## OOP: Introduction

**Object-oriented programming** is a way of writing computer programs using "objects".

**Computer programs that are not object-oriented:** are a list of instructions for the computer, telling it to do certain things in a certain way, which is called **procedural programming**:

- Using functions by grouping commonly used group of statements,
- Using dictionary data type to group related set of data together.
- Functions operate on data (like integers, strings, lists, dict, etc.)

These techniques work very well to make the program short, concise, and modular providing maintainability and readability. However, in **procedural programming** as the programs become more complex with hundreds of variables and hundreds of functions, the above techniques result in programs that are difficult to maintain and debug.

**Object-oriented programming**, is a way of writing computer programs using "**objects**" to represent data and methods that can talk to each other and exchange the data in a way that the user wants. In another way is based on the concept of objects and classes. A class can be thought of as a 'blueprint' for objects. These can have their own attributes (variables or characteristics they possess), and methods (actions they perform).

Another way, object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on. OOP models real-world entities as software objects that have some data associated with them and can perform certain functions.

The ***objects*** are entities that contain both data and functionality together, called object's *methods*.

Because of the way object-oriented programming is designed, it helps the developer by allowing for code to be easily reused by other parts of the program or even by other people.

The advantages of Object-Oriented Programming are that:

- it results in higher-quality software; being faster and easier to execute
- Providing a clear structure for a program and easier to reuse code;
- The OOP programs are modular, making code easier to modify, debug and maintain.

**Object-oriented programming** is a better way to write programs.

Computationally, OOP software is slower, and uses more memory since more lines of code have to be written.

Python allows for computer programs to be written both in object-oriented programming and in procedural programming.

There are many programming languages that allow you to write computer programs in object-oriented programming. like: C++, Java, Ruby, Perl, Emarald, Sapphire, PHP, Python, etc.

Let us first review theory of class, object and methods before we get into OOP software programming.

**Classes**: are like the sand baking sets that you can use to mold sand into various shapes. Classes are basically these plastics that have a specific shape.
**Objects**: are the molded sand we get from these plastics. Why are they objects? – because they inherit every bit of curve, size, and corner of the plastic.

**Functions** are a way to group commonly used set of statements together to provide a certain functionality. it is useful to have a means to bind data and functionality together, especially when the programs become large.
**Classes** is an abstract concept; it provides a means of binding data and functionality together. This functionality is called **methods** of the data type.

Example:

The term 'dog' is a generic term for the animal dog, whereas a bulldog, or golden retriever, or a poodle is a variety (breed) of dog. the term 'dog' is called a *class* and a bulldog, or golden retriever, or poodle is referred to as an ***instance* of *class*** 'dog' and are called an ***object*** in the class 'dog'.

When we write OOP programs, one of the first things we do is to create an ***object*** of a ***class*** data type, we call it an ***instance*** of a class. Creating an instance (*object*) of a class is called **instantiation**.

**Instantiation** describes the mechanism by which objects are created from a class, used as an "object factory". This mechanism relies on the class having at least a default constructor.

For instance, an object could represent a person with **properties** like a name, age, and address and **behaviors** such as walking, talking, breathing, and running.

## Define a Class in Python:

**Primitive data structures** (like numbers, strings, and lists) are designed to represent simple pieces of information, such as the cost of an apple, the name of a poem, or your favorite colors, respectively.

What if you want to represent something more complex?

For example, let's say you want to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a list:

```
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

**There are a number of issues with this approach.**

1- It can make larger code files more difficult to manage. If you reference kirk[0] several lines away from where the kirk list is declared, will you remember that the element with index 0 is the employee's name?

2- It can introduce errors if not every employee has the same number of elements in the list. In the mccoy list above, the age is missing, so mccoy[1] will return "Chief Medical Officer" instead of Dr. McCoy's age.

A great way to make this type of code more manageable and more maintainable is to use classes.

**Classes vs Instances**

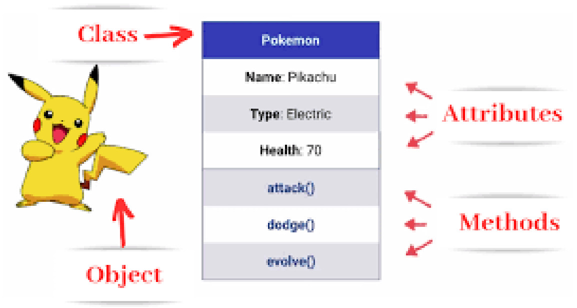
- Classes are used to create user-defined data structures.
- Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

In the below example we'll create a Dog class that stores some information about the characteristics and behaviors that an individual dog can have.

**A class** is a blueprint for how something should be defined. It doesn't actually contain any data. The Dog class specifies that a name and an age are necessary for defining a dog, but it doesn't contain the name or age of any specific dog.

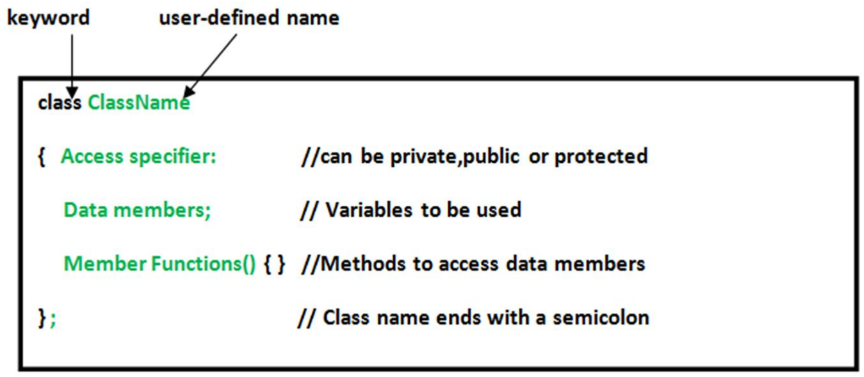
An **instance** is an object that is built from a class and contains real data. An instance of the Dog class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

Another way, **a class is like a form or questionnaire**. **An instance is like a form that has been filled out with information**. Just like many people can fill out the same form with their own unique information, many instances can be created from a single class.



**How to Define a Class**

All class definitions start with the class keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class's body.



```
class ClassName
{  Access specifier:          //can be private,public or protected
   Data members;              // Variables to be used
   Member Functions() { }     //Methods to access data members
};                            // Class name ends with a semicolon
```

**(C++ example)**
When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created)

```
class person
{
    char name[20];
    int id;
public:
    void getdetails() {}
};
int main()
{
    person p1; // p1 is a object
}
```

**Declaring Objects:** When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Example of a Dog class:

```
class Dog:
    pass
```

This creates a new Dog class with no attributes or methods.

pass is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

by defining some properties that all Dog objects should have. There are a number of properties that we can choose from, including name, age, coat color, and breed. To keep things simple, we'll just use name and age.

**Creating __init__ method for the class (Constructor Method or** instance attributes**)**

The __init__() method is also named "constructor." It called Python each time we instantiate an object. The constructor creates the object's initial state with the minimum set of parameters it needs to exist.it is a very special method which is the main method associated with every class.

The properties that all Dog objects must have are defined in.__init__(). Every time a new instance (Dog) object is created, .__init__() sets the initial **state** of the object by assigning the values of the object's properties.

You can give .__init__() any number of parameters, but the first parameter will always be a variable called **self**. When a new class instance is created, the instance is automatically passed to the self parameter in .__init__() so that new **attributes** can be defined on the object.

Let's update the Dog class with an .__init__() method that creates .name and .age attributes:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

def is a keyword in python that we use specifically which defining method for an object.

In the body of .__init__(), there are two statements using the self variable:

1. **self.name = name** creates an **attribute** called name and assigns to it the value of the name parameter.
2. **self.age = age** creates an **attribute** called age and assigns to it the value of the age parameter.

All Dog objects have a name and an age, but the values for the name and age attributes will vary depending on the Dog instance.

On the other hand, **class attributes** are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of .__init__(). For example, the following Dog class has a class attribute called species with the value "Canis familiars":

When you create a list object, you can use print() to display a string that looks like the list:

```
names = ["Fletcher", "David", "Dan"]
print(names)
OUTPUT:  ['Fletcher', 'David', 'Dan']
```

**Class and Instance Attributes**

Now create a new Dog class with a class attribute called. species and two instance attributes called .name and. age:

```
class Dog:
    # Class attribute
    species = "Canis familiars"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Class attributes are defined directly beneath the first line of the class name, they must always be assigned an initial value. When an instance of the class is created, class attributes are automatically created and assigned to their initial values.

Use class attributes to define properties that should have the same value for every class instance. Use instance attributes for properties that vary from one instance to another.

'self' parameter is **required** and must be the first one.

The Dog class's .__init__() method has three parameters, so why are only two arguments passed to it in the example?

**Method __init__ for Initialization of Objects:**

Python has a *method* called __init__ that is used to initialize the *object's* state.
The __init__method is run as soon as the *object* of a *class* is instantiated.
The __init__ *method* initializes the attributes of the new class instance (*object*) we created. Every class needs to have the __init__ method at the beginning of class definition. The following code snippet illustrates the use of __init__ method.

When you instantiate a Dog object, Python creates a new instance and passes it to the first parameter of .__init__(). This essentially removes the self parameter, so you only need to worry about the name and age parameters.

**Instantiate an Object**
Creating a new object from a class is called **instantiating** an object.

To instantiate objects of this Dog class, you need to provide values for the name and age. If you don't, then Python raises a TypeError:

```
a=Dog()
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    Dog()
TypeError: __init__() missing 2 required positional arguments: 'name' and 'age'
```

To pass arguments to the name and age parameters, put values into the parentheses after the class name:

```
a = Dog('Rix',2)
```

Now instantiate a second Dog object:
To see this another way, type the following:

```
a = Dog('Rix',2)
b = Dog('Rox',1)
a == b
OUTPUT:  False
```

In this code, you create two new Dog objects and assign them to the variables a and b. When you compare a and b using the == operator, the result is False. Even though a and b are both instances of the Dog class, they represent two distinct objects in memory.
After you create the Dog instances, you can access their instance attributes using **dot notation**:

```
Print(a.name)
OUTPUT:  'Rix'
Print(a.age)
OUTPUT:  2
```

You can access class attributes the same way:

```
Print(buddy.species)
OUTPUT:  'Canis familiaris'
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. All Dog instances have .species, .name, and .age attributes, so you can use those attributes with confidence knowing that they will always return a value.

Although the attributes are guaranteed to exist, their values *can* be changed dynamically:

```
a.age = 10
print(a.age)
```

```
OUTPUT:  10
a.species = "Felis silvestris"
Print(a.species)
OUTPUT:  'Felis silvestris'
```

In this example, you change the .age attribute of the a object to 10. Then you change the .species attribute of the a object to "Felis silvestris", which is a species of dog. That makes a a pretty strange dog, but it is valid Python!

The key takeaway here is that custom objects are mutable by default. An object is mutable if it can be altered dynamically. For example, lists and dictionaries are mutable, but strings and tuples are immutable.

**Instance Methods**
are functions that are defined inside a class and can only be called from an instance of that class. Just like .__init__(), an instance method's first parameter is always self.

a new Dog class:

```
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"
    def bark(self):
        print("bark bark!")
    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

This Dog class has two instance methods:

1. **.description()** returns a string displaying the name and age of the dog.
2. **.speak()** has one parameter called sound and returns a string containing the dog's name and the sound the dog makes.

Type the following to see your instance methods in action:

```
miles = Dog("Miles", 4)
print(miles.description())
OUTPUT: 'Miles is 4 years old'

Print(miles.speak("Woof Woof"))
OUTPUT: 'Miles says Woof Woof'
```

In the above Dog class, .description() returns a string containing information about
the Dog instance miles. When writing your own classes, it's a good idea to have a method that returns a string containing useful information about an instance of the class.
However, .description() isn't the most Pythonic way of doing this.

```
a = Dog('Rix',2)
print(a)
OUTPUT: <__main__.Dog object at 0x106702d30>
```

Note that the address you see on your screen will be different. When you print(miles), you get a cryptic looking message telling you that a is a Dog object at the memory address 0x106702d30. This message isn't very helpful. You can change what gets printed by defining a special instance method called .__str__().

change the name of the Dog class's .description() method to .__str__():

```
class Dog:
    # Leave other parts of Dog class as-is
    # Replace .description() with __str__()
    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

Now, when you print(miles), you get a much friendlier output:

```
>>> miles = Dog("Miles", 4)
>>> print(miles)
'Miles is 4 years old'
```

Methods like .__init__() and .__str__() are called **dunder methods** because they begin and end with double underscores. There are many dunder methods that you can use to customize classes in Python. Although too advanced a topic for a beginning Python book, understanding dunder methods is an important part of mastering object-oriented programming in Python.

To find out what operations (*methods*) are associated with a particular class instance *object*, the Python built-in function called dir() can be used from the Python shell (IDLE).

As Example: dir(str).

To find out what a particular *method* does, you can use the **help** command.

As Eample: help(str.capitalize)

## OOP in Python:

In Python, "everything" is treated the same way, everything is a class: functions and methods are values just like lists, integers or floats. Each of these are instances of their corresponding classes.

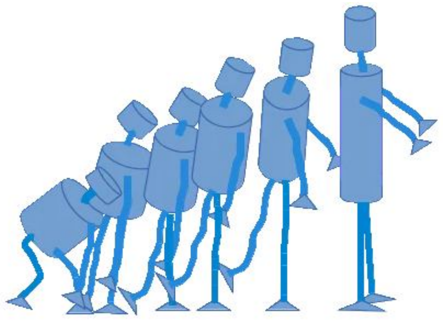
```
x = 42
type(x)
OUTPUT: int

def f(x):
   return x + 1
type(f)
OUTPUT: function
```

One of the many integrated classes in Python is the list class, which we have quite often used in our exercises and examples. The list class provides a wealth of methods to build lists, to access and change elements, or to remove elements:

```
x = [3,6,9]
y = [45, "abc"]
```

The variables x and y denote two instances of the list class.

**A Minimal Class in Python:**



We will design and use a robot class in Python as an example to demonstrate the most important terms and ideas of object orientation. We will start with the simplest class in Python.

```
class Robot:
    pass

if __name__ == "__main__":
    x = Robot()
    y = Robot()
    y2 = y
```

We have created two different robots x and y in our example. Besides this, we have created a reference y2 to y, i.e. y2 is an alias name for y.

Attributes in Python: is the variable in the class

**Methods in python**: are a function in the class

Let's define a function "hi", which takes an object "obj" as an argument and assumes that this object has an attribute "name". We will also define our basic Robot class again:

```
def hi(obj):
    print("Hi, I am " + obj.name + "!")
class Robot:
    pass
x = Robot()
x.name = "Marvin"
hi(x)
```

We will now bind the function „hi“ to a class attribute „say_hi“!

```
def hi(obj):
        print("Hi, I am " + obj.name)
class Robot:
    say_hi = hi
x = Robot()
x.name = "Marvin"

Robot.say_hi(x)
```

"say_hi" is called a method. Usually, it will be called like this:

x.say_hi()

It is possible to define methods like this, but you shouldn't do it.

The proper way to do it:

- Instead of defining a function outside of a class definition and binding it to a class attribute, we define a method directly inside (indented) of a class definition.
- A method is "just" a function which is defined inside a class.
- The first parameter is used a reference to the calling instance.
- This parameter is usually called self.
- Self corresponds to the Robot object x.

We have seen that a method differs from a function only in two aspects:

- It belongs to a class, and it is defined within a class
- The first parameter in the definition of a method has to be a reference to the instance, which called the method. This parameter is usually called "self".

As a matter of fact, "self" is not a Python keyword. It's just a naming convention! So C++ or Java programmers are free to call it "this", but this way they are risking that others might have greater difficulties in understanding their code!

Most other object-oriented programming languages pass the reference to the object (self) as a hidden parameter to the methods.

You saw before that the calls Robot.say_hi(x)". and "x.say_hi()" are equivalent. "x.say_hi()" can be seen as an "abbreviated" form, i.e. Python automatically binds it to the instance name. Besides this "x.say_hi()" is the usual way to call methods in Python and in other object oriented languages.

For a Class C, an instance x of C and a method m of C the following three method calls are equivalent:

- type(x).m(x, ...)
- C.m(x, ...)
- x.m(...)

There is more than one thing about this code, which may disturb you, but the essential problem at the moment is the fact that we create a robot and that after the creation, we shouldn't forget about naming it! If we forget it, say_hi will raise an error.

We need a mechanism to initialize an instance right after its creation. This is the __init__-method, which we cover in the next section.

**The __init__ Method:**

We want to define the attributes of an instance right after its creation. __init__ is a method which is immediately and automatically called after an instance has been created. This name is fixed and it is not possible to choose another name. __init__ is one of the so-called magic methods,we will get to know it with some more details later. The __init__ method is used to initialize an instance. There is no explicit constructor or destructor method in Python, as they are known in C++ and Java. The __init__ method can be anywhere in a class definition, but it is usually the first method of a class, i.e. it follows right after the class header.

