Syllabus - Module IV

Design with classes - Objects and Classes, Methods, Instance Variables, Constructor, Accessors and Mutators. Structuring classes with Inheritance and Polymorphism. Abstract Classes. Exceptions - Handle a single exception, handle multiple exceptions.

Overview of OOP (Object Oriented Programming)

- Python is a multi-paradigm programming language. It supports different programming approaches.

- One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

(1) attributes

(2) behavior

Suppose a parrot is an object, as it has the following properties:

name, age, color as attributes

singing, dancing as behavior

Object-oriented vs. Procedure-oriented Programming Languages

Object-oriented Programming	Procedural Programming
Object-oriented programming is the problem-solving approach and used where computation is done by using objects.	Procedural programming uses a list of instructions to do computation step by step.
It makes the development and maintenance easier	In procedural programming, It is not easy to maintain the codes when the project becomes lengthy.
It simulates the real world entity. So real-world problems can be easily solved through oops.	It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions.
It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere.	Procedural language doesn't provide any proper way for data binding, so it is less secure.
Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc.	Example of procedural languages are: C, Fortran, Pascal, VB etc.

Objects and Classes

- Object is an instance of a class.
- A class is a collection of data(variables) and methods(functions).
- A class is the basic structure of an object and is a set of attributes, which can be data members or method members.

Some important terms in OOP are as follows:

- ✓ Class - They are defined by the user. The class provide basic structure for an object.
- ✓ Data Member - A variable defined in either a class or an object. It holds the data associated with the class or object.
- ✓ Instance variable - A variable that is defined in a method; its scope is only within the object that defines it.
- ✓ Class Variable - A variable that is defined in the class and can be used by all the instances of the class.

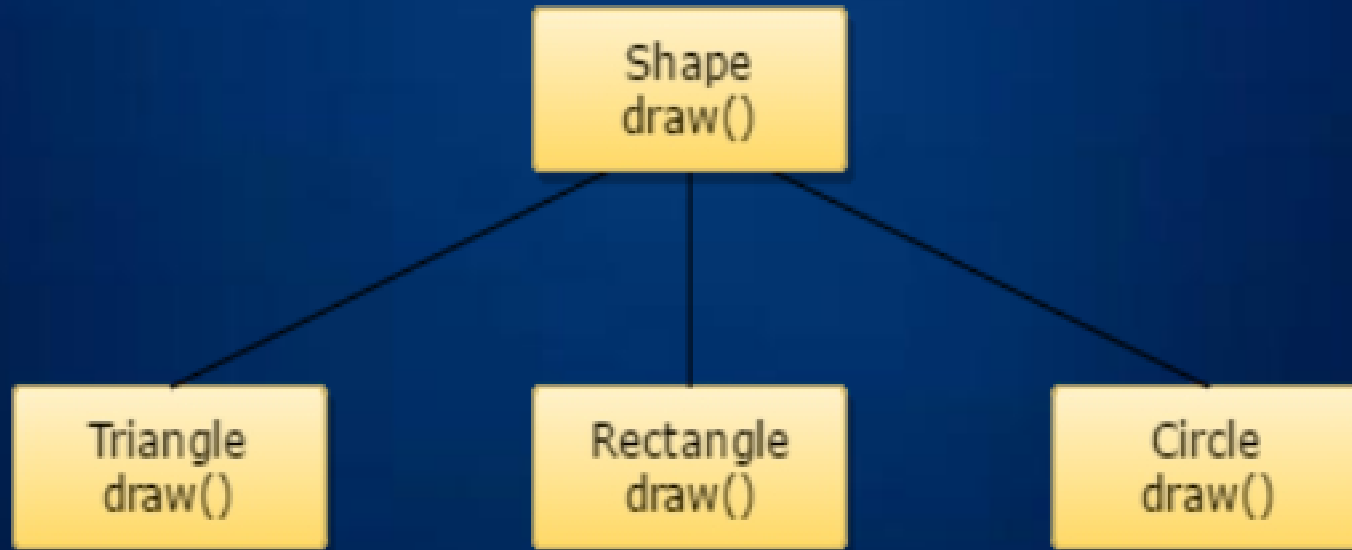
- ✓Instance - An object is an instance of the class.
- ✓Instantiation - The process of creation of an object of a class.
- ✓Method - Methods are the functions that are defined in the definition of class and are used by various instances of the class.
- ✓Function Overloading - A function defined more than one time with different behaviours is known as function overloading. The operations performed by these functions are different.
- ✓Inheritance - a class A that can use the characteristics of another class B is said to be a derived class. ie., a class inherited from B. The process is called inheritance.

Data Encapsulation:

- In OOP, restrictions can be imposed on the access to methods and variables. Such restrictions can be used to avoid accidental modification in the data and are known as Encapsulation.

- Polymorphism

The term polymorphism means that the object of a class can have many different forms to respond in different ways to any message or action.



Class Definition

- A class is a “blueprint” for creating objects.
- It can also be defined as a group of objects that share similar attributes and relationships with each other.

eg., Fruit is a class and apple, mango and banana are its objects.

