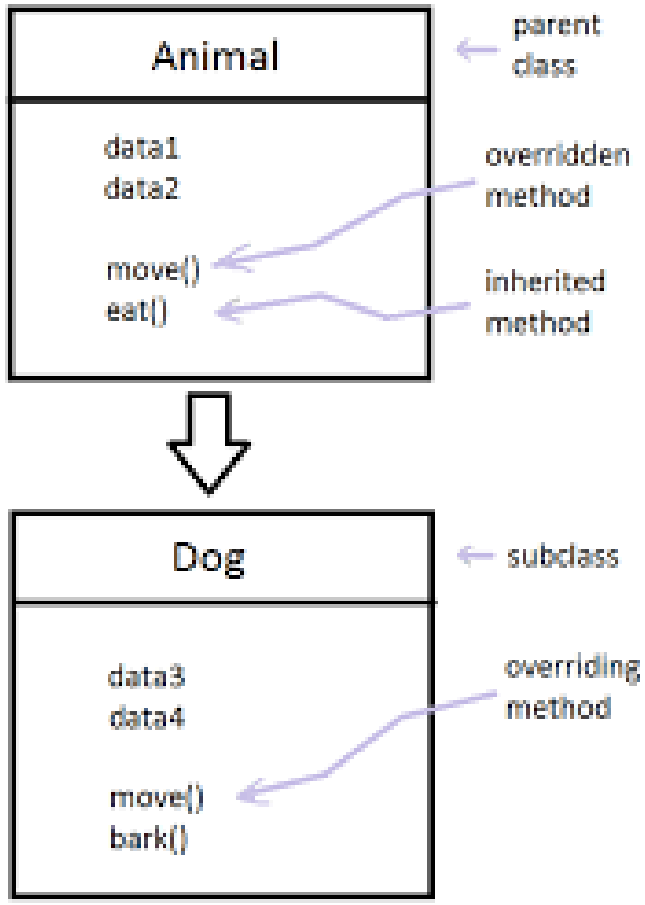
Inheritance, Polymorphism, Method overriding, Decorators ,Exception Handling

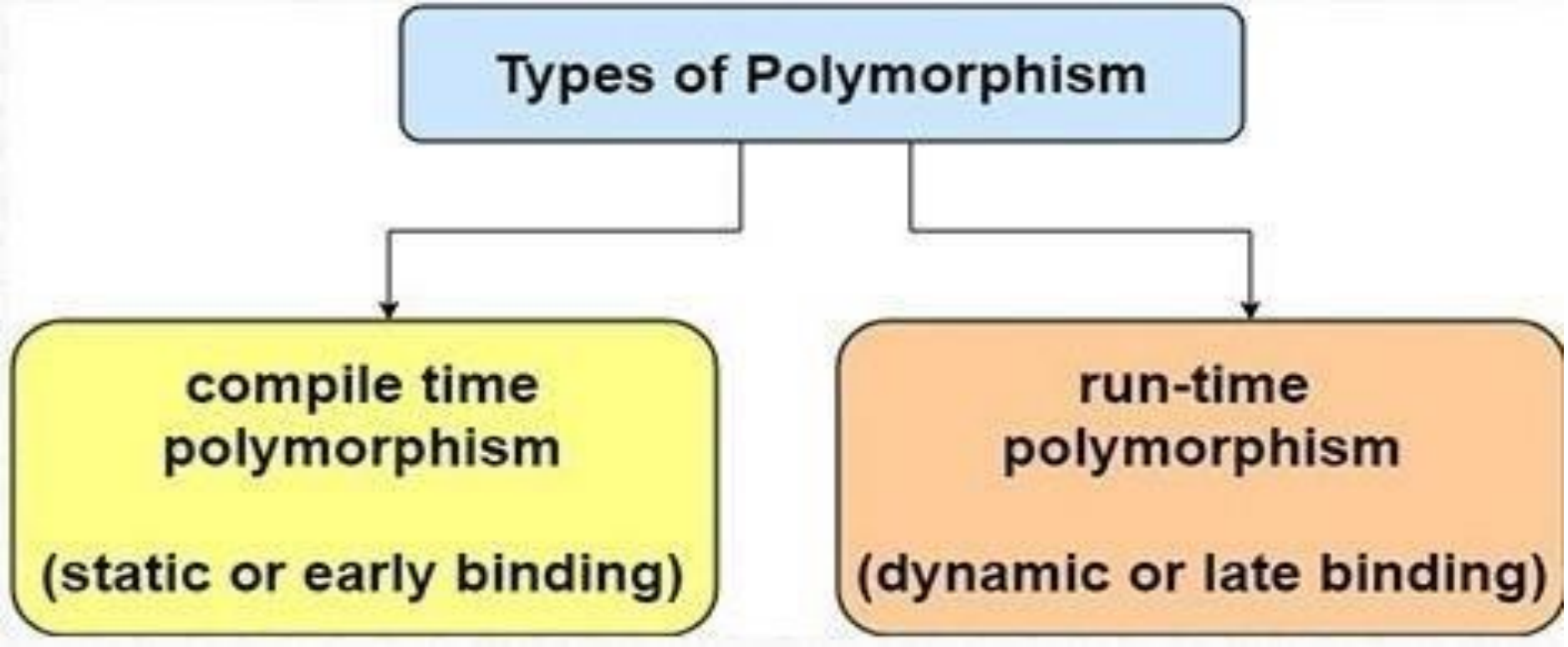
Recompiled -
Naiswita Parmar
Assistant Professor,
CSE, SET,
Navrachana University

Polymorphism

- Greek-term means *"ability to take more than one form"*
- Same method name but different implementations.



Method Overloading – Compile time polymorphism



Method overloading

Method overriding

Polymorphism

- Method overloading:
- 2 methods having same name in same class, BUT different no. and type of arguments.

Method Overloading

- python does not support method overloading by default. But there are different ways to achieve method overloading in Python
- The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method

Method Overloading

```
# First product method.
# Takes two argument and print their
# product
def product(a, b):
        p = a * b
        print(p)
```

```
# Second product method
# Takes three argument and print their
# product
def product(a, b, c):
        p = a * b*c
        print(p)

# Uncommenting the below line shows an error
# product(4, 5)

# This line will call the second product method
product(4, 5, 5)
```

```
# First product method.
# Takes two argument and print their
# product
def product(a, b):
    p = a * b
    print(p)

# Second product method
# Takes three argument and print their
# product
def product(a, b, c):
    p = a * b*c
    print(p)

# Uncommenting the below line shows an error
# product(4, 5)

# This line will call the second product method
product(4, 5, 5)
```

Output:

```
100
```

Method Overloading

- In the above code, we have defined two product method, but we can only use the second product method, as python does not support method overloading.
- We may define many methods of the same name and different arguments, but we can only use the latest defined method. Calling the other method will produce an error.
- Like here calling **W**ill produce an error as the latest defined product method takes three arguments.
- Thus, to overcome the above problem we can use different ways to achieve the method overloading.

Method Overloading

- **Method 1 (Not The Most Efficient Method):**
  We can use the arguments to make the same function work differently i.e. as per the arguments.
- The problem with below code is that makes code more complex with multiple if/else statement and is not the desired way to achieve the method overloading.

```
# 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 ', 'NUV')
```

```python
# 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 ', 'NUV')
```

```
11
Hi NUV


...Program finished with exit code 0
Press ENTER to exit console.
```

Method Overloading

- **Method 2 (Efficient One):**
  By Using Multiple Dispatch Decorator
  Multiple Dispatch Decorator Can be installed by:

```
pip3 install multipledispatch
```

- **Multiple dispatch in Python**
- Multiple dispatch (aka multimethods, generic functions, and function overloading) is choosing which among several function bodies to run, depending upon the arguments of a call.
- In Backend, Dispatcher creates an object which stores different implementation and on runtime, it selects the appropriate method as the type and number of parameters passed.

```
from multipledispatch import dispatch

#passing one parameter
@dispatch(int,int)
def product(first,second):
    result = first*second
    print(result);

#passing two parameters
@dispatch(int,int,int)
def product(first,second,third):
    result  = first * second * third
    print(result);
```

```
#you can also pass data type of any value as per requirement
@dispatch(float,float,float)
def product(first,second,third):
    result  = first * second * third
    print(result);


#calling product method with 2 arguments
product(2,3,2) #this will give output of 12
product(2.2,3.4,2.3) # this will give output of 17.985999999999997
```

```python
from multipledispatch import dispatch

#passing one parameter
@dispatch(int,int)
def product(first,second):
    result = first*second
    print(result);

#passing two parameters
@dispatch(int,int,int)
def product(first,second,third):
    result = first * second * third
    print(result);

#you can also pass data type of any value as per requirement
@dispatch(float,float,float)
def product(first,second,third):
    result = first * second * third
    print(result);


#calling product method with 2 arguments
product(2,3,2) #this will give output of 12
product(2.2,3.4,2.3) # this will give output of 17.985999999999997
```

Output:

```
12
17.985999999999997
```

2 ways for Method Overloading

- Method prototype (signature):

  def methodName(arg1,arg2)

- Method in Python can be said to be overloaded using optional arguments

Eg

- **1**. def methodName(arg1,arg2=somevalue,arg3=somevalue)
- **2**. def methodname(*args)

3. from multipledispatch import dispatch

@dispatch(int,int)

def sum(x,y):

    ans = x*y

    print(ans)

Method Overloading – Compile time polymorphism

Method Overloading:

having multiple methods with same name but with different signature (number, type and order of parameters).

