PYTHON PROGRAMMING (PS01CMCA31)

Unit – III

Composite Data Types, Functions and Exception Handling

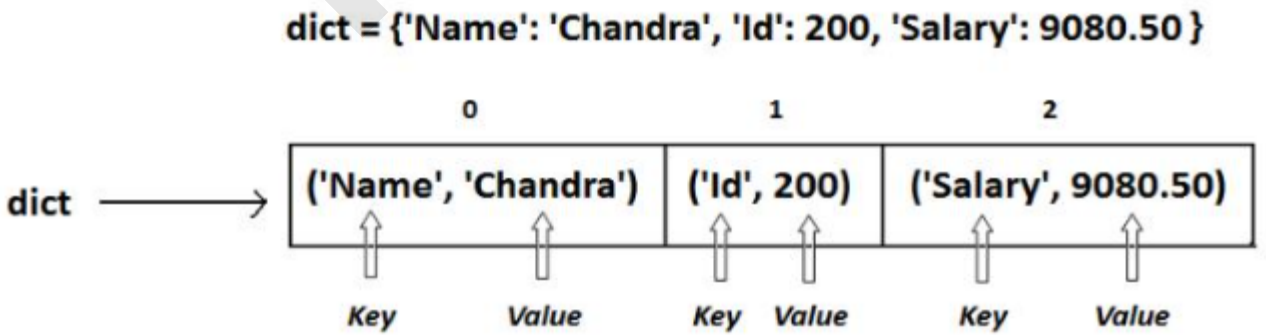
- Mapping type : dict
- Set type: set
- Functions
- Exception handling

Mapping type : dict

- A dictionary represents a group of elements arranged in the form of key-value pairs.
- The first element is considered as 'key' and the immediate next element is taken as its 'value'.
- The key and its value are separated by a colon (:)
- All key and value pairs in a dictionary are inserted in curly braces { }.

E.g.

dict = {'Name' : 'Chandra', 'Id' : 200, 'Salary' : 9080.50 }



Mapping type : dict

- **When 'key' is provided , we get back its 'value'.**

We can mention the key name inside the square braces, as dict['Name'].

```
dict = {'Name' : 'Chandra', 'Id' : 200, 'Salary' : 9080.50 }
print (dict['Name']);    print (dict['Id']);          print (dict['Salary'])
```

- **Operations on dictionaries**
  - To know how many key-value pairs in a dictionary use **len** function.

E.g.
```
dict = {'Name' : 'Chandra', 'Id' : 200, 'Salary' : 9080.50 }
    print('No. of key-value pairs ::',len(dict))
```

  - **To modify the existing value of a key.**

E.g.
```
dict['Salary'] = 10500
    dict['Id'] =10
     dict['Name']='Priya'
```

Mapping type : dict

- **Operations on dictionaries**
  - **To insert a new key-value pair in an existing dictionary.**

  E.g.

```
dict = {'Name' : 'Chandra', 'Id' : 200, 'Salary' : 9080.50 }
dict['Dept'] = 'Sales'
print('No. of key-value pairs ::',len(dict))
print(dict)
```

  - **To delete a key-value pair in an existing dictionary.**

  E.g.

```
dict = {'Name' : 'Chandra', 'Id' : 200, 'Salary' : 9080.50 }
del dict['Id']
print('No. of key-value pairs ::',len(dict))
print(dict)
```

Mapping type : dict

- **Operations on dictionaries**
  - **To test a whether key is available in an existing dictionary.**

E.g.

```
dict = {'Name' : 'Chandra', 'Id' : 200, 'Salary' : 9080.50 }
print(dict)
flag = 'Dept' in dict
print (' Is Dept key is available in dictionary or not ::',flag)
flag = 'Id' in dict
print (' Is Id key is available in dictionary or not ::',flag)
flag = 'Id' not in dict
print (' Is Id key is available in dictionary or not ::',flag)
```

Mapping type : dict

- **Operations on dictionaries**
  - **Duplicate keys are not allowed**. If we enter the same key again , the **old key will be overwritten and only the new key will be available.**

E.g.

```
dict = {'Name' : 'Chandra', 'Id' : 200, 'Salary' : 9080.50 }
print (' Dictionary key-value pairs ::',dict)
dict['Name'] = 'Abhay'
print (' Dictionary key-value pairs ::',dict)
```