The attributes of these objects can be color, taste etc.

Syntax for defining a class

In python class is created by using a keyword class

After that, the first statement can be a docstring that contains the information about the class.

In the body of class, the attributes (data members or method members) are defined.

```
class <class name>(<parent class name>):
 <method definition-1>
 ...
 <method definition-n>
```

- In python, as soon as we define a class, the iterpreter instantly creates an object that has the same name as the class name.
- We can create more objects of the same class. With the help of objects, we can access the attributes defined in the class.

syntax for creating objects for a class

```
<object name> = <classname>
```

Creating class Student

```
class Student:
    '''Student details'''
    def fill_details(self,name,branch,year):
        self.name = name
        self.branch = branch
        self.year = year
        print("A student detail object is created...")
    def print_details(self):
        print("Name: ",self.name)
        print("Branch: ",self.branch)
        print("Year: ",self.year)
```

Creating an object of Student class

```
>>> s1 = Student()
>>> s2 = Student()
>>> s1.fill_details('Rahul','CSE','2020')
A student detail object is created...
>>> s1.print_details()
Name:  Rahul
Branch:  CSE
Year:  2020
>>> s2.fill_details('Akhil','ECE','2010')
A student detail object is created...
>>> s2.print_details()
Name:  Akhil
Branch:  ECE
Year:  2010
```

Constructors in Python

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Creating the constructor in python

Class functions that begin with double underscore "__" are called special functions as they have special meaning.

In Python, the method the __init__() simulates the constructor of the class. This method is called when the class is instantiated. It accepts the self-keyword as a first argument which allows accessing the attributes or method of the class.

Example : Display a Complex Number

```
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def getNumber(self):
        print(f'{self.real} + j{self.imag}')

num1 = ComplexNumber(2, 3)
num1.getNumber()
```

Output : 2+j3

Example : Display Employee Details

```
class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name

    def display(self):
        print("Id: %d\n Name: %s" % (self.id, self.name))


emp1 = Employee(1, "john")
emp2 = Employee(2, "Anu")

emp1.display()
emp2.display()
```

Output:

```
Id: 1
Name: john
Id: 2
Name: Anu
```

Python Non-Parameterized Constructor

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument.

```
class Student:
    def __init__(self):
        print("This is non parameterized 
constructor")

    def show(self, name):
        print("Hello ", name)

s1 = Student()
s1.show("Roshan")
```

output:

This is non parameterized constructor
Hello Roshan

Python Parameterized Constructor

The parameterized constructor has multiple parameters along with the self.

```
class Student:
    def __init__(self, name):
        print("This is parameterized constructor")
        self.name = name

    def show(self):
        print("Hello ", self.name)


s1 = Student("Rahul")
s1.show()
```

output:

This is parameterized constructor
Hello Rahul

Python Default Constructor

When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor. It does not perform any task but initializes the objects.

```
class Student:
    name = "Jeevan"
    rollno = 101

    def show(self):
        print("Name: ", self.name)
        print("Roll No: ", self.rollno)
s1 = Student()
s1.show()
```

Output:

```
Name:  Jeevan
Roll No:  101
```

Example : The Student class

A course-management application needs to represent information about students in a course. Each student has a name and a list of test scores. We can use these as the attributes of a class named Student. The Student class should allow the user to view a student's name, view a test score at a given position (counting from 1), reset a test score at a given position, view the highest test score, view the average test score, and obtain a string representation of the student's information.

The __str__ Method :

Many built-in Python classes usually include an __str__ method. This method builds and returns a string representation of an object's state. When the str function is called with an object, that object's __str__ method is automatically invoked to obtain the string that str returns.

```
"""
File: student.py
Resources to manage a student's name and test scores.
"""
class Student(object):
 """Represents a student."""
def__init__(self, name, number):
 """Constructor creates a Student with the given
 name and number of scores and sets all scores
 to 0."""
 self.name = name
 self.scores = []
 for count in range(number):
 self.scores.append(0)
```

```
def getName(self):
"""Returns the student's name."""
return self.name
def setScore(self, i, score):
"""Resets the ith score, counting from 1."""
self.scores[i - 1] = score
def getScore(self, i):
"""Returns the ith score, counting from 1."""
return self.scores[i - 1]
def getAverage(self):
"""Returns the average score."""
return sum(self.scores) / len(self.scores)
```

```
def getHighScore(self):
"""Returns the highest score."""
return max(self.scores)
def __str__(self):
"""Returns the string representation of the
student."""
return "Name: " + self.name + "\nScores: " + \
" ".join(map(str, self.scores))
```

Accessors and Mutators

- The Methods that allow a user to observe but not change the state of an object are called accessors.
- Methods that allow a user to modify an object's state are called mutators.

Accessor Method: This method is used to access the state of the object i.e, the data hidden in the object can be accessed from this method. However, this method cannot change the state of the object, it can only access the data hidden. We can name these methods with the word get.

Mutator Method: This method is used to mutate/modify the state of an object i.e, it alters the hidden value of the data variable. It can set the value of a variable instantly to a new value. This method is also called as update method. Moreover, we can name these methods with the word set.