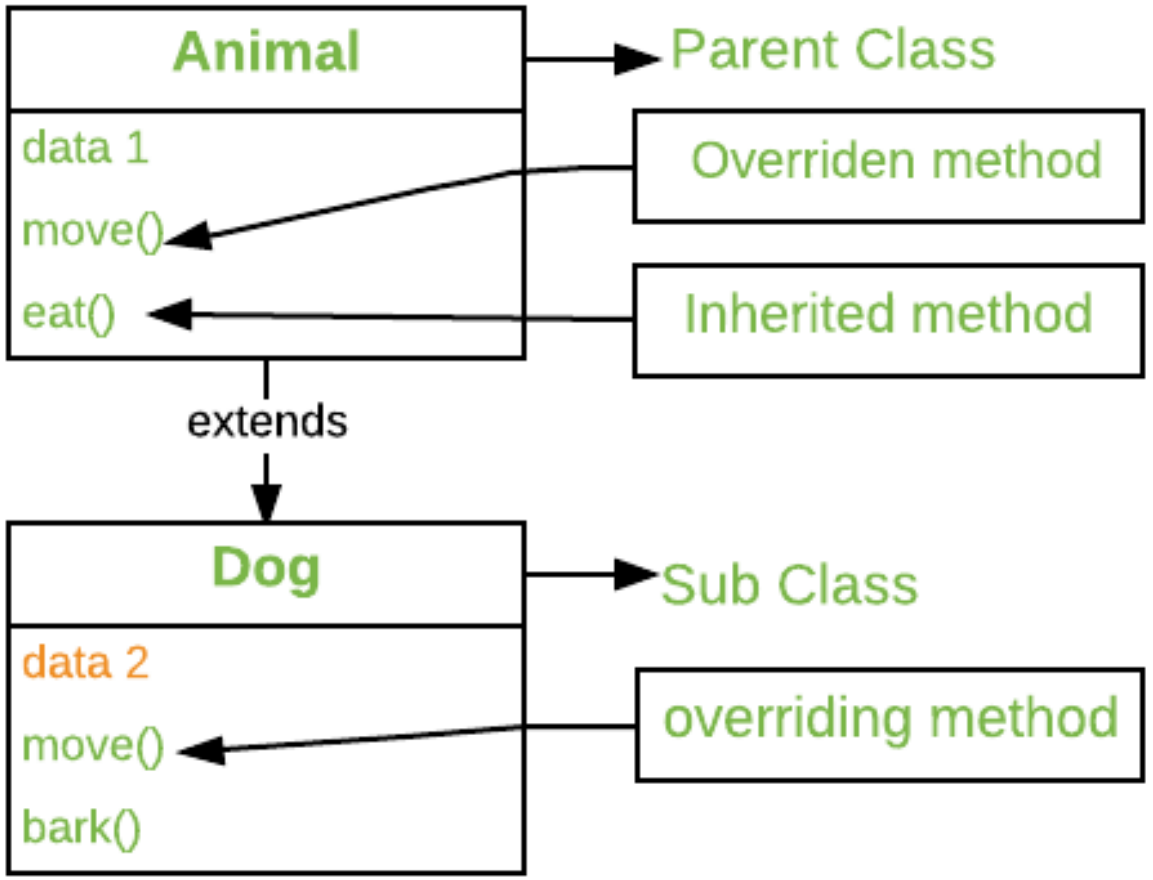
Method Overriding:

When a subclass contains a method with the same name and signature as in the super class then it is called as method overriding.

Method Overriding in Python

- Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
- When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.

Method Overriding in Python



Method Overriding in Python

- The version of a method that is executed will be determined by the object that is used to invoke it.
- If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.
- In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

```
# Python program to demonstrate
# method overriding


# Defining parent class
class Parent():

        # Constructor
        def __init__(self):
                self.value = "Inside Parent"

        # Parent's show method
        def show(self):
                print(self.value)

# Defining child class
class Child(Parent):

        # Constructor
        def __init__(self):
                self.value = "Inside Child"

        # Child's show method
        def show(self):
                print(self.value)


obj1 = Parent()
obj2 = Child()

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

```python
class Parent():

    # Constructor
    def __init__(self):
        self.value = "Inside Parent"

    # Parent's show method
    def show(self):
        print(self.value)
```

```python
# Defining child class
class Child(Parent):

    # Constructor
    def __init__(self):
        self.value = "Inside Child"

    # Child's show method
    def show(self):
        print(self.value)
```

```python
# Driver's code
obj1 = Parent()
obj2 = Child()

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

```
Inside Parent
Inside Child


...Program finished with exit code 0
Press ENTER to exit console.
```

Method Overriding in Python

- Same thing possible in multiple as well as multilevel inheritance

Method Overriding in Python

- **Calling the Parent's method within the overridden method**
- Parent class methods can also be called within the overridden methods. This can generally be achieved by two ways.
- **Using Classname:** Parent's class methods can be called by using the Parent classname.method inside the overridden method.

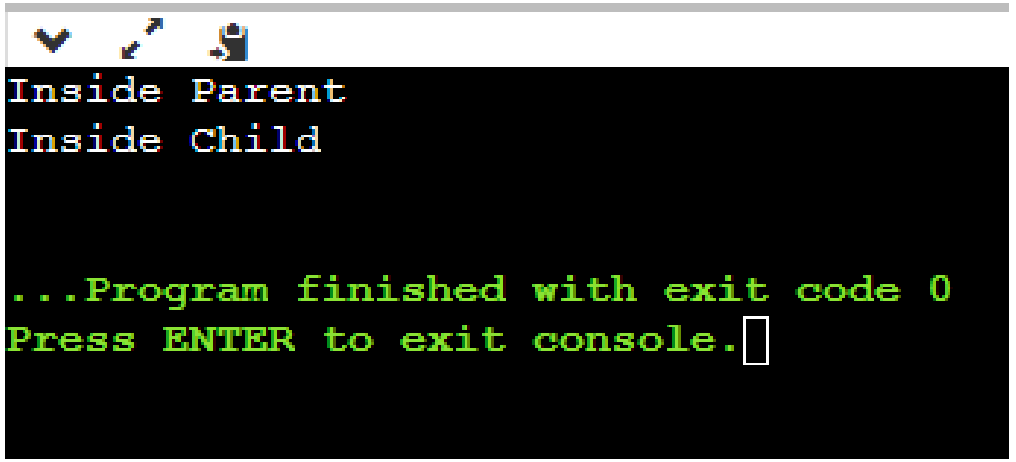
Method Overriding in Python

```
class Parent():

    def show(self):
        print("Inside Parent")
class Child(Parent):

    def show(self):

        # Calling the parent's class
        # method
        Parent.show(self)
        print("Inside Child")
```

```
obj = Child()
obj.show()
```

Method Overriding in Python

```
class Parent():

    def show(self):
        print("Inside Parent")

class Child(Parent):

    def show(self):

        # Calling the parent's class
        # method
        Parent.show(self)
        print("Inside Child")

obj = Child()
obj.show()
```



Method Overriding in Python

- **Using Super():** Python super() function provides us the facility to refer to the parent class explicitly.
- It is basically useful where we have to call superclass functions.
- It returns the proxy object that allows us to refer parent class by ‘super’.

Method Overriding in Python

```
class Parent():

    def show(self):
        print("Inside Parent")
```

```
class Child(Parent):

    def show(self):
        super().show()
        print("Inside Child")

obj = Child()
obj.show()
```

```python
class Parent():

    def show(self):
        print("Inside Parent")

class Child(Parent):

    def show(self):

        # Calling the parent's class
        # method
        super().show()
        print("Inside Child")

# Driver's code
obj = Child()
obj.show()
```

```
Inside Parent
Inside Child


...Program finished with exit code 0
Press ENTER to exit console.
```

```
class Parent():
    def show(self):
        print("Inside Parent")

class Child1(Parent):
    def show(self):
            super().show()
            print("Inside Child1")
class Child(Child1):
        def show(self):
            super().show()
            print("Inside Child")
```

```
obj = Child()
obj.show()
```

