
        return self.signs[sign_number]

    # let's overload this method
    def attack(self, is_monster = False):
        if is_monster == True:
            return f'Bare Silver Sword\nSign {self.pick_sign()} Is Ready'
        else:
            return f'Bare Usual Sword\nSign {self.pick_sign()} Is Ready'

geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180, age = 48)

# For people set is_monster by default (it's switched off)
print(geralt.attack())
print('\n')

# For Monsters set is_monster = True (switch it on)
print(geralt.attack(is_monster=True))
```

```
Geralt has been successfully created
Bare Usual Sword
Sign Aard Is Ready


Bare Silver Sword
Sign Axii Is Ready
```

As you can see it isn't difficult at all. We controlled the logic of the function **attack()** by **switching on and off** the argument **is_monster.** This is how the method overloading works. You can define as many states or switchers as you want but the concept will be the same: **Different function logic depending on initial states of parameters.**

We've overloaded the **custom function** because it was defined by us but what to do if we want to **overload built-in functions?** Well, just write this built-in function and make it do what you want. There is a huge list of built-in functions and you can find them in the documentation:

- https://docs.python.org/3/reference/datamodel.html
- https://docs.python.org/3/library/functions.html

A list of **magic methods:** https://www.tutorialsteacher.com/python/magic-methods-in-python

let's overload the magic method **__str__()**. Overloading of this method allows defining what the object must return if it has been taken as a string.

In [61]:

```
print(geralt)
```

```
<__main__.Witcher object at 0x7f845a6228d0>
```

This is probably not what we want to see when printing the object. let's change that:

In [62]:

```
class Witcher(GameCharacter):

    __signs = ('Axii', 'Qven', 'Ignii', 'Aard', 'Yrden')

    def __init__(self, name, hair_color, height, age, witcher_school = 'Wolf'):
        super().__init__(name, hair_color, height, age)
        self.witcher_school = witcher_school
        self.signs = self.__class__.__signs

    # Overloading
    def __str__(self):
        return f'This Is {self.name} Object'

geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180, age = 48)
print(geralt)
```

```
Geralt has been successfully created
This Is Geralt Object
```

7.2 Operator Overloading

It changes the logic of operator such as addition, multiplication and so forth apart from its default operation (e.g. numbers will be added, strings will be concatenated and lists will be merged). The best and simple example here is probably coordinates of a point which we want to add with another point (i.e. add two objects together). But this is just an example and I again come back to my favourite computer game.

In the game, the Witchers create different elixirs. It gives them extra abilities and power. Let's assume that if we add several elixirs, we combine their effects. We will create the **class Elixir** and pass in its constructor the following attributes:

- Name: name of the elixir (str);
- Properties: set of properties which an elixir provides (list);
- Toxicity: each elixir has its toxicity which causes harm effects (int , seconds);
- Duration: duration of each elixir (int , seconds)

let's start coding!

In [63]:

```
class Elixir:

    # Define constructor of the Class with the following attributes.
    # All attributes will be private (encapsulated)
    def __init__(self, name, properties, toxicity, duration):
        self.__name = name
        self.__properties = list(properties)
        self.__toxicity = toxicity
        self.__duration = duration

    # Define a method for extracting information for a particular elixir
    def get_properties(self):

        print(f'Information About {self.__name} Elixir:',
              f'\nProperties: {self.__properties}',
              f'\nToxicity: {self.__toxicity}',
              f'\nDuration: {self.__duration}')

    # Let's overload add operator.
    # Once again, we want add method to combine properties of elixirs,
    # display total toxicity and duration of each elixir
    def __add__(self, elixr_n):
        combined_properties = self.__properties  + elixr_n.__properties
        total_toxicity = self.__toxicity + elixr_n.__toxicity
        duration_per_elixir = {self.__name:self.__duration,
                               elixr_n.__name:elixr_n.__duration}

        print(f'Combined Properties: {combined_properties}',
              f'\nTotal Toxicity: {total_toxicity}',
              f'\nRemaining Time: {duration_per_elixir}')


# Create 2 different elixirs
thunder = Elixir(name = 'Thunder',
                 properties = ['Extra Power'],
                 toxicity = 25,
                 duration = 30)

cat = Elixir(name = 'Cat',
             properties = ['Night Vision'],
             toxicity = 15,
             duration = 240)

# Let's get info about the first elixir
thunder.get_properties()
print('\n')

# Let's get info about the second elixir
cat.get_properties()
print('\n')

# Combine properties of 2 elixirs:
thunder + cat
```

```
Information About Thunder Elixir:
Properties: ['Extra Power']
Toxicity: 25
Duration: 30


