Python_OOP

Saturday, January 1, 2022 7:31 PM

OOPS(Object oriented programming)

Q:What are the pillars of oop?

1. Inheritance.
2. Polymorphism.
3. Encapsulation.
4. Abstraction.

uses a **constructor(__init__)** which decides how much particular properties get initialized inside the class.

```
class Car(): # making a class. Class name should be capital.
    def __init__(self, windows, doors, enginetype):#constructer.
        self.windows=windows #this __init__ block is
        self.doors=doors # constructer.
        self.enginetype=enginetype
    def drive(self):
        print("The Person drives the car")

car=Car(4,5,"diesel") #car is object being created. Self actually
indicates/points to the object being created.

print(car.windows)
car.drive()
```

```
4

The Person drives the car
```

Constructer vs Destructer:

Constructer:
The task of constructors is to initialize(assign values) to the data members of the class when an object of class is created.
In python it's "**__init__**".

Destructors:
Destructers are called **when an object gets destroyed.** In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically. The __del__() method is a known as a destructor method in Python.

Method types:

1. **Instance method**:
   An instance method is a method that belongs to instances of a class, not to the class itself.
2. **Static method**:
   We generally use class method to create factory methods. Factory methods return class objects (similar to a constructor) for different use cases.
3. **Class method:**
   We generally use static methods to create utility functions.

Class vs Instance Attributes

"Class attributes are common to all object of a particular class."

#like human class its class attributes could be blue_eyes, tall, beautiful,etc.

Each instance/object of a class have thier own **INSTANCE ATTRIBUTES.**
But for CLASS ATTRIBUTES they are same for all the instances/objects of class.
For example :
'All humans can have different names(instance attributes)but all humans have same no. of eyes(class attributes).'
Naming convention:
   Object also called instance.
   Variable also called attribute.

```
class Point():

    default_color = "red"  # this is a CLASS ATTRIBUTE.

     # this is instance methods
     def __init__(self, x, y):

        # these are INSTANCE ATTRIBUTES.
         self.x = x
        self.y = y

     # this is instance methods
    def draw(self):
```

```
red
red
Point(10, 20)
Point(10, 20)
```

```
        print(f"Point({self.x}, {self.y})")

point = Point(10, 20)  # object/instance of class Point

print(point.default_colour)  # object level reading of class
attribute(object referencing).
print(Point.default_color) #class reference reading of class
attribute(class referencing)

point.draw()      # call the method draw
```

Class vs Instance methods

"Class method makes your object creation process easier by enabling you to create factory method instead of providing all parameters of the class at object creation step."

# like human class its classmethod could be walk, run, talk, etc.

Explanation:
Using instance/object of class we can access the instance methods.But sometimes we don't need to create the object of class to access the methods.For accessing class methods we use class reference to access it.Sometimes object initialization is hard and for this we need to create class method that have the attributes initialization. Below class method return a object that have already x and y attributes initialized as 0, when we call class method by class reference then we have another point object with x,y as 0.

- ways to access the instance methods:
    - class reference
    - object reference

- Naming convention:
    - Object also called instance.
    - Variable also called attribute.

```
class Point():

    def __init__(self, x, y):  # this is INSTANCE METHOD.
        self.x = x  # these are INSTANCE ATTRIBUTES.
        self.y = y

    @classmethod
    def zero(cls):
        # return object of class Point which  has x=0 and y=0 attributes.
        return cls(0, 0) # cls(0,0) = Point(0,0)

    def draw(self):  # this is INSTANCE METHOD.
        print(f"Point({self.x}, {self.y})")

# method1:accessing instance method using object reference.
point = Point(0, 0)
#point.draw()

# These two same point objects initialized as x=0,y=0.But the second one
handy for sometimes when there are many specific attributes to provide and
we don't want to create object for it.

# method2:accessing class method using class reference.
point = Point.zero() # you only called 'zero' method instead of providing
point.draw() # 0,0 parameters. classmethod is handy when providing large
parameters.
```

```
Point(0, 0)
```