Example : Accessors and Mutators

```
Class Fruit:
    def __init__(self, name):
        self.name = name

    def setFruitName(self, name):
        self.name = name

    def getFruitName(self):
        return self.name

f1 = Fruit("Apple")
print("First fruit name: ", f1.getFruitName())
f1.setFruitName("Grape")
print("Second fruit name: ", f1.getFruitName())
```

Output:

```
First fruit name: Apple
Second fruit name:  Grape
```

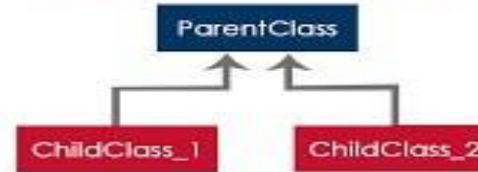
Inheritance in Python

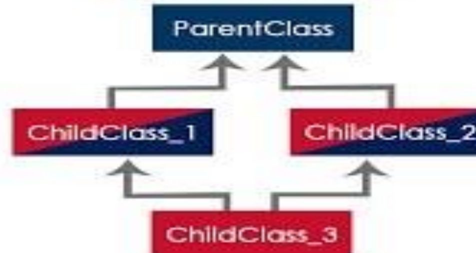
- Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.

Syntax:

```
class derived class name(base class):
        <class-suite>
```

Types of Inheritance





www.btechsmartclass.com

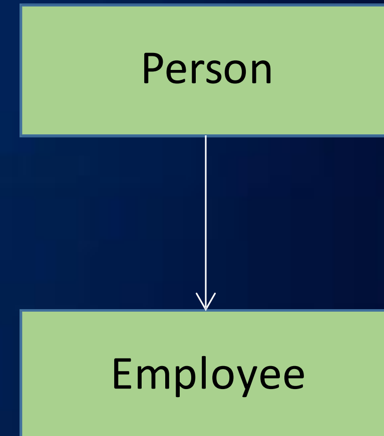
Single Inheritance

- When a child class inherits from only one parent class, it is called single inheritance.
- *class derive-class(<base class >):*

  <class - suite>

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

    def getName(self):
        return self.name

    def isEmployee(self):
        return False
```

```
class Employee(Person):
    def isEmployee(self):
        return True


p = Person("Anu")
print(p.getName(),
p.isEmployee())

e = Employee("Ammu")
print(e.getName(),
e.isEmployee())
```

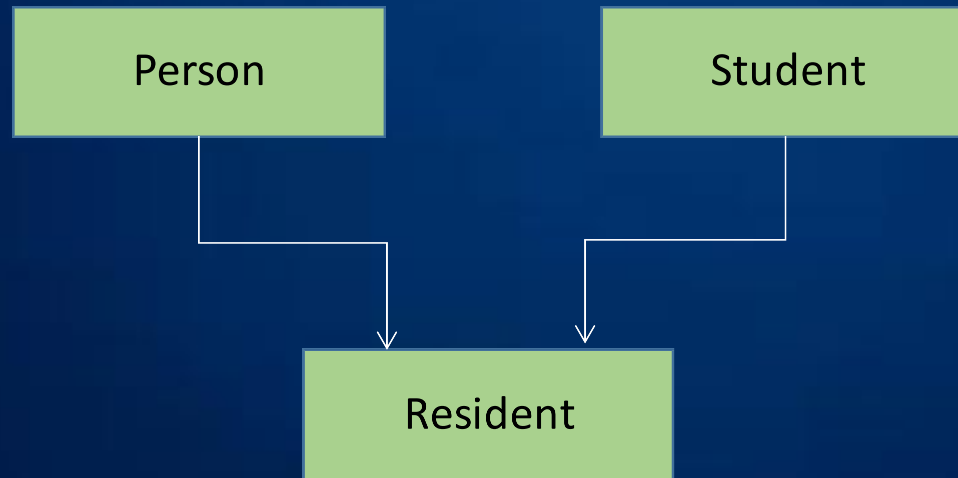


Output:
Anu False
Ammu True

Multiple inheritance

- When a child class inherits from multiple parent classes, it is called multiple inheritance.
- class derive-class(<base class 1>, <base class 2>, ... <base class n>):

  <class - suite>



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

    def showName(self):
        print(self.name)

    def showAge(self):
        print(self.age)
```

```
class Student:
    def __init__(self, rollno):
        self.rollno = rollno

    def getRollno(self):
        print(self.rollno)

class Resident(Person, Student):
    def __init__(self, name, age, rollno):
        Person.__init__(self, name, age)
        Student.__init__(self, rollno)
```

```
r = Resident("Roshan", 21, 101)
r.showName()
r.showAge()
r.getRollno()
```

Output:

```
Roshan
21
101
```

Resolving the Conflict with Python Multiple Inheritance:

```
class A:
    def __init__(self):
        self.name = "John"
        self.age = 23

    def getName(self):
        return self.name


class B:
    def __init__(self):
        self.name = "Richard"
        self.id = 32

    def getName(self):
        return self.name
```

```
class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)

    def getName(self):
        return self.name

C1 = C()
print(C1.getName())
```

Output:

Richard

Method Resolution Order (MRO)

MRO works in a depth first left to right way. super() in the __init__ method indicates the class that is in the next hierarchy. At first the the super() of C indicates A.