Information About Cat Elixir:
Properties: ['Night Vision']
Toxicity: 15
Duration: 240


Combined Properties: ['Extra Power', 'Night Vision']
Total Toxicity: 40
Remaining Time: {'Thunder': 30, 'Cat': 240}
```

Awesome, now we can manipulate elixirs and combine their properties, keep track on total toxicity value (high value can kill the Witcher) and see remaining time of each elixir. Sounds good.

However, in my code, it's possible to combine only two elixirs. For combining many elixirs you have to change the overloading method...leave it to you as homework.

Also, I'd like to leave here a reference to the documentation where you can find the main overloading operators:

- https://docs.python.org/3/reference/datamodel.html (Chapter 3.3.8. Emulating numeric types)

Plus, an article about methods and operators overloading:

- https://stackabuse.com/overloading-functions-and-operators-in-python/

7.3 Main Points of the Chapter

- **Method overloading** - different logic of a method depending on parameters of its arguments;
- **Operator overloading** - different behavior of certain operators (e.g. _*add__( ),* _div__( ) and so forth);

8. What is *args and **kwargs

In order to figure it out, we have to comprehend the concept of **positional** and **keywords arguments**.

Let's deal with them separately.

8.1 *args

*args stands for **arguments. It allows creating a tuple of positional arguments with arbitrary length.** The main operator here is **the star** whereas **arg** is just a variable name and can be any. Star operator allows **unpacking elements of objects such as tuples and lists.** With the help of *args we can provide any number of arguments for a function which provides function flexibility and stability. Let's have a look:

In [64]:

```
# Ordinary function with fixed number of arguments
def no_args_example(a,b,c):
    return sum((a,b,c))

no_args_example(1,2,3)
```

Out[64]:

```
6
```

In [65]:

```
# Above function is not flexible because adding more arguments raises an error
no_args_example(1,2,3,4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-65-67b46dcf9319> in <module>
```

```
      1 # Above function is not flexible because adding more arguments raises an
error
----> 2 no_args_example(1,2,3,4)

TypeError: no_args_example() takes 3 positional arguments but 4 were given
```

In [66]:

```
# Providing *args allows providing any number of argumets
def args_example(*args):
    print(type(args)) # for those who doesn't believe that *args returns a tuple
    return sum(args)

args_example(1,2,3,4)
```

```
<class 'tuple'>
```

Out[66]:

```
10
```

Splendidly, *args allows providing **any number of arguments** by packing all positional argument into a tuple.

8.2 **kwargs

kwargs stands for **keyword arguments. It allows creating a **dictionary of keyword arguments with arbitrary length.** The main operator here is a double star, kwargs is again just a variable name and can be any.

In [67]:

```
# let's create a function with a fixed number of keyword arguments
def no_kwargs_example(current_gold = 1000, item_price = 250):
    return print('Remaining Gold: ', current_gold - item_price)
no_kwargs_example()
```

```
Remaining Gold:  750
```

In [68]:

```
# Again, providing a new keyword argument raises an error. To prevent this,
let's use **kwargs
# Let's assume that first argument is always current_gold whereas other
arguments will be bought items
def kwargs_example(**kwargs):
    print(type(kwargs)) # to make sure that **kwargs return a dictionary
    keys = list(kwargs.keys()) # save keys of the dictionary for later
iterations
    for key in keys[1:]:
        if kwargs[keys[0]] >= kwargs[key]:
            kwargs[keys[0]] -= kwargs[key]
            print(f'{key} was Bought. Remaining Gold: {kwargs[keys[0]]}')
        else:
            print('Not Enough Gold')
```

```
kwargs_example(current_gold = 1000, sun_rune = 250, moon_rune = 450)