Magic methods

__init__ : useful for initializing attributes.

__str__ : whenever we print object it gives us name of the class and memory address, but you reimplement it.e.g

```
# using point class above
def __str__(self):
    return f"Point ({self.x}, {self.y})"

point = Point(0, 0)
print(point)
```

```
Point (0, 0)
```

__eq__ : Compairing two object if they are equal with parameters.e.g:

```
# using point class above
def __eq__(self, other):
    return self.x == other.x and self.y == other.y

point = Point(10, 20)
other = Point(10, 20)
print(point == other)
```

```
True
```

__gt__ :Compairing two object if one is greater than other.e.g:

```
# using point class above
def __gt__(self, other):
    return self.x > other.x and self.y > other.y

point = Point(10, 20)
other = Point(1,2)

print(point == other)
print(point > other)
print(point < other)
```

```
False
True
False
```

__add__ :adding two objects together.e.g

```
# using point class above
def __add__(self, other):
    return Point(self.x+other.x, self.y+other.y)

point = Point(10, 20)
other = Point(1,2)

combined = point + other
print(combined)
```

Making custom container:

"To add more functionality of our own to the standard python data types like list, dictionary, etc."

All the functionality/complexity is hidden from the rest of the program, we can say that all the complexity is **encapsulated** from the rest of the program so that our code looks simpler and cleaner.e.g:

```
#in this class we no longer totake care of uppper case and lower case tags because we've implemented our own encapsulated method.

class TagCloud:
    def __init__(self):
        self.tags = {}

    #using .get() method to input key and default value 0 in
    dictionary.
    def add(self, tag):
        self.tags[tag.lower()] = self.tags.get(tag.lower(), 0) + 1

cloud = TagCloud()
cloud.add("python")
cloud.add("java")
cloud.add("Python")
print(cloud.tags)
```

```
{'python': 2, 'java': 1}
```

```
#this magic method is used to get items from the dictionary using square brackets.e.g object["key"]

def __getitem__(self, tag):
    return self.tags.get(tag.lower(), 0)

#used to set value to a key of the dictionary.
def __setitem__(self, tag, count):
    self.tags[tag.lower()] = count

#gives the length of the dictionary
def __len__(self):
    return len(self.tags)

#gives the iterator and we can iterate using loop on it.
def __iter__(self):
    return iter(self.tags)
```

Encapsulation:
"It describes the idea of wrapping data and the methods that work on data within one unit."
This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.

"Access modifiers assist the implementation of encapsulation in python."

Public members:
"Accessible from anywhere in the program."

Private members:"Accessible within the class only but gives user warning !"

python doesn't have anything called private member variable in Python. However, adding **two underlines(__)** at the beginning makes a variable or a method private is the convention used by most python code.e.g:

```
def __init__(self):
    self.__tags = {}
```

Protected members:
"The members of a class that are declared protected are only accessible to a class and the classes derived from it."

protected members of a class can be accessed by other members within the class and are also available to their subclasses. No other entity can access these members. In order to do so, they can inherit the parent class. Python has a unique convention to make a member protected: Add a **prefix _** (single underscore).
e.g:

```
def __init__(self):
    self._tags = {}
```

Properties in python:

"Properties sit in front of attributes and allow us to get and set value of an attribute easily(you can both set and get using only one property object) and cleanly(you don’t have to use get_price and set_price methods)."

Properties and two internal methods called **getter and setter.**

```
#Below code prevents to put (-ve) value in the class attributes, but it does not uses properties.
class Product:
    def __init__(self, value):
        self.set_price(value)

    def get_price(self):
        return self.__price

    def set_price(self, value):
        if value < 0:
            raise ValueError("Price cannot be negative")
        self.__price = value

product = Product(10)
print(product.get_price())
```

```
10
```

```
#A better way of the above code is using python properties as following.

class Product:
    def __init__(self, value):
        self.set_price(value)

    def get_price(self):
        return self.__price

    def set_price(self, value):
        if value < 0:
            raise ValueError("Price cannot be negative")
        self.__price = value

    price = property(get_price, set_price)

product = Product(10)

product.price = 20 #getter
print(product.price) # setter
```

```
20
```

And best Approach is to use property with decorators as:

```
class Product:

#using property you can simply use our price like a regular attribute.
    def __init__(self, price):
        self.price = price

#property object name is the the name of the function(here property name is 'price'), this is the getter function.
    @property
    def price(self):
        return self.__price

#this is the setter function used to set value to attribute.
    @price.setter
    def price(self, value):
        if value < 0:
            raise ValueError("Price cannot be negative")
        self.__price = value

product = Product(10) # creating object

product.price = 20 #setting value to 20
print(product.price) #getting value
```

Inheritance:
"Inherite attributes and methods from other classes."