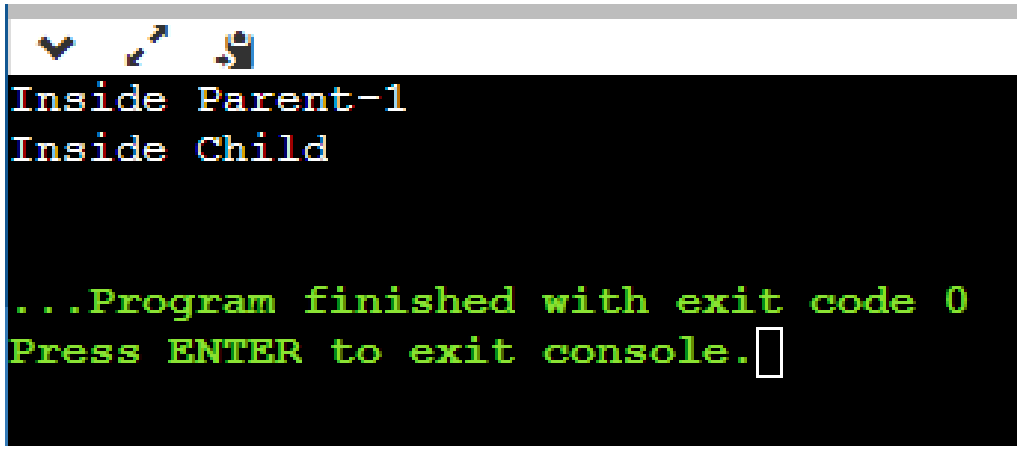
```python
class Parent():

    def show(self):
        print("Inside Parent")

class Child1(Parent):

    def show(self):

        # Calling the parent's class
        # method
        super().show()
        print("Inside Child1")
class Child(Child1):

    def show(self):

        # Calling the parent's class
        # method
        super().show()
        print("Inside Child")

# Driver's code
obj = Child()
obj.show()
```

```
Inside Parent
Inside Child1
Inside Child


...Program finished with exit code 0
Press ENTER to exit console.
```

```
class Parent1():
    def show(self):
        print("Inside Parent-1")
 class Parent2:
    def show(self):
        print("Inside Parent-2")
```

```
class Child(Parent1, Parent2):

    def show(self):
        super().show()
        print("Inside Child")

obj = Child()
obj.show()
```

```python
class Parent1():

    def show(self):
        print("Inside Parent-1")

class Parent2:

    def show(self):
        print("Inside Parent-2")

class Child(Parent1, Parent2):

    def show(self):
        super().show()
        print("Inside Child")

obj = Child()
obj.show()
```



DIAMOND PROBLEM

Diamond Problem

- The diamond problem occurs when two classes have a common ancestor, and another class has both those classes as base classes
- In some languages, because of how inheritance is implemented, when you call d.do_thing(), it is ambiguous whether you actually want the overridden do_thing from B, or the one from C.

```
class A:
    def do_thing(self):
        print('From A')

class B(A):
    def do_thing(self):
        print('From B')

class C(A):
    def do_thing(self):
        print('From C')

class D(B, C):
    pass

d = D()
d.do_thing()
```

- Python doesn't have this problem because of the method resolution order. Briefly, when you inherit from multiple classes, if their method names conflict, the first one named takes precedence. Since we have specified D(B, C), B.do_thing is called before C.do_thing.
- That is also why you get that problem. Consider this: since B inherits from A in your example, B's methods will come before A's. Say we have another class, B_derived, that inherits from B. The method resolution order will then be as follows:

- B_derived -> B -> A

- Now, we have D in the place of B_derived, therefore we can substitute it in to get this:

- D -> B -> A
- However, note that you have also specified that D inherits from A before B, and by the rule above, A must also come before B in the method resolution order. That means we get an inconsistent chain:

- D -> A -> B -> A

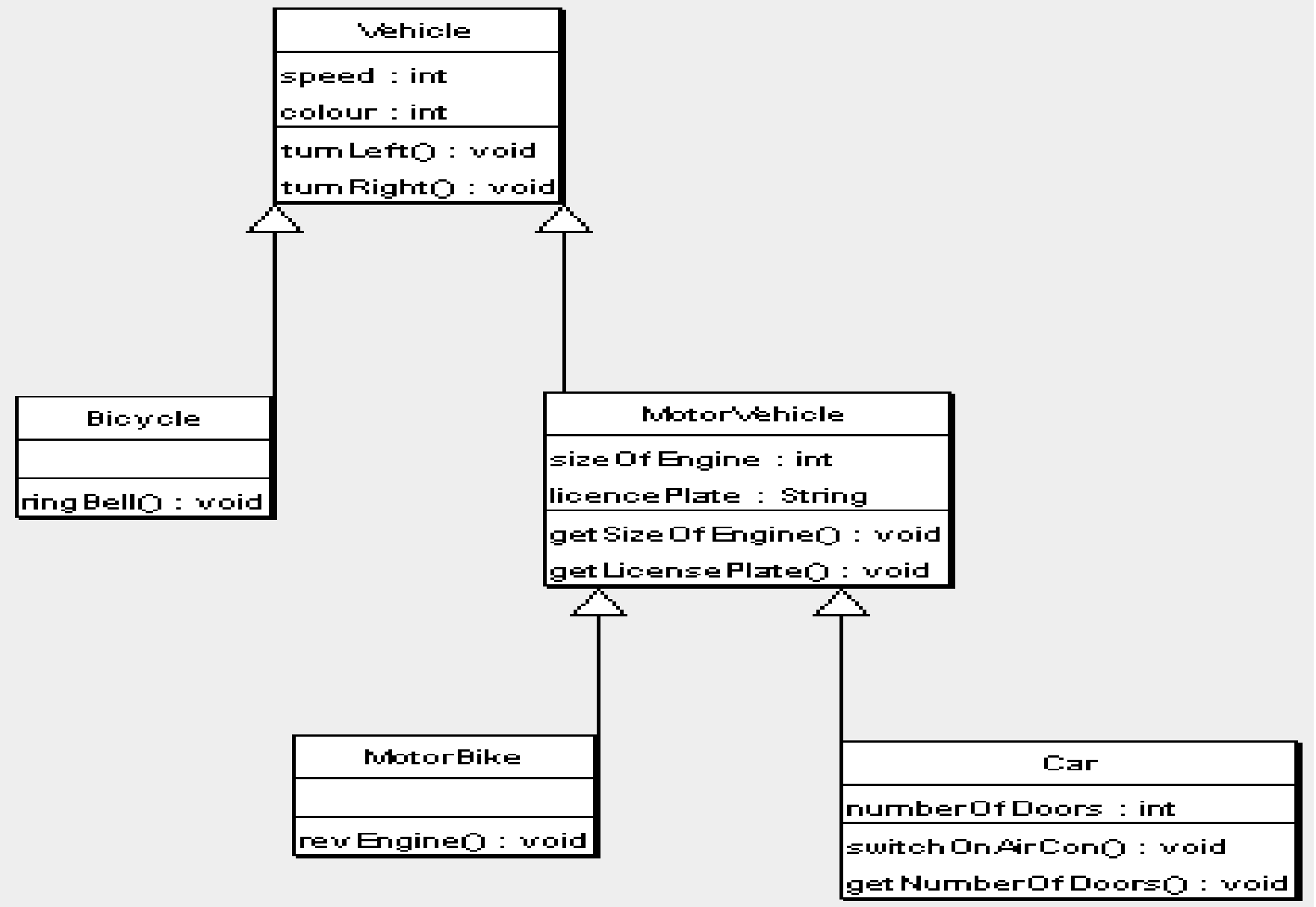
Inheritance

- The mechanism of deriving a Child class (New class) from Parent Class (old-one) is called inheritance
- Child class inherits the data-members and methods of Parent class
- Define new classes that are built upon existing classes, Moreover, you can add new methods and fields in your child class also.
- Helps in code reusability
- Represents real-life scenario in many problem definitions:
- E.g. BankAccount → CurrentAccount, SavingAccount, FDAccount, LoanAccount etc..
- Person (InUniversity) → Student, Faculty, Staff

Syntax of Inheritance

```
class ParentClassName:


class ChildClassName1 (ParentClassName):
        pass


Class ChildClassName2(ChildClassName1):
      pass
```

Inheritance – types of Vehicles



Advantage of Inheritance

- Reusability – Programmer can re-use existing code that already exists, rather than writing it again. Reduce LOC (Line of Code)

- Reduce redundancy of data/code – common data/code shared by creating Parent Class

- Represents real-life concepts/problem domain in software program E.g. Vehicle can be 2 wheeler, 4 wheeler

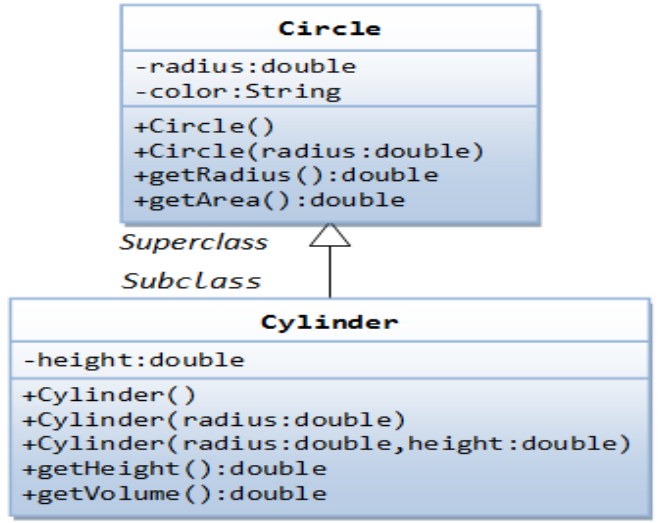
Advantage of Inheritance

- Extensibility – New sub-classes can be added to provide new features in software
- E.g. BankAccount – CurrentAccount and SavingAccount