<class 'dict'>
sun_rune was Bought. Remaining Gold: 750
moon_rune was Bought. Remaining Gold: 300
```

As we can see **kwargs allows providing an arbitrary number of keyword arguments by unpacking them. So this all you have to know about **kwargs in Python. I do hope that it will help to make your future functions flexible and stable.

8.3 Main Points of the Chapter

- *args stands for **positional** arguments, **kwargs stands for **keyword** arguments;
- *args and **kwargs allow providing an **arbitrary** number of positional/keyword arguments;
- **Arguments and parameters** of a function are **different terms;**
- *args unpacks **tuple/list** into positional variables,**kwargs is similar but can be applied only for a dictionary;**
- Impossible to provide **several *args and **kwargs** for a single function because it's unclear how to split arguments between them;
- Positional and keyword arguments must follow the following order (**first - positional, then keyword arguments**)

9. Decorators</a>

We've already seen decorators in action when we were dealing with **static and class** methods and I thought that we need to cover at least the basics before finishing our amazing journey in OOP. Decorators aren't related with OOP paradigm, they came from the realm of **functional programming** (it's another programming paradigm where a program is constructed by applying only functions). In functional paradigm functions are **first-class functions.** First-class functions are just more flexible and have the following **properties:**

- Functions can be saved into variables;
- Functions can be defined inside other functions (function nesting);
- Functions can be passed as arguments into another function or functions

9.1 What Is a Decorator?

Decorator is just a function which takes another function as an argument and changes or extends its logic without changing any line of code (I personally think that **extends is a more appropriate word** because decorated function must be executed anyway and only after we change the logic). Defining only a decorator once we can apply it many times and change the logic of any function, amazing, isn't it?

Decorators can be applied to:

- Classes (in this case decorator takes a class as an argument. However, **applying a decorator on a class doesn't affect class methods**)
- Methods (takes a method as an argument)

Function logic can be extended with the help of another function which is nested in a decorator, it's called a **wrapper**. The wrapper executes decorated function and extends its behaviour. From this point, you have to firstly face with a term **closure**.

9.2 What Is Closure?

Closure is simply when a nested function has an access to arguments of the outermost function. Let's have a look at the following example

In [69]:

```
# For simplicity, let's define only one method with a nested function
# A method for buying something in the game
def buy_item(price): # this is the enclosing function
    current_gold = 1000
    def substract_gold(): # this is the nested function
        if current_gold >= price:
            remaining_gold = current_gold - price
            return print('Remaining Gold: ', remaining_gold)
        else:
            return 'Not Enough Gold'
    return substract_gold
```

Above we've created the nested function and applied the closure. **substract_gold()** function has an access to all variables of the main function **buy_item().**

Let's execute the function:

In [70]:

```
# Bought something that costed 50 gold coins
buy_item(50)
```

Out[70]:

```
<function __main__.buy_item.<locals>.substract_gold()>
```

We can see that calling **buy_item(50)** leads to getting only the address of the inner function. This is because the function **buy_item()** returns only a reference to the **substract_gold()** function. To execute the function we should have written the function **remaining_gold** with parenthesis: **remaining_gold()**...but we do not need that.

Let's save the result of the function into a variable

In [71]:

```
# func_res keeps the result of buy_item() function which is a reference to the
nested function
func_res = buy_item(50)

# Now variable func_res is a function and we can execute it. Let's try
func_res()
```

```
Remaining Gold:  950
```

In [72]:

```
# Actually, this option is also possible.
# First, get a reference to the nested function (first parenthesis)
# Then execute it (the second parenthesis)
buy_item(50)()
```

```
Remaining Gold:  950
```

This is probably all you need to know about **closures** and now we can easily proceed with decorators, especially with the **wrapper function.**

9.3 What Is a Wrapper Function?

It's just a **nested function of a decorator** which executes the decorated function and extends its behaviour. You might have already got bored with nesting but this is the concept on which decorators rely on. Understanding the Closre concept will help us in writing the first decorator.

let's start with the first simple example:

In [73]:

```
# Define the decorator
# Decorator structure is just nested functions
def show_message(func): # the main function
    def wrapper(): # nested function
        # execution of decorated function
        # (no need to save into a variable because it just prints 5)
        func()
        print('Time to decorate!')
        print('Decoration has finished')
    return wrapper

# Write a function to be decorated. The function simply prints 5
def show_5():
    print(5)

# Decorate show_5():
show_5 = show_message(show_5)
show_5()
```

```
5
Time to decorate!
Decoration has finished
```

Congrats, we've coped with the task and written the first decorator. However, it was probably the simplest example ever. In most cases, methods will have different arguments and the wrapper function must be able to take them all as well. For this purpose, we have to deal with ***args and **kwargs.** Luckily, we already know the topic and can easily implement them.

9.4 Decorator Implementation

Let it be simple. We may want to know who is using the method. Adding print to each code might be inconvenient and only in some cases, we may want to know who has used a certain method. As an alternative solution, you may use a method overloading and it will be working. However, it's inconvenient because you have to overload each method. When you have this situation applying a decorator can be a good solution for the problem. Let's overcome the problem by creating the following decorator:

In [74]:

```
# Decorator Creation (i.e. function inside a function. Nested function is a
wrapper)
def show_name(func):
    # function can take in positional as well as keyword arguments
    def wrapper(self, *args, **kwargs):
        # pack all possible argumetns. self must be treated separately
        func(self, *args, **kwargs)
        print(f'{self.name} has called the method')
    return wrapper