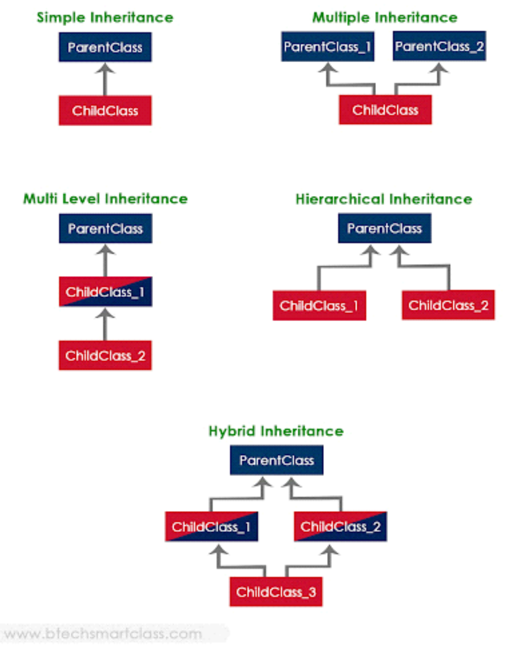
- Prevents code duplication.
- Code reusablity.

Types:
- **Single.**
- **Multi-level.**
- **Multiple.**
- **Hierarchical.**
- **Hybrid**

```
#Animal is the base/parent class.
class Animal:
    def __init__(self):
        self.age = 1

    def eat(self):
        print('eat')

#Mammal is the child class, Mammal is inheriting from Animal class.
class Mammal(Animal):
    def walk(self):
        print('walk')

class Fish(Animal):
    def swim(self):
    print("swim")

m = Mammal()
m.eat() #inheriting method.
print(m.age) #inheriting attribute.
m.walk()
```

```
eat
1
walk
```

Types of inheritance:



```
print(isinstance(m, Mammal))
print(issubclass(Mammal, Animal))
```

```
True
True
```

Method overriding/run time Polymorphism
"In this condition child class constructor replace the parent class constructor."

It is run time Polymorphism because both constructors have same name '__init__' but different functionalities.

```
#In the below code self.age is not initialized as it get replaced by the child class constructor.
class Animal:
    def __init__(self):
        self.age = 1

    def eat(self):
        print('eat')

class Mammal(Animal):
    def __init__(self):
        self.weight = 2

    def walk(self):
        print('walk')

m = Mammal()
print(m.age)
print(m.weight)
```

```
AttributeError: 'Mammal' object has no attribute 'age'
```

Super() function:

So if you want to call both parent and child constructor then use **super()** function.e.g:

```
mammal constructor
animal constructor
1
2
```

```python
class Animal:
    def __init__(self):
        self.age = 1
        print("animal constructor")

    def eat(self):
        print('eat')

#in this class Animal constructor is inherited, if you don’t use
super method then the child(Mammal) class constructor simply replace
the parent(Animal) class constructor.
class Mammal(Animal):
    def __init__(self):
        print("mammal constructor")
        self.weight = 2
        super().__init__() #inherite parent class constructor

    def walk(self):
        print('walk')

m = Mammal()
print(m.age)
print(m.weight)
```

Multi-level Inheritance:
"Inherite from parent class and then child, then the child to another child and so on."

This is not recommended. If you want to do it just do 2 to 3 levels more levels add complexity to the code.

```python
#The problem here is that in real-world chicken is a bird but can't
fly. So that much levels is normally not recommended.
class Animal:
    def eat(self):
        print('eat')

class Bird(Animal):
    def fly(self):
        print('fly')

class Chicken(Bird):
    pass
```

Multiple Inheritance:
"From multiple classes import attributes or methods in a particular class."