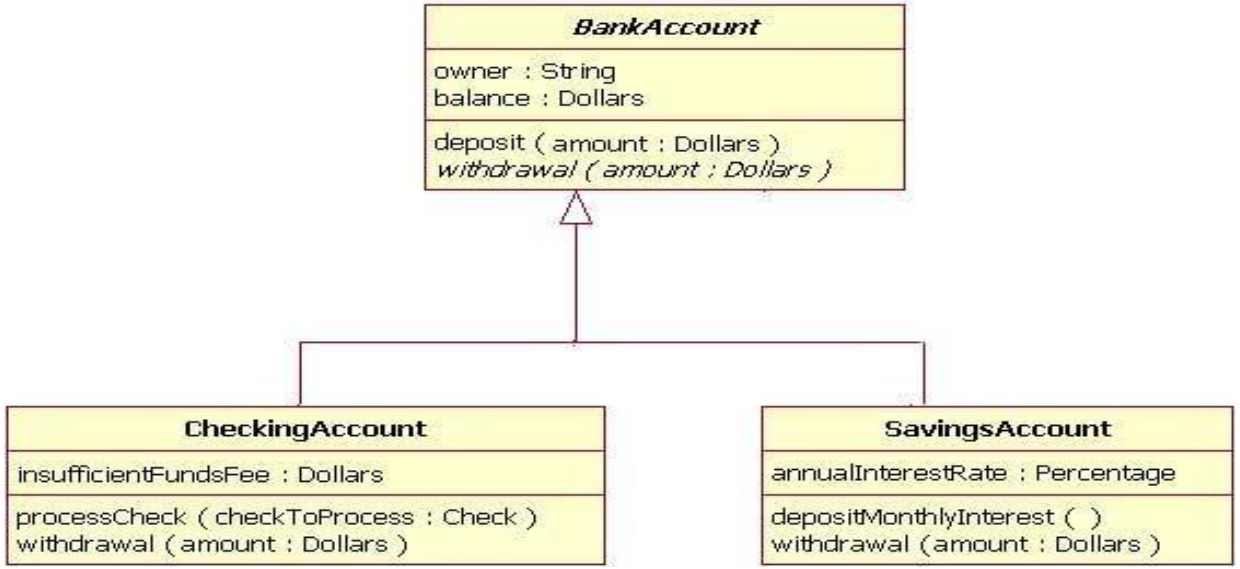
New types of accounts - FixedDepositAccount and LoanAccount can be added (by inheriting BankAccount)

Types of Inheritance

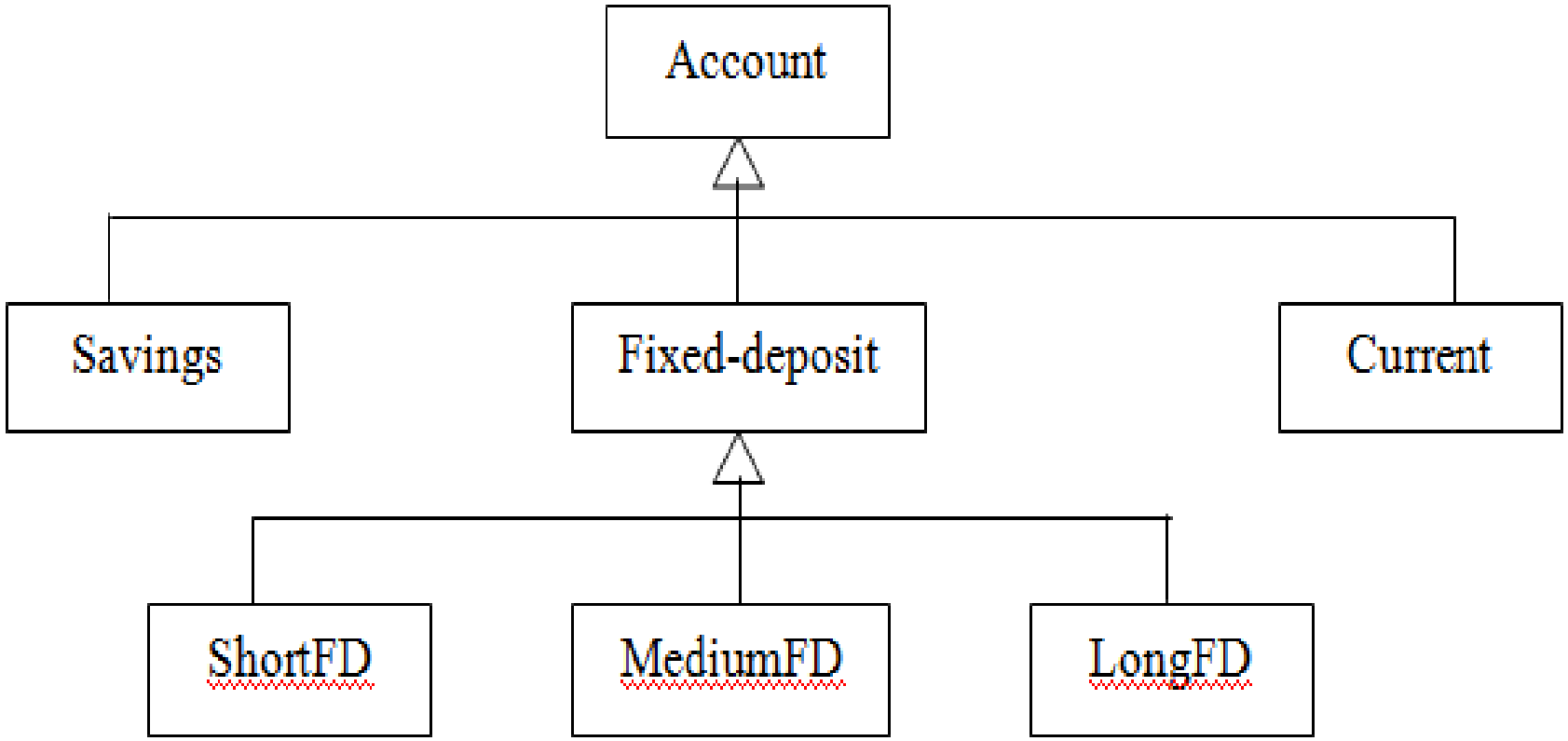
- Single (1 parent-class, 1 child-class)



- Hierarchical (1 parent-class, many child-class)

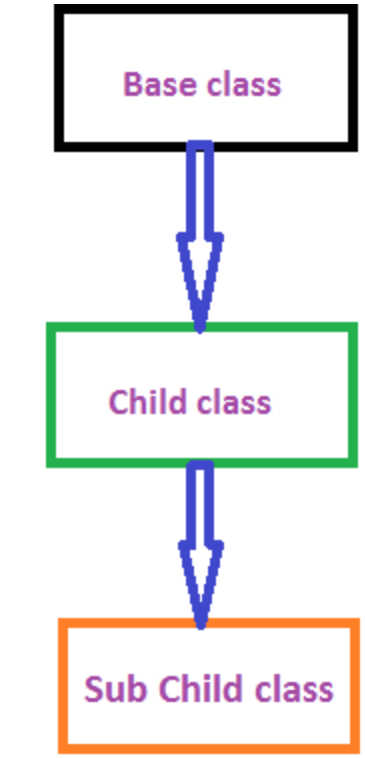


Hierarchical Inheritance

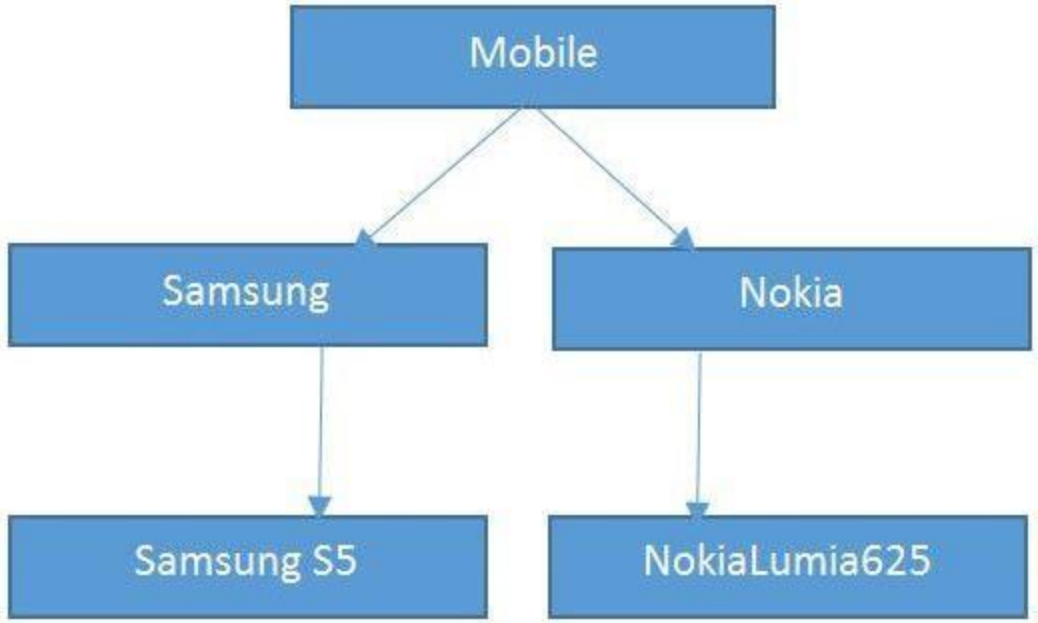


Types of Inheritance

- Multi-level, (1 parent-class having 1 child-class ,that Child-class having another Child-class)