Then super in the constructor of A searches for its superclass. If it doesn't find any, it executes the rest of the code and returns. So the order in which constructors are called here is:

C -> A -> B

Resolving the Conflict with Python Multiple Inheritance:

```
class A:
    def __init__(self):
        super().__init__()
        self.name = 'John'
        self.age = 23

    def getName(self):
        return self.name

class B:
    def __init__(self):
        super().__init__()
        self.name = 'Richard'
        self.id = '32'

    def getName(self):
        return self.name

class C(A, B):
    def __init__(self):
        super().__init__()

    def getName(self):
        return self.name

C1 = C()
print(C1.getName())
print(C.__mro__)
```

Output : John

Multilevel Inheritance

- This is achieved when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is achieved in python.

- *Syntax*

class class1:

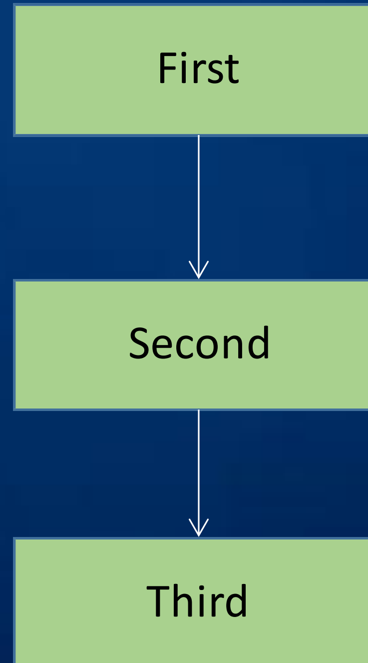
<class-suite>

class class2(class1):

<class suite>

class class3(class2):

<class suite>



```python
class First:
    def first(self):
        print("I am the first class")

class Second(First):
    def second(self):
        print("I am the second class")

class Third(Second):
    def third(self):
        print("I am the third class")
```

```python
t = Third()
t.first()
t.second()
t.third()
```

Output:

```
I am the first class
I am the second class
I am the third class
```

Hierarchical Inheritance

- When more than one derived classes are created from a single base – it is called hierarchical inheritance.
- *Syntax*

class class1:

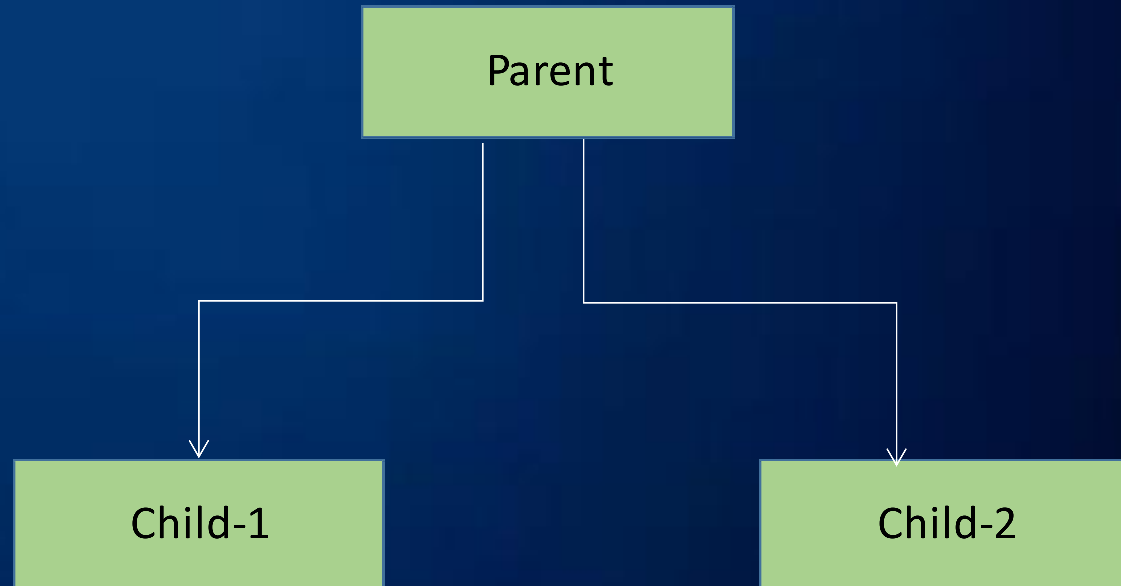
<class-suite>

class class2(class1):

<class suite>

class class3(class1):

<class suite>



```
class Parent:
    def func1(self):
        print("This function is in
Parent")


class Child1(Parent):
    def func2(self):
        print("This function is in
child1")


class Child2(Parent):
    def func3(self):
        print("This function is in
child3")
```