# Familiar code
class Witcher(GameCharacter):
    __signs = ('Axii', 'Qven', 'Ignii', 'Aard', 'Yrden')

    def __init__(self, name, hair_color, height, age, witcher_school = 'Wolf'):
        super().__init__(name, hair_color, height, age)
        self.witcher_school = witcher_school
        self.signs = self.__class__.__signs

    def pick_sign(self):
        sign_number = random.sample(range(0,len(self.signs)),1)[0]
        return self.signs[sign_number]

    @show_name
    def attack(self, is_monster = False):
        if is_monster == True:
            print(f'Bare Silver Sword\nSign {self.pick_sign()} Is Ready')
        else:
            print(f'Bare Usual Sword\nSign {self.pick_sign()} Is Ready')
    @show_name
    def greeting(self):
        print(f"My name is {self.name}\nI'm the Witcher and I'm looking for the
monsters")


geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180, age = 48)
ciri = Witcher(name = 'Ciri', hair_color = 'White', height = 180, age = 22)
vesimir = Witcher(name = 'Vesimir', hair_color = 'Grey', height = 180, age = 88)

print('\n')
geralt.attack(is_monster = True)

print('\n')
ciri.pick_sign()

print('\n')
vesimir.greeting()
```

```
Geralt has been successfully created
```

```
Ciri has been successfully created
Vesimir has been successfully created
```

```
Bare Silver Sword
Sign Qven Is Ready
Geralt has called the method
```

```
My name is Vesimir
I'm the Witcher and I'm looking for the monsters
Vesimir has called the method
```

Unbelievable, defining the decorator **@show_name** only once we were able to extend the logic of all defined function in the **Witcher class.** I hope, the example has demonstrated how powerful decorators can be.

9.5 Module functool.wraps

I have to point out one moment. If we will try to get the name of a decorated function, we get what we don't expect:

In [75]:

```
geralt.greeting.__name__
```

Out[75]:

```
'wrapper'
```

It says that the function **greeting()** is a wrapper but this isn't what we want to see. In addition, we can't access and see attributes of the function as well. To overcome the problem, we have to manually rewrite the function **show_name()** and a reference to the documentation in the **wrapper()** function. Have a look:

In [76]:

```
def show_name(func):

    def wrapper(self, *args, **kwargs):
        func(self, *args, **kwargs)
        print(f'{self.name} has called the method')
    # Rewrite the function name and the reference to the documentation
    wrapper.__name__ = func.__name__
    wrapper.__doc__ = func.__doc__
    return wrapper


class Witcher(GameCharacter):

    __signs = ('Axii', 'Qven', 'Ignii', 'Aard', 'Yrden')
    def __init__(self, name, hair_color, height, age, witcher_school = 'Wolf'):
        super().__init__(name, hair_color, height, age)
        self.witcher_school = witcher_school
        self.signs = self.__class__.__signs

    @show_name
    def greeting(self):
        print(f"My name is {self.name}\nI'm the Witcher and I'm looking for the
monsters")

geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180, age = 48)

# Now everything is right
geralt.greeting.__name__
```

```
Geralt has been successfully created
```

Out[76]:

```
'greeting'
```

However, there is a better solution which provides **module functool.wraps**. All we have to do is just add a decorator **@wraps** to the function **wrapper()** and it solves all previous problems with the help of a single line of code.

In [77]:

```
from functools import wraps

def show_name(func):
    # Only need to add a decorator
    @wraps(func)
    def wrapper(self, *args, **kwargs):
        func(self, *args, **kwargs)
        print(f'{self.name} has called the method')
    return wrapper

class Witcher(GameCharacter):

    __signs = ('Axii', 'Qven', 'Ignii', 'Aard', 'Yrden')

    def __init__(self, name, hair_color, height, age, witcher_school = 'Wolf'):
        super().__init__(name, hair_color, height, age)
        self.witcher_school = witcher_school
        self.signs = self.__class__.__signs

    @show_name
    def greeting(self):
        print(f"My name is {self.name}\nI'm the Witcher and I'm looking for the
monsters")

geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180, age = 48)

# Now everything is right
geralt.greeting.__name__
```

```
Geralt has been successfully created
```

Out[77]:

```
'greeting'
```

For more info, check this huge article about python decorators:

- https://realpython.com/primer-on-python-decorators/#more-real-world-examples

9.6 Main Built-in Decorators in Python

Python includes **3 main build-in decorators:**

- @classmethod;
- @staticmethod;
- @property

The first two we already know but **@property** is something new for us. With the help of this decorator we can:

- **"Transform"** a function into a property (field);
- Create methods for **getting/setting and deleting property values** in a more convenient way

9.7 How Can I Get a Property From a Function?

Decorator **@property** allows transforming a function into a property. For instance, we may want to define a function that combines some properties or make a result of a function a new property. All you have to do is just apply **@property** on a function.

Let's come back to **Witcher class** and transform a function **pick_sign()** into a property.

In [78]:

```
class Witcher(GameCharacter):

    __signs = ('Axii', 'Qven', 'Ignii', 'Aard', 'Yrden')

    def __init__(self, name, hair_color, height, age, witcher_school = 'Wolf'):
        super().__init__(name, hair_color, height, age)
        self.witcher_school = witcher_school
        self.signs = self.__class__.__signs

    # Define a new property
    @property
    def pick_sign(self):
        sign_number = random.sample(range(0,len(self.signs)),1)[0]
        return self.signs[sign_number]

ciri = Witcher(name = 'Ciri', hair_color = 'White', height = 180, age = 22)

# Now instead of a method Ciri has a new property
ciri.pick_sign
```

```
Ciri has been successfully created
```

Out[78]:

```
'Axii'
```

You can make as above but I personally prefer defining a property like that:

In [79]:

```
class Witcher(GameCharacter):

    __signs = ('Axii', 'Qven', 'Ignii', 'Aard', 'Yrden')

    def __init__(self, name, hair_color, height, age, witcher_school = 'Wolf'):
        GameCharacter.__init__(self, name, hair_color, height, age)
        self.witcher_school = witcher_school
        self.signs = self.__class__.__signs

    # Don't touch origianl function
    def pick_sign(self):
        sign_number = random.sample(range(0,len(self.signs)),1)[0]
        return self.signs[sign_number]

    # Now define a new property name (the name of a function will be a new
property)
    @property
    def picked_sign(self):
        return self.pick_sign()

ciri = Witcher(name = 'Ciri', hair_color = 'White', height = 180, age = 22)

# let's call the created property
ciri.picked_sign
```

```
Ciri has been successfully created
```

Out[79]:

```
'Aard'
```

In [80]:

```
# However, we can't change or delete this property.
# For this purpose, we have to define the setter and deleter
ciri.picked_sign = 'Qven'
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-80-eb4c8236362d> in <module>
      1 # However, we can't change or delete this property.
      2 # For this purpose, we have to define the setter and deleter
----> 3 ciri.picked_sign = 'Qven'

AttributeError: can't set attribute
```

We've already covered **getters, setters and deleters**. However, there is a more convenient way of defining them.

Keep in mind that there are two options of defining the getters, setters and deleters:

- Using **@property decorator;**

- Using **property class**

Let's deal with them one by one

9.7.1 Property Class Implementation</a>

I suggest coming back to the example that was covered in 5.3.3 Getting and Setting Encapsulated Attributes and rewrite it by using **property class.**

In [81]:

```
class GameCharacter:

    _counter = 0

    def __init__(self, name, hair_color, height):
        self.__name = name
        self.__hair_color = hair_color
        self.__height = height
        self.__class__._counter += 1

    def __del__(self):
        self.__class__._counter -= 1

    # Define the getter, setter and deleter
    def get_counter(self):
        return self.__class__._counter

    def set_counter(self, new_value):
        # Only int is allowed
        if isinstance(new_value, int):
            self.__class__._counter = new_value
        else:
            raise TypeError('Counter Must Be Int!')

    def drop_counter(self):
        print('Counter Has Been Deleted')
        del self.__class__._counter

    # Now we can name our property. Let's name it counter.
    # We define counter property by using the special class property
    # and pass the created getter, setter and deleter as arguments

    # As forth argument description of the property may be passed
    counter = property(get_counter, set_counter, drop_counter, 'Counter of Game
Characters')

geralt = GameCharacter(name = 'Geralt', hair_color = 'White', height = 180)
ciri = GameCharacter(name = 'Ciri', hair_color = 'White', height = 180)

# The class property allows not calling get_counter(), set_counter() and
drop_counter() methods
# We can get, set and delete the property counter using the dot notation
print('Initial Number of Created Game Characters: ', ciri.counter)

ciri.counter = 10
print('New Counter Value: ', ciri.counter)

del ciri.counter
```

```
Initial Number of Created Game Characters:  2
New Counter Value:  10
```

```
Counter Has Been Deleted
```

It looks much better, isn't it?

9.7.2 Property Decorator Implementation

Here we will cover another example. Let's apply **@property** on **Witcher class** and define its getter, setter and deleter methods.

In [82]:

```
# We have to rewrite GameCharacter class because:
# 1. Make the field counter protected because private attributes can't be
inherited
# 2. In above code __counter becomes the property .counter
class GameCharacter:

    _counter = 0

    def __init__(self, name, hair_color, height):
        self.__name = name
        self.__hair_color = hair_color
        self.__height = height
        self.__class__._counter += 1

    def __del__(self):
        self.__class__._counter -= 1

class Witcher(GameCharacter):