(2 parents and one child)

You can combine the features/attribute of many classes into one class as in the Manager class.

```python
class Employee:
    def greet(self):
        print("Employee greet!")

class Person:
    def greet(self):
        print("Person greet!")

# First the python interpreter looks for greet method in Manager
class then Employee and then Person class.
class Manager(Employee, Person):
    pass

m = Manager()
m.greet()
```

```
Employee greet!
```

Good Example of inheritance:

```python
#creating custom exception class.
class InvalidOperationError(Exception):
    pass

class Stream:
    def __init__(self):
        self.opened = False

    def open(self):
       # If self.opened true
        if self.opened:
            raise InvalidOperationError("Stream is already open")
        self.opened = True
    def close(self):
        if not self.opened:
            raise InvalidOperationError("Stream is already closed")
        self.opened = False

#Below two classes inherite from the parent Stream class.
class FileStream(Stream):
    def read(self):
        print("Reading data from a file")

class NetworkStream(Stream):
    def read(self):
        print("Reading data from a network")
```

Abstract Base Class:
(Its like half baked cokkie its not ready to be eaten.)
"Abstract base class provide some common code and a common interface to its derivatives."

"**Abstraction** is a technique used in OOPS paradigm which shows only relevant details to the user rather than showing unnecessary information on the screen helping in reduction of the program complexity and efforts to understand."

Issues:
Problem 1:
Stream class is a general concept the main idea is like which is type of stream either it is FileStream or NetworkStream more specifically.
Solution 1:
So a better approach is to make a abstract class(in our case Stream) and put **common functionalities in it that will carry to the other classes.**

Problem 2:
If we've to implement more classes we should have to maintain a common interface across all the classes it is a nicer approach, like to have **same methods name across all types of Streams.**
Solution 2:
So define a abstract method inside a **abstract class will enforce a common interface across all classes by enforcing each class to have the same abstract method in it.**

If classes don't have abstract method we define in the abstract class then that class will also be considered abstract so define that abstract method(in our case read()) in all the other classes which are inherited from Stream too.

```
from abc import ABC, abstractmethod

class InvalidOperationError(Exception):
    pass

class Stream(ABC):
    def __init__(self):
        self.opened = False

    def open(self):
        if self.opened:
            raise InvalidOperationError("Stream is already open")
        self.opened = True

    def close(self):
        if not self.opened:
            raise InvalidOperationError("Stream is already closed")
        self.opened = False

    @abstractmethod
    def read(self):
        pass

# following are the concrete classes that use the
# functionalities of the abstract class(problem 1 solved).

class FileStream(Stream):
    def read(self):
        print("Reading data from a file")

class NetworkStream(Stream):
    def read(self):
        print("Reading data from a network")
```

```python
# MemoryStream is new class that have read() method
# following the same interface (read method as other concrete
classes) as the other classes(problem 2 solved).

class MemoryStream(Stream):
    def read(self):
        print("Reading data from a memory")

#If a class have a abstract method then it is considered a abstract
# class and we cannot create instance of an abstract class.
# We can only create instances of concrete classes.

stream = MemoryStream()
stream.open()
```

Polymorphism(poly:Many, phism:Forms):
"One function taking many forms at the runtime."

But the classes should have same method names for polymorphism (in our case draw).

Types:

1. **Method overloading/Compile time polymorphism(python don't support it!)**
2. **Method overriding/run time polymorphism/duck typing.**

Method Overriding/runtime polymorphism/duck typing:
Method overriding is an example of run time polymorphism. In this, the specific implementation of the method that is already provided by the parent class is provided by the child class. **It is used to change the behavior of existing methods** and there is a need for at least two classes for method overriding. In method overriding, **inheritance is always required** as it is done between parent class(superclass) and child class methods.

```python
from abc import ABC, abstractmethod

# abstract method to enforce that the methods in clases should have
same name.
class UIControl(ABC):
    @abstractmethod
    def draw(self):
        pass

class TextBox(UIControl):
    def draw(self):
        print("drawing TextBox")

class DropDownList(UIControl):
    def draw(self):
        print("drawing DropDownList")

# this draw method taking different forms based on the object passed
to it at # the runtime.

def draw(control):
    control.draw()

text = TextBox()
ddl = DropDownList()

# this draw method is taking many shapes based on the object you pass
draw(text)
draw(ddl)
```

```
drawing TextBox
drawing DropDownList
```

Duck typing/run time polymorphism:

"If it looks like a duck and quacks like a duck, it's a duck."