```
class A:
    def __init__(self):
        print("__init__ has been executed!")
x = A()
```

We add an __init__-method to our robot class:

```
class Robot:
    def __init__(self, name=None):
        self.name = name
    def say_hi(self):
        if self.name:
            print("Hi, I am " + self.name)
        else:
            print("Hi, I am a robot without a name")
x = Robot()
x.say_hi()
y = Robot("Marvin")
y.say_hi()
```

## Constructors in C++

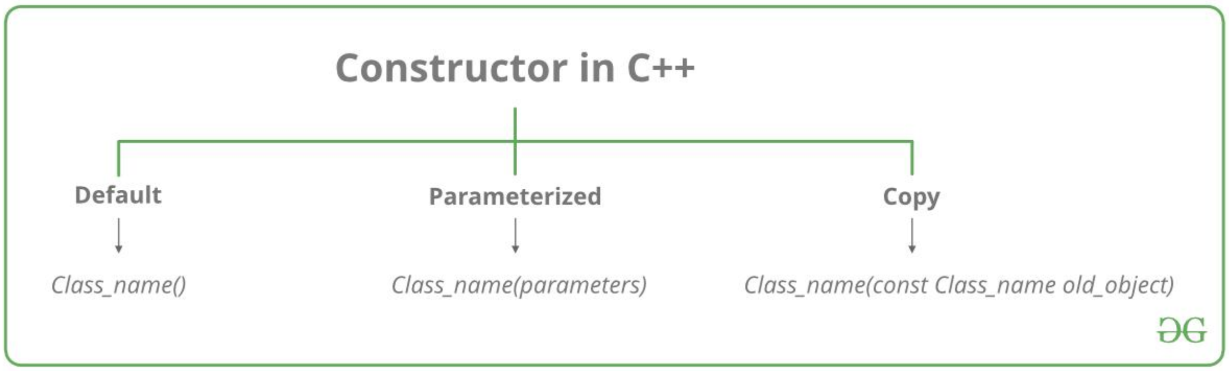
**What is constructor?**

A constructor is a special type of member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class because it does not have any return type.

A method implementing a constructor of a class can't return a value
A class would usually have many attributes (variables) which may not all be initialisable at once. For example with a class Point, having X & Y as attributes, the programmer may want to instantiate a Point with a known value for X but not for Y.
How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- It must be placed in public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).



Let us understand the types of constructors in C++ by taking a real-world example.

Suppose you went to a shop to buy a marker. When you want to buy a marker, what are the options?

1- you go to a shop and say give me a marker. So just saying give me a marker mean that you did not set which brand name and which color, you didn't mention anything just say you want a marker. So when we said just I want a marker so whatever the frequently sold marker is there in the market or in his shop he will simply hand over that. And this is what a default constructor is!

2- The second method you go to a shop and say I want a marker a red in color and XYZ brand. So you are mentioning this and he will give you that marker. So in this case you have given the parameters. And this is what a parameterized constructor is!

3- the third one you go to a shop and say I want a marker like this(a physical marker on your hand). So the shopkeeper will see that marker. Okay, and he will give a new marker for you. So copy of that marker. And that's what copy constructor is!

**Types of Constructors**

**<u>1. Default Constructors:</u>** Default constructor is the constructor which doesn't take any argument. It has no parameters

```
// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
using namespace std;

class construct
{
public:
   int a, b;

   // Default Constructor
   construct()
   {
      a = 10;
      b = 20;
   }
};

int main()
{
   // Default constructor called automatically
   // when the object is created
   construct c;
   cout << "a: " << c.a << endl
```

```
        << "b: " << c.b;
    return 1;
}
```

**What is the significance of the default constructor?**
They are used to create objects, which do not have any having specific initial value.

**Is a default constructor automatically provided?**
If no constructors are explicitly declared in the class, a default constructor is provided automatically.

**Will there be any code inserted by the compiler to the user implemented default constructor behind the scenes?**
The compiler will implicitly declare the default constructor if not provided by the programmer, will define it when in need. The compiler-defined default constructor is required to do certain initialization of class internals.

Consider a class derived from another class with the default constructor, or a class containing another class object with the default constructor. The compiler needs to insert code to call the default constructors of the base class/embedded object.

```
// CPP program to demonstrate Default constructors
#include <iostream>
using namespace std;

class Base {
public:
    // compiler "declares" constructor
};

class A {
public:
    // User defined constructor
    A() { cout << "A Constructor" << endl; }

    // uninitialized
    int size;
};

class B : public A {
    // compiler defines default constructor of B, and
    // inserts stub to call A constructor

    // compiler won't initialize any data of A
};

class C : public A {
public:
    C()
    {
        // User defined default constructor of C
        // Compiler inserts stub to call A's constructor
        cout << "C Constructor" << endl;

        // compiler won't initialize any data of A
    }
};

class D {
public:
    D()
    {
        // User defined default constructor of D
        // a - constructor to be called, compiler inserts
        // stub to call A constructor
        cout << "D Constructor" << endl;

        // compiler won't initialize any data of 'a'
    }

private:
    A a;
};

// Driver Code
int main()
{
    Base base;

    B b;
    C c;
    D d;

    return 0;
}
```

**Note:** Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly

**2. Parameterized Constructors:** It is possible to pass arguments to constructors. When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
Example e = Example(0, 50); // Explicit call

Example e(0, 50);           // Implicit call
```

- **Uses of Parameterized constructor:**
  1. It is used to initialize the various data elements of different objects with different values when they are created.
  2. It is used to overload constructors.
- **Can we have more than one constructor in a class?**
  Yes, It is called Constructor Overloading.

**3. Copy Constructor:** A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on Copy Constructor.

Whenever we define one or more non-default constructors ( with parameters ) for a class, a default constructor( without parameters ) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.

```
// Illustration
#include <iostream>
using namespace std;
 class point
{
private:
  double x, y;
 public:
   // Non-default Constructor &
  // default Constructor
  point (double px, double py)
  {
    x = px, y = py;
  }
};
int main(void)
{
  // Define an array of size
  // 10 & of type point
  // This line will cause error
  point a[10];
   // Remove above line and program
  // will compile without error
  point b = point(5, 6);
}
```

**Constructor Overloading in C++**

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments.This concept is known as Constructor Overloading and is quite similar to function overloading.

- Overloaded constructors essentially have the same name (exact name of the class) and differ by number and type of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

```
// C++ program to illustrate Constructor overloading
#include <iostream>
using namespace std;
 class construct
{
 public:
   float area;
    // Constructor with no parameters
   construct()
   {
      area = 0;
   }
    // Constructor with two parameters
   construct(int a, int b)
   {
      area = a * b;
   }
   void disp()
   {
      cout<< area<< endl;
   }
};
 int main()
{
   // Constructor Overloading with two different constructors of class name
   construct o;
   construct o2( 10, 20);
   o.disp();
   o2.disp();
   return 1;
}
```

A destructor is a special member function of the class. The name of the destructor for a class is the tilde (~) character followed by the class name. The destructor is the complement of the constructor. The general form of defining destructor in the class is given below:

~user_define_name();

The above syntax shows the tilde (~) sign before the constructor's name. The name of the constructor is same as that of the class name. A destructor receives no parameters and returns no value. For example: for the class sample, the destructor defined will be ~ sample ().

**Destructor**

A class destructor is called when an object is destroyed -e.g., when program execution leaves the scope in which an object of that class was initiated. The destructor itself does not actually destroy the object. it performs termination housekeeping before the system reclaims the object's memory so that memory may be reused to hold new objects. Therefore, a destructor is equally useful as is a constructor.

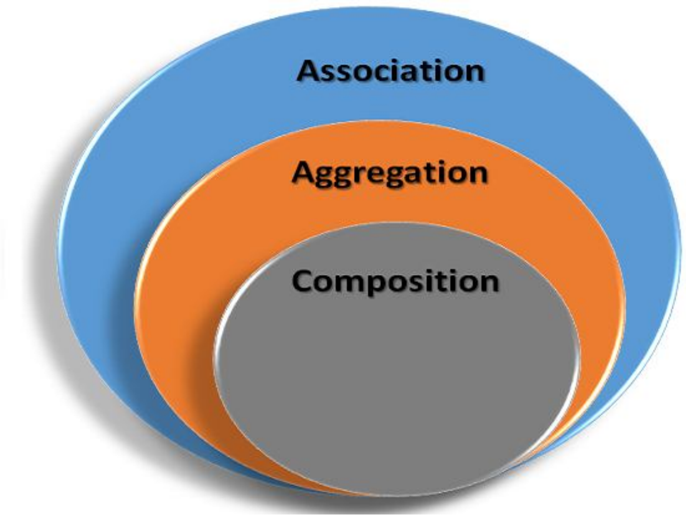
Generally, a destructor is defined under public section of the class, so that its objects can be destroyed in any function. If in case the user fails to define a destructor for a class, the compiler automatically generates one, the default destructor. A class may have only one destructor - destructor over loading is not allowed. The following program shows the use of both constructor and destructor.

**Characteristics of a Destructor**

1. A destructor is invoked automatically by the compiler upon exit from the program and cleans up the memory that is no longer needed.
2. A destructor does not return any value.
3. A destructor cannot be declared as static or const.
4. A destructor must be declared in public section of data.
5. A destructor does not accept arguments and therefore it cannot be overloaded.
6. Destructor methods do not have to be implemented, but if some class attributes have been dynamically allocated during the objects lifecycle, one destructor must be created per class attribute which had been dynamically allocated
7. The destructor method is essential in two cases:
    1. Housekeeping before destroying an object, releasing connections to database, closing network links, etc.
    2. Deallocating dynamically allocated memory from class attributes, this being done explicitly when the programming language has manual memory management (e.g., C++ Operator delete) or implicitly through garbage collector mechanism (e.g., getting an object variable to null in java)

## Aggregation in OOP:

is defined as a relation that exists between two or more two objects which individually have their own individual life cycle along with the ownership. Aggregation in OOPS constitutes 2 words (aggregation and OOPS). An aggregation is a special form of semantically weak relation which happens between unrelatable objects. Aggregation is one of the 5 types of relationships that exist in a Unified modeling language acronym'd as UML.



An aggregate object is one which contains other objects. For example, an Airplane class would contain Engine, Wing, Tail, Crew objects. Sometimes the class aggregation corresponds to physical containment in the model (like the airplane). But sometimes it is more abstract (e.g. Club and Members).
Whereas the test for inheritance is "isa", the test for aggregation is to see if there is a whole/part relationship between two classes ("hasa"). A synonym for this is "part-of".
Being an aggregated object of some class means that objects of the containing class can message you to do work. Aggregation provides an easy way for two objects to know about each other (and hence message each other). *Can the contained object message the container object? Is the relationship symmetrical?*

What we said about constructors holds true for destructors as well. There is no "real" destructor, but something similar, i.e. the method __del__. It is called when the instance is about to be destroyed and if there is no other reference to this instance. If a base class has a __del__() method, the derived class's __del__() method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

The following script is an example with __init__ and __del__:

```
class Robot():
    def __init__(self, name):
        print(name + " has been created!")
    def __del__(self):
        print ("Robot has been destroyed")
if __name__ == "__main__":
    x = Robot("Tik-Tok")
    y = Robot("Jenkins")
    z = x
    print("Deleting x")
    del x
    print("Deleting z")
    del z
    del y
```

**OUTPUT:**

```
Tik-Tok has been created!
Jenkins has been created!
Deleting x
Deleting z
Robot has been destroyed
Robot has been destroyed
```

The usage of the __del__ method is very problematic. If we change the previous code to personalize the deletion of a robot, we create an error:

```
class Robot():
    def __init__(self, name):
        print(name + " has been created!")
    def __del__(self):
        print (self.name + " says bye-bye!")
if __name__ == "__main__":
    x = Robot("Tik-Tok")
    y = Robot("Jenkins")
    z = x
    print("Deleting x")
    del x
    print("Deleting z")
    del z
    del y
```

**OUTPUT:**

```
Tik-Tok has been created!
```

```
Jenkins has been created!
Deleting x
Deleting z
```

We are accessing an attribute which doesn't exist anymore. We will learn later, why this is the case.

**Footnotes**:

1. The picture on the right side is taken in the Library of the Court of Appeal for Ontario, located downtown Toronto in historic Osgoode Hall
2. "Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer", code is also represented by objects.) Every object has an identity, a type and a value." (excerpt from the official Python Language Reference)
3. "attribute" stems from the Latin verb "attribuere" which means "to associate with"
4. Jogger ticketed for trespassing
5. There is a way to access a private attribute directly. In our example, we can do it like this: x._Robot__build_year You shouldn't do this under any circumstances!

## Principles of OOP

The four basic principles of object-oriented programming are:

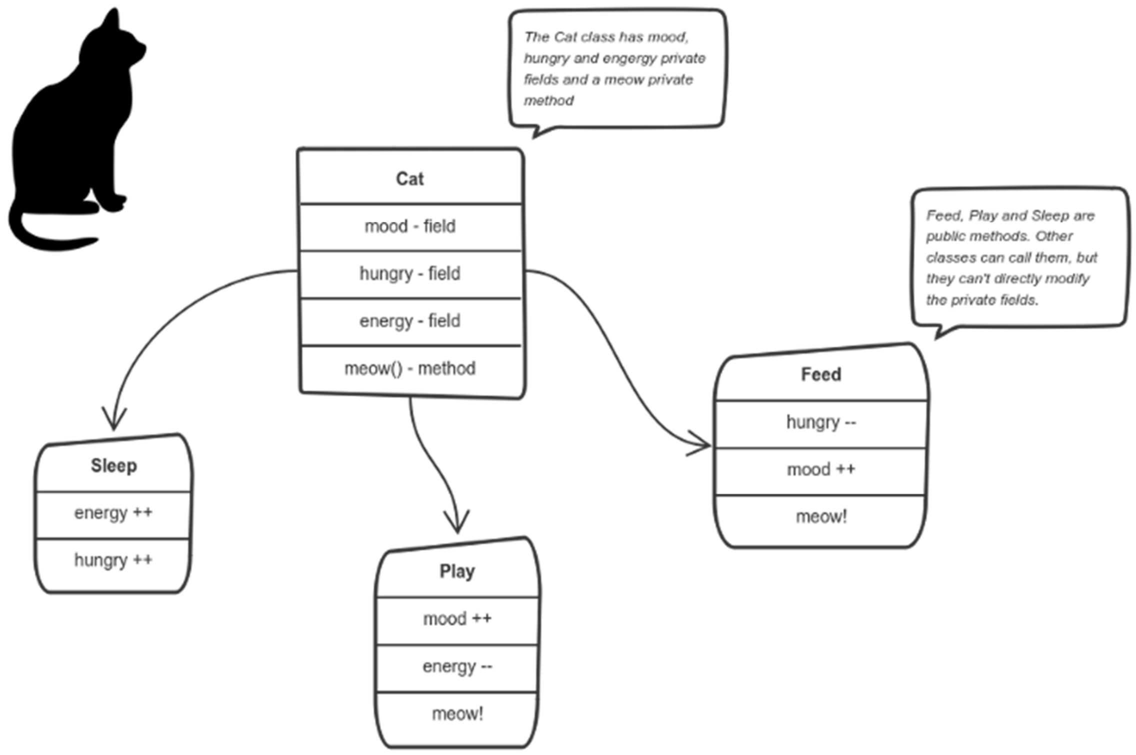
1. **Encapsulation** (the most important properties of OOP)
2. Abstraction,
3. Inheritance,
4. Polymorphism.

### 1- Encapsulation

The principle of encapsulation entails that all the properties and methods of an object is kept private and safe from interference by other objects. In each object we can have both private and public variables and methods. Private variables and methods cannot be called or used by other objects, whereas public ones can. **Encapsulation** is the process in which we protect the internal integrity of data in a class.

Encapsulation is achieved when each object keeps its state **private**, inside a class. Other objects don't have direct access to this state. Instead, they can only call a list of public functions — called methods.

Let's say we're building a tiny Sims game. There are people and there is a cat. They communicate with each other. We want to apply encapsulation, so we encapsulate all "cat" logic into a Cat class. It may look like this:


The Cat class has mood, hungry and engergy private fields and a meow private method
Cat
mood - field
hungry - field
energy - field
meow() - method
Feed, Play and Sleep are public methods. Other classes can call them, but they can't directly modify the private fields.
Sleep
energy ++
hungry ++
Play
mood ++
energy --
meow!
Feed
hungry --
mood ++
meow!

You can feed the cat. But you can't directly change how hungry the cat is.

Here the "state" of the cat is the **private variables** mood, hungry and energy. It also has a private method meow(). It can call it whenever it wants, the other classes can't tell the cat when to meow.
What they can do is defined in the **public methods** sleep(), play() and feed(). Each of them modifies the internal state somehow and may invoke meow(). Thus, the binding between the private state and public methods is made.
This is encapsulation.

Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single _ or double __.

**Example Data Encapsulation in Python**

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

we defined a Computer class.

We used __init__() method to store the maximum selling price of Computer. Here,

Here, we have tried to modify the value of __maxprice outside of the class. However, since __maxprice is a private variable, this modification is not seen on the output.

As shown, to change the value, we have to use a setter function i.e setMaxPrice() which takes price as a parameter.

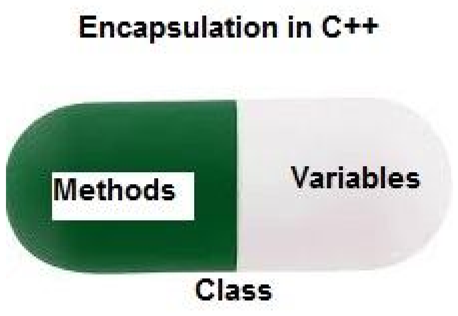
Encapsulation in C++

In normal terms **Encapsulation** is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them

Consider a real-life example of encapsulation,
in a company there are different sections like:

- the accounts section,
- finance section,
  - handles all the financial transactions and keep records of all the data related to finance.
- sales section etc.
  - handles all the sales related activities and keep records of all the sales

Now there may arise a situation when for some reason an official from finance section needs all the data about sales in a particular month.
In this case, he is not allowed to directly access the data of sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of sales section and the employees that can manipulate them are wrapped under a single name "sales section".



Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like sales, finance or accounts is hidden from any other section.

```
// c++ program to explain
// Encapsulation

#include<iostream>
using namespace std;

class Encapsulation
{
   private:
      // data hidden from outside world
      int x;
```

```
    public:
        // function to set value of
        // variable x
        void set(int a)
        {
            x =a;
        }

        // function to return value of
        // variable x
        int get()
        {
            return x;
        }
};

// main function
int main()
{
    Encapsulation obj;

    obj.set(5);

    cout<<obj.get();
    return 0;
}
```

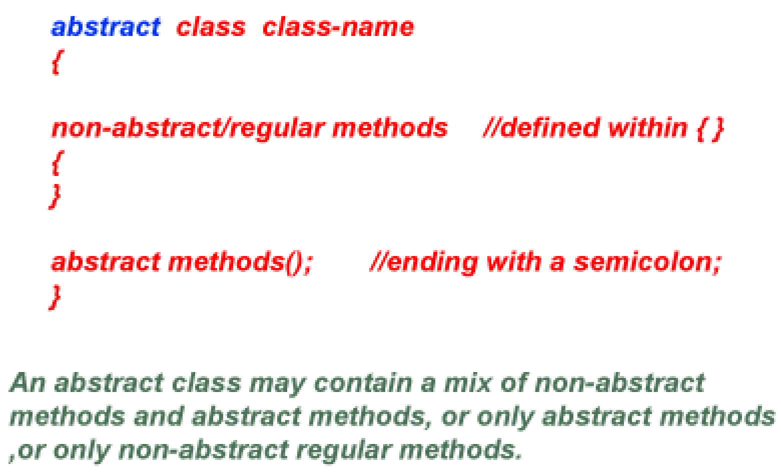
In the above program the variable **x** is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are binded together which is nothing but encapsulation.