    __signs = ('Axii', 'Qven', 'Ignii', 'Aard', 'Yrden')

    def __init__(self, name, hair_color, height, witcher_school = 'Wolf'):
        super().__init__(name, hair_color, height)
        self.witcher_school = witcher_school
        self.signs = self.__class__.__signs

    def pick_sign(self):
        # I had to modify this function due to signs[sign_number].
        # The explanation will be described after the code section
        if isinstance(self.signs, tuple):
            sign_number = random.sample(range(0,len(self.signs)),1)[0]
            return self.signs[sign_number]
        else:
            return self.signs

    @property
    def picked_sign(self):
        return self.pick_sign()


    # For setter use the following syntax: @property_name.setter
    @picked_sign.setter
    def picked_sign(self, new_value):
        # Allow only string values
        if isinstance(new_value, str):
            self.signs = new_value
        else:
            raise TypeError('New Value Must Have a String Format')
    # For deleter use the following syntax: @property_name.deleter
    @picked_sign.deleter
    def picked_sign(self):
        print(f'Property {self.signs} Has Been Successfully Deleted!')
        del self.signs

ciri = Witcher(name = 'Ciri', hair_color = 'White', height = 180)
print('Currently Picked Sign: ', ciri.picked_sign)

# Let's set another value for property pick_sign
ciri.picked_sign = 'Ignii'
print('New Set Sign Value: ', ciri.picked_sign)

# Can delete as well
del ciri.picked_sign
```

```
Currently Picked Sign:  Ignii
New Set Sign Value:  Ignii
Property Ignii Has Been Successfully Deleted!

Exception ignored in: <function GameCharacter.__del__ at 0x7f845a5e1f80>
Traceback (most recent call last):
  File "<ipython-input-81-cfbc52eb5ee3>", line 12, in __del__
AttributeError: type object 'GameCharacter' has no attribute '_counter'
```

In the above code, we've implemented the **setter and deleter methods** by using the special decorator **@property**. I've decided to make the function **pick_sign() a property.** After it became a property, whenever we want to call this property it **executes** the function **pick_sign()**.

In my case, I had to **reimplement** this function because it was returning a tuple with a random index. Can you guess what kind of a problem it leads? When setting a new property value, it executes the function again and instead of a new sign name we get only a new letter of that sign name. To overcome the problem, I've defined the following checking (first calling self.signs always returns a tuple, whereas the second one - a string). Luckily, it works.

So you've seen that we can transform a function into a new property and we can do that in two different ways either using the decorator or the class **property.** In addition, we can not only create a new property but also change it using setters and deleters.

9.8 Main Points of the Chapter

- Functions are **first-class objects** in Python;
- **Decorator** is a function which extends the logic of a decorated function;
- **Wrapper function** is nested into a decorator. It executes a decorated function and extends its logic;
- Decorator can be applied to **classes and objects;**
- Decoration of a class doesn't decorate its methods.
- **@wraps** copies decorated function name, signature and documentation.

10. Desriptors</a>

Descriptor is an object attribute with binding behaviour (i.e. an object attribute whose behaviour is being overridden by descriptor methods). I'm sure that the definition is complex and probably doesn't make any sense. For this purpose, let's use another definition: Descriptor is a class where there are 3 special methods:

- __get__() # getter
- __set__() # setter
- __delete__() # deleter

Besides, descriptors can be divided into two groups:

- **Data-Descriptor** # has all methods
- **Non-Data Descriptor** # has only get() method

We already familiar with these methods but once we've faced with a problem. We could define the getters, setters and deleters only for one attribute. With the help of descriptors, our life becomes much easier. We only need to find attributes that are similar in terms of behaviour (i.e attributes that have binding behaviour) and define descriptors for them. Once the descriptors defined, they provide similar behaviour for binding attributes (i.e. defining the getters, setters and deleters only once, we can get, change and delete binding attributes) and this is super cool!

let's come back to the example that was covered in chapter 5.3.3 Getting, Setting and Deleting Encapsulated Attributes and apply descriptors

In [83]:

```
# First we have to define class for descriptors. Let's name it Descriptor
class Descriptor:

    def __init__(self, name):
        self.__name = name

    def __get__(self, instance, owner):
        return instance.__dict__[self.__name]

    def __set__(self, instance, value):
        if self.__name in ['name','hair_color']:
            if isinstance(value, str):
                instance.__dict__[self.__name] = value
            else:
                raise TypeError
        else:
            if isinstance(value, int):
                instance.__dict__[self.__name] = value
            else:
                raise TypeError

    def __delete__(self, instance):
        print(f'Property {self.__name} Has Been Deleted')
        del instance.__dict__[self.__name]


class GameCharacter:

    # First define descriptors as below
    name = Descriptor('name')
    hair_color = Descriptor('hair_color')
    height = Descriptor('height')

    # All the rest looks the same
    __counter = 0
    def __init__(self, name, hair_color, height):
        self.name = name
        self.hair_color = hair_color
        self.height = height

        self.__class__.__counter += 1

geralt = GameCharacter(name = 'Geralt', hair_color = 'White', height = 180)
bob = GameCharacter(name = 'bob', hair_color = 'White', height = 180)

# Now we get, set and delete properties (attributes) much easier. Plus, we've
defined parameters check before assigning them
print('Using Get To Access a Property Value: ', geralt.name)

# New value setting
geralt.height = 175
print('New Height Value: ', geralt.height)

# Let's delete property hair_clor
del geralt.hair_color
```

```
Using Get To Access a Property Value:  Geralt
New Height Value:  175
Property hair_color Has Been Deleted

Exception ignored in: <function GameCharacter.__del__ at 0x7f845a5e1f80>
Traceback (most recent call last):
  File "<ipython-input-81-cfbc52eb5ee3>", line 12, in __del__
AttributeError: type object 'GameCharacter' has no attribute '_counter'
```

let's understand what was going on in the code above. We should always start with defining the **descriptor class**. The name of the class can be any (not obligatory descriptor, I've just named it like that because this name is pretty straightforward).

The first method in the descriptor class is **__init__()**. In this method, we save the name of local property into property **__name.**

In **get**() method, we define what the method should return when we call the property. As it's the **get()** method it must return a calling attribute. Also, have a look at arguments which the method take in:

- **self** # belongs to the descriptor property (descriptor instance);
- **instance** # instance of the class (e.g. ciri, geralt and so forth);
- **owner** # a class in which the descriptors are defined (Witcher class)

Here **instance** argument will refer to an instance for which the descriptor was called. Then for this instance, a descriptor property is being created. In **get()** method, we call this property, whereas in the **set()** method we assign a new value for this property as well as check value type. In **deleter**, the logic is the same.

Important Notes

__init__() method from Witcher class doesn't create local attributes for Witcher class. Instead, here it assigns values to the descriptors. New properties aren't created due to descriptors operators overloading and method **set**() from descriptor class is called.

Starting from Python 3.6+ a more convenient method has appeared.

All we need to do is just write instead of **__init__()** another method **__set_name__()**

In [84]:

```
class Descriptor:

    def __set_name(self, owner, name):
        self.__name = name

    def __get__(self, instance, owner):
        return instance.__dict__[self.__name]

    def __set__(self, instance, value):
        if self.__name in ['name','hair_color']:
            if isinstance(value, str):
                instance.__dict__[self.__name] = value
            else:
                raise TypeError
        else:
            if isinstance(value, int):
                instance.__dict__[self.__name] = value
            else:
                raise TypeError

    def __delete__(self, instance):
        print(f'Property {self.__name} Has Been Deleted')
        del instance.__dict__[self.__name]

class GameCharacter:

    # There is now need to rpovide names as parameters
    name = Descriptor()
    hair_color = Descriptor()
    height = Descriptor()

    # All the rest is the same
    __counter = 0

    def __init__(self, name, hair_color, height):
        self.name = name
        self.hair_color = hair_color
        self.height = height

        self.__class__.__counter += 1
```

11. Abstract Classes. Why Do We Need Them?

First of all, let's understand what is an abstract class:

- **Abstract class** is a class which has at least **one abstract method or property** (a method which is defined but not implemented)

Abstract classes provide a **mutual interface** (a set of attributes) for its child classes. It helps to make sure that we defined all methods for all child classes and didn't forget anything. If we did forget, we aren't able to even create an object and this is better because we can know where exactly we forgot something and solve a problem faster.

Using abstract classes leads to an **increase in modularity and readability of the code.** It gives more insights to the developers of what is going on in the code as well as handle possible errors much faster.

In Python, abstract classes don't exist naturally. To create them we have to import a special module called abc (**Abstract Base Class**).

11.1 Abstract Base Class Module

Abstract Base Class module (ABC) provides an opportunity to create abstract classes in Python. There are several options for how you can create abstract classes. Have a look:

In [85]:

```
# Import ABC module
from abc import ABC, ABCMeta

# First option (My favourite one)
class AbstractClass(ABC):
        pass

# The second option
class AbstractClass(metaclass = ABCMeta):
    pass
```

From classes above we can create an object because we haven't defined any abstract methods or properties (there is nothing overload). Have a look:

In [86]:

```
abstract_object = AbstractClass()
abstract_object
```

Out[86]:

```
<__main__.AbstractClass at 0x7f845a5b9c10>
```

To define abstract methods and properties in an abstract class, we have to import **abstractmethod** and **abstractproperty** explicitly.

In [87]:

```
from abc import ABC, abstractmethod, abstractproperty

class AbstractClass(ABC):
    # Define one abstract method by using the following decorator
    @abstractmethod
    def attack(self):
        pass
    # The abstract property
    @abstractproperty
    def greeting(self):
        pass
```

After we've defined at least one abstract method or property we can't create an object any more. Have a look:

In [88]:

```
abstract_object = AbstractClass()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-88-047c7a6ae952> in <module>
----> 1 abstract_object = AbstractClass()

TypeError: Can't instantiate abstract class AbstractClass with abstract methods
attack, greeting
```

From my point of view, abstract classes are used for providing a **mutual interface.**

What does mutual interface mean?

Well, the simplest explanation may sound like this: a set of class attributes and methods is an **interface**, thus classes that inherit abstract classes must have these attributes and methods (overloaded!) as well as some new attributes. Even though some classes will have different attributes, always will be a set of attributes or methods defined in both of them (mutual interface). I hope you got the idea.

11.2 Abstract Class Implementation

This time let's make **GameCharacter** class abstract and inherit two classes (Villager and Witcher). Have a look:

In [89]:

```
import random
from abc import ABC, abstractmethod

# Let's make GameCharacter class abstract
class AbstractClassGameCharacter(ABC):

    def __init__(self, name, hair_color, height):
        self.name = name
        self.hair_color = hair_color
        self.height = height

    # Each class must have these methods (mutual interface)
    @abstractmethod
    def greet(self):
        pass

    @abstractmethod
    def farewell(self):
        return 'Farewell!'

# Create the new class Villager
class Villager(AbstractClassGameCharacter):
    # Overload abstract methods
    def greet(self):
        return f'Hello My Name Is {self.name}'

    def farewell(self):
        return f'I Was Glad to See You'

class Witcher(AbstractClassGameCharacter):

    __signs = ('Axii', 'Qven', 'Ignii', 'Aard', 'Yrden')

    def __init__(self, name, hair_color, height, witcher_school = 'Wolf'):
        super().__init__(name, hair_color, height)
        self.witcher_school = witcher_school
        self.signs = self.__class__.__signs

    # Don't forget to overload the  abstract methods
    def greet(self):
        return f"I am glad to see you, my name is Geralt.\nI'm the Witcher"

    # Instead of overriding we can extend the method
    def farewell(self):
        extension = super().farewell()
        return f'Thanks for talking.{extension}'

    # Unique methods of Withcer Class (let's leave only pick_sign for
simplicity)
    def pick_sign(self):
        sign_number = random.sample(range(0,len(self.signs)),1)[0]
        return self.signs[sign_number]

# All abstract methods were overridden, thus no errors
geralt = Witcher(name = 'Geralt', hair_color = 'White', height = 180)
bob_villager = Villager(name = 'Bob Funny Villager', hair_color = 'Dark', height
= 175)

# Mutual Interface
print(geralt.farewell())
print('\n')
print(bob_villager.greet())
```

```
Thanks for talking.Farewell!


Hello My Name Is Bob Funny Villager
```

11.3 Main Points of the Chapter

- **Abstract class** is a class that has at least one abstract method or property;
- To define abstract class, import the **module abc;**
- Objects from abstract classes can't be created;
- All abstract methods or properties must be overridden;
- Static and class methods can be abstract as well;
- An **interface** is a set of class attributes and methods;
- Abstract classes provide a **mutual interface** between child classes

12. OOP Advantages and Disadvantages

In the last chapter, I would like to slightly pay your attention to the advantages and disadvantages of OOP paradigm.

12.1 Advantages

Undoubtedly, the main advantage is **code consistency**. Each class has its predefined fields and methods preventing the growth of new fields and methods for new objects. An object may be considered as a container with instruments (methods) and fields - variables.

The next one is **code reusability**. Inheritance allows to reuse, override or extend attributes and methods from parental classes.

The last one is probably **time**. Using OOP paradigm allows creating a complex system faster.

12.2 Disadvantages

You have to study a particular field carefully and determine relationships between classes. Wrong relationships and classes may destruct the whole system.

Conclusion

What an amazing journey it was! I foresee that many of you will think that the tutorial is rather lengthy...yes it is. However, have a look at how many important topics we've covered. I hope that you've strengthened your knowledge and topics such as overloading, decorators or encapsulation won't cause any difficulties any more. Thank you all guys and remember don't overfit, generalize.