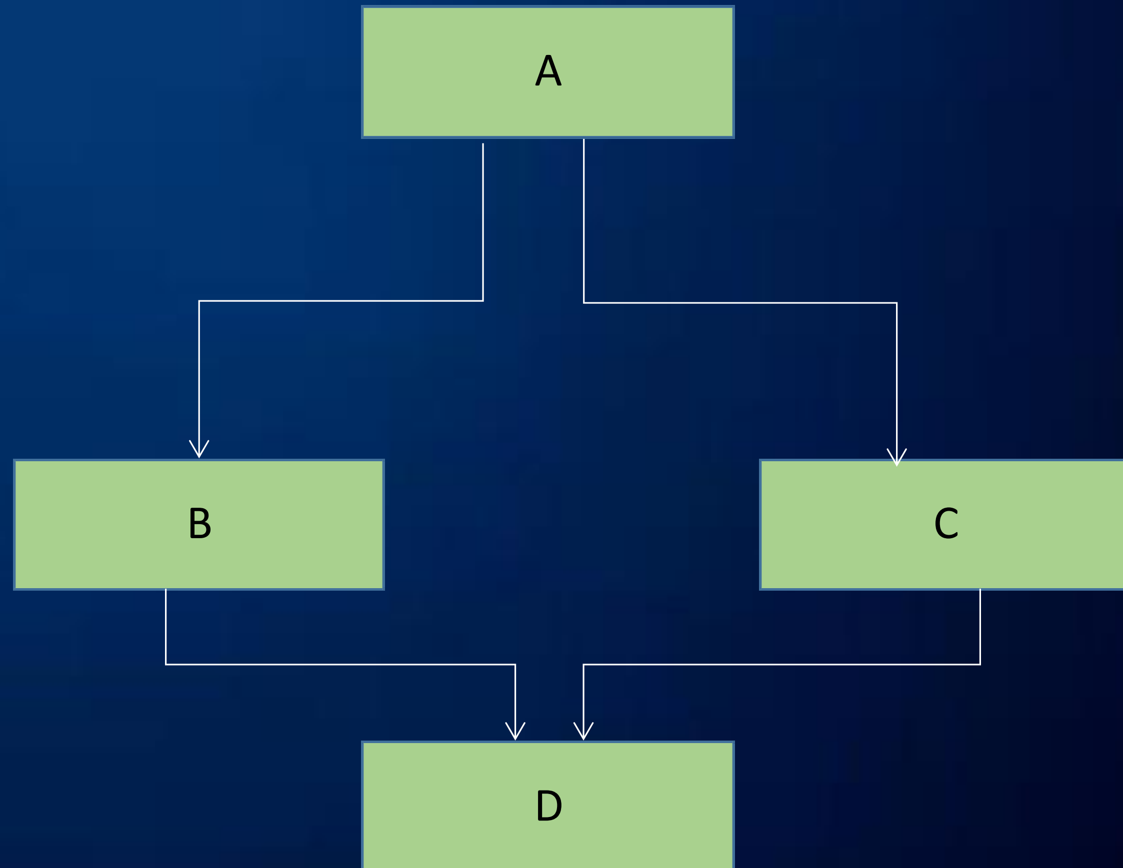
```
c1 = Child1()
c2 = Child2()

c1.func1()
c1.func2()

c2.func1()
c2.func3()
```

Output:

```
This function is in child1
This function is in Parent
This function is in child3
```

Hybrid Inheritance

- The hybrid inheritance is the combination of more than one type of inheritance. We may use any combination as a single with multiple inheritances, multi-level with multiple inheritances, etc.



Use the super() Function

- Python also has a super() function that will make the child class inherit all the methods and properties from its parent:
- By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

```
x = Student("John", "Samuel")
x.printname()
```

Output:
John Samuel

Polymorphism in Python

- The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being uses for different types.

- It is a very important concept in programming. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

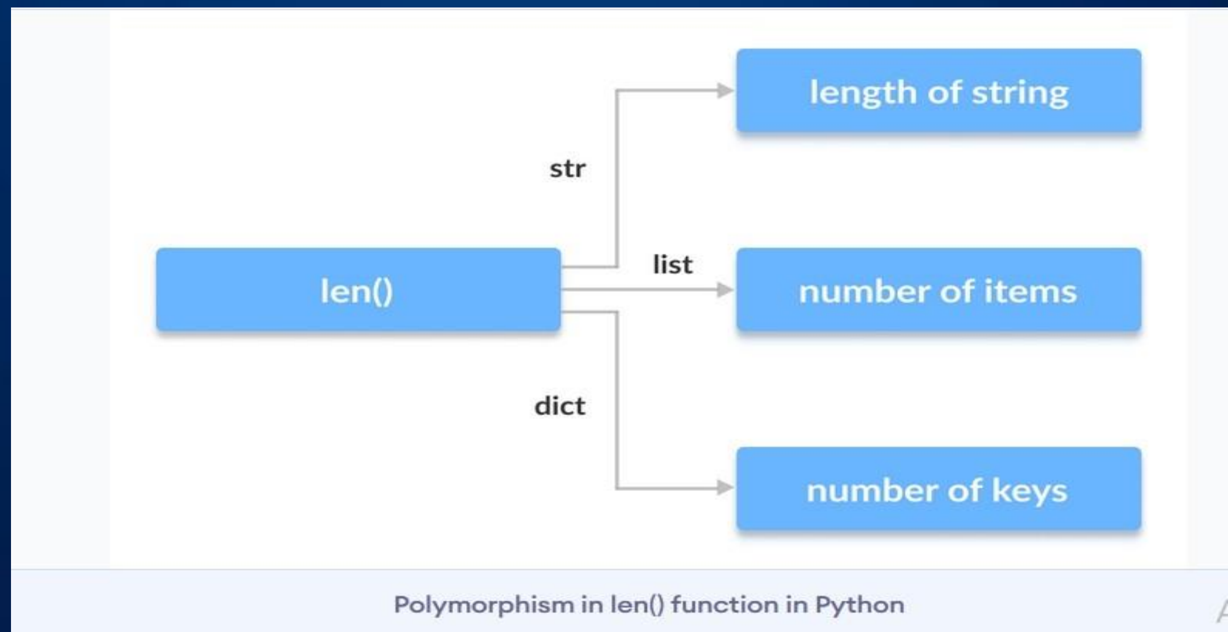
- **Polymorphism in addition operator**

```
num1 = 1
num2 = 2
print(num1+num2)
o/p : 3
```

```
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
o/p : Python Programming
```