Mapping type : dict

- **Dictionary Methods**

Method	Example	Description
clear()	d.clear()	Removes all key-value pairs from dictionary 'd'.
copy()	d1= d.copy()	Copies all elements from 'd' into a new dictionary 'd1'.
fromkeys()	d.fromkeys(s [,v])	Create a new dictionary with keys from sequence 's' and values all set to 'v'.
get()	d.get(k [,v])	Returns the value associated with key 'k'. If key is not found, it returns 'v'.
items()	d.items()	Returns an object that contains key-value pairs of 'd'. The pairs are stored as tuples in the object.
keys()	d.keys()	Returns a sequence of keys from the dictionary 'd'.
values()	d.values()	Returns a sequence of values from the dictionary 'd'.
update()	d.update(x)	Adds all elements from dictionary 'x' to 'd'.
pop()	d.pop(k [,v])	Removes the key 'k' and its value from 'd' and returns the value. If key is not found, then the value 'v' is returned. If key is not found and 'v' is not mentioned then 'KeyError' is raised.
setdefault()	d.setdefault(k [,v])	If key 'k' is found, its value is returned. If key is not found, then the k, v pair is stored into the dictionary 'd'.

Mapping type : dict

- **Dictionary Methods**

```
dict = {'Name' : 'Chandra', 'Id' : 200, 'Salary' : 9080.50 }
print (' Dictionary key-value pairs ::',dict)
print (' Only keys  ::',dict.keys())
print (' Only values  ::',dict.values())
print (' Key and value pairs as tuples  ::',dict.items())
```

E.g. Create a dictionary and find the sum of values.

```
dict = eval(input('Enter elements in { } """'))
ans = sum (dict.values())
print('Sum of values in the dictionary ::',ans)
```

Output

```
Enter elements (key-value pairs) in { }{'A':1, 'B':2, 'C':3}
Sum of values in the dictionary :: 6
```

Mapping type : dict

- **Dictionary Methods**

```
dict = {'Name' : 'Chandra', 'Id' : 200, 'Salary' : 9080.50 }
print('Display only keys')
print('----------------------')
for k in dict:
    print(k)
print('Display only values')
print('------------------------')
for k in dict:
    print(dict[k])
print('Display key & value pair')
print('------------------------------')
for k , v in dict.items():
    print('key={} and value={}'.format(k,v))
```

Mapping type : dict

- **Dictionary Methods**

E.g. Create a dictionary from keyboard and display the elements.

dict = {}

```
print('How many elements ? ', end='')
n = int(input())

for i in range(n):
    print('Enter key::',end= ' '); key = input();
    print('Enter Value::',end= ' '); val = int(input());
    dict.update({key:val})

print('Dictionary is::',dict)
```

Mapping type : dict

- **Dictionary Methods**

E.g. Create a dictionary with cricket player names and scores. Also search player name and retrieve score.

```
dict = {}
print('How many players ? ', end='')
n = int(input())
for i in range(n):
    print('Enter player name::',end= ' '); key = input();
    print('Enter score ::',end= ' '); val = int(input());
    dict.update({key:val})
print('Players in this match')
for pname in dict.keys():
     print(pname)
print('Enter player name::',end= ' ');
name = input();
runs = dict.get(name,-1)
if (runs == -1):    print('Player not found');
else:    print('{} made {}runs'.format(name,runs));
```

Set Type: set

- Mathematically a set is a collection of items not in any particular order. A Python set is similar to this mathematical definition with below additional conditions.
  - The elements in the set **cannot be duplicates**.
  - The **elements in the set are immutable**(cannot be modified) but the **set as a whole is mutable**.
  - There is **no index** attached to any element in a python set. So they do not support any indexing or slicing operation.

Set Operations

- The sets in python are typically used for mathematical operations like **union, intersection, difference and complement etc.**

Set Type: set

- **Creating a set**

A set is created by using the set() function or placing all the elements within a pair of curly braces.