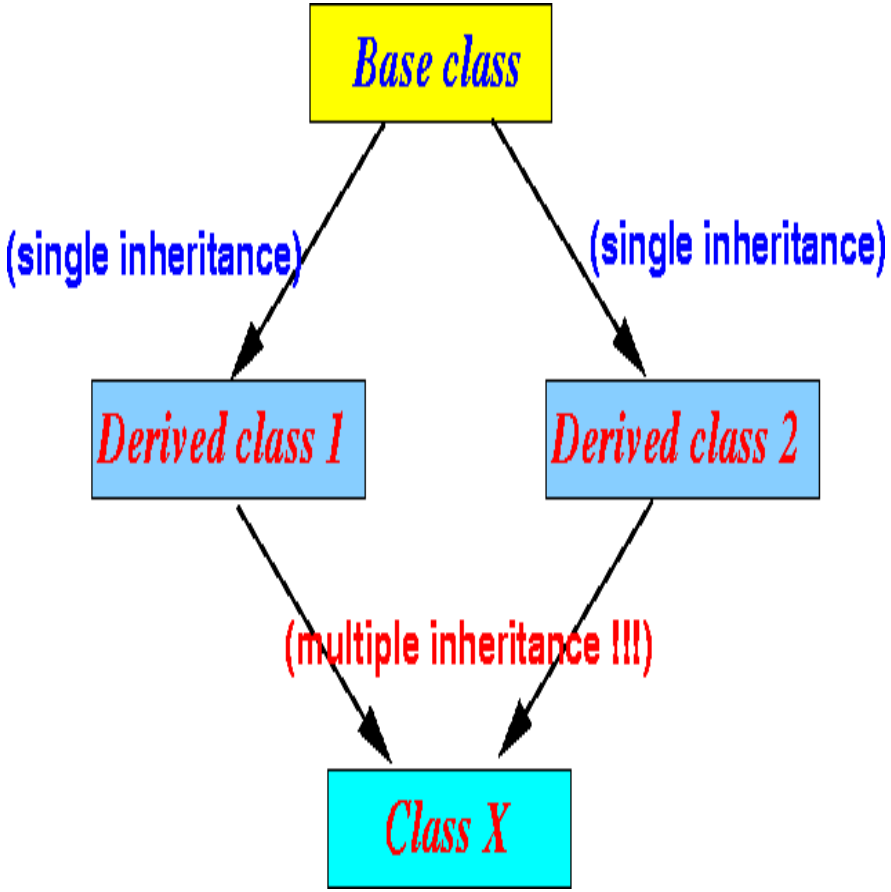


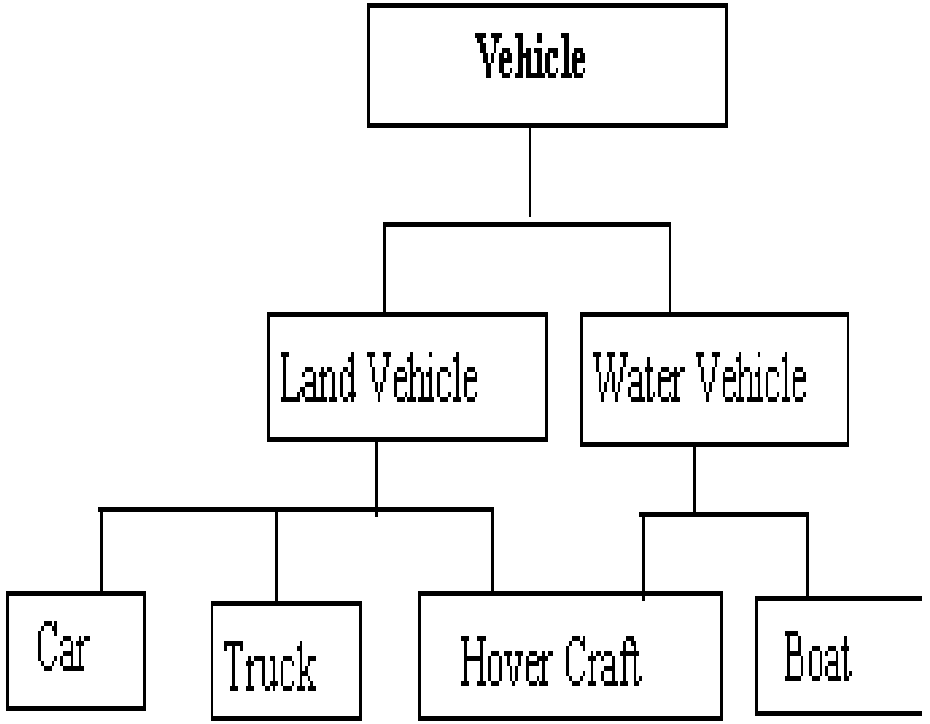
MULTI LEVEL INHERITANCE



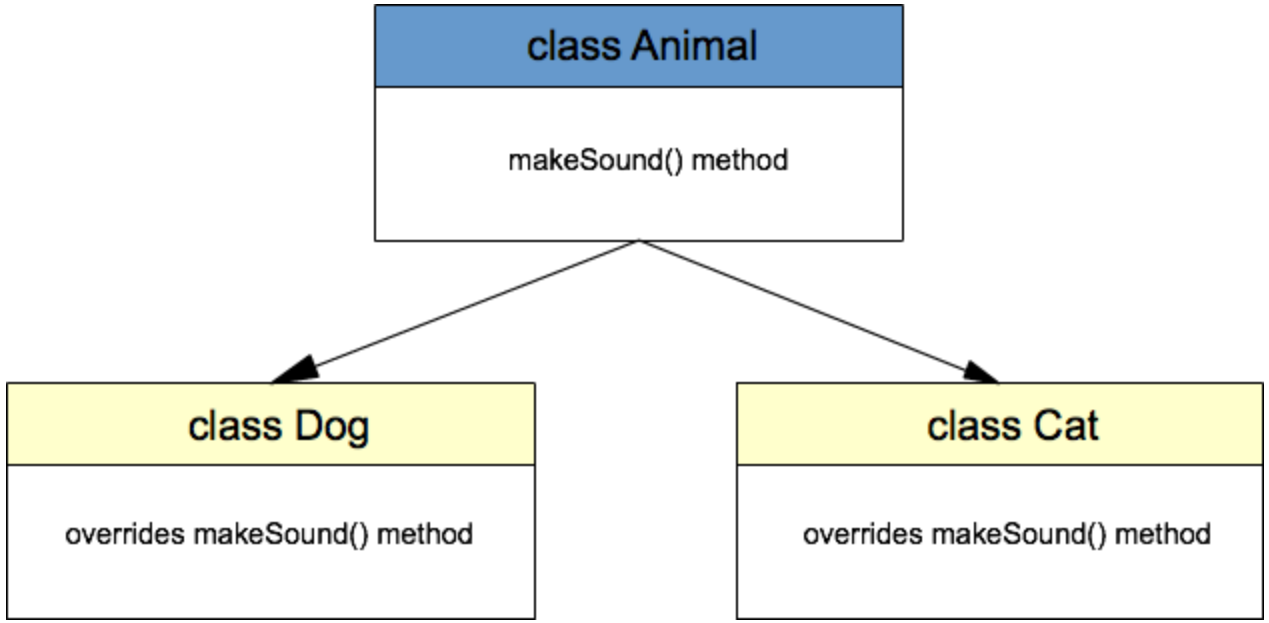
Types of Inheritance

- Multiple inheritance

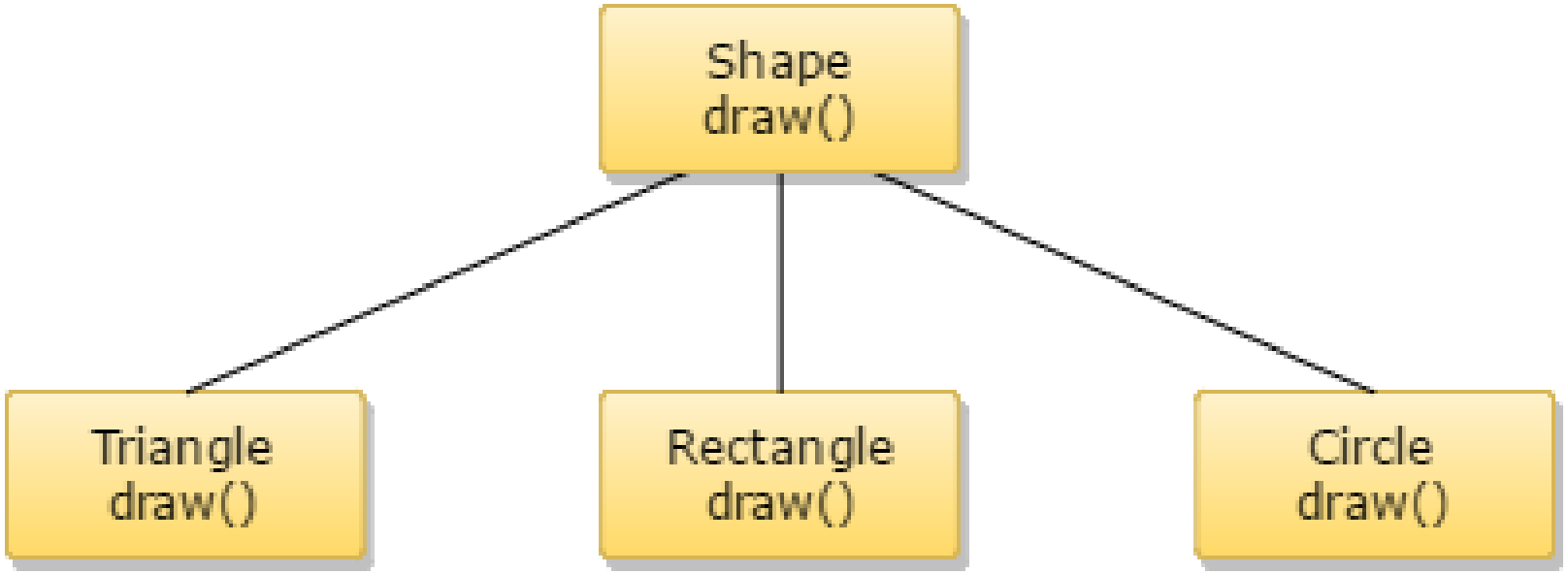




Method Overriding (Inheritance)



Method Overriding (Inheritance)



Polymorphism

Method Overriding (Inheritance)

- In method overriding, Parent and Child class - methods have same name with same prototypes (no. and type of arguments)
- While compiling the Program - with overriden method calls → Compiler is not able to resolve such method calls
- Hence, overridden methods are resolve/linked with correct Implementation at run-time by PVM.
- Method overriding – is also called as – Late binding OR Runtime polymorphism OR Dynamic binding.

Inheritance – Method Overriding

```
class One:
    def __init__(self):
        self.x=10
        print("One Constructor")

    def display(self):
        print("Class One ",self.x)

class Two(One):
    def __init__(self):
        super().__init__() #or One.__init__(self)
        self.y=20
        print("Two Constructor")

    def display(self):
        print("Class Two ",self.x)
        print("Class Two ",self.y)
        One.display(self)

#########################
t=Two()
t.display()
```

Use of Super keyword – in Inheritance

1) Used to call parent-class constructor – from Child class

- RULE: If parent class is having a parameterized Constructor.

2) To access the over-ridden methods and data-members members of Super class START EDITING FROM HERE

Syntax – usage in Child class: super.member

```
def __init__(self):
    super().__init__() #or One.__init__(self)
```

Super keyword – access overridden data

```
class One:
    def __init__(self):
        self.x=1  #attributes
        print("One Constructor")

    def display(self): # #function signature or prototype
        print("Display of class One ",self.x)


class Two(One):
    def __init__(self):
        self.y=2
        #super().__init__() #super() refers to parent class in python
        One.__init__(self)
        print("Two Constructor")
        # Method/Function overriding :- A function / method is rewritten in child class which is already mentioned in its parent class with same name and signature
        def display(self):
            super().display()
            print("Class Two display ",self.x, self.y)
```

Super keyword – access overridden data

```
class Three(Two):
    def __init__(self):
        self.z=3
        super().__init__() # will invoke class Two constructor
        print("Three Constructor") #super() refers to the parent class