- **Function polymorphism in python**

```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```



Polymorphism in len() function in Python

- **Polymorphism in Class Methods**

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def findArea(self):
        print("Area: ", self.length * self.width)

    def printInfo(self):
        print("This is a Rectangle")
```

```
class Shape:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def findArea(self):
        print("Area: ", self.length * self.width)

    def printInfo(self):
        print("This is a geometric shape")
```

```python
rect1 =  Rectangle(12, 10)
sh1 = Shape(10, 13)

for value in (rect1, sh1):
    value.printInfo()
    value.findArea()
```

Output:

This is a Rectangle
Area: 120
This is a geometric shape
Area: 130

- **Polymorphism and Inheritance**

Like in other programming languages, the child classes in Python also inherit methods and attributes from the parent class. We can redefine certain methods and attributes specifically to fit the child class, which is known as Method Overriding.

In Python, to override a method, you have to meet certain conditions, and they are:

- You can't override a method within the same class. It means you have to do it in the child class using the Inheritance concept.

- To override the Parent Class method, you have to create a method in the Child class with the same name and the same number of parameters.

```python
class Parent:
    def __init__(self):
        self.value = "Inside Parent"

    def show(self):
        print(self.value)


class Child(Parent):
    def __init__(self):
        self.value = "Inside Child"

    def show(self):
        print(self.value)
```

```python
obj1 = Parent()
obj2 = Child()

obj1.show()
obj2.show()
```

<u>Output:</u>

Inside Parent
Inside Child

Abstraction

Abstraction means hiding the complexity and only showing the essential features of the object. So in a way, Abstraction means hiding the real implementation and we, as a user, knowing only how to use it.

Abstract Class in Python

- An abstract class is a class that contains one or more abstract methods.
- An Abstract method is a method that generally doesn't have any implementation, it is left to the sub classes to provide implementation for the abstract methods.
- Abstract class can't be instantiated so it is not possible to create objects of an abstract class.

Abstract Class in Python

In Python abstract class is created by deriving from the meta class ABC which belongs to the abc (Abstract Base Class) module.

Syntax for creating Abstract Class

```
from abc import ABC
class MyClass(ABC):
```

Abstract Method in Python

- For defining abstract methods in an abstract class, method has to be decorated with @abstractmethod decorator.
- From abc module @abstractmethod decorator has to be imported to use that annotation.

Syntax for defining abstract method in abstract class in Python

```
from abc import ABC, abstractmethod
class MyClass(ABC):
@abstractmethod
        def mymethod(self):
          pass #empty body
```

Important points about abstract class in Python

- Abstract class can have both concrete methods as well as abstract methods.
- Abstract class works as a template for other classes.
- Abstract class can’t be instantiated so it is not possible to create objects of an abstract class.
- Generally abstract methods defined in abstract class don’t have any body but it is possible to have abstract methods with implementation in abstract class.
- If any abstract method is not implemented by the derived class Python throws an error.

```python
from abc import ABC, abstractmethod

class Parent(ABC):
    def common(self):
        print("I am the common of parent")

    @abstractmethod
    def vary(self):
        pass

class Child1(Parent):
    def vary(self):
        print("I am vary of child1")

class Child2(Parent):
    def vary(self):
        print("I am vary method of child2")

obj1 = Child1()
obj1.common()
obj1.vary()
obj2 = Child2()
obj2.common()
obj2.vary()
```

o/p

```
I am the common of parent
I am vary of child1
I am the common of parent
I am vary method of child2
```

Encapsulation

- Encapsulation in Python describes the concept of **bundling data and methods within a single unit**
- Using encapsulation, we can hide an object's internal representation from the outside. This is called information hiding.
- Also, encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class.
- Encapsulation is a way to can restrict access to methods and variables from outside of class. Whenever we are working with the class and dealing with sensitive data, providing access to all variables used within the class is not a good choice.

