A language is Typed if the specification of every operation defines types of data to which the operation is applicable, with the implication that it is not applicable to other types. A typed langnguage is either weakly typed (allows one type of value is treated as another type like string as number Ex: Perl, java script) or strongly typed (do not allows implicit type converion & rises error when there is operation on wrong type of values Ex. python) language.

A language is Un-typed allows any operation to be performed on any data, which are generally considered to be sequences of bits of various lengths **Assembly languages.**

Class and Objects

Class: In general Class is a group of similar objects, example Fruits, Vehicle, Employee, Student etc. Class is a blueprint to create objects which specifies attributes(properties) and functions(behaviour) of the object.

Class is user defined datastructure which tieds data and functions together, it is a blueprint of objects. Class definition will not reserve memory
In python a class can be defined using the syntax

```
class ClassName:
'Optional class documentation string'
Data members # to access in all the objects
Functions definition
```

class is a keyword and class name can be any user defined name, normally the class name will be of title case.
A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.
Document string is not mandetory, it can accessed using the magic function __doc__ using the syntax. Class_name.__doc__ . docstring is used to brief description about the class.
Example:

```
class Student:
““““A class representing a student ”””
    Student-count=0
def __init__(self , n, a):
   self.full_name = n
   self.age = a
def get_age(self):       #Method
```

```
    return self.age
```

Student_count is a data member that can be accesed in all the methods.

__init__( )

Class functions that begins with double underscore (__) are called special functions as they have special meaning.

__init__ is one such function that gets called when a new object is created. This function is normally used to initialize all the instance variables.

get_age() is a method i,e a member function of student class.

Self: The first argument of every method is a reference to the current instance of the class.

In __init__, self refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called.

Object: Object is a thing which has data and exits Ex. Mango, Bike. Object is an instance (Variable) of type Class, memmory will be resorved once the object is created. Object is a combination of functions, variables and data structures.

Once a class is defined user can create any number of objects using the syntax

    Object name=Class-name(values)
    Ex: f = student("Python", 14)

Class attributes can be accessed using the syntax

    Object-name.method()
    Object-name.data-member

Class attributes can also be accessed using the python built in functions

getattr(obj, attribute): to access the attribute of object.(Retruns value of attribute for that object)

hasattr(obj,attribute): to check if an attribute exists or not.(Returns true if exists)

setattr(obj,name,value): to set new value to an attribute. If attribute does not exist, then it would be created.

delattr(obj, name): to delete an attribute

One major difference between objects and class is the way attributes and methods are treated in objects and classes. A class is a definition about objects; the attributes and methods in a class are thus declarations that do not contain values. However, objects are created instances of a class. Each has its own attributes and methods. The values of the set of attributes describe the state of the objects.

Class members: (Data member and Member function)

1. Data Members: Variables defined in a class are called data memebers. There are types of fields

Class Variables: The variables accessed by all the objects(instances) of a class are called class variable. There is only copy of the class variable and when any one object makes a change to a class variable, the change is reflected in all the other instances as well. Class variables can be accessed using the syntax

Class_name.Class variable

Object_name. Class variable

Note: if user changes the value of class variable using class name, then it reflects to all objects

if user changes the value of class variable using object name, then it reflects only to that objects

Object/instance Variable: Variables owned by each individual object (instance of a class) are called object variables. Each object has its own copy of the field i.e. they are not shared and are not related in any way to the field by the same name in a different instance of the same class.

In the above Student class example Student-count is Class variable and full_name is a Object variable.

2. Member function(method): A **method** is a function that is defined inside a class. Methods are members of classes. A method is a function that is available for a given object. There are different types of methods like Class methods, Instance methods, Service methods, and Support methods.

Class method: A "class method" is a method where the initial parameter is not an instance object(self), but the class object , which by convention is called cls . This is implemented with the @classmethod decorator. The class method does not require object to invoke it.

Ex:

```
class Foo(object):
    @classmethod
    def hello(cls):
        print("hello from %s" % cls.__name__)
Foo.hello()
-> "Hello from Foo"
Foo().hello()
-> "Hello from Foo"
```

Instance Method: An "Instance method" is a method where the initial parameter is an instance of object(self) and requires no decorator. These are the most common methods used.

```
class Foo(object):
    def hello(self):
        print("hello”)

f=Foo()

f.hello()
```

Built-in Class attributes: Every Python class keeps the following built-in attributes and they can be accessed using dot operator like any other attribute.