Example

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
Months={"Jan","Feb","Mar"}
Dates={21,22,17}
print(Days)
print(Months)
print(Dates)
```

- **A set with strings, integers and boolean values: E.g.**

```
set1 = {"abc", 34, True, 40, "male"}
print(set1)
```

Set Type: set

- **Accessing Values in a Set**

We cannot access individual values in a set. We can only access all the elements together. But we can also get a list of individual elements by looping through the set. **E.g.**

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
 for d in Days:
   print(d)
```

- **Adding Items to a Set**

We can add elements to a set by using **add() method**. **E.g.**

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
Days.add("Sun")
print(Days)
```

Set Type: set

- **Removing Item from a Set**

We can remove elements from a set by using discard() or remove() method.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
Days.discard("Sun")
Days.remove("Sat")
print(Days)
```

- **Union of Sets**

The union operation on two sets produces a new set containing all the **distinct elements from both the sets. E.g.**

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA|DaysB; print(AllDays)
AllDays = DaysA.union(DaysB); print(AllDays)
```

Set Type: set

- **Intersection of Sets**

The intersection operation on two sets produces a new set containing only the common elements from both the sets. In the below example the element “Wed” is present in both the sets. **E.g.**

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA & DaysB
print(AllDays)
AllDays = DaysA.intersection(DaysB)
print(AllDays)
```

Set Type: set

- **Difference of Sets**

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set.

difference() method returns a new set, without removes the unwanted items from the original set.

difference_update() method removes the unwanted items from the original set.

Set Type: set

- **Difference of Sets**

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA - DaysB
print(AllDays)
AllDays = DaysA.difference(DaysB)
print(AllDays)
print(DaysA)
print(DaysB)
AllDays = DaysA.difference_update(DaysB)
print(AllDays)
print(DaysA)
print(DaysB)
```

Set Type: set

- **Compare Sets**
- We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets. **E.g.**

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
SubsetRes = DaysA <= DaysB
print(SubsetRes)
SubsetRes = DaysA.issubset(DaysB)
print(SubsetRes)
SupersetRes = DaysB >= DaysA
print(SupersetRes)
SupersetRes = DaysB.issuperset(DaysA)
print(SupersetRes)
```

Set Type: set

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set as intersection of 2 or more sets
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether set contains another set or not
remove()	Removes the specified element
union()	Return a set containing the union of sets

Set Type: set

- **List** is a collection which is ordered and changeable.

  Allows duplicate members.

- **Tuple** is a collection which is ordered and unchangeable.

  Allows duplicate members.

- **Set** is a collection which is unordered, unchangeable, and unindexed.

  No duplicate members.

- **Dictionary** is a collection which is ordered and changeable.

  No duplicate members.

Functions

- It is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks.
- Once a function is written, it can be reused as and when required.
- It will reduce the length of program.
- **Syntax of Function**

```
def function_name(parameters):
        """docstring"""
        statement(s)
```

- Keyword **def** that marks the start of the function header.
- A **function name** to uniquely identify the function.
- **Parameters** to pass values to a function. They are optional.
- A **colon (:)** to mark the end of the function header.
- Optional **docstring** to describe what the function does.
- An optional return statement to return a value from the function.

Functions

- **E.g. A function that accepts two values and finds their sums.**

```
#creating a function
def sum(a,b):
        """ This function find sum of two numbers"""
        c = a + b
        print( c )
#calling a function
sum (10,5)
sum(1.5,2.5)
sum('Good','Morning')
```

The parameters do not know which type of values they are receive till the values are passed at the time of calling function.

Functions

- **Returning results from a function**
- We can return the result or output from the function using a return statement in the body of the function.**E.g.**

```
def sum(a,b):
        """ This function find sum of two numbers"""
        c = a + b
        return c
#calling a function
ans = sum (10,5);          print (ans);
ans = sum(1.5,2.5) ;       print (ans);
ans = sum('Good','Morning') ; print (ans);
```

Functions

- **A program to find factorial of a function.**

```
def fact(n):
        fact = 1
        while(n>=1):
                fact *= n
                n = n - 1
        return fact
#calling a function
ans = fact (2);          print (ans);
ans = fact (3);          print (ans);
ans = fact (4);          print (ans);
```

Functions

Exercise:

A function to find addition, subtraction, multiplication and division of two numbers.

A function to find the length of string.

A function to find the factorial of a number.

A function to concatenate two input string.

A function to test whether a number is ever or odd.

A function to find factorial of given number.

A function to find the entered number is prime or not.