    def display(self): #overridden method of class Two
        super().display()
        print("Navrachana University Class Three ",self.x,self.y,self.z)

#########################################
t=Three()
t.display()
```

Important

Abstract Class

- An abstract class can be considered as a blueprint for other classes.
- It allows you to create a set of methods that must be created within any child classes built from the abstract class.
- A class which contains one or more abstract methods is called an abstract class.
- An abstract method is a method that has a declaration but does not have an implementation.
- While we are designing large functional units we use an abstract class.
- When we want to provide a common interface for different implementations of a component, we use an abstract class.

Why use Abstract Base Classes ?

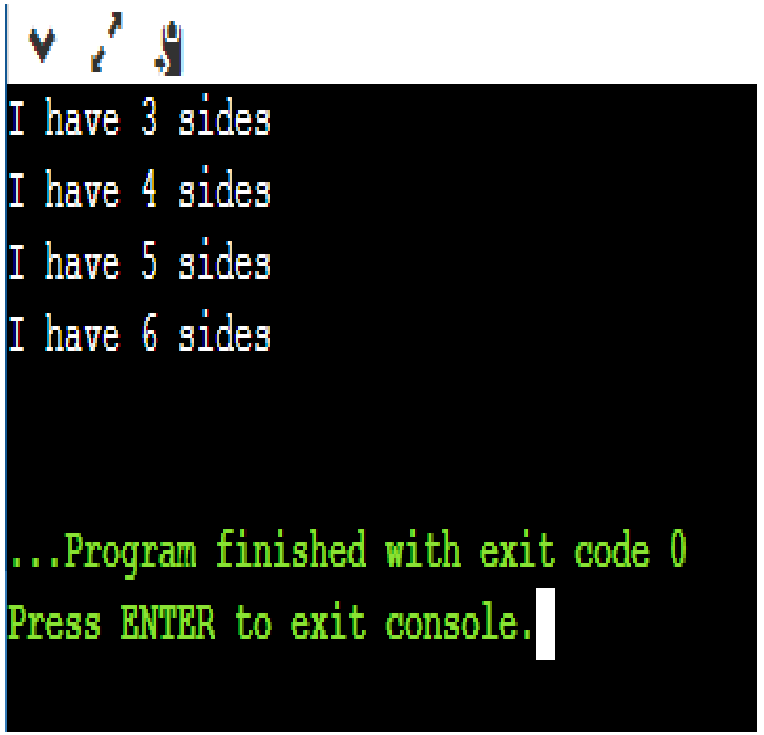
- By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses.
- This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible.

How Abstract Base classes work ?

- By default, Python does not provide abstract classes.
- Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is ABC.
- ***ABC*** works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base.
- A method becomes abstract when decorated with the keyword @abstractmethod.

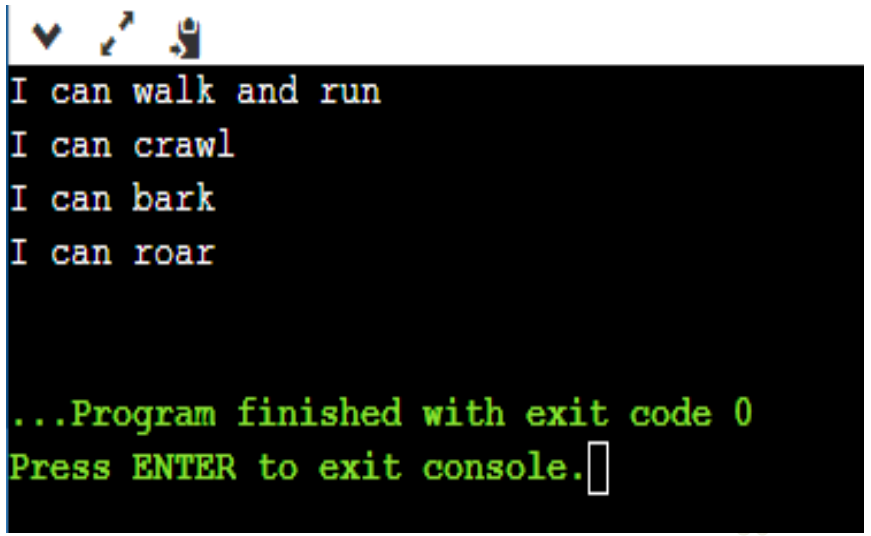
```
from abc import ABC, abstractmethod
class Polygon(ABC):
        @abstractmethod
        def noofsides(self):
                pass
class Triangle(Polygon):
        # overriding abstract method
        def noofsides(self):
                print("I have 3 sides")
class Pentagon(Polygon):
        # overriding abstract method
        def noofsides(self):
                print("I have 5 sides")
class Hexagon(Polygon):
        # overriding abstract method
        def noofsides(self):
                print("I have 6 sides")
```

```
class Quadrilateral(Polygon):
        # overriding abstract method
        def noofsides(self):
                print("I have 4 sides")


# Driver code
R = Triangle()
R.noofsides()

K = Quadrilateral()
K.noofsides()

R = Pentagon()
R.noofsides()

K = Hexagon()
K.noofsides()
```



```
from abc import ABC, abstractmethod
class Animal(ABC):
        def move(self):
                pass
class Human(Animal):
        def move(self):
                print("I can walk and run")
class Snake(Animal):
        def move(self):
                print("I can crawl")
class Dog(Animal):
        def move(self):
                print("I can bark")
class Lion(Animal):
        def move(self):
                print("I can roar")
```