"To achieve polymorphic behaviour the object passed to the function must have the certain method(in our case draw()) we are looking for, we are not concerned about the type of object passed because python is dynamic language."

Duck typing means code will simply accept any object that has a particular method(in our case draw()). Let's say we have the following code: **animal.quack()**. If the given object **animal** has the method we want to call then we're good (no additional type requirements needed). It does not matter whether animal is actually a **Duck** or a different animal which also happens to quack. That's why it is called duck typing: if it looks like a duck (e.g., it has a method called **quack()** then we can act as if that object is a duck).

To have polymorphism you don't need abstract base class but it is a

good practice in other languages.

```
class TextBox:
    def draw(self):
        print("drawing TextBox")

class DropDownList:
    def draw(self):
        print("drawing DropDownList")

# object passed here should have draw() method. we are not concerned
whats the type of object passed to it.

def draw(controls):
    for control in controls:
        control.draw() #this is the duck typing.

text = TextBox()
ddl = DropDownList()
draw([text, ddl])
```

Method overriding(run-time Polymorphism)
"In this condition child class method replace the parent class method."

It is a type of polymorphism because both constructors have same name '__init__' but different functionalities.

```
#In the below code self.age is not initialized as it get replaced by the child
class constructor.
class Animal:
    def __init__(self):
        self.age = 1

    def eat(self):
        print('eat')

class Mammal(Animal):
    def __init__(self):
        self.weight = 2

    def walk(self):
        print('walk')

m = Mammal()
print(m.age)
print(m.weight)
```

```
AttributeError: 'Mammal' object has no attribute 'age'
```

Method Overloading/Compile time polymorphism:
Method Overloading is an example of **Compile time polymorphism.** In this, more than one method of the same class shares the same method name having different signatures. Method overloading is used to add more to the behavior of methods and there is no need of more than one class for method overloading.
Note: Python does not support method overloading. We may overload the methods but can only use the latest defined method.

```
# Function to take multiple arguments
def add(datatype, *args):
    # if datatype is int
    # initialize answer as 0
    if datatype =='int':
        answer = 0

    # if datatype is str
    # initialize answer as ''
    if datatype =='str':
        answer =''
    # Traverse through the arguments
    for x in args:
        # This will do addition if the
        # arguments are int. Or concatenation
        # if the arguments are str
        answer = answer + x
    print(answer)

# Integer
add('int', 5, 6)

# String
add('str', 'Hi ', 'Geeks')
```

```
11
Hi Geeks
```

Extending builtin classes:
"Extend the functionalities of builtin types in python."

```python
class Text(str):
    #self represents current object. Here its string.
    def duplicate(self):
        return self + self

text = Text("Python")
print(text.duplicate())
```

Data Classes:

If our class only have data then use **namedtuple** instead of simple tuple.

- It has keyword argument to provide itsprameter.
- It can compare two objects and you don't have to implement extra magic method for it.

```python
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
p1 = Point(x=1, y=2)
p2 = Point(x=3, y=2)
print(p1 > p2)
```

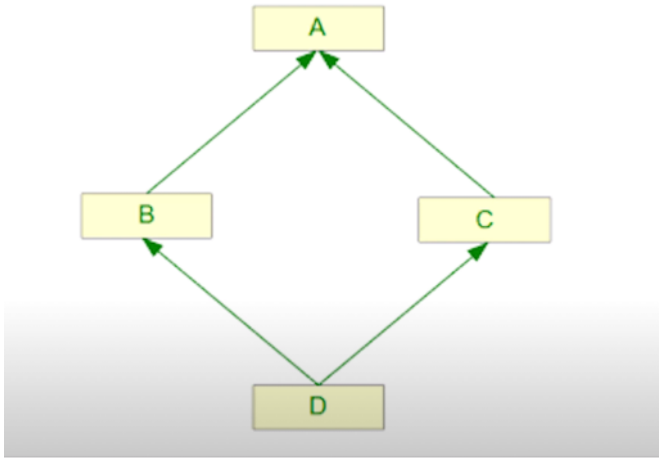
```
False
```

Questions - OOP

Q.Pillars of OOP
Q.Types of Inheritance
Q.Overriding & Overloading
Q.Constructor & Destructor
Q.Access Modifiers