__dict__: Dictionary containing the class's namespace.

__doc__: Class documentation string or none, if undefined.

__name__: Class name.

__module__: Module name in which the class is defined. This attribute is "__main__" in interactive mode

Example:

```
class Employee:
	'Common base class for all employees'
	empCount = 0
	def __init__(self, name, salary):
		self.name = name
		self.salary = salary
		Employee.empCount += 1
	def displayCount(self):
		print ("Total Employee %d" % Employee.empCount)
	def displayEmployee(self):
		print ("Name : ", self.name,", Salary: ", self.salary)
emp1 = Employee("Ranga", 2000)
emp2 = Employee("Soma", 5000)
print ("Employee.__doc__:", Employee.__doc__)
print ("Employee.__name__:", Employee.__name__)
print ("Employee.__module__:", Employee.__module__)
print ("Employee.__bases__:", Employee.__bases__)
print ("Employee.__dict__:", Employee.__dict__ )
```

Classes and functions: Program

Encapsulation : Encapsulation is the mechanism of embeding the data with function for restricting the access to some of an objects's components, this means, that the internal representation of an object can't be seen from outside of the objects definition.

Encapsulation provides the different levels of accessing to the class members this is called visibility of class members. OOPs languages provides three access specifiers to controll the accessing of class members**.**

Private: This specifies that the class members (variables and methods) can be accessed only within that class means that only the member functions can use the class members and can not be used outside the class. In C++ all the members are private by default. In python to make the class member private begin the name of the member with double under_score(__)

Example:

```
class Visibility:
    def __init__(self,a,b):
        self.atta=a
        self.__attb=b #private data
    def __disp(self):# private method
        print ("Inside private method",self.atta,self.__attb)
    def display(self):
        print("Inside public method",self.atta)
        self.__disp()#calling private method
ob1=Visibility(2,20)
ob1.display()
ob1.__disp()# will not work as it can not access outside
```

Public: This specifies that all attributes and methods are visible to every one. Class members can be accesed even outside the class using **object_name.class member.** In C++ to make the class members public, have all the members in the **public:** section.

In pyhton by defalut all the members are public.

Example:

```
class Cup:
    def __init__(self):
            self.color = None
             self.content = None

    def fill(self, beverage):
            self.content = beverage
    def empty(self):
             self.content = None
redCup = Cup()
redCup.color = "red"
redCup.content = "tea"
redCup.empty()
redCup.fill("coffee")
```

Shows all members are public

Protected: This specifies that class members are accessible only within the class and its sub-class. In python to make the class member protected begin the name of the member with single under_score(_).

Example:

```
class Cup:
    def __init__(self):
        self.color = None
        self._content = None # protected variable

    def fill(self, beverage):
        self._content = beverage
    def empty(self):
        self._content = None
redCup = Cup("red")
redCup._Cup__content = "tea"
```

Note :Even the variable content is protected it is accessable.

Object initialization: Assiging the values to objects when they are created is called object intialization. **Constructor** is used to initialize objects. Constructor is a special method which is called when a new instance of a class (object) is created. In python __init__( ) method is used as initialization method called class constructor.

Example:

```
 class Student:
...     def __init__(self,na,ag,fe):
...         self.name=na
...         self.age=ag
...         self.fee=fe
...
>>> s1=Student('Ranga',22,25000)
>>> s2=Student('Soma',25,38000)
```

Here the objects S1 and S2 are initialized to specified values.

Destructor: Destructor is member function, which is invoked when the instance is about to be destroyed. This method is used to cleanup the resources used by an instance. Destructor is used for destroying the objects which are not needed, to free the memory space. The process of periodically reclaiming the block of memory that are no longer needed is called **Garbage collection.** In python Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

In pyhton __del__(self) method is used as destructor.

Example:

```
class Point:
        def __init__( self, x=0, y=0):
                self.x = x
                self.y = y
        def __del__(self):
                class_name = self.__class__.__name__
                print (class_name, "destroyed")
pt1 = Point()
pt2 = pt1
pt3 = pt1
print (id(pt1), id(pt2)) # prints the ids of the obejcts)
del pt1
del pt2
del pt3
```

Method and Message:

Message: Objects communicate with one another by sending messages. A message is a method call from a message-sending object to a message-receiving object. A message- sending object is a sender while a message-receiving object is a receiver. The set of methods collectively defines the dynamic behavior of an object. An object may have as many methods as required. A message contains 3 components.