Functions

Formal & Actual Arguments

- When a function is defined the parameters used to retrieve values from outside the function is called **formal parameters.**
- When we call the function, we should pass the values or data to the function are called **actual parameters.**

```
def sum(a,b):      # a and b are formal parameters.
        """ This function find sum of two numbers"""
        c = a + b
        return c
#calling a function
x = 10; y = 20
ans = sum (x,y);          # x and y are actual parameters.
print (ans);
```

Functions

- Actual parameters are four types.
  - Positional arguments
  - Keyword arguments
  - Default arguments
  - Variable length arguments
- **Positional arguments :** The number of arguments and their positions in the function definition should match exactly with the number and position of the argument in the function call.
- **Keyword arguments :** The parameters are identified by their names.
- **Default arguments :** The default value for the function arguments.
- **Variable length arguments:** Sometimes the programmer does not know how many values a function may receiver. E.g. To write a function to sum of N numbers.

Functions

- **Positional arguments : E.g.**

```
def sum(a,b):      # a and b are formal parameters.
        """ This function find sum of two numbers"""
        c = a + b
        return c
#calling a function
x = 10; y = 20
ans = sum (x,y);          # x and y are positional parameters.
print (ans);
```

Functions

- **Keyword arguments :**

```
def product(iname, price):      # a and b are formal parameters.
        """ This function find sum of two numbers"""
        print('Item name is ::',iname);
        print('Item price is ::',price);
#calling a function
product (iname='Pencile',price=20); # keyword parameters.
product (price=40, iname='Pen',); # keyword parameters.
x = 'Notebook' ; y = 120
product (iname=x,price=y); # x and y are keyword parameters.
```

Functions

- **Default arguments :**

```
def product(iname, price = 50):      # a and b are formal parameters.
        """ This function find sum of two numbers"""
        print('Item name is ::',iname);
        print('Item price is ::',price);
#calling a function
product (iname='Pencile'); # default parameters.
product (iname='Pen', price=60); # default parameters.
```

- **Variable length arguments:**

It is an argument that can accept any number of values. It is written with a ' * ' symbol in the function definition.

E.g. def function-name (farg , *args) :

farg is a formal argument & args is a variable length argument. We can pass one or more values to this args and it will store them all in a tuple.

Functions

- **Variable length arguments:**

```
def add(farg, *args):
    print("Formal argument ::",farg)
    sum = 0
    for i in args:
        sum+=i
    print('sum of all numbers =',farg+sum)
add (5,10)
add(2,10,20)
def add(*args):
    sum = 0
    for i in args:
        sum+=i
    print('sum of all numbers =',sum)
add (5,10)
add(2,10,20)
```

Set Type: set

- **Variable length arguments:**

A keyword variable length argument is an argument that can accept any number of values provided in the format of keys and values. We can use it with '**' before the argument.

```
def display(farg, **args):
        print("Formal argument ::",farg)
        for x, y in args.items():
                print('key =',x)
                print('value=',y)
display(5,rno=10)
print()
display(10,rno=20,name='Mahesh')
```

Functions

- **Local and Global variable**

When we declare a variable inside a function, it becomes a local variable. Its scope is limited only to that function. It could not be access outside the function.

E.g.

```
def display():
    a = 1
    a = a + 5
    print("Value of a  ::",a)
display()

# print statement gives an error
# print("Value of a  ::",a)
```

Functions

- **Local and Global variable**

When we declare a variable above a function, it becomes a global variable. Its scope is the entire program body written below it.

E.g.

```
a=10    # Global variable
def display():
        b=10        # Local variable
        b = b + 5
        print("Value of b within a function ::",b)
        print("Value of a within a function ::",a)
display()
# print statement gives an error
print("Value of a  outside the function::",a)
```

Functions

- **Local and Global variable**

Sometimes local and global variable have the same name. In that function by default refers to the local variable and ignore the global variable. So global variable is not accessible inside function, but accessible outside the function.