**Abstraction in C++**

Data abstraction is one of the most essential and important feature of object oriented programming in C++.
Abstraction means displaying only essential information and hiding the details.
Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

An **abstract class** is declared with the **abstract** keyword. Unlike a regular class in C#, an abstract class may not only contain the **regular methods**, *defined with curly braces { }* but may also contain the **abstract methods**, *ending with a semicolon;* or a mix of **regular** and **abstract methods**.



```
abstract class class-name
{

non-abstract/regular methods    //defined within { }
{
}

abstract methods();     //ending with a semicolon;
}
```

An abstract class may contain a mix of non-abstract methods and abstract methods, or only abstract methods ,or only non-abstract regular methods.

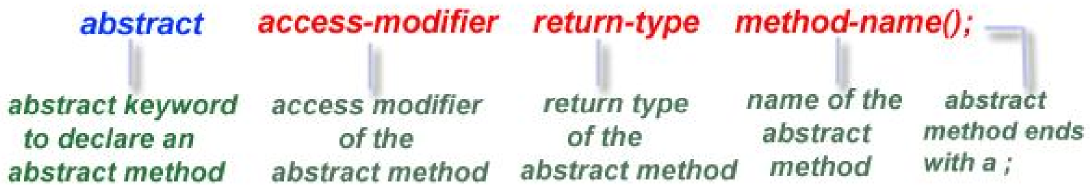
Syntax of Abstract class

*What are abstract methods?*

**Abstract methods** are not implemented in the abstract class and are declared with -:

- An **abstract** keyword.
- An *access modifier*.
- A *return-type* of this abstract method.
- A *method name* and parameters(*if there are any*) to be passed to it.

Please remember, the *internal logic of an abstract method is not provided*, which means that an abstract method is not defined within a pair of curly braces {} and it ends with a semicolon **;**



```
abstract   access-modifier   return-type   method-name();
```

abstract keyword to declare an abstract method
access modifier of the abstract method
return type of the abstract method
name of the abstract method
abstract method ends with a ;

Syntax of abstract method

An example of an abstract method –

**abstract public void add();**

In the example, we have declared a method **add** with an **abstract** keyword, *which makes it an abstract method*. T his method has a **public** access modifier and a **void** return-type, which means that this abstract method will not return any value. And as it should be, this abstract method ends with a **semicolon ;**

- **When a regular class inherits an abstract class, it must implement the abstract methods of an abstract class, or a compile error is issued.**

**Polymorphism**
The word polymorphism means having many forms. In simple words,A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

Polymorphism happens in a classes' inheritance hierarchies.

A method called M1 in classes A, B, C, where B derives from A and C derives from B, has the same signature but a different implementation: M1 is polymorphic.

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle).

However we could use the same method to color any shape. This concept is called

Polymorphism.

```
class Parrot:

    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")

# common interface
```

```
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

In the above program, we defined two classes `Parrot` and `Penguin`. Each of them have a common `fly()` method. However, their functions are different.

To use polymorphism, we created a common interface i.e `flying_test()` function that takes any object and calls the object's `fly()` method. Thus, when we passed the `blu` and `peggy` objects in the `flying_test()` function, it ran effectively.

We're down to the most complex word! Polymorphism means "many shapes" in Greek.
So we already know the power of inheritance and happily use it. But there comes this problem. Say we have a parent class and a few child classes which inherit from it. Sometimes we want to use a collection — for example a list — which contains a mix of all these classes. Or we have a method implemented for the parent class — but we'd like to use it for the children, too.
This can be solved by using polymorphism.

Simply put, polymorphism gives a way to use a class exactly like its parent so there's no confusion with mixing types. But each child class keeps its own methods as they are.
This typically happens by defining a (parent) interface to be reused. It outlines a bunch of common methods. Then, each child class implements its own version of these methods.

Any time a collection (such as a list) or a method expects an instance of the parent (where common methods are outlined), the language takes care of evaluating the right implementation of the common method — regardless of which child is passed.

Take a look at a sketch of geometric figures implementation. They reuse a common interface for calculating surface area and perimeter:

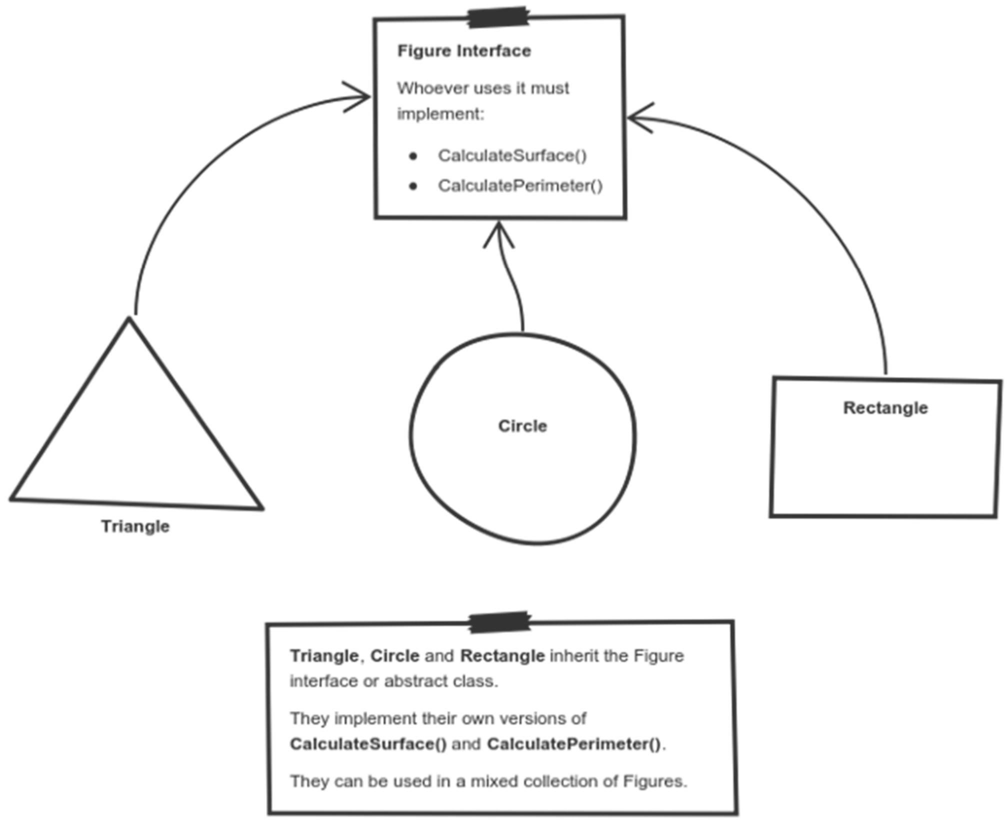


Figure Interface
Whoever uses it must implement:
• CalculateSurface()
• CalculatePerimeter()
Triangle
Circle
Rectangle
Triangle, Circle and Rectangle inherit the Figure interface or abstract class.
They implement their own versions of CalculateSurface() and CalculatePerimeter().
They can be used in a mixed collection of Figures.

Triangle, Circle, and Rectangle now can be used in the same collection
Having these three figures inheriting the parent Figure Interface lets you create a list of mixed triangles, circles, and rectangles. And treat them like the same type of object.
Then, if this list attempts to calculate the surface for an element, the correct method is found and executed. If the element is a triangle, triangle's CalculateSurface() is called. If it's a circle — then circle's CalculateSurface() is called. And so on.
If you have a function which operates with a figure by using its parameter, you don't have to define it three times — once for a triangle, a circle, and a rectangle.

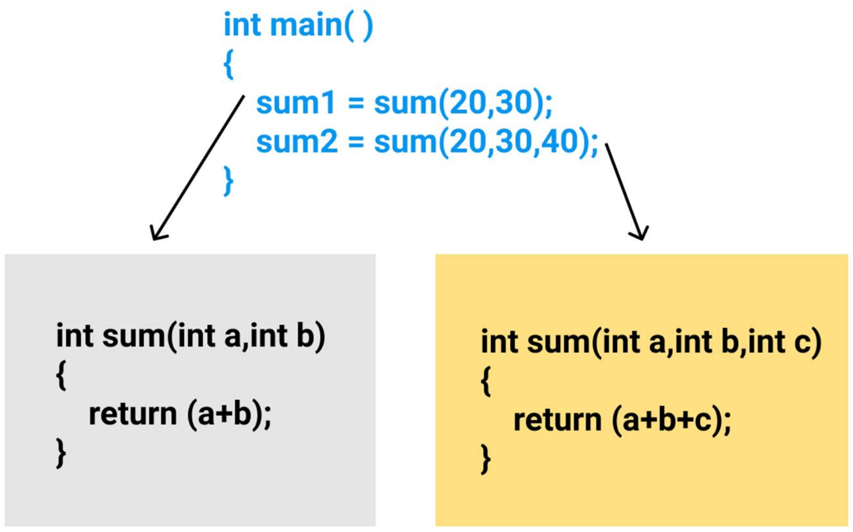
You can define it once and accept a Figure as an argument. Whether you pass a triangle, circle or a rectangle — as long as they implement CalculateParamter(), their type doesn't matter.
I hope this helped. You can directly use these exact same explanations at job interviews.

If you find something still difficult to understand — don't hesitate to ask in the comments below.

An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- *Operator Overloading*: The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- *Function Overloading*: Function overloading is using a single function name to perform different types of tasks.
  Polymorphism is extensively used in implementing inheritance.



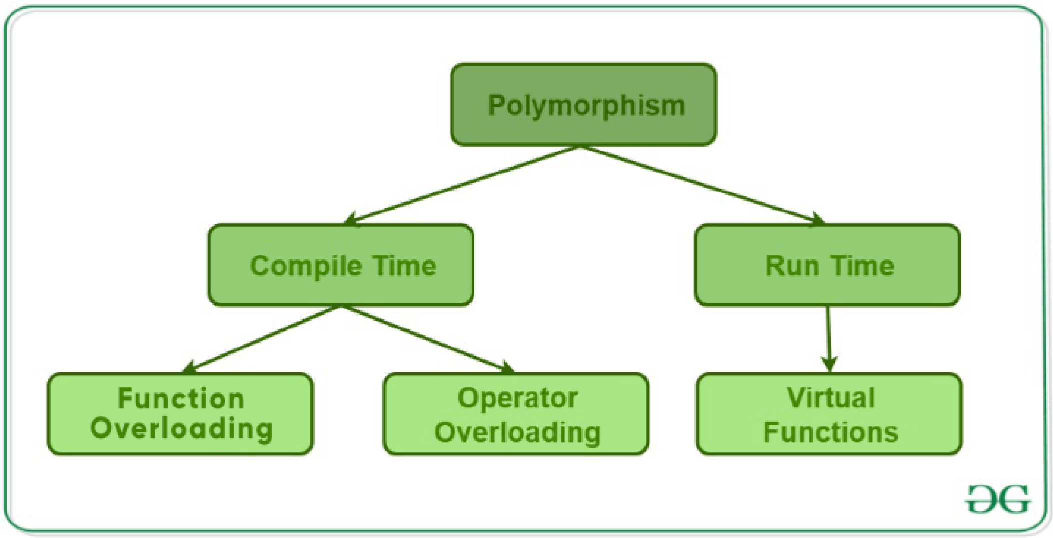
```
int main( )
{
    sum1 = sum(20,30);
    sum2 = sum(20,30,40);
}
```

```
int sum(int a,int b)
{
    return (a+b);
}
```

```
int sum(int a,int b,int c)
{
    return (a+b+c);
}
```

**In C++ polymorphism is mainly divided into two types:**

- Compile time Polymorphism
- Runtime Polymorphism



1- **Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.

Function Overloading: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

In C++, following function declarations **cannot** be overloaded

1) Function declarations that differ only in the return type. For example, the following program fails in compilation.

```
#include<iostream>
int foo() {
  return 10;
}

char foo() {
  return 'a';
}

int main()
{
   char x = foo();
   getchar();
   return 0;
}
```

2) Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.

```
void h(int ());
void h(int (*)()); // redeclaration of h(int())
```

3) Two parameter declarations that differ only in their default arguments are equivalent. For example, following program fails in compilation with error *"redefinition of `int f(int, int)' "*

```
#include<iostream>
#include<stdio.h>
using namespace std;
int f ( int x, int y) {
   return x+10;
}
int f ( int x, int y = 10) {
   return x+y;
}
int main() {
  getchar();
  return 0;
}
```

Consider the following example where a base class has been derived by other two classes –

```
#include <iostream>
using namespace std;

class Shape {
  protected:
    int width, height;

  public:
    Shape( int a = 0, int b = 0){
      width = a;
      height = b;
    }
   virtual  int area() {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
class Rectangle: public Shape {
  public:
    Rectangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
};

class Triangle: public Shape {
  public:
    Triangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};

// Main function for the program
int main() {
  Shape *shape;
  Rectangle rec(10,7);
  Triangle  tri(10,5);
  // store the address of Rectangle
  shape = &rec;

  // call rectangle area.
  shape->area();

  // store the address of Triangle
  shape = &tri;

  // call triangle area.
  shape->area();

  return 0;
}
```

the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Another example, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

```
// Base class
class Animal {
  public:
    void animalSound() {
    cout << "The animal makes a sound \n" ;
  }
};

// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
    cout << "The pig says: wee wee \n" ;
  }
};

// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
```

```
    cout << "The dog says: bow wow \n" ;
  }
};
int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound();
  myPig.animalSound();
  myDog.animalSound();
  return 0;
}
```

**Function Overloading**

Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters.

When a function name is overloaded with different jobs it is called Function Overloading.

In Function Overloading "Function" name should be the same and the arguments should be different.

Function overloading can be considered as an example of polymorphism feature

```
#include <iostream>
using namespace std;

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char const *c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

## Operator Overloading

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

### A simple and complete example

```
#include <iostream>
using namespace std;

class Complex {
private:
   int real, imag;
public:
   Complex(int r = 0, int i = 0)  {real = r;   imag = i;}
   // This is automatically called when '+' is used with between two Complex objects
   Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
   }
   void print() { cout << real << " + i" << imag << endl; }
};
int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2;
   c3.print();
}
```

### Can we overload all operators?

Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

```
. (dot)

  ::

  ?:

  sizeof
```

**Overloading**
*overloading* is what happens when you have two methods with the same name but different signatures. At *compile time*, the compiler works out which one it's going to call, based on the compile time types of the arguments and the target of the method call.

Overloaded methods may arise within classes or even outside the context of a class, should the programming language allows methods implementation outside classes (e.g. C++).
A method M1 can be overloaded N times assuming the N+1 methods' signatures bear the same names but different formal parameters.
Overloaded methods share the same algorithm, processed with different input (formal) parameters.

Simple cases

```
using System;

class Test
{
    static void Foo(int x)
    {
        Console.WriteLine("Foo(int x)");
    }

    static void Foo(string y)
    {
        Console.WriteLine("Foo(string y)");
    }

    static void Main()
    {
        Foo("text");
    }
}
```

Multiple parameters

```
using System;

class Test
{
    static void Foo(int x, int y)
    {
        Console.WriteLine("Foo(int x, int y)");
    }

    static void Foo(int x, double y)
    {
        Console.WriteLine("Foo(int x, double y)");
    }

    static void Foo(double x, int y)
    {
        Console.WriteLine("Foo(double x, int y)");
    }

    static void Main()
    {
        Foo(5, 10);
    }
}
```

Method Overloading is creating a method with the **same name as an existing method in a class**. Hence in simple words, method overloading allows us to have **multiple versions of a method within a class**

*Why method overloading is used?*

Method overloading is mainly used in a situation when we want to create multiple specialized versions of a method in a class, with each version having the same name and doing some specific specialized task.

*How method overloading is achieved?*

- Multiple versions of a method can be created in the same class.
- Multiple versions of a method can be created in a subclass, through inheritance(explained at the end of this article).

*Rules to method overloading -*

1. **Overloaded methods *MUST* have different arguments.**
2. **Overloaded methods *may/may not* have different return types.**

Inheritance
OK, we saw how encapsulation and abstraction can help us develop and maintain a big codebase.

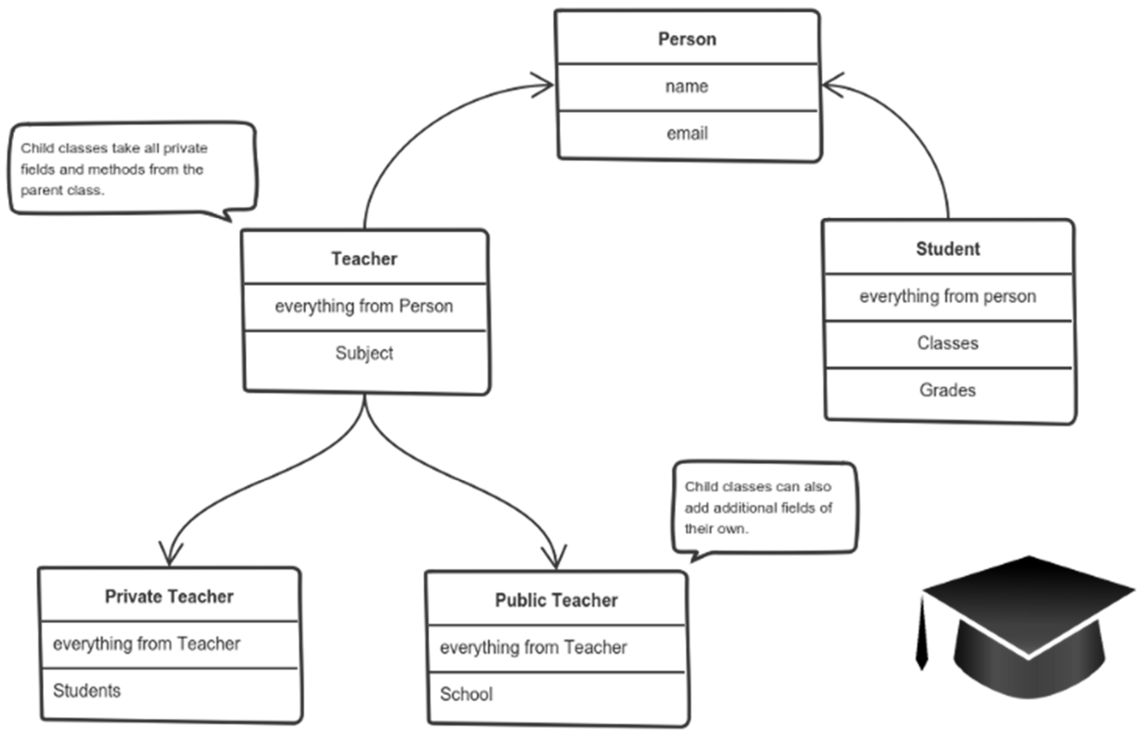
But do you know what is another common problem in OOP design?