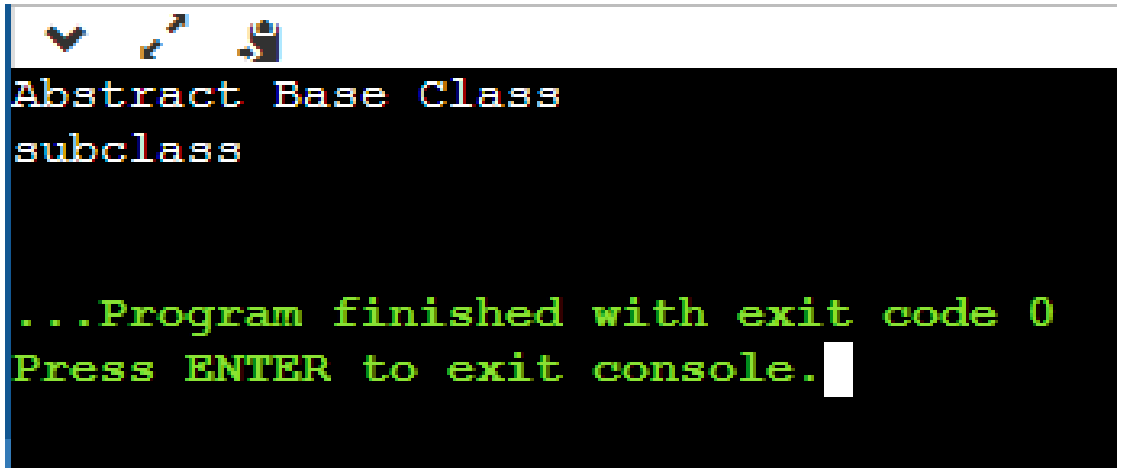
```
# Driver code
R = Human()
R.move()

K = Snake()
K.move()

R = Dog()
R.move()

K = Lion()
K.move()
```



Concrete Methods in Abstract Base Classes

- Concrete classes contain only concrete (normal)methods whereas abstract classes may contain both concrete methods and abstract methods.
- The concrete class provides an implementation of abstract methods, the abstract base class can also provide an implementation by invoking the methods via super().

```
import abc
from abc import ABC, abstractmethod


class R(ABC):
        def rk(self):
                print("Abstract Base Class")
class K(R):
        def rk(self):
                super().rk()
                print("subclass ")
# Driver code
r = K()
r.rk()
```

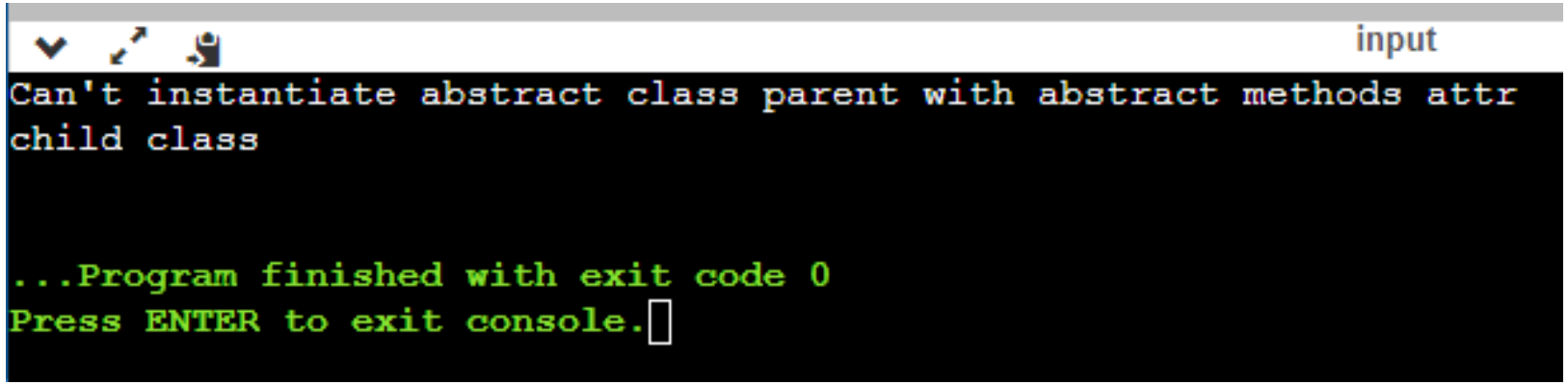


Abstract Properties

- Abstract classes include attributes in addition to methods, you can require the attributes in concrete classes by defining them with @abstractproperty.

```
import abc
from abc import ABC, abstractmethod

class parent(ABC):
        @abc.abstractproperty
        def attr(self):
                return "parent class"
class child(parent):
        @property
        def attr(self):
                return "child class"
```

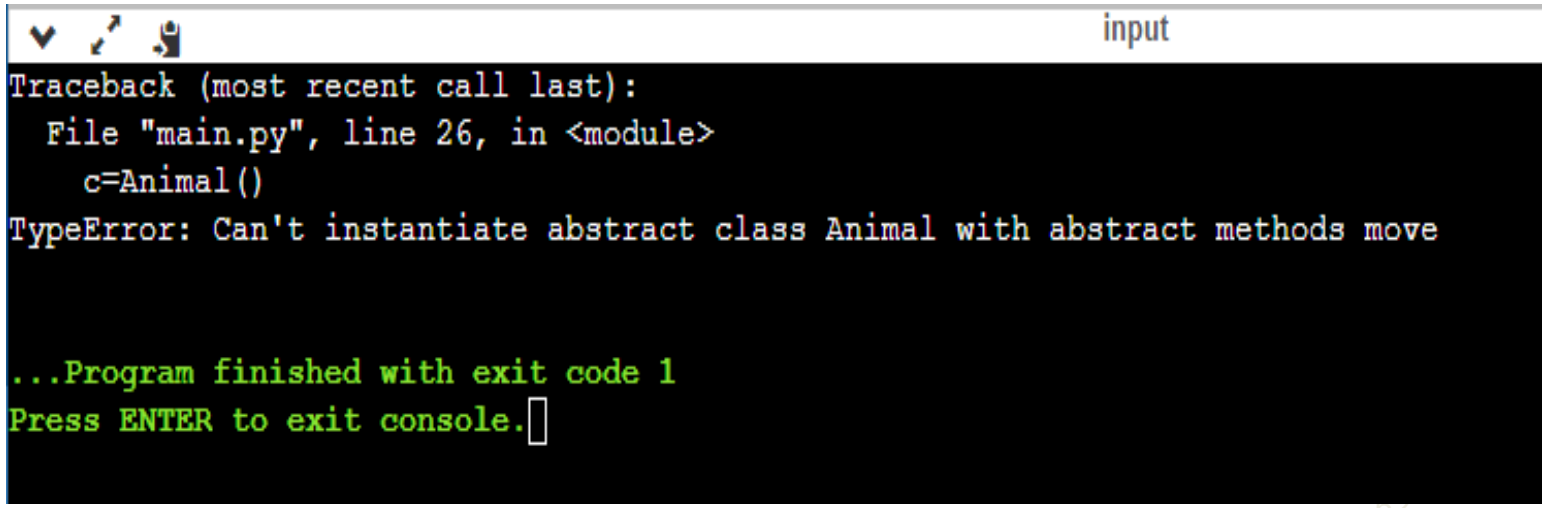
```
try:
        r =parent()
        print( r.attr)
except Exception as err:
        print (err)

r = child()
print (r.attr)
```



Abstract Class Instantiation

- Abstract classes are incomplete because they have methods that have nobody.
- If python allows creating an object for abstract classes then using that object if anyone calls the abstract method, but there is no actual implementation to invoke.
- So we use an abstract class as a template and according to the need, we extend it and build on it before we can use it.
- Due to the fact, an abstract class is not a concrete class, it cannot be instantiated.
- When we create an object for the abstract class it raises an *error*.

```
from abc import ABC,abstractmethod

class Animal(ABC):
        @abstractmethod
        def move(self):
                pass
class Human(Animal):
        def move(self):
                print("I can walk and run")
class Snake(Animal):
        def move(self):
                print("I can crawl")
class Dog(Animal):
        def move(self):
                print("I can bark")
class Lion(Animal):
        def move(self):
                print("I can roar")

c=Animal()
```



Practice Problem

- The Employee class represents an employee, either full-time or hourly. The Employee class should be an abstract class because there're only full-time employees and hourly employees, no general employees exist.
  - The Employee class should have a property that returns the full name of an employee. In addition, it should have a method that calculates salary. The method for calculating salary should be an abstract method.
  - The FulltimeEmployee class inherits from the Employee class. It'll provide the implementation for the get_salary() method.
  - The HourlyEmployee also inherits from the Employee class. However, hourly employees get paid by working hours and their rates. Therefore, you can initialize this information in the constructor of the class.
  - To calculate the salary for the hourly employees, you multiply the working hours and rates.
  - The Payroll class will have a method that adds an employee to the employee list and print out the payroll

final Decorator

```
from typing import final
@final   #disallows inheritance
class Person:
    def __init__(self):
        self.person_name="Raj"

class Employee(Person):
    def __init__(self):
        super().__init__() #inherits class person

    @final     #disallows overriding
    def display(self):
        print("Display Method of Employee")

class Manager(Employee):
    def __init__(self):
        super().__init__() #inherits class Employee
    @final
    def display(self):
        print("Display Method of Manager")

man=Manager()
man.display()
```

Class Decorators

- @staticmethod – Used to declare a method as static/class method
- @property - for creating getter method
- @methodname.setter - for creating setter method
- @final – To prohibit inheritance and method overriding (import package as "from typing import final")
- @abstractmethod – To create abstract method (import package as "from abc import ABC, abstractmethod")

Exception Handling

What is an Exception?