E.g.

```
a=10    # Global variable
def display():
        a=10          # Local variable
        a = a + 5
        print("Value of a within a function ::",a)
display()
# print statement gives an error
print("Value of a  outside the function::",a)
```

Functions

Passing group of elements to a function

- To passing a group of elements like numbers or strings, we can accep them into a list and then pass the list to the function.
- **Program to accept a numbers and display its sum & average.**

```
def sum_avg (lst):
    n = len(lst)
    sum=0
    for i in lst:
        sum+=i
    avg = sum/n
    return sum , avg
print('Enter numbers separated by space::')
lst = [ int(x) for x in input().split()]
s , a = sum_avg(lst)
print('Sum of all numbers::',s)
print('Aveerage of all numbers::',a)
```

Functions

- **Recursive function**

A function call itself is known as recursive function.

```
def factorial(n):
    if n == 0:
        result = 1
    else:
        result = n * factorial(n-1)
    return result
for i in range(1,6):
    print('factorial of {} is {}'.format(i,factorial(i)))
```

Functions

- **Anonymous functions or Lambdas**

A function without a name is called 'anonymous function'. They are defined using the keyword lambda and hence they are also called 'Lambda functions'. **Format : lambda argument_list : expression**

Normal function

```
def square(x) :
        return x * x
```

Lambda function

```
lambda x : x * x
```

Lambda function returns a function, so we write it as

```
f = lambda x : x * x
```

Here f is the function name to which the lambda expression is assigned.

```
f = lambda x : x * x
value = f(3)
print('Square of 3 is ::',value)
```

Exception

- As a human being it is possible to commit **errors** either in the design of the software or in writing the code. The errors in the software are called **bugs** and the process of removing them is called **'debugging'.**
- **Types of errors**
  - Compile time errors
  - Runtime errors
  - Logical errors
- **Compile time errors :** These are syntactical errors found in the code, due to which a program fails to compile. **E.g.**

```
x = 1
if x == 1
    print('Error of colon sign is missing')
```

- **Runtime errors :** When PVM cannot execute the byte code, it flags runtime error. Runtime errors are not detected by the Python compiler. They are detected by the PVM, only at runtime. **E.g.**

```
a = 'Hi';  b = 10 ; print (a+b)
```

Exception

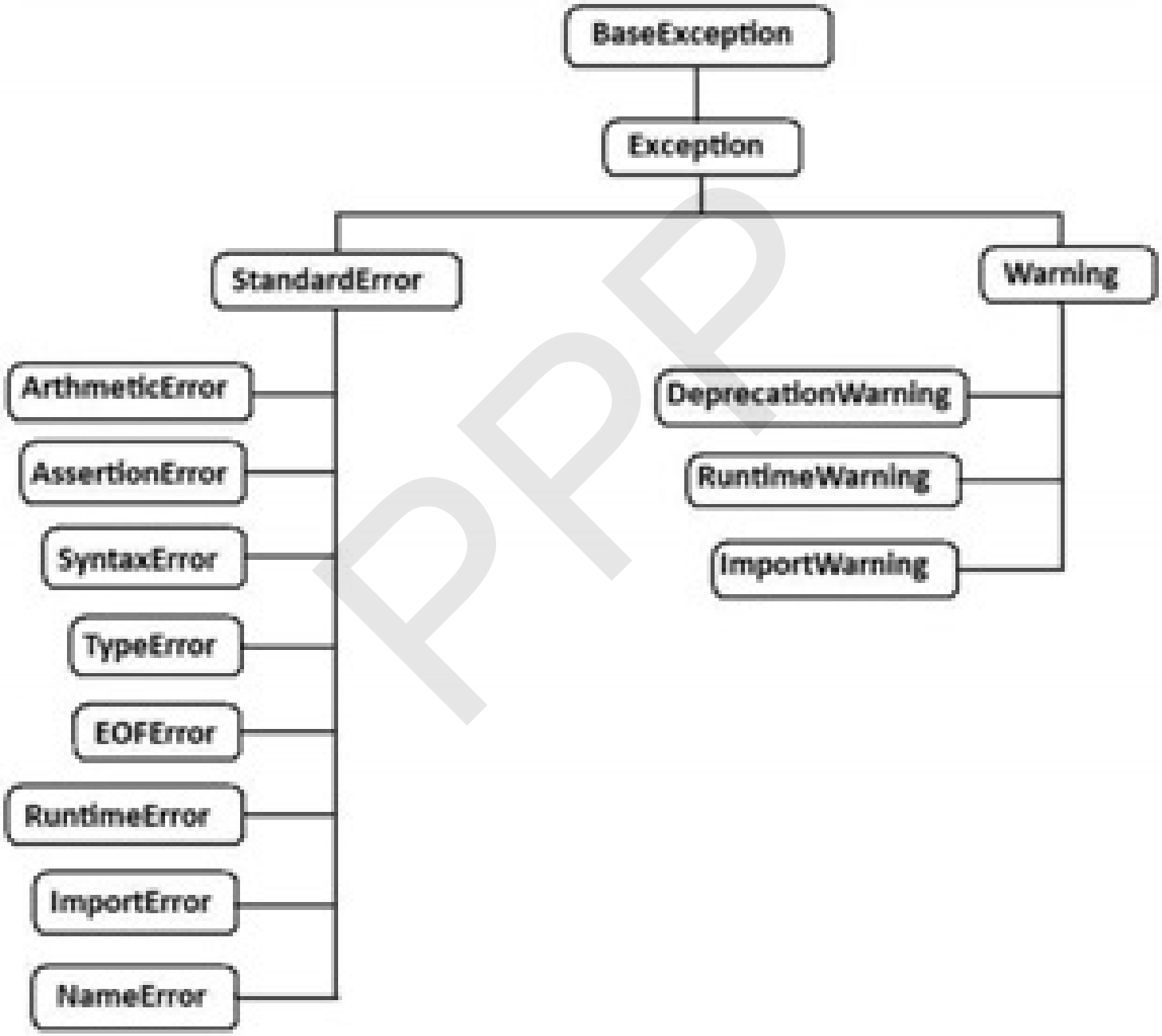
- Logical errors : These errors depict flaws in the login of the program. The programmer might be using wrong formula or the design of the program is wrong. Logical errors are not detected either by Python compiler or PVM. **E.g.**

```
sal = 2000
# increment the salary by 20%
sal = sal * 20 / 100
# sal = sal + sal * 20 / 100
print (sal)
```