Q.What is Diamond Problem?
In multiple inheritance if at the calling object program confuses which method to call a the b and c level. It don't exist in python because in python we can arrange the objects by the place of the parameters in the child class.

```python
class A:
    def met(self):
        print("This is a method from class A")

class B(A):
    def met(self):
        print("This is a method from class B")

class C(A):
    def met(self):
        print("This is a method from class C")

class D(C, B):
    def met(self):
        print("This is a method from class D")

a = A()
b = B()
c = C()
d = D()
d.met()
```



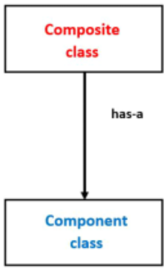
Composition , Aggregation & Association

1. **Composition (represents "Has a" relationship):**
   composition is denoted as a strong association.

   "Composition is better than inheritance"

   It can be defined as when a class can reference one or more objects of another class inside its **instance variable.**

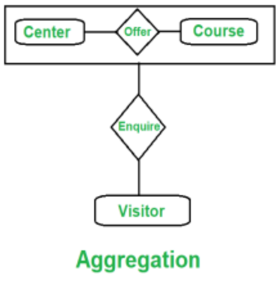
```python
    class A:
        pass

    class B:

    obj = A()
```



2. **Aggregation**
   It is often denoted as a weak association

   When an object can access another object then that relationship is called aggregation.

```python
class B(object):
    pass
```

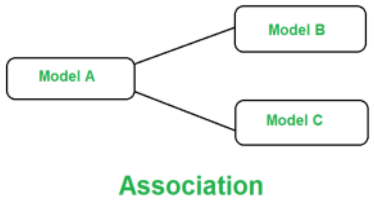


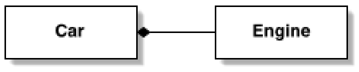
```
class A(object):
    def __init__(self, b):
        self.b = b

b = B()
a = A(b)
```

3. Association

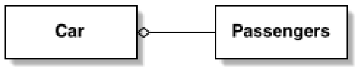
It is a weak type of relationship

Its a relationship between two classes and that relationship is established through their objects. Each object has its own life cycle and there is no owner object.





Composition: every car has an engine.



Aggregation: cars may have passengers, they come and go

Composition vs Inheritance

It's big confusing among most of the people that both the concepts are pointing to **Code Reusability** then **what is** the **difference b/w Inheritance and Composition and when to use Inheritance and when to use Composition?**

- **Inheritance** is used where a class wants to derive the nature of parent class and then modify or extend the functionality of it. **Inheritance** will extend the functionality with extra features allows **overriding of methods.**
- But in the case of **Composition**, we can only use that class we can not modify or extend the functionality of it. It will not provide extra features. Thus, when one needs to use the class as it without any modification, the composition is recommended and when one needs to change the behavior of the method in another class, then inheritance is recommended.

Difference between Aggregation and Association:

Aggregation	Association
• Aggregation describes a special type of an association which specifies a whole and part relationship.	• Association is a relationship between two classes where one class use another.
• It in flexibile in nature	• It is inflexible in nature
• Special kind of association where there is whole-part relation between two objects	• It means there is almost always a link between objects
• It is represented by a "has a"+ "whole-part" relationship	• It is represented by a "has a" relationship
• Diamond shape structure is used next to the assembly class.	• Line segment is used between the components or the class

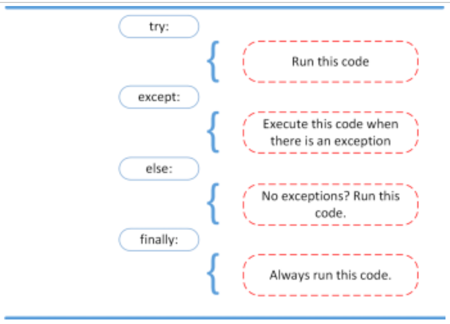
Static & Dynamic Binding

Python is a dynamically typed language. It doesn't know about the type of the variable until the code is run. So declaration is of no use. What it does is, It stores that value at some memory location and then binds that variable name to that memory container. And makes the contents of the container accessible through that variable name. So the data type does not matter. As it will get to know the type of the value at run-time.