- is an abnormal condition that occurs in program at run-time (disrupts normal flow of execution)
- It is a run-time error.
- Abnormally terminates the program OR Disrupts the normal flow of the program.
- Reasons:
- Due to wrong user input
- logical mistakes (trying to access collection (list, set, dictionary element outside size))

Difference between Syntax Error and Exceptions

- **Syntax Error:** As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

```
amount = 10000
if(amount > 2999)
print("You are eligible to purchase Dsa Self Paced")
```

```
  File "/home/ac35380186f4ca7978956ff46697139b.py", line 4
    if(amount>2999)
                  ^
SyntaxError: invalid syntax
```

Difference between Syntax Error and Exceptions

- **Exceptions:** Exceptions are raised when the program is syntactically correct, but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

```
# initialize the amount variable
marks = 10000

# perform division with 0
a = marks / 0
print(a)
```

```
Traceback (most recent call last):
  File "/home/f3ad05420ab851d4bd106ffb04229907.py", line 4, in <module>
    a=marks/0
ZeroDivisionError: division by zero
```

- **Note:** Exception is the base class for all the exceptions in Python. You can check the exception hierarchy on the below given link

- https://docs.python.org/2/library/exceptions.html#exception-hierarchy

What is an Exception?

```
try:
    #statement 1
    #statement 2
except:
    # statement 3
    # statement 4
else:
    # statement 5 (execute if no exception)
finally:
    # statement 6 .....(always executed)
```

Finally

- The finally keyword is used in try...except blocks. It defines a block of code to run when the try...except...else block is final.

- The finally block will be executed no matter if the try block raises an error or not.

- **This can be useful to close objects and clean up resources.**

Exception Types

- except ZeroDivisonError (if a number is divided by Zero)
- except ArithmeticError (if a number is divided by Zero)
- except NameError (if the accessed variable is not defined)
- except IndexError (if collection is accessed with no variable in range)
- except ImportError (if environment is unable to import the package)
- except KeyError (if a matching key not found in dictionary)
- except MemoryError (if a program runs out of memory)
- except OSError (if program is unable to access OS functionality)
- except URLError (if a request is not responded or wrong url)

Examples of Exception

```
def display():
    print("One")
    lst=[2,4]
    try:
        #k=5/0
        print(lst[4])
    except IndexError:
        print("Exception Handled 1")
    except ZeroDivisionError:
        print("Exception Handled 2")
    print("Two")


display()
```

Exception Handling

- Mechanism to handle runtime errors in well defined way , Writing try-except block of code

Advantages

- Avoid abnormal termination of the program
- To maintain the normal flow of the application
- To handle possible error conditions and ensure reliable and error free execution
- Robust and reliable Software to run on different platforms/environments

Generating Exception Explicitly

raise keyword is used to generate exception explicitly from program. It is used when library programmer wants to create his own custom exception

Example

```
try:
    raise ZeroDivisionError(“Don’t divide number by zero")  # Raise Error
except ZeroDivisonError:
    print ("An exception")
```

Custom Exception

- User can also create his own Exception.
- To create custom exception, inherit Exception class to your own class.
- Exception class the parent of all the Exceptions in Python.
- Eg. IndexError, DivideByZeroError, FileNotFoundError are child of Exception class.

```
class MarksNotInRangeException(Exception):#Exception is an inbuilt class from which new exception can be derived
    def __init__(self,marks,msg="Marks not in Range - 0 to 100"): #Registers the Exception message under PVM by invoking Exception class constructor using super in the next line.
        super().__init__(msg)
############################
print("One")
english=1001
if not 0 <= english <= 100:
    raise MarksNotInRangeException(english)
print("Two")
```

Assertion

- An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.
- The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.
- Assertions are carried out by the assert statement
- Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The *assert* Statement

- When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an AssertionError exception.
- The syntax for assert is −

  assert Expression[, Arguments]

- If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

The *assert* Statement

- Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature −

```
def KelvinToFahrenheit(Temperature):
   assert (Temperature >= 0),"Colder than absolute zero!"
   return ((Temperature-273)*1.8)+32
print KelvinToFahrenheit(273)
print int(KelvinToFahrenheit(505.78))
print KelvinToFahrenheit(-5)
```

```
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in <module>
print KelvinToFahrenheit(-5)
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

Assertion

- Used when you want to "stop" the script based on a certain condition and return something to help debug faster:

```
list_ = ["a","b","x"]
assert "x" in list_, "x is not in the list"
print("passed")
#>> prints passed

list_ = ["a","b","c"]
assert "x" in list_, "x is not in the list"
print("passed")
#>>
Traceback (most recent call last):
  File "python", line 2, in <module>
AssertionError: x is not in the list
```

Assertion vs Exception

- The key differences between exceptions and assertions are:
- Assertions are *intended* to be used solely as a means of detecting programming errors, aka bugs.
- By contrast, an exception can indicate other kinds of error or "exceptional" condition; e.g. invalid user input, missing files, heap full and so on.

raise vs assert

- try/except blocks let you catch and we can manage exception or add custom exceptions. Exceptions can be triggered by raise, assert, and a large number of errors such as trying to index an empty list or Integrity errors.
- **raise** is typically used when you have detected an error condition or some condition does not satisfy.
- **assert** is similar but the exception is only raised if a condition is met.
- raise and assert have a different philosophy.

raise vs assert

- raise - raise an exception.
- assert - raise an exception if a given condition is meet.
- try - execute some code that might raise an exception, and if so, catch it.

- Python’s assert statement is a debugging aid, not a mechanism for handling run-time errors.
- The goal of using assertions is to let developers find the likely root cause of a bug more quickly.
- An assertion error should never be raised unless there’s a bug in your program.

raise vs assert

- There are many "normal" errors in code that you detect and raise errors. like HTTP error code (2xx, 4xx).
- Assertions are generally reserved for “I swear this cannot happen” issues that seem to happen anyway. Its more like runtime debugging than normal runtime error detection.
- Assertions can be disabled if you use the -O flag or run from .pyo files instead of .pyc files, so they should not be part of regular error detection.
- If production quality code raises an exception, then figure out what you did wrong. If it raises an AssertionError, you've got a bigger problem.

- In short:

Error vs Exception

- Errors cannot be handled, while Python exceptions can be handled at the run time.
- An error can be a syntax (parsing) error, while there can be many types of exceptions that could occur during the execution and are not unconditionally inoperable.
- Errors can be of various types:
  - **Syntax Error**
  - **Out of Memory Error**
  - **Recursion Error**
  - **Exceptions**

Syntax Error

- **Syntax** errors often called as parsing errors, are predominantly caused when the parser detects a syntactic issue in your code.

```
a = 8
b = 10
c = a b
```

```
  File "<ipython-input-8-3b3ffcedf995>", line 3
    c = a b
          ^
SyntaxError: invalid syntax
```

Out of Memory Error

- Memory errors are mostly dependent on your systems RAM and are related to Heap.
- If you have large objects (or) referenced objects in memory, then you will see OutofMemoryError (Source).
- It can be caused due to various reasons:
  - Using a 32-bit Python Architecture (Maximum Memory Allocation given is very low, between 2GB - 4GB).
  - Loading a very large data file
  - Running a Machine Learning/Deep Learning model and many more.

Out of Memory Error

- You can handle the memory error with the help of exception handling, a fallback exception for when the interpreter entirely runs out of memory and must immediately stop the current execution.
- In these rare instances, Python raises an OutofMemoryError, allowing the script to somehow catch itself and break out of the memory error and recover itself.
- **neither it is a good practice to use exception handling for such an error, nor it is advisable.**

Recursion Error

- It is related to stack and occurs when you call functions.
- As the name suggests, recursion error transpires when too many methods, one inside another is executed (one with an infinite recursion), which is limited by the size of the stack.

For your reference

https://stackoverflow.com/questions/60708789/exceptions-vs-errors-in-python

References

- Michael H. Goldwasser, David Letscher , “Object-oriented Programming in Python “ , Pearson Prentice Hall, 2008
- Dusty Phillips , “Python 3 Object Oriented Programming” ,PACLT publications.
- Retrieved and referred from docs.python.org, July 2021.
- Retrieved and referred from https://www.pcmag.com/encyclopedia, July 2021.
- Retrieved and referred from https://stackoverflow.com, July 2021.
- Retrieved and referred from https://wikipedia.org, July 2021.
- Retrieved and referred from https://w3schools.com, July 2021.