Exception

- An exception is a runtime error which can be handled by the programmer. If the programmer cannot do anything in case of an error, then it is called an **error** and not an **exception.**
- All exceptions are represented as classes in Python.
- Exceptions already available in python are called **'built-in' exception.**
- The base class for all built-in exceptions is **'BaseException'** class.
- From BaseException class, the sub class **'Exception'** is derived.
- From **Exception class**, the sub classes **'StandardError'** and **'Warning'** are derived.
- All **errors (or exceptions)** are defined as sub classes of **StandardError.** An error should be compulsorily handled otherwise the program will not execute. Similarly, all **warnings** are derived as sub classes from **'Warning' class**. A warning represents a caution and even though it is not handled, the program will execute. So, warnings can be neglected but errors cannot be neglected.

Exception



Exception

To handle exceptions, the programmer should perform the following three steps:

Step 1: The programmer should observe the statements, where there may be a possibility of exceptions. Such statements should be written inside a 'try' block. A try block looks like as follows:

```
try: statements
```

if some exception arises inside it, the program will not be terminated. When PVM understands that there is an exception, it jumps into an 'except' block.

Step 2: The programmer should write the 'except' block where he should display the exception details to the user. The programmer should also display a message regarding what can be done to avoid this error. Except block looks like as follows:

```
except exceptionname:
        statements #these statements form handler
```

The statements written inside an except block are **called 'handlers'** since they handle the situation when the exception occurs.

Exception

Step 3: Lastly, the programmer should perform clean up actions like closing the files and terminating any other processes which are running. The programmer should write this code in the finally block. **'finally' block is executed either exception occurred or not.** Finally block looks like as follows:

finally: statements

E.g.

```
#an exception handling example
try:
   a = 10
   b = 0
   c = a / b
   print('Value of c :',c)
except ZeroDivisionError:
   print('Division by zero happened')
   print('Please do not enter 0 in denominator')
finally:
   print('Final block is executed')
```

Exception

- **Important built-in exceptions in Python**

Exception Class Name	Description
Exception	Represents any type of exception. All exceptions are sub classes of this class.
ArithmeticError	Represents the base class for arithmetic errors like OverflowError,ZeroDivisionError,FloatingPointError.
IndexError	Raised when a sequence index or subscript is out of range.
KeyError	Raised when a mapping (dictionary) key is not found in the set of existing keys.
RuntimeError	Raised when an error is detected that doesn't fall in any of the other categories.
StopIteration	Raised by an iterator's next() method to signal that there are no more elements.
IndentationError	Raised when indentation is not specified properly.
ValueError	Raised when a built-in operation or function receives an argument that has right datatype but wrong value.
ZeroDivisionError	Raised when the denominator is zero in adivision or modulus operation.