- an object identifier that indicates the message receiver,
- a method name (corresponding to a method of the receiver), and
- arguments (additional information required for the execution of the method).

Example:

Ranga as an Object

Attributes:

```
name = " Ranga "
address = " 1, MCA,RVCE "
budget = " 2000 "
```

Methods:

```
    purchase() {
        Soma.takeOrder("Ranga", "sofa", "1, mca rvce",
        "12 November")
    }
getBudget()
    {return budget}
```

The message Soma.takeOrder(who , stock , address , date) is interpreted as follows:

- Soma is the receiver of the message;
- takeOrder() is a method call on Soma;
- "Ranga" , "stock" , "address" , "date" are arguments of the message.

Method: A method in object-oriented programming (OOP) is a procedure associated with a message and an object. A message is valid if the receiver has a method that corresponds to the method named in the message and the appropriate arguments, if any, are supplied with the message.

Method Signature: A method signature is part of the method declaration. It is the combination of the method name and the parameter list.

Method Overloading: It is the capability of definig the several functions with same name that performs similar task, but on different data types. In python Method over loading is not supported as it conciders only the recenst(last) function definition.

```
Example: def add(a,b):
                  return a+b
              def add(a,b,c):
                  return a*b*c
         print add(2,3)# will not work as python takes only the  recent definition.
```

Operator Overloading: The feature that allows the defination of different behaviour for a language built-in operator, depending on the data type is called Operator overloading. For example + operator in can be used perform arithmetic addition of two numbers and can also be used concatinate two strings.

In python for overloading the existing operators user have to define the respective spe-cial function as shown in the table 1 and table 2 respectively.

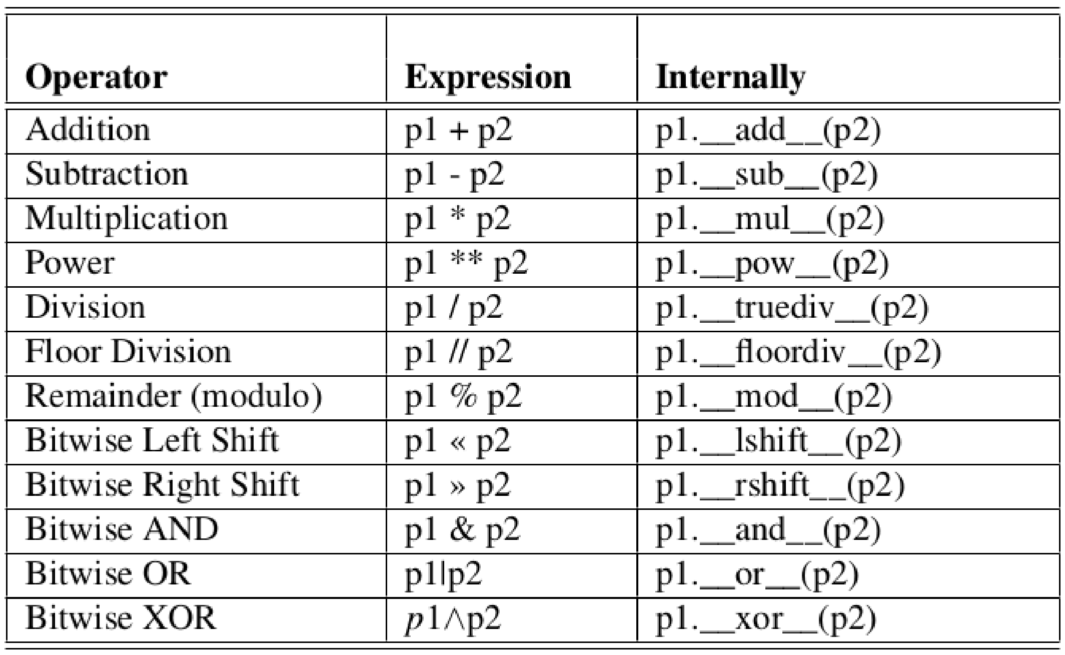


TABLE 1: Special Function for Overloading Arithmetic Operators

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 « p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 » p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1|p2	p1.__or__(p2)
Bitwise XOR	$p1 \wedge$p2	p1.__xor__(p2)