Objects are often very similar. They share common logic. But they're not **entirely** the same. Ugh…
So how do we reuse the common logic and extract the unique logic into a separate class? One way to achieve this is inheritance.

It means that you create a (child) class by deriving from another (parent) class. This way, we form a hierarchy.

The child class reuses all fields and methods of the parent class (common part) and can implement its own (unique part).

For example:



A private teacher is a type of teacher. And any teacher is a type of Person.
If our program needs to manage public and private teachers, but also other types of people like students, we can implement this class hierarchy.

This way, each class adds only what is necessary for it while reusing common logic with the parent classes.

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

```
# parent class
class Bird:

  def __init__(self):
     print("Bird is ready")
```

```
    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):

    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")

    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from the swim() method. Again, the child class modified the behavior of the parent class. We can see this from the whoisThis() method. Furthermore, we extend the functions of the parent class, by creating a new run() method.

Additionally, we use the super() function inside the __init__() method. This allows us to run the __init__() method of the parent class inside the child class.

## Inheritance

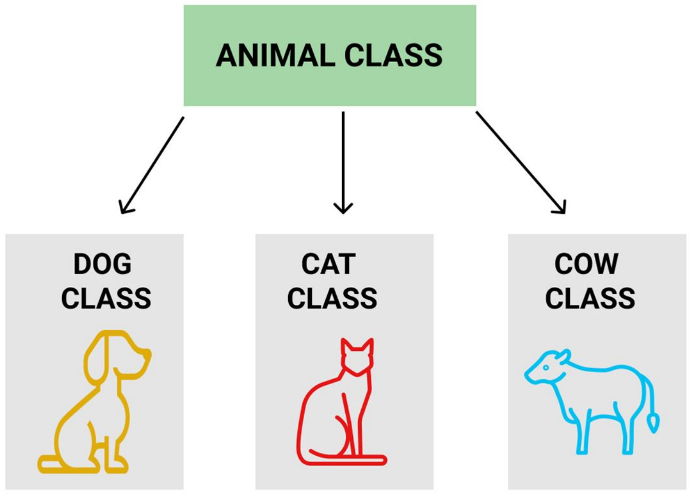
Inheritance can cause a confusing effect. When the compiler goes looking for instance method overloads, it considers the compile-time class of the "target" of the call, and looks at methods declared there. If it can't find anything suitable, it then looks at the parent class... then the grandparent class, etc. This means that if there are two methods at different levels of the hierarchy, the "deeper" one will be chosen first, even if it isn't a "better function member" for the call. Here's a fairly simple example:

```
using System;

class Parent
{
   public void Foo(int x)
   {
      Console.WriteLine("Parent.Foo(int x)");
   }
}

class Child : Parent
{
   public void Foo(double y)
   {
      Console.WriteLine("Child.Foo(double y)");
   }
}


class Test
{
   static void Main()
   {
      Child c = new Child();
      c.Foo(10);
   }
}
```

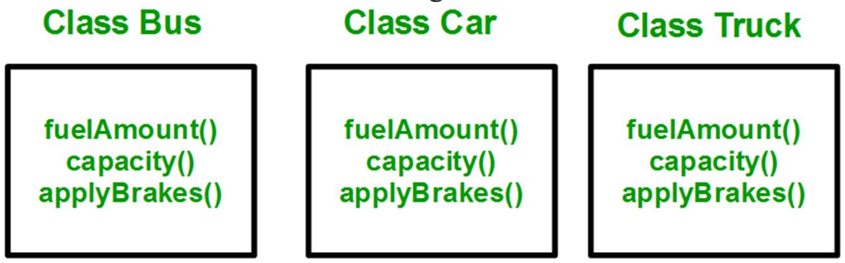
Inheritance mechanisms, which can be summarised as sharing properties (attributes) and abilities (methods), are extremely useful to position data and code in the right locations of a class hierarchy to maximise code reuse.
An almost universal example (except in C++ !!) is for a programming language to have an "Object" class, made mother of all classes, in which a toString() method is provided and will be available in all subclasses.
Inheritance is deeply intertwined with polymorphism, which allows inherited methods to behave differently across the classes' hierarchy.

**Inheritance in C++**
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object-Oriented Programming.
**Sub Class:** The class that inherits properties from another class is called Sub class or Derived

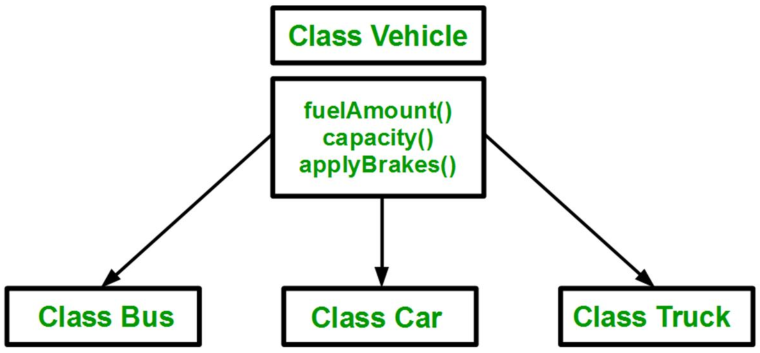
Class.
**Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class.
**Example**: Dog, Cat, Cow can be Derived Class of Animal Base Class.



**Why and when to use inheritance?**
Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.
**Note**: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

```
// C++ program to demonstrate implementation
// of Inheritance

#include <bits/stdc++.h>
using namespace std;

//Base class
class Parent
{
   public:
     int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
   public:
     int id_c;
};

//main function
int main()
   {

       Child obj1;

       // An object of class child has all data members
       // and member functions of class parent
       obj1.id_c = 7;
       obj1.id_p = 91;
       cout << "Child id is " <<  obj1.id_c << endl;
```

```
    cout << "Parent id is " <<  obj1.id_p << endl;

    return 0;
}
```

In the above program the ‘Child’ class is publicly inherited from the ‘Parent’ class so the public data members of the class ‘Parent’ will also be inherited by the class ‘Child’.

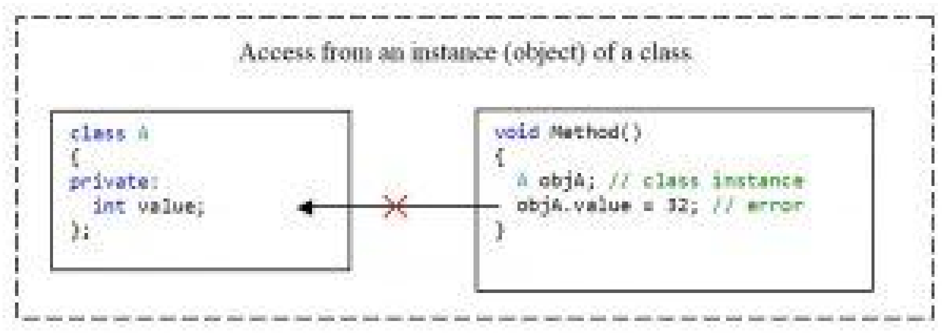
**Modes of Inheritance**

1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class

2.1. The access modifier private. Features of use

If an element (function, data member) is declared in a class in the private section, then the following access rules apply to it:

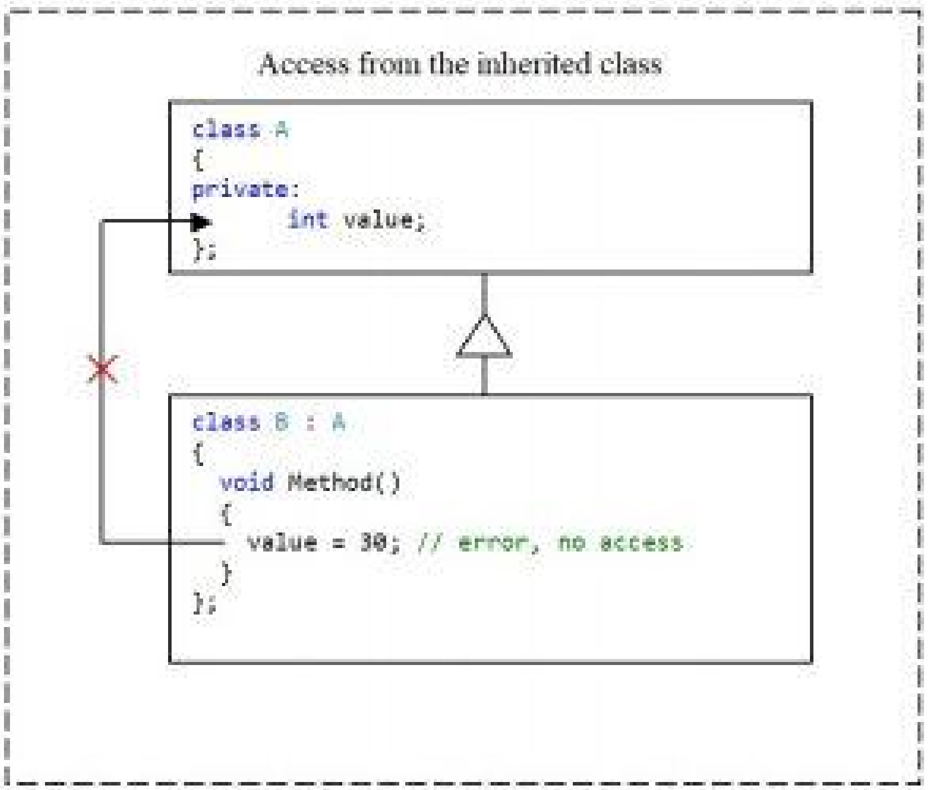
- the element is inaccessible to any instances of methods of other classes or methods that are not “friendly” to the class (Figure 1);
- the element is inaccessible from inherited classes (Figure 2);
- the element is accessible from methods that are implemented in the class (Figure 3);
- the element is accessible from friendly functions (Figure 4);
- the element is accessible from methods of friendly classes (Figure 5).



```
class A
{
private:
  int value;
};
```

```
void Method()
{
  A objA; // class instance
  objA.value = 32; // error
}
```

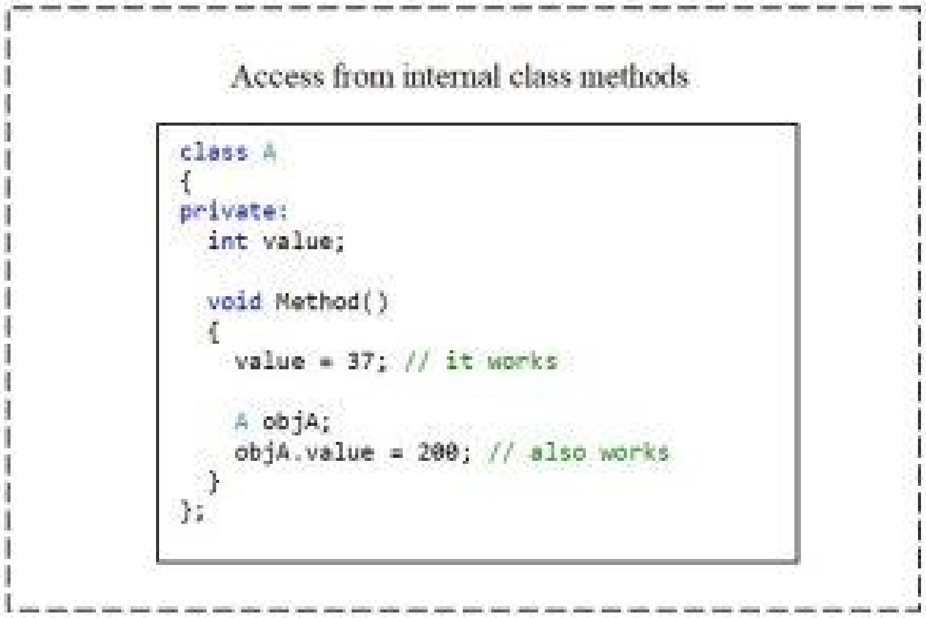
Figure 1. The private access modifier. No access from the class instance to the private-element value of the class



```
class A
{
private:
      int value;
};
```

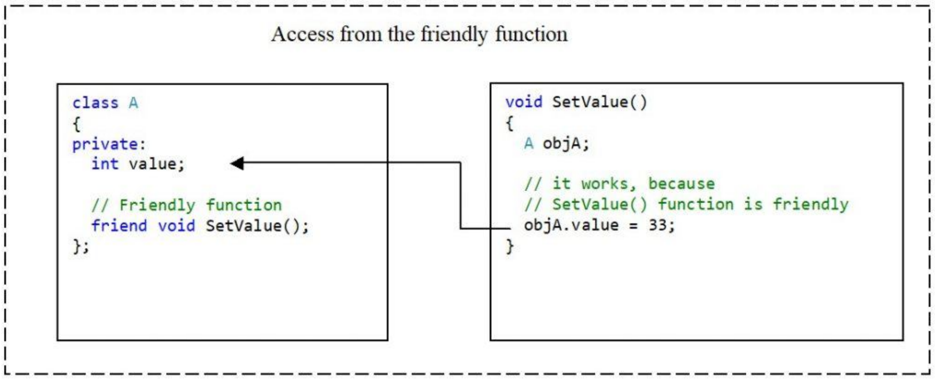
```
class B : A
{
  void Method()
  {
    value = 30; // error, no access
  }
};
```

Figure 2. The private access modifier. There is no access to the private-member of the class from the inherited class



```
class A
{
private:
  int value;

  void Method()
  {
    value = 37; // it works

    A objA;
    objA.value = 200; // also works
  }
};
```

Figure 3. The private access modifier. Access to the elements of the class from the internal class method



```
class A
{
private:
  int value;

  // Friendly function
  friend void SetValue();
};
```

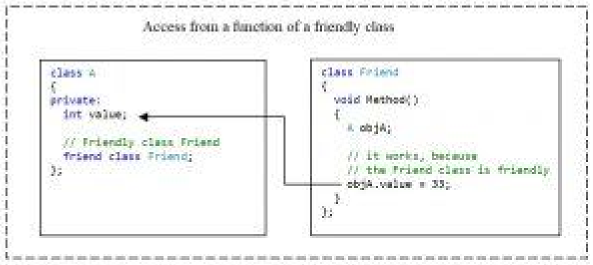
```
void SetValue()
{
  A objA;

  // it works, because
  // SetValue() function is friendly
  objA.value = 33;
}
```

Figure 4. The private access modifier. Access to a class member from a friendly function (method)

If in the code shown in Figure 4, in the declaration of the class A, before the name of the SetValue() function, the keyword friend is removed, then the access to the variable value of the class will not be available. As a result, in the SetValue() function in the line
objA.value = 33;

the compiler will throw an error like

Member A::value is inaccessible



```
class A
{
private:
  int value;

  // Friendly class Friend
  friend class Friend;
};
```

```
class Friend
{
  void Method()
  {
    A objA;

    // it works, because
    // the Friend class is friendly
    objA.value = 33;
  }
};
```

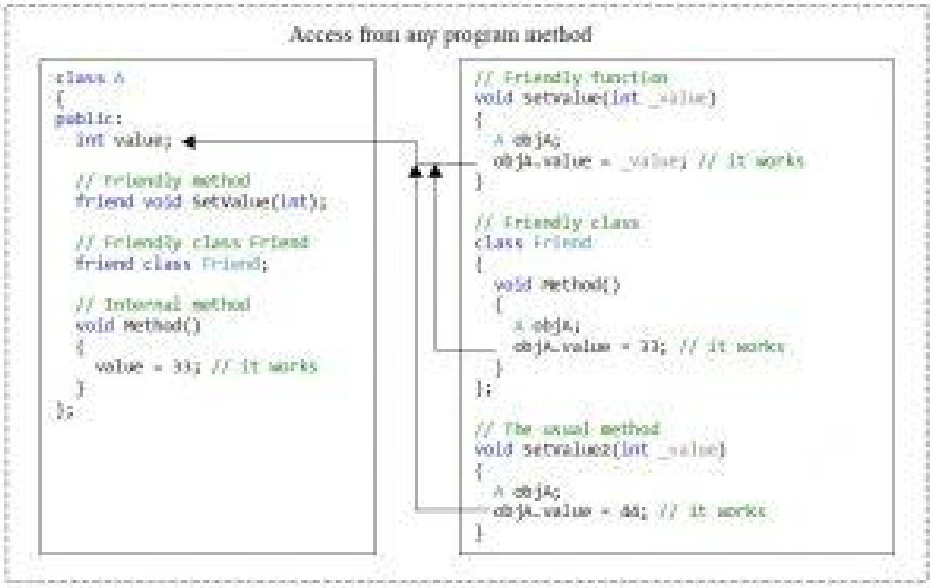
Figure 5. The private access modifier. Access to a private member of a class from a method of a friendly class

. Access modifier public. Features of use

If an element with the public access modifier is defined in a class, then the rule is true:

- the element is available to all methods in the program.

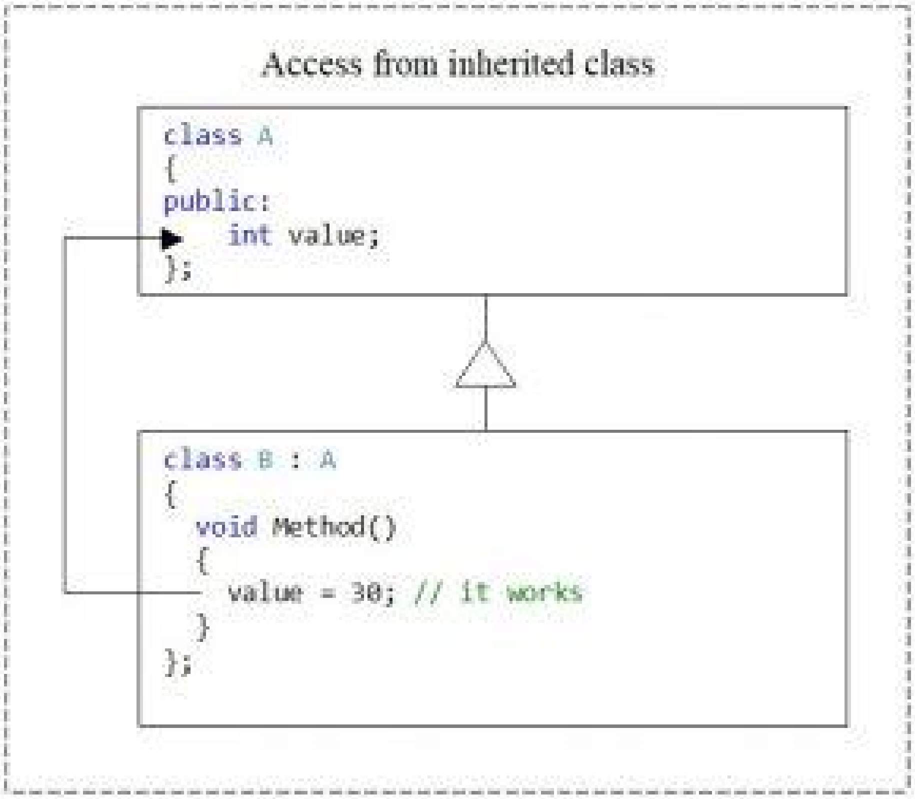
An exception is the case when the class is inherited as private. Then even public members of this class will not be available in inherited classes. You can read more about using class access modifiers **here**.