Access modifiers limit access to the variables and methods of a class. Python provides three types of access modifiers private, public, and protected.

- **Public Member**: Accessible anywhere from otside oclass.
- **Private Member**: Accessible within the class
- **Protected Member**: Accessible within the class and its sub-classes

```
class Employee:

    def __init__(self, name, salary):

        self.name = name
        self._project = project
        self.__salary = salary
```

Errors and Exceptions

- Two common kinds of errors that you may have to deal with are

✓ Syntax errors

✓ Exceptions

Syntax Errors:

It occur when you type the code incorrectly.

Exceptions:

They are different from syntax errors. They occur during the execution of a program when something unexpected happens.

SYNTAX ERRORS

Syntax Errors

Syntax errors are errors in your code that the computer cannot interpret. In Python these errors are often:

- spelling errors
- the omission of important characters (such is a missing colon)
- Inconsistent use of / wrong indentation

Very common for new programmers...

Runtime Errors

Theres are errors that are not detected until run time.

These are often caused by:

- It can't find some data because it doesn't exist
- It can't perform an action on the data it has been given because it is an invalid type of data.

These types of error only found at runtime and often through lots of testing….

Logical Errors

These are errors in the code that do not throw an error at all, but simply do not do what you intended the code to do.

These are the most difficult to spot, because they can only be found through full and extensive testing.

RUNTIME ERRORS(EXCEPTIONS)

Some Common Built-inExceptions

- NameError: This exception is raised when the program cannot find a local or global name. The name that could not be found is included in the error message.

- TypeError: This exception is raised when a function is passed an object of the inappropriate type as its argument. More details about the wrong type are provided in the error message.

- ValueError: This exception occurs when a function argument has the right type but an inappropriate value.

- NotImplementedError: This exception is raised when an object is supposed to support an operation but it has not been implemented yet. You should not use this error when the given function is not meant to support the type of input argument. In those situations, raising a TypeError exception is more appropriate.

- ZeroDivisionError: This exception is raised when you provide the second argument for a division or modulo operation as zero.

- FileNotFoundError: This exception is raised when the file or directory that the program requested does not exist

IndexError

The `IndexError` is thrown when trying to access an item at an invalid index.

Example: IndexError

```
>>> L1=[1,2,3]
>>> L1[3]
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>

L1[3]
IndexError: list index out of range
```

ModuleNotFoundError

The `ModuleNotFoundError` is thrown when a module could not be found.

Example: ModuleNotFoundError

```
>>> import notamodule
Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>

import notamodule
ModuleNotFoundError: No module named 'notamodule'
```

KeyError

The `KeyError` is thrown when a key is not found.

Example: KeyError

```
>>> D1={'1':"aa", '2':"bb", '3':"cc"}
>>> D1['4']
Traceback (most recent call last):
File "<pyshell#15>", line 1, in <module>


D1['4']
KeyError: '4'
```

ImportError

The `ImportError` is thrown when a specified function can not be found.

Example: ImportError

```
>>> from math import cube
Traceback (most recent call last):
File "<pyshell#16>", line 1, in <module>

from math import cube
ImportError: cannot import name 'cube'
```

Example: StopIteration

```
>>> it=iter([1,2,3])
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
File "<pyshell#23>", line 1, in <module>

next(it)
StopIteration
```

TypeError

The `TypeError` is thrown when an operation or function is applied to an object of an inappropriate type.

Example: TypeError

```
>>> '2'+2
Traceback (most recent call last):
File "<pyshell#23>", line 1, in <module>

'2'+2
```

StopIteration

The `StopIteration` is thrown when the `next()` function goes beyond the iterato
items.

Example: StopIteration

```
>>> it=iter([1,2,3])
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
File "<pyshell#23>", line 1, in <module>

next(it)
StopIteration
```

ValueError

The `ValueError` is thrown when a function's argument is of an inappropriate type.

Example: ValueError

```
>>> int('xyz')
Traceback (most recent call last):
File "<pyshell#14>", line 1, in <module>

int('xyz')
ValueError: invalid literal for int() with base 10: 'xyz'
```

NameError

The `NameError` is thrown when an object could not be found.

Example: NameError

```
>>> age
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>

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

ZeroDivisionError

The `ZeroDivisionError` is thrown when the second operator in the division is zero.

Example: ZeroDivisionError

```
>>> x=100/0
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module>

x=100/0
ZeroDivisionError: division by zero
```

KeyboardInterrupt

The `KeyboardInterrupt` is thrown when the user hits the interrupt key (normally Control-C) during the execution of the program.

Example: KeyboardInterrupt

```
>>> name=input('enter your name')
enter your name^c
Traceback (most recent call last):
File "<pyshell#9>", line 1, in <module>

name=input('enter your name')
KeyboardInterrupt
```

Handling an Exception

```
a = True
while a:
    x = int(input("Enter a number: "))
    print("Dividing 50 by", x, "will give you:", 50/x)
```

You will get Value Error on entering decimal number or string as input.

To handle the exception,

The first step of the process is to include the code that you think might raise an exception inside the try clause. The next step is to use the except keyword to handle the exception that occurred in the above code