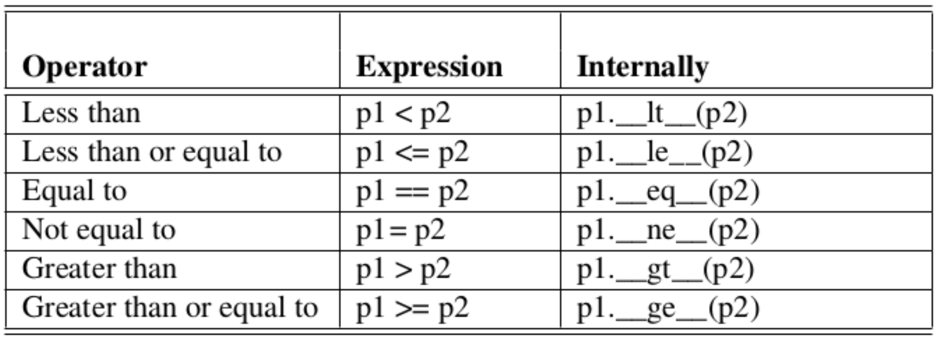


TABLE 2: Special Function for Logical Arithmetic Operators

Operator	Expression	Internally
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1= p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

Inheritance: In object Oriented Programming inheritance is the ability to define a new class that is a modified version of an existing class.

The primary advantage of this feature is that you can add new methods to a class without modifying the existing class. The existing class is called as Super Class/Parent class. A Subclass, "derived class", heir class, or child class is a derivative class which inherits the properties (methods and attributes) from one are more super classes.

Inheritance is a powerful feature. Some programs that would be complicated without inheritance can be written concisely and simply with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them.

Disadvantage of inheritance is that it make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be scattered among several modules.

In Inheritance base class and child classes are tightly coupled. Hence If you change the code of parent class, it will get affects to the all the child classes.

Types of Inheritance: Single: In this inheritance, a derived class is created from a single base class.

Multi level: In this inheritance, a derived class is created from another derived class.

Multiple: In this inheritance, a derived class is created from more than one base class. This inheritance is not supported by .NET Languages like C#, F# etc.

Hierarchical inheritance: In this inheritance, more than one derived classes are created from a single base.

Hybrid inheritance: This is combination of more than one inheritance. Hence, it may be a combination of Multilevel and Multiple inheritance or Hierarchical and Multilevel inheritance or Hierarchical and Multipath inheritance or Hierarchical, Multilevel and Multiple inheritance.

Uninheritable classes:

Inheritance in python: Python supports multiple inheritance i.e a sub-class can be derived from more than one base class. The child class inherits the attributes of its parent class, and user can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes
to inherit from is given after the class name −

class **SubClassName** (ParentClass1[, ParentClass2, ...]):
'Optional class documentation string'
class_suite.

Example:

```
class Player(object):
    ''' base class'''
    def __init__(self,name,mobile):
        self.name=name
        self.mobile=mobile
    def __str__(self):
        return "%s mobile number is %d" % (self.name, self.mobile)
class Crick_player(Player):
    def __init__(self, name,mobile,Noruns,Nowickets):
        Player.__init__(self, name,mobile)
        self.Noruns=Noruns
        self.Nowickets=Nowickets
    def __str__(self):
        return "%s scored %d runs and taken %d wickets, his mobile number is %d"
```

class, indicates that we are creating a class, The second word, Player, indicates the name of the class, **object** is a special variable in Python that you should include in the parentheses when you are creating a new class. In the second class defination **Crick_player** is a sub-class and **Player** is a base-class

The **issubclass(sub, sup)** boolean function returns **True**, if the given subclass **sub** is indeed a subclass of the superclass sup.

The **isinstance(obj, Class)** boolean function returns True, if obj is an instance of class Class or is an instance of a subclass of Class.

Method Overriding: Method overriding, in object oriented programming, is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has same name, same parameters or signature, and same return type as the method in the parent class(wiki).

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.

Example:

```
class Thought(object):
    def __init__(self):
        pass
    def message(self):
        print "I feel like I am diagonally parked in a parallel universe."

class Advice(Thought):
    def __init__(self):
        super(Advice, self).__init__()
    def message(self):
        print "Warning: Dates in calendar are closer than they appear"
        super(Advice, self).message()
```

Method Resolution Order in Python

The Python Method Resolution Order defines the class search path used by Python to search for the right method to use in classes having multi-inheritance.