```
class A
{
public:
  int value;

  // Friendly method
  friend void SetValue(int);

  // Friendly class Friend
  friend class Friend;

  // Internal method
  void Method()
  {
    value = 33; // it works
  }
};
```

```
// Friendly function
void SetValue(int _value)
{
  A objA;
  objA.value = _value; // it works
}

// Friendly class
class Friend
{
  void Method()
  {
    A objA;
    objA.value = 33; // it works
  }
};

// The usual method
void SetValue2(int _value)
{
  A objA;
  objA.value = 44; // it works
}
```

Figure 6. The public access modifier. Access to the elements of the class from any method in the program



```
class A
{
public:
    int value;
};
```

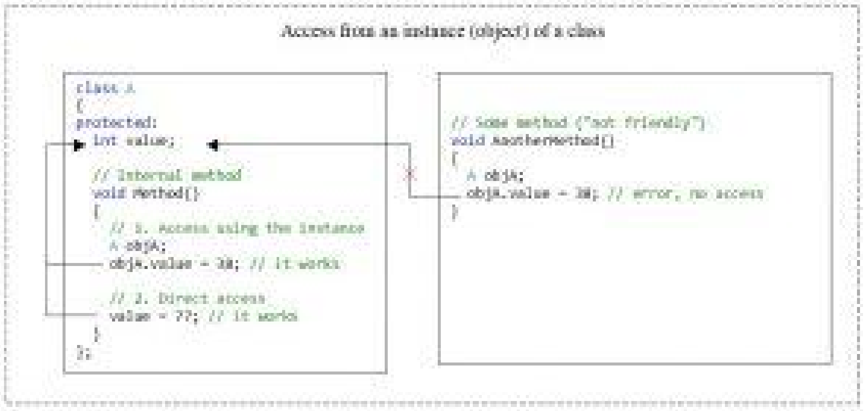
```
class B : A
{
  void Method()
  {
    value = 30; // it works
  }
};
```

Figure 7. The public access modifier. Access from an inherited class

## 2.3. The protected access modifier. Features of use

The protected access modifier is relevant in cases where classes form an inheritance hierarchy. If an element (function, data member) with the protected access modifier is declared in a class, then the following rules apply to it:

- a class element is not accessible from any external method (Figure 8) if this method is not friendly;
- the class element is accessible from the internal methods of the class (Figure 8). It should be noted here that a protected-element of a class is also accessible from an instance of a class, if this instance is declared in an internal method of the class;
- the element of the class is accessible from the friendly functions of the class (Figure 9);
- a class element is accessible from methods of a friendly class (Figure 9);
- the class element is accessible from the methods of the inherited class (Figure 10);
- the class element is inaccessible from instances of the inherited class (Figure 11). This rule does not apply to "friendly" methods and methods of "friendly" classes.



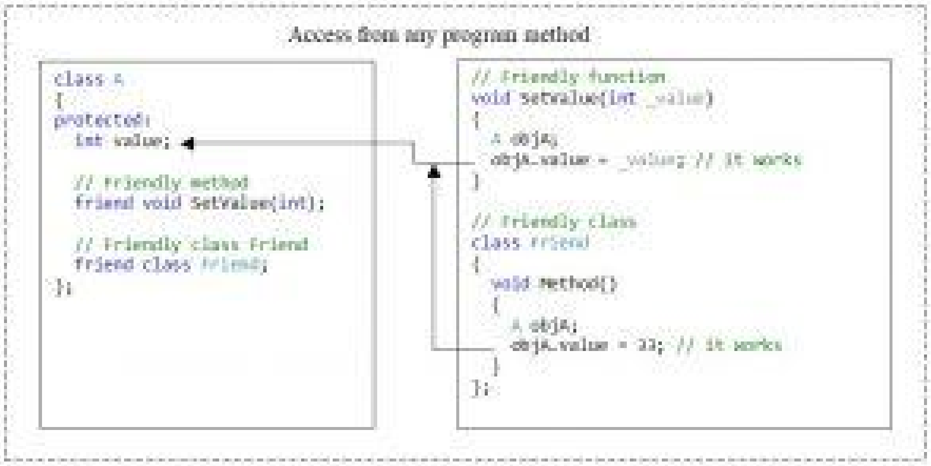
```
class A
{
protected:
  int value;

  // Internal method
  void Method()
  {
    // 1. Access using the instance
    A objA;
    objA.value = 38; // it works

    // 2. Direct access
    value = 77; // it works
  }
};
```

```
// Some method ("not friendly")
void AnotherMethod()
{
  A objA;
  objA.value = 38; // error, no access
}
```

Figure 8. The protected access modifier. Access is denied from a class instance if this instance is created in an "unfriendly" method

Figure 8 demonstrates two types of access to a protected class element:

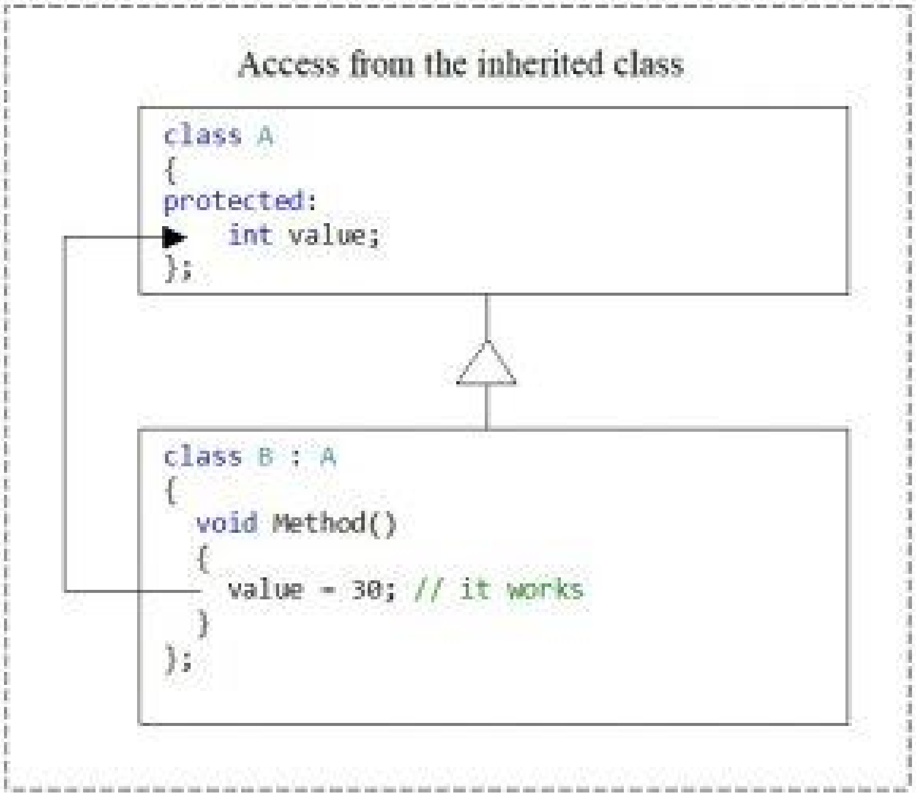
- through an instance (object) of the class, which is declared in the internal Method() method of the class;
- direct access as a member of the class data.



```
class A
{
protected:
  int value;

  // Friendly method
  friend void SetValue(int);

  // Friendly class Friend
  friend class Friend;
};
```

```
// Friendly function
void SetValue(int _value)
{
  A objA;
  objA.value = _value; // it works
}

// Friendly class
class Friend
{
  void Method()
  {
    A objA;
    objA.value = 33; // it works
  }
};
```

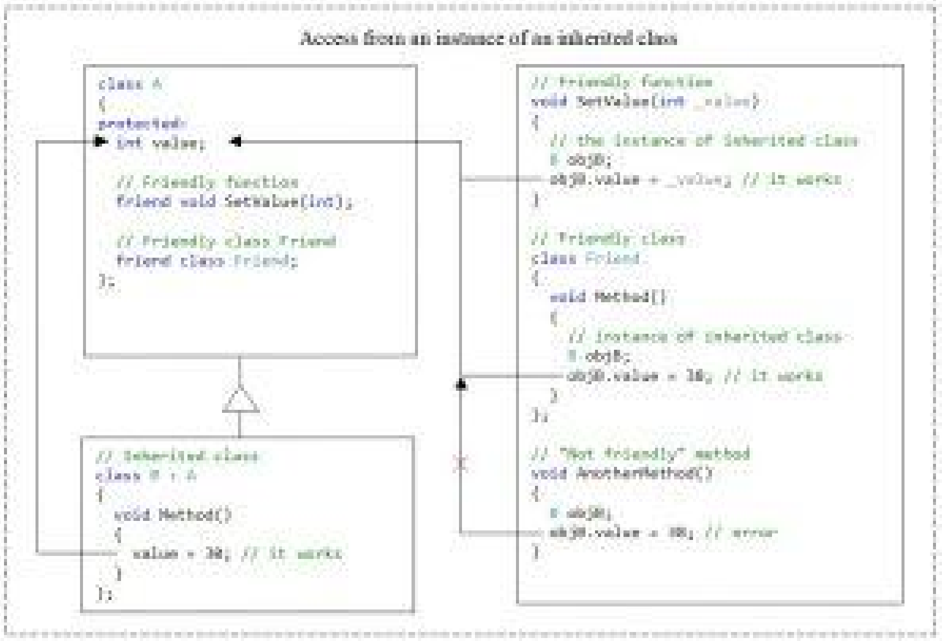
Figure 9. The protected access modifier. Access from “friendly” methods and methods of “friendly” classes



```
class A
{
protected:
    int value;
};
```

```
class B : A
{
  void Method()
  {
    value = 30; // it works
  }
};
```

Figure 10. The protected access modifier. Access from methods of an inherited class



```
class A
{
protected:
  int value;

  // Friendly function
  friend void SetValue(int);

  // Friendly class Friend
  friend class Friend;
};
```

```
// Inherited class
class B : A
{
  void Method()
  {
    value = 10; // it works
  }
};
```

```
// Friendly function
void SetValue(int _value)
{
  // the instance of inherited class
  B objB;
  objB.value = _value; // it works
}

// Friendly class
class Friend
{
  void Method()
  {
    // instance of inherited class
    B objB;
    objB.value = 10; // it works
  }
};

// "Not friendly" method
void AnotherMethod()
{
  B objB;
  objB.value = 10; // error
}
```

Figure 11. The protected access modifier. No access from inherited class instances

**Public, - Protected-, and Private Attributes**

Who doesn't know those trigger-happy farmers from films? Shooting as soon as somebody enters their property. This "somebody" has of course neglected the "no trespassing" sign, indicating that the land is private property. Maybe he hasn't seen the sign, maybe the sign is hard to be seen? Imagine a jogger, running the same course five times a week for more than a year, but then he receives a $50 fine for trespassing in the Winchester Fells. Trespassing is a criminal offence in Massachusetts. He was innocent anyway, because the signage was inadequate in the area**.

Even though no trespassing signs and strict laws do protect the private property, some surround their property with fences to keep off unwanted "visitors". Should the fence keep the dog in the yard or the burglar on the street? Choose your fence: Wood panel fencing, post-and-rail fencing, chain-link fencing with or without barbed wire and so on.

We have a similar situation in the design of object-oriented programming languages. The first decision to take is how to protect the data which should be private. The second decision is what to do if trespassing, i.e. accessing or changing private data, occurs. Of course, the private data may be protected in a way that it can't be accessed under any circumstances. This is hardly possible in practice, as we know from the old saying "Where there's a will, there's a way"!

Some owners allow a restricted access to their property. Joggers or hikers may find signs like "Enter at your own risk". A third kind of property might be public property like streets or parks, where it is perfectly legal to be.

We have the same classification again in object-oriented programming:

- Private attributes should only be used by the owner, i.e. inside of the class definition itself.
- Protected (restricted) Attributes may be used, but at your own risk. Essentially, they should only be used under certain conditions.
- Public Attributes can and should be freely used.

Python uses a special naming scheme for attributes to control the accessibility of the attributes. So far, we have used attribute names, which can be freely used inside or outside of a class definition, as we have seen. This corresponds to public attributes of course. There are two ways to restrict the access to class attributes:

- First, we can prefix an attribute name with a leading underscore "_". This marks the attribute as protected. It tells users of the class not to use this attribute unless, they write a subclass. We will learn about inheritance and subclassing in the next chapter of our tutorial.
- Second, we can prefix an attribute name with two leading underscores "__". The attribute is now inaccessible and invisible from outside. It's neither possible to read nor write to those attributes except inside the class definition itself*.

To summarize the attribute types:

| **Naming** | **Type** | **Meaning** |
|---|---|---|
| name | Public | These attributes can be freely used inside or outside a class definition. |
| _name | Protected | Protected attributes should not be used outside the class definition, unless inside a subclass definition. |
| __name | Private | This kind of attribute is inaccessible and invisible. It's neither possible to read nor write to those attributes, except inside the class definition itself. |

We want to demonstrate the behaviour of these attribute types with an example class:

```
class A():
    def __init__(self):
        self.__priv = "I am private"
        self._prot = "I am protected"
        self.pub = "I am public"
```

We store this class (attribute_tests.py) and test its behaviour in the following interactive Python shell:

```
from attribute_tests import A
x = A()
x.pub
```

**OUTPUT:**

```
    'I am public'
```

```
x.pub = x.pub + " and my value can be changed"
x.pub
```

**OUTPUT:**

```
    'I am public and my value can be changed'
```

```
x._prot
```

**OUTPUT:**

```
    'I am protected'
```

```
x.__priv
```

**OUTPUT:**

```
    ---------------------------------------------------------------------------
    AttributeError                            Traceback (most recent call last)
    <ipython-input-6-f75b36b98afa> in <module>
    ----> 1x.__priv
    AttributeError: 'A' object has no attribute '__priv'
```

The error message is very interesting. One might have expected a message like "__priv is private". We get the message "AttributeError: 'A' object has no attribute __priv instead, which looks like a "lie". There is such an attribute, but we are told that there isn't. This is perfect information hiding. Telling a user that an attribute name is private, means that we make some information visible, i.e. the existence or non-existence of a private variable.

Our next task is rewriting our Robot class. Though we have Getter and Setter methods for the name and the build_year, we can access the attributes directly as well, because we have defined them as public attributes. Data Encapsulation means, that we should only be able to access private attributes via getters and setters.

We have to replace each occurrence of self.name and self.build_year by self.__name and self.__build_year.

The listing of our revised class:

```
class Robot:
    def __init__(self, name=None, build_year=2000):
        self.__name = name
        self.__build_year = build_year
    def say_hi(self):
        if self.__name:
            print("Hi, I am " + self.__name)
        else:
            print("Hi, I am a robot without a name")
    def set_name(self, name):
        self.__name = name
    def get_name(self):
        return self.__name
    def set_build_year(self, by):
        self.__build_year = by
    def get_build_year(self):
        return self.__build_year
    def __repr__(self):
        return "Robot('" + self.__name + "', " +  str(self.__build_year) +  ")"
    def __str__(self):
        return "Name: " + self.__name + ", Build Year: " +  str(self.__build_year)
if __name__ == "__main__":
    x = Robot("Marvin", 1979)
    y = Robot("Caliban", 1943)
    for rob in [x, y]:
        rob.say_hi()
        if rob.get_name() == "Caliban":
            rob.set_build_year(1993)
        print("I was built in the year " + str(rob.get_build_year()) + "!")
```

**OUTPUT:**

```
Hi, I am Marvin
I was built in the year 1979!
Hi, I am Caliban
I was built in the year 1993!
```

Every private attribute of our class has a getter and a setter. There are IDEs for object-oriented programming languages, who automatically provide getters and setters for every private attribute as soon as an attribute is created.

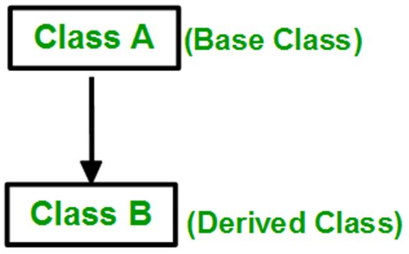
This may look like the following class:

```
class A():
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
    def GetX(self):
        return self.__x
    def GetY(self):
        return self.__y
    def SetX(self, x):
        self.__x = x
    def SetY(self, y):
        self.__y = y
```

There are at least two good reasons against such an approach. First of all not every private attribute needs to be accessed from outside. Second, we will create non-pythonic code this way, as you will learn soon.

**Types of Inheritance in C++**

**1. Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.

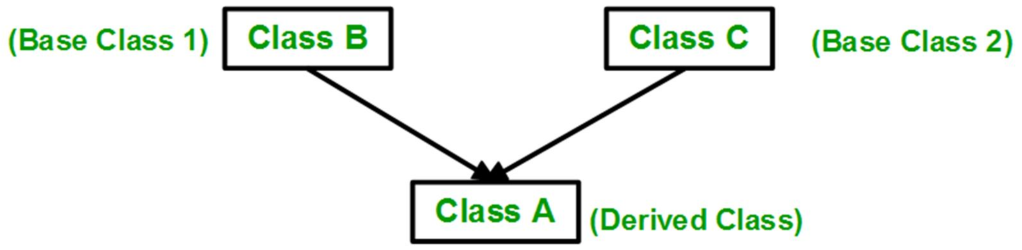


```
class subclass_name : access_mode base_class

{

  //body of subclass

};
```

```
// C++ program to explain
// Single inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from a single base classes
class Car: public Vehicle{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
```

```
   Car obj;
   return 0;
}
```

3. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



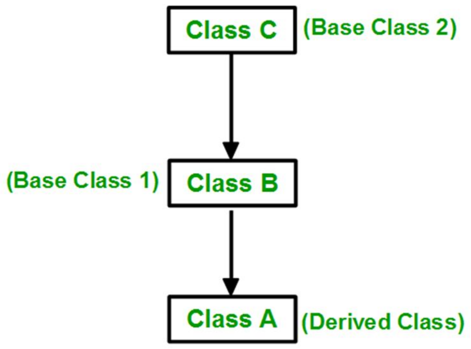
```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  //body of subclass
};
```

Here, the number of base classes will be separated by a comma (', ') and access mode for every base class must be specified.

