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**
- **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 programming is almost inevitable. Even not knowing OOP paradigm in Python we utilize objects and classes which hadn't been cre

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

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. I more appropriate for newly created classes. However, these names are interchangeable and both can be used. For better understa


source: imgur.com

Above picture depicts that even classes such as Int, Float..Tuple are objects of the main metaclass Type. If it's difficult to understa 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 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 fie understand the topic by implementing OOP on own examples instead of mine. However, you can first go step by step through my e

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 clas 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 sou

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 t 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 s 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

In order to access class attributes we need an object of that class first and then call an attribute with the help of the foll (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 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 t 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 reas attributes and methods can be found only in its class from which can be inherited many objects. Thus, when a method i 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 wor

By the way, do you know why we got Vesimir is Saying: None? Well, this is because the function say() doesn't return an

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', 'heigh
unction GameCharacter.say at 0x7f845a6115f0>, '__dict__': <attribute '__dict__' of 'GameCharacter' objec
kref__' 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 object we have to either define a constructor (will be covered later) or assign object attributes explicitly. For example, b value weight and can delete only this attribute because other attributes exist only for the class. However, they can be a

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. L 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?

```
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 ti** 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 th

In Python **methods with underscores** belong to a **special group of methods** called **overloading or magic methods**. We do **are called automatically** when an object takes part in some action (e.g. when an object is being created **__init__()** metho 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 definec define default arguments in a method. In this case, you have to pass only obligatory arguments in a method, default val

We can not only change current attribute values but also create new ones. However, it isn't recommended as introduces

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 faci one class, they will be different in terms of attribute values. If we look at our previous implementation, we can notice tha 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

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}")
```

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 underscore
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 ir
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, othe

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 e
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 defa**
- **__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, at
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 **(outsid**

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 tho
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

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 a**
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 dea attributes and methods of its class when is being created. Taking this fact into account we can say that a new class can 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 something unique. Rewriting the same code millions of times is tedious and not a good idea, we need a better solution.. have to do is just define the main class (it's called parental class) with the main attributes and inherit as many classes a classes will have not only own unique attributes and methods but also attributes and methods from parental class or cl 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, vi all are game characters, they have some unique attributes (e.g. the witchers and monsters have extraordinary abilities location (clothes, type of voice and so on))

I hope you got the idea.

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 co ultimately find it. We can create __init__() for Witcher class as well. In this case, __init__() from Witcher class will overrid Witchers belong to different schools (e.g. Cat School, Wolf School, Bear School and others). Let's add this unique attrib

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 GameC

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 [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 t added several lines of code and initialized a new object with the unique method signs_ability() because in the game onl 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
```

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 bee
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._

However, this option isn't reliable because when it comes to multiple inheritance the order of which parental classes are introduces chaos and errors. To avoid this, use reliable construction: super().__init__(). The function super() will go thro 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). Previous get access to parental attributes. Thus, we can make extensions on other functions as well. In other words, we can call 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 th

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 mar
just an abstract term. Basically, polymorphism in OOP is just methods with the same names but absolutely different log**

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

**Let's implement polymorphism. For this purpose, simply create the method greeting() (the same method name but diffe
exists in GameCharacter class and defining the same method name in Witcher class will override the method, this is ex**

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 t
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 t
be huge and complex and inside can be many auxiliary attributes and methods which must not be used outside a certai
provide stability for a class. Thus, we may want to protect stability elements of a class and not allow changing them in a
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

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 r

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'
```

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**

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

**You may say: if we can access attributes height and _hair_color and change both of them what is the difference then? W
with a single underscore only indicates or tells that it's protected and can be used only in a certain class and its child cl
don't exist in Python.**

**In fact, protected and public attributes are identical. Single underscore only tells that the attribute protected and we sh
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 r
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 outs

**To better understand the topic let's cover one more example. For example, we may want to keep track of how many gam
really important because we may want to create an only certain number of characters for a certain location of the game
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)
```

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 at
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 attrib
attribute let's create a special method called get_counter(). Then create a child class Witcher and try calling the private**

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

How can we solve this problem?

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 private attribute has been defined, its name changes and has a prefix of a class to which belong. In our case, it belongs must have a prefix of that class.

Let's find out local attributes of Witcher class.

In [ ]:
```
Witcher.__dict__
```

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 __counte

They aren't going to work for the rest attributes (e.g. __name, __hair_color and __height). As a solution, we can create in 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 and deleters isn't the only one and not the best. There are more "convenient" ways based on the class/decorator proper

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 ch the state of objects. These methods have a different scope of variables and understanding the difference between ther

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 These methods can change only the state of a class because objects attributes aren't available for them (these method 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 met

It can be clearly seen that the name of private attribute __counter has changed to _GameCharacter__counter. Because c 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

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

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(class attribute. Thus, it must be a class method. The current implementation is more appropriate and the first sight at th 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. A available for them and they can't change the state of classes and objects. They can operate only those arguments whic static methods that they don't depend on classes and objects. In addition, they can be called directly through a class w 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 randomly create age values. Static methods might be a good choice if you need functions that will be responsible for so 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 get In addition, with the syntax: self.__class__ they can get access to class attributes and manipulate them as well. All these class. The following example will be rather lengthy because this is how I want to demonstrate the flexibility and power o

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 th 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

This time class Witcher had its own private attribute called __signs (a tuple with signs names). The constructor of Witch had a new attribute called signs. It was a new object attribute. Two new methods were defined. The most interesting he attributes from the classes as well as object attributes (i.e. instance methods had access to both object and class attrib 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 at**
- **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**