```python
a = True
while a:
    try:
        x = int(input("Please enter a number: "))
        print("Dividing 50 by", x, "will give you: ", 50/x)

    except ValueError:
        print("The input was not an integer. Please try again...")
```

Output:

```
Please enter a number: a
The input was not an integer. Please try again...
Please enter a number: 2
Dividing 50 by 2 will give you:  25.0
```

- If no exception was raised, the program skips the except clause and the rest of the code executes normally.
- If an exception is raised, the program skips the remaining code inside the try clause and the type of the exception is matched with the name of the exception after the except keyword.

In case of a match, the code inside the except clause is executed first, and then the rest of the code after the try clause is executed normally.

- When you enter an integer as an input, the program gives you the final result of the division.
- When a non-integral value is provided, the program prints a message asking you to try and enter an integer again.
- Note that this time, the program does not abruptly quit when you provide some invalid input

- You can also handle multiple exceptions using a single except clause by passing these exceptions to the clause as a tuple.

```
except (ZeroDivisionError, ValueError, TypeError):
    print("Something has gone wrong..")
```

- One possible use of catching all exceptions is to properly print out the exception error on screen like the following code:

```
import math
import sys
try:
    result = math.factorial(2.4)
except:
    print("Something Unexpected has happened.",sys.exc_info()[0])
else:
    print("The factorial is", result)
```

Using the Else Clause

- The else clause is meant to contain code that needs to be executed if the try clause did not raise any exceptions.
- if you decide to use an else clause, you should include it after all the except clauses but before the finally block.

```
a = True
while a:
    try:
        x = int(input("Please enter a number: "))
    except ValueError:
        print("The input was not a valid integer. Please try again...")
    else:
        print("Dividing 50 by", x, "will give you :", 50 / x)
```

Using the Finally Clause

- The code inside the finally clause is always executed irrespective of whether the try block raised an exception.

```
a = True
while a:
    try:
        x = int(input("Please enter a number: "))
    except ValueError:
        print("The input was not a valid integer. Please try again...")
    else:
        print("Dividing 50 by", x, "will give you :", 50/x)
    finally:
        print("Already did everything necessary.")
```

Output:

Please enter a number: 2

Dividing 50 by 2 will give you : 25.0

Already did everything necessary.

Please enter a number: d

The input was not a valid integer. Please try again...

Already did everything necessary.

Please enter a number:

Exception with Arguments

Using arguments for Exceptions in Python is useful for the following reasons:

- ✓ It can be used to gain additional information about the error encountered.
- ✓ As contents of an Argument can vary depending upon different types of Exceptions in Python, Variables can be supplied to the Exceptions to capture the essence of the encountered errors. Same error can occur of different causes, Arguments helps us identify the specific cause for an error using the except clause.
- ✓ It can also be used to trap multiple exceptions, by using a variable to follow the tuple of Exceptions.

Arguments in Built-in Exceptions

```
try:
    b = float(100+50/0)
except Exception as argument:
    print("This is the argument\n", argument)
```

Output:

This is the argument

division by zero

```
s = "Hello Python"
try:
    b = float(s/20)
except Exception as argument:
    print("This is the argument: \n",
argument)
```

Output:

This is the argument:

unsupported operand type(s) for /: 'str' and 'int'

Raising exceptions

An exception can be raised forcefully by using the raise clause in Python. It is useful in that scenario where we need to raise an exception to stop the execution of the program.

Syntax: raise Exception_class,<value>

- To raise an exception, the raise statement is used. The exception class name follows it.
- An exception can be provided with a value that can be given in the parenthesis.
- To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.
- We can pass the value to an exception to specify the exception type.

```python
try:
    age = int(input("Enter the age:"))
    if(age<18):
        raise ValueError
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

Output:

```
Enter the age:16
The age is not valid
```

User-Defined Exceptions

In Python, users can define custom exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from the built-in Exception class. Most of the built-in exceptions are also derived from this class.

```
class customError(Exception):
    pass
raise customError
```

Here, we have created a user-defined exception called CustomError which inherits from the Exception class. This new exception, like other exceptions, can be raised using the raise statement with an optional error message.

Deriving Error from Superclass

- ✓Super class exceptions are created when a module needs to handle several distinct errors.
- ✓One of the common way of doing this is to create a base class for exceptions defined by that module.
- ✓Further, various sub classes are defined to create specific exception classes for different error conditions.

```python
class Error(Exception):
    pass
class ValueTooSmallError(Error):
    pass
class ValueTooLargeError(Error):
    pass
number = 10
while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")

    except ValueTooLargeError:
        print("This value is too large, try again!")

print("Congratulations! You guessed it correctly.")
```

Output:

Enter a number: 3

This value is too small, try again!

Enter a number: 5

This value is too small, try again!

Enter a number: 11

This value is too large, try again!

Enter a number: 10

Congratulations! You guessed it correctly.

Thank You