```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// second base class
class FourWheeler {
  public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};
```

```
// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

// main function
int main()
{
   // creating object of sub class will
   // invoke the constructor of base classes
   Car obj;
   return 0;
}
```

4. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.



```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
   Vehicle()
   {
     cout << "This is a Vehicle" << endl;
   }
};

// first sub_class derived from class vehicle
class fourWheeler: public Vehicle
{  public:
   fourWheeler()
   {
      cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from the derived base class fourWheeler
class Car: public fourWheeler{
   public:
     Car()
     {
       cout<<"Car has 4 Wheels"<<endl;
     }
};

// main function
int main()
{
    //creating object of sub class will
    //invoke the constructor of base classes
    Car obj;
    return 0;
}
```

**C++ Aggregation Definition:** In C++, aggregation is a process in which one class (as an entity reference) defines another class. It provides another way to reuse the class. It represents a **HAS-A** relationship association, or it has class and relationship.
**Note**: HAS-A relation simply means dynamic or run-time binding.
To qualify as an aggregation, an object must define the following relationships:

1. The member must be a part of a class.

- A member can belong to or more classes at a time.
- The member does not have any existence managed by the object.
- The member is unknown about the object's existence.
- The relationship is uni-directional.

Collectively, aggregation is a part-whole relationship, where the parts (member) are contained within the entire uni-directional relationship. However, unlike composition, members can belong to one object at a time. The entire object doesn't need to be responsible for the existence and lifespan of the members. In simple and sober language, **aggregation is not responsible for creating or destroying the members or parts**.
For instance, consider a relationship between a person and their home address. However, the same address may belong to more than one person at a time in some significant order. Although, it is not managed by the person; the address existed even before the man and will tend to exist even after it. Additionally, a person knows where he/she lives, but the address does not have any information about what person lives. Therefore, it is an example of an aggregate relationship.
Let us dive into the syntax of aggregation.

```
Class PartClass
//instance variables
```

```
//instance methods
class Whole
PartClass*
```

In the above syntax, the whole class represents the class that is a container class. The container class is considered as the class for the other part class, which is again contained in the entire class's object. Here, each object of class as the whole can hold the reference pointer of the part class's object.

**Interface**:
In order to insure the presence of mandatory functionalities in a set of classes C1, C2, ..., Cn, where Cn inherits from C(n-1), a software engineer wants to enforces the whole class hierarchy to implement a set of methods M1, M2, …, Mn , An interface must be defined with M1, M2, …, Mn and applied to each class of the set. The methods are implemented accordingly in each classes.

In order to insure the presence of mandatory functionalities in a set of classes, a software engineer wants to enforces these classes to implement a set of methods M1, M2, …, Mn.

- In absence of explicit interfaces, an abstract class Cabs must be defined with methods M1, M2, …, Mn and then each class of the set must inherit from Cabs.
- An interface must be defined with M1, M2, …, Mn and applied to each class of the set.

MyCode:

```
import math
# 01 - encapsulation : different parameter in one class or unit
class employee:
    #02 - overloading : modify the same object method one time
    def __init__(self, name, salary, project):
        self.name=name
        self.salary=salary
        self.project=project

     # 03 - polymorphism : same object with same parameter but with different algorism
    def __str__(self):
        return "Mr/Ms " + str(self.name) + " is working in " + str(self.project) + " project"

    def show(self):
        print("Name: ", self.name, 'Salary:', self.salary)

    def work(self):
        print(self.name, 'is working on', self.project)

class Point2D:

    def __init__(self,x:float=0.0,y:float=0.0,name:str="O"):
        self.X=float(x)
        self.Y=float(y)
        self.name=name
    def __str__(self):
        return "( " + str(self.X) + " ; " + str(self.Y) + " )"
    def __eq__(self,another):
        if(self.X==another.X and self.Y == another.Y):
            return True
        else:
            return False

class straightLine:
    # 04  - Aggregation : call/use a class inside another class
    def __init__(self,point1:Point2D=None,point2:Point2D=None):
        self.point1=point1
        self.point2=point2
        self.gradientAndIntercept()

    def gradientAndIntercept(self):
        x1=self.point1.X
        x2=self.point2.X

        y1=self.point1.Y
        y2=self.point2.Y
        if (x2-x1 !=0):
            self.gradient=(y2-y1)/(x2-x1)
            self.intercept=y1-self.gradient
        else:
            self.gradient=float("inf")
            self.intercept=float("nan")

class lineSegmant(straightLine):
    def __init__(self,point1:Point2D=None,point2:Point2D=None):
        super().__init__(point1,point2)
        self.computeNorm()
    def computeNorm(self):
        self.norm=math.sqrt((self.point1.X-self.point2.X)**2+(self.point1.Y-self.point2.Y)**2)
    def __lt__(self,anotherLineS):
        return (self.norm < anotherLineS.norm)

test1 = Point2D(3,5,"test1")
test2 = Point2D(8,55,"test2")
print("class directly: ",Point2D)
print("test1 : \n",test1)
test3 = Point2D(9.9,5.5,"test3")
print("test3 : \n",test3)
test3.X="hello"
print("test3 after : \n",test3)

p1=Point2D(4,2,"p1")
print("p1 : \n",p1)
p2=Point2D(5,4,"p2")
print("p2 : \n",p2)
p3=Point2D(4,2)
print("p3 : \n",p3)

if (p1==p3):
    print("p1 == p3 : same")
else:
    print("p1 <> p3 :different")

try:
    stL1=straightLine(p1,p2)
    stL1=straightLine(p1)
except:
    print("unexpected")

emp1=employee("Samy",3000,"eBorder")
emp1.show()
emp1.work()
```

```
print (emp1)

try:
    LS1=lineSegmant(p1,p2)
    LS2=lineSegmant(p3)
except:
    print("unexpected")

pO=Point2D(0,0,"O")
pA=Point2D(5,8,"A")
pB=Point2D(2,4,"B")

LS3=lineSegmant(pO,pA)
LS4=lineSegmant(pO,pB)

if(LS3<LS4):
    print ("LS3 less than LS4")
else:
    print ("LS4 less than LS3")
```

MODULE INSTALLER:ModuleInstaller.py

```
import sys
import subprocess

def isConda()->bool:
    try:
        import conda
    except:
        is_conda=False
    else:
        is_conda=True
    return is_conda

def isPip()->bool:
    try:
        import pip
    except:
        is_pip=False
    else:
        is_pip=True
    return is_pip

def installModule(package:str):
    packageManager="pip"
    if isConda():
        packageManager="conda"
        try:
            subprocess.check_call([sys.executable,"-m",packageManager,"install","-y",package])
        except Exception as e:
            print("not able to install, issue with installation with conda with error: ",e)
    else:
        try:
            subprocess.check_call([sys.executable,"-m",packageManager,"install",package])
        except Exception as e:
            print("not able to install, issue with installation with pip with error: ",e)
```

Content hiding ContentObfuscation.py

```
# 01 import base64
import base64
import myTools.ModuleInstaller as mi

try:
    # 02 import cryptography.fernet
    import cryptography.fernet as f
except:
    mi.installModule("cryptography")
    import cryptography.fernet as f

class ContentObfuscation:

    # 03 generete key
    generate_key = f.Fernet.generate_key()

    # 04 encode that key in base64.b64encode() and save in byts variable as a token
    __fernetk:bytes=base64.b64encode(generate_key)

    print(" test ")
    #
__fernetk:bytes=b'YVlJWHEtSVhSSUtFcTljS3JwcFZrOGhyaXRXa0tsWGNZcW9oQjNobURF
cz0='

    # overwrite __init__ creating cipher variable with encoding the token from the main class in
base64 using fernet
    def __init__(self:object):
        self._cipher_suite = f.Fernet(base64.b64decode(ContentObfuscation.__fernetk))

    # encoding function to encrypt the clear text
    def obfuscate(self:object, clearText:str)-> str:
        return (self._cipher_suite.encrypt(clearText.encode())).decode()
    # dencoding function to dencrypt the clear text
    def deobfuscate(self:object, obfuscatedText:str)-> str:
        return (self._cipher_suite.decrypt(obfuscatedText.encode())).decode()
```

MSSQL_DBConnector.py

```
import myTools.DBConnection as dbc
try:
    import pyodbc
except:
    mi.installModule("pyodbc")
    import pyodbc

class MSSQL_DBConnector(dbc.DBConnector):
    """this class inherits from an abstract class
    DBConnector the pure virtual method, decorated with @abstractmethod
    called selectBestDriverAvailable must be implemented here"""

    def
__init__(self:object,dbserver:str=None,dbname:str=None,dbusername:str=None,dbpassword:str=
None):

        super().__init__(dbserver,dbname,dbusername,dbpassword)

        self.selectBestDriverAvailable()

    def selectBestDriverAvailable(self:object):
        listOfAllAvaillableDriver:list[str]=pyodbc.drivers()
        if(listOfAllAvaillableDriver is not None):
            if("ODBC Driver 17 for SQL Server" in listOfAllAvaillableDriver):
                self._m_driver = "ODBC Driver 17 for SQL Server"
            elif("ODBC Driver 13 for SQL Server" in listOfAllAvaillableDriver):
                self._m_driver = "ODBC Driver 13 for SQL Server"
        return self._m_driver
```

DBConnection.py

```
from abc import ABC, abstractmethod
import myTools.ModuleInstaller as mi
try:
    import pyodbc
except:
    mi.installModule("pyodbc")
    import pyodbc

class DBConnector(ABC):
    """ to make connection """
    def
__init__(self:object,dbserver:str=None,dbname:str=None,dbusername:str=None,dbpassword:str=
None):
        self._m_dbserver = dbserver
        self._m_dbname = dbname
        self._m_dbusername = dbusername
        self._m_dbpassword = dbpassword
        # for Connectivity check, we are not connected to the data source at begining
        self._m_isDBConnectionOpen=False
        # at starting we don't know which driver will be used
        self._m_driver=None

    @abstractmethod # python decorator
    def selectBestDriverAvailable(self:object):
        """ This is virtual method, this will done by a child class of DBConnector"""
        pass

    def open(self:object):
        self._m_conn= pyodbc.connect('DRIVER={'+ self.selectBestDriverAvailable()
+'};SERVER='+ self._m_dbserver +';DATABASE='+ self._m_dbname +';UID='+
self._m_dbusername +';PWD='+ self._m_dbpassword)
        self._m_isDBConnectionOpen=True

    @property # read only property
    def IsConnected(self:object)->bool:
        return self._m_isDBConnectionOpen

    #if we need to set the value like setter
    #@IsConnected.setter
    #def IsConnected(self:object,value:bool):
    #    self._m_isDBConnectionOpen=value

    @property
    def dbUser(self:object)->str:
        return self._m_dbusername
    @dbUser.setter #read/write
    def dbUser(self:object, value:str)->str:
        self._m_dbusername = value

    @property
    def connection(self:object)->str:
        return self._m_conn
```

Decorate_debug

```
#debug example
import functools
def debug(func):
    """Print the function signature and return value"""
    @functools.wraps(func)
    def wrapper_debug(*args, **kwargs):
        args_repr = [repr(a) for a in args]                   # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]  # 2
        signature = ", ".join(args_repr + kwargs_repr)         # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")         # 4
        return value
    return wrapper_debug

@debug
def make_greeting(name, age=None):
    if age is None:
        return f"Howdy {name}!"
    else:
        return f"Whoa {name}! {age} already, you are growing up!"

make_greeting("Benjamin")

# another example for debug
import math

# Apply a decorator to a standard library function
math.factorial = debug(math.factorial)

def approximate_e(terms=18):
    return sum(1 / math.factorial(n) for n in range(terms))

approximate_e(5)

# timer decorator
import time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()    # 1
        value = func(*args, **kwargs)
        end_time = time.perf_counter()      # 2
        run_time = end_time - start_time    # 3
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])

waste_some_time(1)
```

All Days code:

InitialPython:

```
#it seems "empty" with usage of "pass"
#but is it? No, it inherits from the class object
#explicit inheritance required in Python 2.7, not in >=3.0
class myClass:
    pass


a = 50
# a = 50 --> instanciation of an object of class int + a points to this anonymous object
print(id(a)) #id() gives the unique object identifier --> location in RAM of the PyObject for a
#It's the value of the implicit pointer called a

#It's not that straighforward
#a is an implicit pointer to an object of class int
#print really receives a memory address!!
#print receives not particular information (parameter) about what to print on the console
print(a)

#obj1 is an instance of myClass
obj1 = myClass()

#Capture string representation of obj1 in an instance of str
obj1ToStr = obj1.__str__()

print(obj1)
print(myClass)

print(a.__dir__())
```

Pint2D.py

```
import math

class Point2D:
    #In C++, Java, C#, we'd need to define x & y as attributes of the class
    #Python uses automatic variable declaration to maximum extend
    #A variable (an object, instance of a class) can only exist when initialised (constructed)

    # writing a single "x" or "y" in the class definition environment won't work as expected

    #Due to auto var declaration, class attributes must be "declared" in the constructor of the class
    # This is the constructor of a class in Python
    # Only one possible overload of the xtor, __init__ is inherited from the object class
    # Particular kind of method: polymorphic overload
    #polymorphic: inherited from object, implementation with same signature: same name and
same input param
    #One overload allowed (in Python), keep method name, but change its parameters (def of
overloading)

    #Very unique to Python, the parameters of all class methods must precise an explicit "this"
pointer
    # C++, Java (all other OOP) have an implicit "this" pointer
    #Pythonic way is to call it "self", but you don't have to (any name)
    #As long as the first input parameter is understood to be the "this pointer"

    #In order to deal with multiple possible constructions, the input parameters must be defaulted
    # Traditional to set them to None
    def __init__(self, x:float = None, y:float = None, name:str = None):
        #You must remember to use the self pointer explicitely to refer to the class attributes
        if(x is None):
            self.X = 0.0 #by design choice, a None value for x or y means 0 valued
        else:
            self.X = float(x) #call to the xtor of class float, with x as input parameter
        if(y is None):
            self.Y = 0.0
        else:
            self.Y = float(y) #not a type recast !

        if(name is None or name == ""):
            self.name = "O"
        else:
            self.name = name

    def __str__(self):
        return "(" + str(self.X) +";"+ str(self.Y) + ")"
    # Operator == overload
    def __eq__(self, anotherPoint):
        if(self.X == anotherPoint.X and self.Y == anotherPoint.Y):
            return True
        else:
            return False

    #Operator !=
    def __ne__(self, anotherPoint):
        return not(__eq__(self, anotherPoint))


class StraightLine2D:
    def __init__(self, point1:Point2D = None, point2:Point2D = None):
        self.point1 = point1
        self.point2 = point2
        #Compute gradient and intercept based on these two points
        self.computeGradientAndIntercept()

    def computeGradientAndIntercept(self):
        x1 = self.point1.X
        x2 = self.point2.X

        y1 = self.point1.Y
        y2 = self.point2.Y

        if(x2 - x1 != 0):
            self.gradient = (y2 - y1) / (x2 - x1)
            self.intercept = y1 - self.gradient * x1
        else:
            #infinite gradient
            self.gradient = float("inf")
            self.intercept = float("nan")

class LineSegment2D(StraightLine2D):
    #The constructor here __init__ is not straighforward
    ##Same signature, it requires two instances of Point2D
    def __init__(self, point1:Point2D = None, point2:Point2D = None):
        #but you do'nt want to duplicate the code from init in StraightLine
        #We need a way to run the __init__() code from the mother class (StraightLine)
        #use super as an "pointer" to elements of the mother class
        #and then make explicit calls, here call the constructor from the mother class
        super().__init__(point1, point2)
        #We don't need to deal with teh StraightLine aspect of LineSegment
        self.computeLSNorm()
    def computeLSNorm(self):
        #The square root of the sum of squares of coordinates
        #Do we just go for it or do we test whether the points are not None
        self.norm = math.sqrt((self.point1.X - self.point2.X)**2 + (self.point1.Y -
self.point2.Y)**2)

    def __lt__(self, anotherLS):
        #Binary operator < : left hand side operand is self, right hand side, another LS2D object
        return (self.norm < anotherLS.norm)


#named input variables, order can be changed from declaration
p1 = Point2D(y = 4, x = 2, name = "P1")
p1prime = Point2D(y = 4, x = 2, name = "P1Prime")
print(id(p1))
print(id(p1prime))

p2 = Point2D(y = 5.6, name = "Z")
p3 = Point2D() #equivalent of default xtor, with no input values, creates point at (0;0)

#Compairing two Points??
if(p1 == p1prime):
    print("same points")
else:
    print("different points")

try:
    SL1 = StraightLine2D(p1, p3)

    #Typical case of dealing with the exception, Don't complexify the code of StraightLine
    SL2 = StraightLine2D(p1)
except:
    print("Could create a StraigtLine object")

pO = Point2D(0,0,"O")
pA = Point2D(1, 2, "A")
pB = Point2D(2, 4, "B")

LS1 = LineSegment2D(pO,pA)
LS2 = LineSegment2D(pO, pB)

if(LS2 < LS1):
    print("Yes, LS1 is shorter than LS2")

print("bye")
```

**Playground.py**

```
import math #imports a whole module (may have multiple classes and/or functions) - Full load
import sys
#from xxx import yyyy #from a module, import a particular submodule/class/function -- Partial
load
#This one requires further discussions, today afternoon

#How, in a scripting language, could we engineer a "main" function
#Which would be the starting point of execution of the program?

#It's possible in Python
#It's a "trick", but it works!