Exception

The complete exception handling syntax will be in the following format:

```
try:
        statements
except  Exception1:
        handler1
except Exception2:
        handler2
else:
        statements
finally:
        statements
```

The following points are noteworthy:

- A single try block can be followed by several except blocks.
- Multiple except blocks can be used to handle multiple exceptions.
- We cannot write except blocks without a try block.
- We can write a try block without any except blocks.
- Else block and finally blocks are not compulsory.
- When there is no exception, else block is executed after try block.
- Finally block is always executed.

Exception

```
#an exception handling example
import math
try:
   a = 10;   b = 2;   c = a / b ;
   print('Value of c :',c)
   lst = [1,2,3,4]
   for i in range(4):
      print(lst[i])
   x = 10
   print('Sqrt of x is ::',math.sqrt(x))
except ZeroDivisionError:
   print('Division by zero happened')
   print('Please do not enter 0 in denominator')
except IndexError:
   print('Subscript or index specify is out of range ')
else:
   print('Run time error occurs')
 finally:
   print('Final block is executed')
```

Exception

- To catch multiple exceptions, a single except block and write all the exceptions as a tuple inside parentheses as:

  except (Exceptionclass1, Exceptionclass2,...):

- We can catch the exception as an object that contains some description about the exception.

  except Exceptionclass as obj:

- To catch any type of exception where we are not bothered about which type of exception it is, we can write except block without mentioning any Exceptionclass name as:

  except:

E.g.

```
except (TypeError, ZeroDivisionError):
        print('Either TypeError or ZeroDivisionError occurred.')
except:
        print('Some exception occurred.')
```

Exception

- The **assert statement** is useful to ensure that a given condition is True. If it is not true, it raises **AssertionError**. The syntax is as follows:

  assert condition, message

- If the condition is **False**, then the exception by the name AssertionError is raised along with the 'message' written in the assert statement.
- If 'message' is not given in the assert statement, and the condition is False, then also AssertionError is raised without message.

A program using the assert statement and catching AssertionError.

```
try:
  x = int(input('Enter a number between 5 and 10:'))
  assert x>=5 and x<=10
  print('The number entered:', x)
except AssertionError:
  print('The condition is not fulfilled')
```

Exception

A Python program to use the assert statement with a message.

```
try:
   x = int(input('Enter a number between 5 and 10:'))
   assert x>=5 and x<=10, "Input no. must be between 5 and 10"
   print('The number entered:', x)
except AssertionError as obj:
   print(obj)
```

Exception

User- defined exceptions

Like the built-in exceptions of Python, the programmer can also create his/her own exceptions which are called **'User- defined exceptions'** or **'Custom exceptions'**.

Steps to be followed for User Defined Exceptions

1. Since all **exceptions are classes**, the programmer has to create own exception as a class. Also, that **class as a sub class to the in-built 'Exception' class**.

Format

```
class MyException(Exception):
def __init__(self, arg):
self.msg = arg
```

Here, 'MyException' class is the sub class for 'Exception' class.

This class has a constructor where a variable 'msg' receives a message passed from outside through 'arg'.

Exception

2. When the programmer suspects the possibility of exception than use **'raise' statement** as:

raise MyException('message')

Here, raise statement is raising MyException class object that contains the given 'message'.

3. The programmer can insert the code inside a 'try' block and catch the exception using 'except' block as:

try:

code

Except MyException as me:

print(me)

Here, the object 'me' contains the message given in the raise statement.

Exception

A Python program to create our own exception and raise it when needed.

```
class MyException(Exception):
  def __init__(self, arg):
    self.msg = arg
try:
   bank = {'Raj':5000, 'Vani':8000, 'Ajay':2000, 'Naresh':3000}
   for k,v in bank.items():
      print('Name= ' , k) ;      print('Basics= ' , v) ;
      if(v<3000):
         raise MyException('Balance amount is less in the account of '+k)
except MyException as me:
   print(me)
```