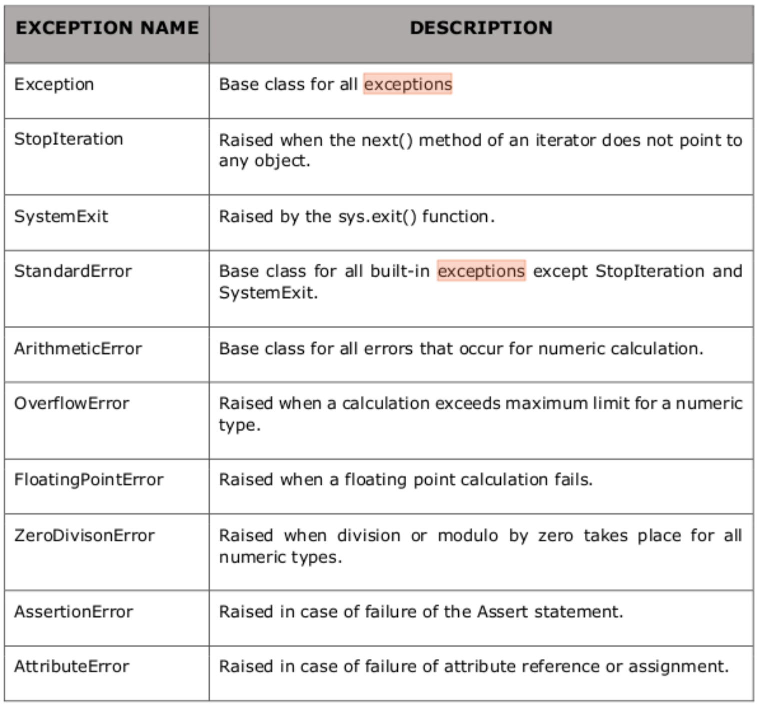
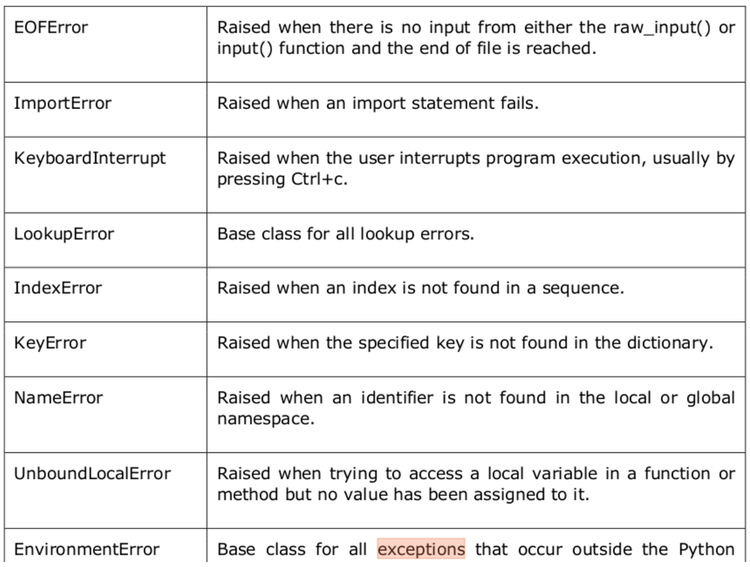
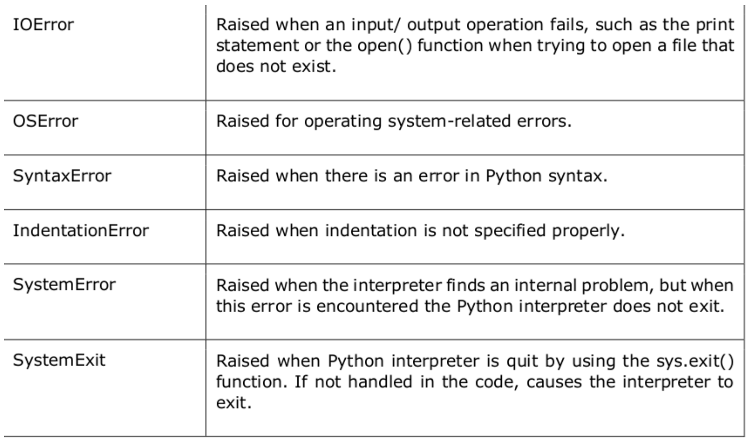
```
class X: pass
class Y: pass
class Z: pass
class A(X,Y): pass
class B(Y,Z): pass
class M(B,A,Z): pass
# Output:
# [<class '__main__.M'>, <class '__main__.B'>,
# <class '__main__.A'>, <class '__main__.X'>,
# <class '__main__.Y'>, <class '__main__.Z'>,
# <class 'object'>]
```

Exception:

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. In Python exception is an object that represents an error.