#import with a foldername.modulename (without .py extension)
#This runs the code of the target module, loads all function definition in memory
import myTools.ModuleInstaller as mi
import myTools.ContentObfuscation as co
import DBConnectors.MSSQL_DBConnector as mssql
import CustomerIDSequencer as cseq

try:
    import pyodbc
except:
    mi.installModule("pyodbc")
    import pyodbc

try:
    import pandas as pd
except:
    mi.installModule("pandas")
    import pandas as pd

def main():
    print("This is the main function of the program")
    #connection to an SQL Server instance named SQL2019 on the local machine EF-CODD
    #connection with the sa user (NEVER TO BE DONE, Super Admin of SQL Server)
    #user: sa
    #password: godsti
    #UNACCEPTABLE, HUGE SECURITY RISK
    testVar = range(10, 0, -1)
    for i in testVar:
        print(i)

    custSequencer = cseq.CustomerIDSequencer()

    nextCustID = custSequencer.__next__()
    for i in custSequencer:
        print(i)

    #Try to instanciate the abstract class DBCOnnector
    myConnector = mssql.MSSQL_DBConnector(dbserver="EF-CODD\SQL2019", \
        dbname = "SQLPlayground_A19", dbusername = "sa", dbpassword="godsti")
    myConnector.open()

    #IsConnected is a read-only property
    result = myConnector.IsConnected

    #dbUser is a read/write property
    myConnector.dbUser = 'anotherUser'

    obfuscator = co.ContentObfuscation()

    dbConnection = \
        pyodbc.connect('DRIVER={ODBC Driver 17 for SQL Server};SERVER=EF-
CODD\SQL2019;' \
        + 'DATABASE=SQLPlayground_A19;UID=sa;PWD='+ \
        obfuscator.deObfuscate('gAAAAABgygMWApiBrtD7ENPIH3f4R6TGS3Od-
J5CepDftXoddiJShPmDIQU2ZmhicVk-UT_rnRnr16CNiaQQWTUz7hLelLVxZQ=='))

    sqlQuery = "SELECT * FROM dummy"
    dfResult = pd.read_sql(sqlQuery, dbConnection)

    return 0

#How do I force the interpreter to execute main() only?
# __name__ is an envrionment variable, maintained by the interpreter
if __name__ == '__main__' :
    main()
    sys.exit
    #How to I make sure that nothing executes after?
```

DebugDecorrator_example.py

```
import functools

#new, own decorator, called "debug" --> @debug

def debug(func):
    """Print the function signature and return value"""
    @functools.wraps(func)
    #@functools does the "behind the scenes job" to pass func into debug
    #this will allow my code to access the environment of the function pointed by func
    #Notably, the input parameters of the function pointed by func
    #In our example, the function pointed by func is make_greeting
    #We now have access to two important collections: *args and **kwargs
       #*args is a tuple with all the unnamed input parameters of func. f1(8) 8 is unnamed
       #**kwargs is a dictionary of all the named input parameters of func f1(x=8) named param
    def wrapper_debug(*args, **kwargs):
        #expansion of the data in args
           #args is a tuple, iterate over the tuple via a for loop
           #Construction of an anonymous list by placing the for loop inside []
        args_repr = [repr(a) for a in args]                    # 1

        #kwargs.items() iterates over all the dict element and returns the current key(k) and the
value (v)
        kwargs_repr = [f"{k}={v}" for k, v in kwargs.items()]  # 2

        signature = ", ".join(args_repr + kwargs_repr)          # 3
        print(f"Calling {func.__name__}({signature})")
        #Alternative way of calling a Python function: passing a tuple of unamed input params +
dict of named params
        value = func(*args, **kwargs)
        # "!r" is an explicit conversion flag that converts the value to a string using repr()
        #Thanks Yassine!
        print(f"{func.__name__!r} returned {value!r}")          # 4
        return value
    #return a function pointer on the internal function
    #this is captured by the "internals" of the  @functools.wraps(func) decorator
    return wrapper_debug

#custom decorator, called debug
#It's purely and simply a function declared above, called debug
#This is syntaxic sugar to call debug and pass make_greeting as a parameter
@debug
def make_greeting(firstname, lastname, age=None):
    if age is None:
        return f"Howdy {firstname} {lastname}!" #example of variable susbstitution in a string
with f in front
```

```
    else:
        return f"Whoa {firstname} {lastname}, {age} already, you are growing up!"

#Standard function call of make_greeting
# As make_greeting has been decorated with debug, the code of def debug + internal
wrapper_debug is called
# The decorator debug act as an "automated hook" --> When a function is called, another one
must be implicitely called too
# Automatic calling of another piece of code from the decorated function
    #The case here is simplified: @debug allows us to see the input parameters of every decorated
function
    #No extra code around the call of greeting_value to achieve the desired output
    #Automatic code execution of debug doesn't disrup the return mechanism of make_greeting
greeting_value = make_greeting("Sébastien", lastname = "Corniglion", age=39)

@debug
def uselessProc():
    pass

uselessProc()

#An import can be placed wherever relevant for the code
import math
# Apply a decorator to a standard library function
# The decorator is purely and simply a function too
# math.factorial is defined in the math module, not my function
# I cannot decorate it like with make_greeting, but I can use the decorator as a standard function
call

#I'm MONKEY PATCHING math.factorial
#Now, in the module math, the function factorial is replaced by debug(math.factorial)
#the code of math.factorial is not deleted but replaced by the call debug(math.factorial)
#This works thanks to functions being first class citizen: math.factorial is an object, loaded in
RAM with import.math
#It's only a function pointer replacement
math.factorial = debug(math.factorial)

#Now the old pointer math.factorial has been replaced by the RAM location of
debug(math.factorial)
#Compution of e = Infinite Sum over n of (1 / n!)
def approximate_e(terms=18):
    return sum(1 / math.factorial(n) for n in range(terms))

e = approximate_e(50)
print(e)
```

```
import time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()    # 1
        value = func(*args, **kwargs)
        end_time = time.perf_counter()      # 2
        run_time = end_time - start_time    # 3
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])

waste_some_time(num_times = 100)
```

**DBConnector.py**

```
from abc import ABC, abstractmethod

try:
    import pyodbc
except:
    mi.installModule("pyodbc")
    import pyodbc

class DBConnector(ABC):
    """(docstrings with triple double quotes)This makes DBConnector, an abstract class"""

    def __init__(self:object, dbserver:str = None, dbname:str = None, dbusername:str = None,
dbpassword:str = None):
        # m_ in front of the name of a instance attribute is an old naming convention
        #to rapidly identify the fact that the variable is part of a class
        self._m_dbserver =  dbserver
        self._m_dbname = dbname
        self._m_dbusername = dbusername
        self._m_dbpassword = dbpassword #this must be obfuscated

        #At construction time, we are not connected to the data source
        self._m_isDBConnectionOpen = False
        #At construction time, in DBConnector class, we don't know which driver will be used
        self._m_driver = None


    @abstractmethod #Python decorator, to be seen later
    def selectBestDriverAvailable(self:object):
        """This is a PURE VIRTUAL METHOD, we cannot, unable to provide an algorithm to
select
       the correct driver, this will done by a child class of DBConnector"""
        pass

    def open(self:object):
        #the pyodbc connect function expects the connection string


        self._m_conduit = pyodbc.connect('DRIVER={' + self.selectBestDriverAvailable() + '};'  \
            + 'SERVER='+self._m_dbserver+';' \
        + 'DATABASE=' + self._m_dbname + ';UID=' + self._m_dbusername + ';PWD=' +
self._m_dbpassword )

        #if the connect call works (no exception), we can assume that
        #the conduit is connected to the data source
        self._m_isDBConnectionOpen = True

    @property #a read only property
    def IsConnected(self:object)->bool:
        return self._m_isDBConnectionOpen

    @property
    def dbUser(self:object)->str:
        return self._m_dbusername
    @dbUser.setter
    def dbUser(self:object, value:str)->str:
        self._m_dbusername = value
```

**MSSQL_DBConnector.py**

```
import DBConnectors.DBConnector as dbc

try:
    import pyodbc
except:
    mi.installModule("pyodbc")
    import pyodbc

class MSSQL_DBConnector(dbc.DBConnector):
    """This class inherits from an abstract class: DBConnector
    The pure virtual method, decorated with @abstractmethod,
    called selectBestDriverAvailable must be implemented here"""

    def __init__(self:object, dbserver:str = None, \
        dbname:str = None, dbusername:str = None, \
        dbpassword:str = None):

        #Constructor a child class, what must be done?
        #CODE REUSE: use the xtor code of your mother
        super().__init__(dbserver, dbname, dbusername, dbpassword)
        self.selectBestDriverAvailable()

    def selectBestDriverAvailable(self:object):
        """Polymorphic implementation of the virtual function in DBConnector
        This method is specialised for connecting to a MS SQL Server instance
        """

        #A good starting point: pyodbc drivers() method
        listOfAllAvailableDriver:list[str] = pyodbc.drivers()

        if(listOfAllAvailableDriver is not None):
            #Thanks to backward compatibility, select the highest version available
            if("ODBC Driver 17 for SQL Server" in listOfAllAvailableDriver):
                self._m_driver = "ODBC Driver 17 for SQL Server"
            elif("ODBC Driver 13 for SQL Server" in listOfAllAvailableDriver):
                self._m_driver = "ODBC Driver 13 for SQL Server"

            #carry on


        return self._m_driver
```

UnitTestExample.py

```
#This is not unit testing, this is OLD, as old as the first programming languages
# Testing by assertion, with an input, a predicate, an expected output
# In Python (and Java) should the assertion fail, an exception is triggered
assert sum([1,2,3]) > 5, "Should be 6"

import unittest

#Object-Oriented Python is mandatory, unittesting is done via a class

class TestSum (unittest.TestCase):
#in this class, as many test cases as required can be implemented
# One test case = one method
# No need to implement the polymorphic constructor, it's all dealt with by inheritance

    def test_sum_list(self):
        self.assertEqual(sum([1,2,3]), 6, "Should be 6")

    def test_sum_tuple(self):
        self.assertEqual(sum((2,2,1)), 5, "Should be 5")

if __name__ == "__main__":
    try:
        unittest.main()
    except Exception as ex:
        print(ex.with_traceback)
    print("bye")
```

**GeneratorsExample.py**

```
import os
#Python Generators?
#Same idea as iterators (going step by step over "something")
#An iterator iterates over an existing data collection

def emit_lines(pattern="DSTI", originPath = "."):
    lines = []
    #browses the directory called test in the current directory of execution
    for dir_path, dir_names, file_names in os.walk(originPath):
        #iterate over the all the files in test
        for file_name in file_names:
            if file_name.endswith('.txt'):
                #if current file has a .txt extension, os.path.join provides the absolute path to file
                for line in open(os.path.join(dir_path, file_name)):
```

```
            #open the file and load its content in memory, by providing the full file path
            #open provide a collection of lines in the file
            if pattern in line:
                #if pattern found in line, add the line in the lines list
                lines.append(line)
    return lines


lines = emit_lines()
for foundLine in lines:
    print(foundLine)


def generate_filenames(directory="."):
    """
    generates a sequence of opened files
    matching a specific extension
    """
    for dir_path, dir_names, file_names in os.walk(directory):
        for file_name in file_names:
            if file_name.endswith('.txt'):
                yield open(os.path.join(dir_path, file_name))

def cat_files(files):
    """
    takes in an iterable of filenames
    """
    for fname in files:
        for line in fname:
            yield line

def grep_files(lines, pattern=None):
    """
    takes in an iterable of lines
    """
    for line in lines:
        if pattern in line:
            yield line

allFiles = generate_filenames(".")
fileContent = cat_files(allFiles)
linesWithPattern = grep_files(fileContent, 'DSTI')

#Please note that none of three functions are being called until iteration is performed below
for foundLine in linesWithPattern:
    print(foundLine)
```

**CustomerIDSequencer.py**

```
class CustomerIDSequencer():
    """ This class is an equivalent to an Oracle Databse Sequencer
        It provides the "next" value for a CustomerID based on CU-XYZ with x,y,z in [0;9]
    """

    def __init__(self):
        self.__currentDigit = 0
        self.__CustIDBase = "CU-"

    #For a class to be iteratable, you must impelement some methods inherited from object
    # Python, like C++, doesn't have the notion of "interfaces", i.e forcing implementation of a list
of methods
    # Python have abstract classes (like C++), but not part of "standard" Python, relies on the
package abc: not widespread

    #What should be given to "for" when being iterated over
    def __iter__(self):
        return self

    def __next__(self):
        #implementation of your algorithm: what should be the "next" value
        if(self.__currentDigit < 999):
            self.__currentDigit = self.__currentDigit + 1
            return self.__CustIDBase + str(self.__currentDigit).zfill(3) #zero fill method of the str
class
        else:
            #Raising an exception to the caller (which can or not capture the exception with a try
block)
            # Call to the constructor of the base class Exception, mother class of all other specialised
exception classes
            raise Exception("Current sequence value eq 999, maximum value reached")
```

**Countdown_SingleThreaded.py**

```
import time
#from Threading import Thread

COUNT = 50000000

def countdown(n):
    while n >0:
        n = n-1
```

```
start = time.time()
countdown(COUNT)
end = time.time()

print("Time taken in seconds (single threaded code - )", end - start)

# Time taken in seconds (single threaded code - 5.836000204086304)
```

Countdown_MultiThreaded.py

```
import time
from threading import Thread

#COUNT = 50000000

def countdown(n):
    while n >0:
        n = n-1

t1 = Thread(target=countdown, args=(25000000000,))
t2 = Thread(target=countdown, args=(25000000000,))
t3 = Thread(target=countdown, args=(25000000000,))
t4 = Thread(target=countdown, args=(25000000000,))

#t4 = Thread(target=countdown, args=(12500000,))

start = time.time()
t1.start() #thread1 launches and starts being executed, apart from current thread
#t1.join() #current thread will now wait for thread1 to finish
          #threads 2, 3 and 4 won't be started until thread1 is done
          #here it's not desirable, we want all threads to start computing

t2.start()
t3.start()
t4.start()

t1.join() #join here means that the current thread will "block", waiting for thread1
          #but t2, t3, t4 are still running
t2.join()   #t2 has less worload to perform
          #it's likely that when t1 ends, and the current thread resumes
            # t2.join() will be immediate, because t2 is already finished
t3.join()
t4.join()

#in order to get the correct data for the final computation
#which is "additive", the final computation is correct
```

```
# if the current has waited for all the computational threads to finish
end = time.time()

print("Time taken in seconds (multi threaded code - )", end - start)

# Time taken in seconds (single threaded code - ) 5.836000204086304
# Time taken in seconds (multi threaded code - ) 5.8460001945495605 (2 threads)
# Time taken in seconds (multi threaded code - ) 5.95702600479126 (4 threads)
```

Countdown_Multiprocessing.py

```
import time
from multiprocessing import Pool

COUNT = 5000000000

def countdown(n):
    while n >0:
        n = n-1

#avoiding code RE-ENTRANCE when the forked Python interpreters are launched by
apply_async
# apply_async will set the __name__ attribute of the forked Python interpreters to the name
#provided as function pointer: here countdown
#This test avoid the forked interpreters to carry on running the code, as __name__ is not ==
'__main__'
#But in the initial launch of the program (initial process), the value of __name__ is == __main__
if __name__ == '__main__':
    pool = Pool(processes=8)
    start = time.time()

    p1 = pool.apply_async(countdown, [COUNT//8])
    p2 = pool.apply_async(countdown, [COUNT//8])
    p3 = pool.apply_async(countdown, [COUNT//8])
    p4 = pool.apply_async(countdown, [COUNT//8])
    p5 = pool.apply_async(countdown, [COUNT//8])
    p6 = pool.apply_async(countdown, [COUNT//8])
    p7 = pool.apply_async(countdown, [COUNT//8])
    p8 = pool.apply_async(countdown, [COUNT//8])

    pool.close()
    pool.join()

    end = time.time()

    print("Time taken in seconds (multiprocessed code - )", end - start)
    #Time taken in seconds (multiprocessed code - ) 0.9979984760284424


    # Time taken in seconds (single threaded code - ) 6.177238702774048
    # Time taken in seconds (single threaded code - ) 5.960912227630615 (2 threads)
    # Time taken in seconds (single threaded code - ) 5.866125106811523 (4 threads)
```

**WrappingUp_Python_s21.py**

```
import os
import numpy as np
import itertools


def apply_to_list(aList: list, lbd):
    """This function passes a lambda function over a list via the ldb parameter"""
    j = 0
    #i is the derefenced iterator: the current element of list
    for i in aList:
        #for the current element: apply a function, given in parameter
        aList[j] = lbd(i)
        j = j + 1
    return aList


test_list = [1, 2, 3, 4]


#the lambda keyword creates an anonymous function, an object, passed as parameter
#the lambda as an input paramter: x
#when writing lambda x --> you skip a function name, which could be myFunction(x)
#convience, to reach genericity
#whatever is done in the body of the lambda function can be changed easily
#Here, in the body, I'm using the xtor of int, but I don't have to

#What could be the use-case motivation to call to int xtor in lambda body?
#to force the results of applying the lambda to be made exclusively of int
#the result of applying the lambda is a monotyped list
apply_to_list(test_list, lambda x : x*3 )

print(test_list)

BoyNames = ["Alan", "Adam", "Wes", "Will", "Albert", "Charles", "Chris"]
first_letter = lambda x : x[0]

#itertools.groupby(BoyNames, first_letter) provides an data source for iteration to the for loop
```

```
#not an "SQL GROUP BY" --> progress element by element in the BoyNames collection (SQL
works on the whole table)

#as itertools.groupby is progressing element by element in BoyNames, grouping key based on
lambda function
#itertools.groupby is comparing to the previously known "group"
#The first group created is from "Alan" --> lambda function says "start with A"
#itertools.groupby will continue grouping within this group --> Adam goes in
#When it goes to "Wes" --> lambda returns W --> new group
#Goes to Albert, it's a new group too, but not put back in an existing group, due to sequential
processing
for letter, names in itertools.groupby(BoyNames, first_letter):
    print(letter, list(names))


#numpy provides C-arrays in Python
#Capsuled data, using the PyCapsule technique
#Data and operations are done in the C-world (compiled), not Python
#numpy is a Python wrapper around the C-world components

#data capsuled in the C world (numpy)
#numpy arrays are mono-typed, as they are C arrays
#data is contiguous in memory
myNPArray = np.arange(1000000)

#data is in the Python world (list)
#a Python list can be multi-typed
#Non-contiguous in memory
myList = list(range(1000000))


#numpy provides "vectorised" operations
#Here the "vectorised" operation will apply the *2 on all array elements