Exception and Error Handling in Python

Errors cannot be handled, while Python exceptions can be handled at the run time.

- **Try:** It will run the code block in which you expect an error to occur.
- **Except:** Here, you will define the type of exception you expect in the try block (built-in or custom).
- **Else:** If there isn't any exception, then this block of code will be executed (consider this as a remedy or a fallback option if you expect a part of your script to produce an exception).
- **Finally:** Irrespective of whether there is an exception or not, this block of code will always be executed.


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

SOLID Principles

The SOLID principles are:

- The Single-Responsibility Principle (SRP)

- The Open-Closed Principle (OCP)
- The Liskov Substitution Principle (LSP)
- The Interface Segregation Principle (ISP)
- The Dependency inversion Principle (DIP)

Abstract Class vs Interface Class
An abstract class can have instance methods that implement a default behavior. An Interface can only declare constants and instance methods, but cannot implement default behavior and all methods are implicitly abstract. An interface has all public members and no implementation.

Singleton Pattern & Factory Pattern

Singleton Pattern
This pattern restricts the instantiation of a class to one object. It is a type of creational pattern and involves only one class to create methods and specified objects.
It provides a global point of access to the instance created.

Factory Pattern
The factory pattern comes under the creational patterns list category. It provides one of the best ways to create an object. In factory pattern, objects are created without exposing the logic to client and referring to the newly created object using a common interface.

Abstraction vs Encapsulation

Abstraction	Encapsulation
Abstraction works on the **design level.**	Encapsulation works on the **application level.**
Abstraction is implemented to **hide unnecessary data** and withdrawing relevant data.	Encapsulation is the mechanism of **hiding the code and the data** together from the outside world or misuse.
It highlights what the work of an object instead of how the object works is	It focuses on the inner details of how the object works. Modifications can be done later to the settings.
Abstraction focuses on outside viewing, for example, shifting the car.	Encapsulation focuses on internal working or inner viewing, for example, the production of the car.
Abstraction is supported in Java with the interface and the abstract class.	Encapsulation is supported using, e.g. **public, private and secure** access modification systems.
In a nutshell, abstraction is **hiding implementation with the help of an interface and an abstract class.**	In a nutshell, **encapsulation is hiding the data with the help of getters and setters.**

Coupling & Cohesion
Coupling: Coupling is the measure of the degree of interdependence between the modules. A good software will have **low coupling**.

Cohesion: Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have **high cohesion.**

Multithreading
Multithreading is defined as the ability of a processor to execute multiple threads concurrently.

Compile Time vs Runtime Polymorphism

S.NO	Method Overloading	Method Overriding
1.	In the method overloading, methods or functions must have the same name and different signatures.	Whereas in the method overriding, methods or functions must have the same name and same signatures.
2.	Method overloading is a example of compile time polymorphism.	Whereas method overriding is a example of run time polymorphism.
3.	In the method overloading, inheritance may or may not be required.	Whereas in method overriding, inheritance always required.
4.	Method overloading is	Whereas method overriding is

	performed between methods within the class.	done between parent class and child class methods.
5.	It is used in order to add more to the behavior of methods.	Whereas it is used in order to change the behavior of existing methods.
6.	In method overloading, there is no need of more than one class.	Whereas in method overriding, there is need of at least of two classes.

Class method vs Static Method

Class Method	Static Method
The class method takes cls (class) as first argument.	The static method does not take any specific parameter.
Class method can access and modify the class state.	Static Method cannot access or modify the class state.
The class method takes the class as parameter to know about the state of that class.	Static methods do not know about class state. These methods are used to do some utility tasks by taking some parameters.
@classmethod decorator is used here.	**@staticmethod** decorator is used here.