Exception Handling: If there is some suspicious code which may raise an exception, then place the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

Standard Exceptions in Python:

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.

Syntax:

```
try:
        statements to be executed
        -------
        -------
except ExceptionI:
        If there is ExceptionI, then execute this block.
        ------------
except ExceptionII:
        If there is ExceptionII, then execute this block.
        .....................
else:
        If there is no exception then execute this block.
        ----------------------------
```

A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.

You can also provide a generic except clause, which handles any exception.

After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.

The else-block is a good place for code that does not need the try: block's protection.

Example:

```
sname=input("Enter Existing file name")
dname=input("Enter new file name")
try:
  sh=open(sname, "r")
  dh = open(dname, "w")
  str=sh.read()
  dh.write(str)
except IOError:
  print ("Error: can\'t find file or read data")
else:
  print ("File successfully copied")
  sh.close()
  dh.close()
```

except clacuse with no exceptions.

```
try:
You do your operations here
......................
except:
If there is any exception, then execute this block.
......................
else:
If there is no exception then execute this block.
```

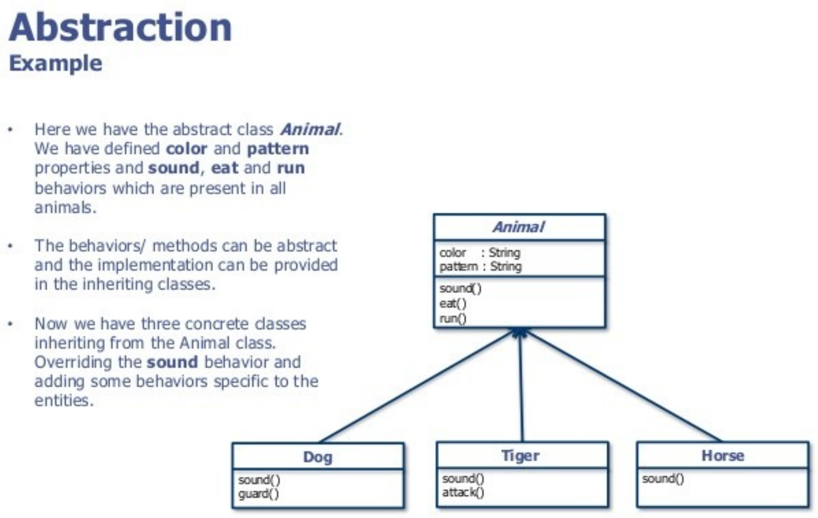
If no specific exception is specified then it catches all the exceptions that occurs. But drawback of this kind is the programmer can not identify the root cause of the problem.

Except clause with multiple exceptions:

```
try:
        You do your operations here
        ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
        If there is any exception from the given exception list,then execute this block.
......................
else:
        If there is no exception then execute this block.
```

Abstract Class: An abstract class is a class that cannot be directly instantiated. Instead, you instantiate an instance of a subclass. Typically, an abstract class has one or more operations that are abstract. An abstract operation has no implementation; it is pure declaration so that clients can bind to the abstract class. The most common way to indicate an abstract class or operation in the UML is to italicize the name.

You can also make properties abstract, indicating an abstract property or accessor methods. Italics are tricky to do on a whiteboards, so you can use the label: {abstract}.



Abstraction

Example

- Here we have the abstract class ***Animal***. We have defined **color** and **pattern** properties and **sound**, **eat** and **run** behaviors which are present in all animals.
- The behaviors/ methods can be abstract and the implementation can be provided in the inheriting classes.
- Now we have three concrete classes inheriting from the Animal class. Overriding the **sound** behavior and adding some behaviors specific to the entities.

Dynamic Binding: Dynamic binding also called dynamic dispatch is the process of linking procedure call to a specific sequence of code (method) at run-time. It means that the code to be executed for a specific procedure call is not known until run-time. Dynamic binding is also known as late binding or run-time binding.

Dynamic binding is an object oriented programming concept and it is related with polymorphism and inheritance.

Class Library: In object-oriented programming , a class library is a collection of prewritten classes or coded templates, any of which can be specified and used by a programmer when developing an application program. The programmer specifies which classes are being used and furnishes data that instantiate s each class as an object that can be called when the program is executed. Access to and use of a class library greatly simplifies the job of the programmer since standard, pretested code is available that the programmer doesn't have to write. In python import stament is used include the prewritten modules into the required script.