#In Computer Science, a "vectorised" operation as a strong meaning
#It's related to a CPU ability: it's hardware matter!
#What can the CPU do "at once", per clock cycle
#Historically, CPUs are SISD: Single Instruction Single Data
                        #Order Single Instruction: ADD, MULTIPLY (microcode, operation in CPU)
                        # To this instruction, Single Data: scalar integer (in binary)
#Modern CPUs: SIMD --> Single Instruction, Multiple Data
#            MIMD --> Multiple Instructions, Multiple Data

#A vectorised operation means a SIMD or MIMD capacity at CPU/GPU level

#numpy "vectorisation" suggest a SIMD behaviour: operation *, multiple data is the array
```

```
#THERE IS NO GARANTY (in fact, it's not happening) THAT NUMPY USES A SIMD CPU
INSTRUCTION
#IN SIMD CPU/GPU architectures, the vector registers size are "tiny" : 512-bit when large (in
2020)

#An array of 1 million elements cannot fit in a large 512-bit vector register
#IT'S NOT A VECTORISED OPERATION --> SYNTAXIC SUGAR
#USEFUL, BUT NO PERFORMANCE OF REAL VECTOR INSTRUCTIONS AND
REGISTERS IN CPU/GPU
#REAL VECTOR OP: DEEP LEARNING COURSE, WHEN USING PyTorch, TensorFlow -->
CUDA library NVIDIA

#AKA: broadcasting the "* 2" operation on the whole numpy array
#It's the proper term
myNPArray_2 = myNPArray * 2
#What is happening here:
#for(int i = 0; i < array_size; i++)
#    myNPArray2[i] = myNPArray[i] * 2

#Impossible with a list "directly", we need a lambda function to do it
myList_2 = [ x * 2 for x in myList] #lambda function
#an anonymous (lambda) function (also an object) is created by Python, return fed to the caller:
xtor of list

#Creating an iteratable class for DB purposes

#The triple quoted text (comments overs multiple lines)
#When placed immediatly after a class definition (or a function)
#sets the magic attribute __doc__ of the object, in-line with the PythonDoc protocol

class CustomerIDSequencer:
    """
    Customer IDs are based on the following model: CU-XYZ, with XYZ being digits [0-9]
    CustomerID is a string value, digits need padded with 0's if necessary
    Eg. CU-001, CU-015
    And, an exception must be leveraged if we have reached CU-999 (with chosen model, cannot
go further)
    """


    def __init__(self):
        self.currentDigit = 0 #start at 0, for when next is called with +1 we get first sequence CU-
001
        self.CustIDBase = "CU-"

    #Overload of __iter__ --> this must return an iterator object
    #Here, we can return ourself (self), we'll see what happens at debug level
    def __iter__(self):
        return self

    def __next__(self):
        if(self.currentDigit < 999 ):
            self.currentDigit = self.currentDigit + 1
            return self.CustIDBase + str(self.currentDigit).zfill(3) #0's in front for max length of 3
        else:
            raise Exception("Current sequence value == 999, maximum value reached")

myCustomerSequence = CustomerIDSequencer()

for currentCustomerID in myCustomerSequence:
    print(currentCustomerID)


#a quick recap on iteration/iterators in Python
# Very similar to C++ iterators of the Standard Template Library

# an iterator is a pointer on one element of a data collection
# In Python, all standard data collection classes (list, tuple, set, etc.) are iteratable

Ex. --> Factorial computation
n = 4
result = 1
data = range(n, 1, -1)

#This could be used to write a while loop based on a > 1, calling i.__next__()
#Indentical to C++ --> the iterator is dereferenced to get the actual value
i = data.__iter__()
a = i.__next__()


#Pythonic usage of iterator
# the Python for loop is very different than C/C++/Java, where we drive a loop variable
# In Python, the for loop is built upon iterators objects: must be provided with iteratable class
# This has lead to the dev of alternative for loops in "old" languages: foreach, for_each -->
Python behaviour
for i in range(n, 1, -1):
    result = result * i

#In the class range, overload of magic method __iter__(self)
#When an instance of range is subjected to a for/while loop, __iter__(self) is implicitely called
```

```
#Python Generators?
#Same idea as iterators (going step by step over "something")
#An iterator iterates over an existing data collection

def emit_lines(pattern="DSTI", originPath = "."):
    lines = []
    #browses the directory called test in the current directory of execution
    for dir_path, dir_names, file_names in os.walk(originPath):
        #iterate over the all the files in test
        for file_name in file_names:
            if file_name.endswith('.txt'):
                #if current file has a .txt extension, os.path.join provides the absolute path to file
                for line in open(os.path.join(dir_path, file_name)):
                    #open the file and load its content in memory, by providing the full file path
                    #open provide a collection of lines in the file
                    if pattern in line:
                        #if pattern found in line, add the line in the lines list
                        lines.append(line)
    return lines

def generate_filenames(directory="./"):
    """
    generates a sequence of opened files
    matching a specific extension
    """
    for dir_path, dir_names, file_names in os.walk(directory):
        for file_name in file_names:
            if file_name.endswith('.txt'):
                yield open(os.path.join(dir_path, file_name))

def cat_files(files):
    """
    takes in an iterable of filenames
    """
    for fname in files:
        for line in fname:
            yield line

def grep_files(lines, pattern=None):
    """
    takes in an iterable of lines
    """
    for line in lines:
        if pattern in line:
            yield line
```

```
#A generator creates data
# Below, we are creating new data: squares of numbers
# The data created by a generator is iteratable

#the instruction yield hides immense complexity, creating and adding, on-the-fly, iteratable data

def generateSquareValues(upTo = 100):
    print("Generating squares of numbers from 1 to {}".format(upTo**2))
    for i in range (1, upTo+1, 1):
        #this "yield" keyword creates a Python generator
        yield i ** 2


squareGen = generateSquareValues()
print(squareGen)

#The loop will be run only when iterating over the function creating a generator
#Computions are only run upon each iteration over the generator object



obj = squareGen.__next__()
#Does it grow in size at each step of __next__ being called (explicitely or implicitely by for
loop)?
#No, as the data source being a generator, the purpose is only:
        #take CPU time for current item computation (if applicable)
        #RAM for current item size
    #When your code goes "next", the item before is lost, unless you "save" it yourself in your
own data collection

#It's only in the for loop here, when iterating over squareGen that the **2 is called, one by by
one
for i in squareGen:
    print(i)

#Generators are useful when data is coming from I/O
    # Data being read from files
    # Resultset of a SQL query

# I/O data read can be done using two main approaches
    # "BULK" read --> load at once, whole file, into RAM
    # "PARTIAL/ROW" read --> load "row by row", item per item

#When data comes as the resultset of an DB query (RDBMS, NoSQL DB), matters are more
complicated
# The data of the resultset comes in bulk by nature: the DB Server executed the query and has
loaded the
```

```
        #resultset in RAM
# When the DB Server is not on the same machine as where your code is running
   #The whole resultset generated by the query is in RAM of the Server
   #The resultset size could (easily) be larger than the available RAM on your code-execution
machine (the client)
      #Reading this in bulk on the client will lead to an OutOfMemory error

#This problem is solved "quicker" when Server & Client are on same machine
#If out of RAM, the Server will fail even before the client can read the data!

#Example with TXT files on disk

#emit_lines is written "standard style", without using generators
# What are the potential drawbacks of NOT using generators?

#Here, the data source is not "unique"
#There ara potentially many TXT files under the directory
#EAch TXT file could be huge, and/or an immense number of files to read
#The call to emit_lines() is BLOCKING until the whole read is finished
#As long as emit_lines has not read all the text files, the Python interpreted is stuck reading

#Typical example of a code difficult to parralelise --> run on multiple CPU/CPU Cores
#3 nested for loops, ending in blocking I/O's
#This runs on one CPU, like all Python "standard" code
read_results = emit_lines()

py_files = generate_filenames(".")
py_file = cat_files(py_files)
lines = grep_files(py_file, 'DSTI')


#Please note that none of three functions are being called until iteration is performed below
for foundLine in lines:
   print(foundLine)
```

```
import os

def extract_lines(pattern="DSTI", originalPath="."):
    lines=[]
    for dir_path, dir_name, file_names in os.walk(originalPath):
        for files in file_names:
            if files.endswith('.txt'):

                if pattern in line:
                    lines.append(line)
    return lines

result= extract_lines()
# all files loaded in memory
for foundLines in result:
    print(foundLines)

# Generator >>

# select all files
def generate_fileNames(directory='.'):
    for directory_path, directory_name,file_names in os.walk(directory):
        for file_name in file_names:
            if file_name.endswith('.txt'):
                yield open(os.path.join(directory_path,file_name))

# select files content
def cat_files(files):
    for fileName in files:
        for line in fileName:
            yield line

# select the lines which have the pattern
def grep_lines(lines,pattern=None):
    for line in lines:
        if pattern in line:
            yield line

allFiles = generate_fileNames(".")
fileContent = cat_files(allFiles)
LinesWithPattern = grep_lines(fileContent,"DSTI")

# calling file by file (all files not loaded in memory but loading in memory step by step)
for foundLines in LinesWithPattern:
    print(foundLines)
```

**Multiprocessing: to gain Maximum hardware resources for your program**

**Process: Meta Data (for OS) to identify a memory space (a RAM where a program is loaded)**

**Due to memory protection mechanism, this zone is exclusively reserved to the program (PID) and if any other program tries to read or write on that memory zone OS will unload or terminate it.**

**Thread: is sub unit of execution within PID (for one process there is at least one thread per process PID which is the main thread, there is the thread 0 of execution)**

**Note: All the thread (which are functions) of the given program shared the same process memory space so they are sharing the data between each other.**

**If 2 processes trying to exchange data where they are not in the same memory space (example to copy data from word to excel) and due to memory protection mechanism if they need to communicate through the kernel API which require enter process communication.**

**Scheduling mechanism is based on threads not process. So Schedular of OS giving the execution time to Threads attached to the process not to process itself.**

**Single Thread example:**

```
import time

count = 50000000

def countdown(n):
    while (n>0):
        n=n-1

start =time.time()
countdown(count)
end=time.time()

print ("the total time: ", end - start)
```

**Multi-Threaded Programing model**

```
# number of thread depend of how many core our processor have for the distribution of them.
import time
from threading import Thread

def countdown(n):
    while (n>0):
        n=n-1
t1 = Thread(target=countdown, args=(12500000,))
t2 = Thread(target=countdown, args=(12500000,))
t3 = Thread(target=countdown, args=(12500000,))
t4 = Thread(target=countdown, args=(12500000,))

start =time.time()
# start executing the thread
t1.start()
t2.start()
t3.start()
t4.start()
# stop the lunching thread execution until part t0 to finished and so on but in this case the total
time will be same like as one thread. But without the join for sure will be too much faster.
t1.join()
t2.join()
t3.join()
t4.join()
end=time.time()
print ("the total time: ", end - start)
print ("")
```

**Multi-Processes Programing model**

```
import time
from multiprocessing import Pool
ITEM=50000000
def countdown(n):
    while(n>0):
        n=n-1

# Avoiding code re-enter when Python interpreters are lunched by apply_async.
if __name__ == '__main__':
    pool=Pool(processes=4)
    start = time.time()

    p1 = pool.apply_async(countdown,[ITEM//4])
    p2 = pool.apply_async(countdown,[ITEM//4])
    p3 = pool.apply_async(countdown,[ITEM//4])
    p4 = pool.apply_async(countdown,[ITEM//4])

    pool.close()
    pool.join()

    end = time.time()

    print("the total process time", end-start)
    print("")
```

**Lambda Function:**

Is an anonymous function (no def function name but only body of code).

```
#the result of applying the lambda is a monotyped list
#numpy arrays are mono-typed, as they are C arrays

import os
import numpy as np
import itertools

def apply_to_list(alist:list,lbd):
    """this function passes a lambda function over a list via the lbd parameter"""
    j=0
    for i in alist:
        alist[j]=lbd[i]
        j=j+1
    return alist

test_list = [1,2,3,4]
```

Itertools group by

```
names=["Alan","Adam","Wes","Will","Albert","Charles","Chris","Ali","Amin"]
j=0
for i in names:
   print(i[0], i )

print("============ using itertools ============")
first_letter=lambda x : x[0]
#using itertools.group by (names, firstletter) to group the names by the first letter (it will got all similer at once ) then change the group for the next different returned value
for letter, name in itertools.groupby(names,first_letter):
   print(letter,list(name))

# encoding=utf-8

import threading
import time

Try: # python2
   from Queue import Queue
 Except ModuleNotFoundError: # python3
   from queue import Queue

"""
 A synchronous, thread-safe queue class is provided in Python's Queue module.
       Includes FIFO (First In First Out) queue Queue, LIFO (Last In First Out) queue LifoQueue, and Priority Queue PriorityQueue.
       These queues implement lock primitives (which can be understood as atomic operations, ie either do not, or do) and can be used directly in multithreading.
       Queues can be used to synchronize between threads.
 Queue use
       For Queue, plays an important role in multi-threaded communication
       Add data to the queue, using the put() method
       Take data from the queue, use the get() method
       Determine if there is any data in the queue, use the qsize() method
 Why use producer and consumer models?
       In the thread world, producers are the threads that produce data, and consumers are the threads that consume data.
       In multi-threaded development, if the producer is processing fast and the consumer processing speed is slow, then the producer must wait for the consumer to finish processing before continuing to produce data.
       By the same token, if the consumer's processing power is greater than the producer, then the consumer must wait for the producer.
       In order to solve this problem, a producer and consumer model was introduced.
 What is the producer consumer model?
```

The producer consumer model solves the problem of strong coupling between producers and consumers through a container.

The producer and the consumer do not communicate directly with each other, but communicate through the blocking queue, so the producer does not have to wait for the consumer to process after the data is produced.

Directly thrown to the blocking queue, the consumer does not find the producer to ask for data, but directly from the blocking queue, the blocking queue is equivalent to a buffer.

Balance the processing power of producers and consumers.

This blocking queue is used to decouple producers and consumers. Looking at most design patterns, you will find a third party to decouple.

The code for implementing producer and consumer problems with FIFO queues is as follows:
""""

```
class Producer(threading.Thread):
    def run(self):
        global queue
        count = 0
        while True:
            if queue.qsize() < 1000:
                for i in range(100):
                    count = count + 1
                                   Msg = 'generate product' + str(count)
                    queue.put(msg)
                    print(msg)
            time.sleep(0.5)

class Consumer(threading.Thread):
    def run(self):
        global queue
        while True:
            if queue.qsize() > 100:
                for i in range(3):
                                   Msg = self.name + 'consumed' + queue.get()
                    print(msg)
            time.sleep(1)
if __name__ == '__main__':
    queue = Queue()

    for i in range(500):
          Queue.put('initial product' + str(i))
    for i in range(2):
        p = Producer()
        p.start()
    for i in range(5):
        c = Consumer()
        c.start()
```

**From Last session**
**(DB name QueueDB, Table QueueExample (MessageID, MessageContent,InProcess)**
**( 2 procesuers dequeue_message, enqueue_message)**

```
import pyodbc
import os
import logging
import sys
import time

from PIL import Image
import mutliprocessing

def consumer(pollingInterval):
    logging.basicConfig(stream=sys.stderr, level=logging.FATAL)

    thumbnailSize = (256,256)

    mssqlInstanceName='SEB-WFH'
    mssqlDatabase = 'QueueDB'
    mssqlDriver='{ODBC Driver 17 for SQL Server}'


dbconnector=pyodbc.connect('DRIVER={driver};SERVER={instance};DATABASE={database
};UID='+username+';PWD='+ password)
    dbconnector.autocommit = True
    sql= """\
                    DECLEARE @message nvarchar )max);
                    DECLEARE @resultState smallint;

                    SET @messageContent = ?;

                    EXECUTE [dbo].[dequeue_message] @message OUTPUT, @resultState
OUTPUT;
                    SELECT @messageas [Message], @resultState AS ResultState;
        """
    stopReadingQueue = False

    while stopReadingQueue is False:
        cursor = dbConnector.cursor()

        try:
            cursor.execute(sql)

            rows = cursor.fetchall()
            message, returnCode = rows[0].Message, rows[0].ResultState
            cursor.close()
        except Exception as excp:
            logging.fatal("""\ Error while polling the queue, message integrity is not guarant
                          Message content (if available): {messageContent} \n
                          Exception: {excpMessage}""")

        if(returnCode == -1):
            stopReadingQueue = True

        elif(returnCode == 0 and (message is not None or message != '')):
            print("CONSUMER - DEQUEUED: " + message)
            fileFullPath = './thumbnails/' + os.path.basename(message)
            try:
                with Image.open(message) as imgToProcess:
                    imgToProcess.thumbnail(thumbnailSize)
                    imgToProcess.save(fileFullPath)
                    print("CONSUMER - PROCESSED: " + fileFullPath)

            except Exception as excp:
                logging.fatal("Error while trying to save the thumbnail: {filePath} \n")

        time.sleep(pollingInterval)

def producer():
    logging.basicConfig(stream=sys.stderr, level=logging.FATAL)

    unsplashAPIKey = '' #create your

    mssqlInstanceName='SEB-WFH'
    mssqlDatabase = 'QueueDB'
    mssqlDriver='{ODBC Driver 17 for SQL Server}'


dbconnector=pyodbc.connect('DRIVER={driver};SERVER={instance};DATABASE={database
};UID='+username+';PWD='+ password)
    dbconnector.autocommit = True

    url =
"https://api.unsplash.com/photos/random/?client_id={key}".format(key=unsplashAPIKey)
    response = requests.get(url)
    img_url = response.json()["urls"]["raw"]

    response = request.get(img_url)

    if (response.ok is True):
        contentType = response.headers['content-type']
        fileExtension = mimetypes.guess_extension(contentType)
        fileName = str(uuid.uuid4())
        fileRelativePath =
'./originals/{filename}{extention}'.format(filename=filename,extension=extension)

        with open(fileRelativePath, 'wb') as f:
           f.write(response.content)
           fileFullPath = os.path.abspath(fileRelativePath)
           sql= """\
                    DECLEARE @message nvarchar )max);
                    DECLEARE @resultState smallint;

                    SET @messageContent = ?;

                    EXECUTE [dbo].[enqueue_message] @messageContent, @resultState OUTPUT;
                    SELECT @resultState AS ResultState;
              """
           param = (fileFullPath,)
           cursor = dbConnector.cursor()
           cursor.execute(sql,params)

           rows = cursor.fetchall()
           returnCode = row[0].ResultState
```