OOP Using Python

➢ Handling Exceptions :

An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids program to crash. Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error then exception handling can be used. An exception can be raised in program by using the "**`raise exception`"** statement. Raising an exception breaks current code execution and returns the exception back until it is handled.

Syntax :

```
try:
    some statements here
except:
    exception handling statements
```

Example	Out Put

try:
`i = int(input("Enter Value : "))`
`print ("Square of ",i, " is : ",i * i);`
except:
`print ("Oops....Invalid Input!!!!")`
	`Enter Value : 8`
	`Square of 8 is : 64`
	
	`Enter Value : a`
	`Oops....Invalid Input!!!!`

➢ Exceptions as a control flow mechanism :

There's one other form of flow control that is common within Python, exception handling. One thing that differs compared to many other languages is that in Python exceptions are relatively lightweight. This means they aren't only meant to be used in the most extreme circumstances; instead it is not uncommon to use them as a type of control flow.

It is also possible to have two more clauses in a try-except block. "**else"** is called if no exception was caught. "**finally**" is caused no matter what, and is useful for cleaning up resources.

- **Common Exceptions :**

Exception	Description
BaseException	The base exception, catching this will catch all exceptions.
Exception	The lowest-level non-system exiting exception. Typically this is the lowest level exception you'd want to catch.
AttributeError	Raised when attempting to access an attribute of an object that doesn't exist.
ImportError	Raised when something cannot be imported.
IndexError	Raised when a sequence index is out of range.
KeyError	Raised when trying to access a dictionary key that does not exist.
StopIteration	Raised when an iterable is exhausted.
TypeError	Raised when an operation is invalid for a specific type.
ValueError	Raised when a function receives a value of an appropriate type but inappropriate value.
ZeroDivisionError	Raised when attempting to divide by zero.

Example	Out Put

try:
  i = int(input("Enter 1st Value : "))
  j = int(input("Enter 2nd Value : "))
  k = i / j
except ZeroDivisionError:
  print("Invalid Second Value")
except :
  print("Invalid Input!!!")
else :
  print (i, "/", j, " = " ,k);
finally:
  print ("Thanks!!!!")
	Enter 1st Value : 5
	Enter 2nd Value : 2
	5 / 2 = 2.5
	Thanks!!!!
	
	Enter 1st Value : 5
	Enter 2nd Value : 0
	Invalid Second Value
	Thanks!!!!
	
	Enter 1st Value : a
	Invalid Input
	Thanks!!!!

➢ Assertions :

- **What is Assertion? :** Assertions are statements that assert or state a fact confidently in python program. **For example**, while writing a division function, divisor shouldn't be zero, programmers assert divisor is not equal to zero. Assertions are simply **Boolean** expressions that checks if the conditions return **true** or **not**. If it is **true**, the program does nothing and moves to the next line of code. However, if it's **false**, the program stops and throws an error. It is also a **debugging tool** as it brings the program on halt as soon as any error is occurred and shows on which point of the program error has occurred.
- So, Assertions are the **condition or boolean expression** which are always supposed to be true in the code. Assert statement takes an expression and optional message. Assert statement is used to check types, values of argument and the output of the function. Assert statement is used as debugging tool as it halts the program at the point where an error occurs.

Syntax :

```
assert <condition>,<error message>
```

Example	Out Put

`i = int(input("Enter 1st Value : "))`
`j = int(input("Enter 2nd Value : "))`
`assert j != 0, "Invalid Second Value."`
`print (i,"/",j," = ",i/j)`
	`Enter 1st Value : 5`
	`Enter 2nd Value : 2`
	`5 / 2 = 2.5`
	.....
	`Enter 1st Value : 5`
	`Enter 2nd Value : 0`
	
	`Traceback (most recent call last):`
	`File "C:/assert.py", line 3, in <module>`
	`assert j != 0,`
	`"Invalid Second Value."`
	`AssertionError: Invalid Second Value.`

➢ Abstract Data Types and Classes :

Abstraction is one of the most powerful ideas in computer science. It separates "**The What from The How**". Abstraction provides modularity. Classes are the Python representation for "Abstract Data Types" (ADT) a very useful notion in any programming language. An ADT involves both data and operations on that data.

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.

The user of data type need not know that data type is implemented, for example, we have been using **`int, float, char`** data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it. We can think of **ADT as a black box** which hides the inner structure and design of the data type. `LIST` and `Dictionary` are best example of ADT.

In object oriented programming, an `abstract class` is like a `normal class` that **cannot be instantiated**. It's a way for the class designer to provide a **blueprint** of a class, so that its' methods have to be implemented by the developer writing a class that inherits from it.

- **CLASSES in Python :**

  Python is an object oriented programming language. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stress on objects. Object is simply a collection of data (variables) and methods (functions) that act on those data. And, class is a blueprint for the object.

  We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object. As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called ***instantiation***.

  - **Defining a CLASS in Python :**

    Like function definitions begin with the keyword def, in Python, we define a class using the keyword class. The first string is called ***docstring*** and has a brief description about the class. Although not mandatory, this is recommended.

    A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions. There are also special attributes in it that begins with double **underscores (__)**. For example, **__doc__** gives us the ***docstring*** of that class. As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

Syntax :

```
Class myClass:
      "This is a docstring."
      ##Class Members
```

Example

```
class MyClass:
      "This is my First class"
      a = 100
      def func(self):
            print("Hello From Function in Class")
print("Value of Class Variable a = ",MyClass.a)
print(MyClass.func)
print(MyClass.__doc__)
##Create object of a Class and call Class Function
myObj = MyClass()
myObj.func()
```

Out Put

```
Value of Class Variable a =  100
<function MyClass.func at 0x00000000004BC1E0>
This is my First class
Hello From Function in Class
```

- **Constructors in Python :**

A class function that begins with ***double underscore (__)*** is called ***special function*** as they have special meaning. One particular interest is the **__init__()** function. This special function gets called whenever a new object of that class is instantiated. This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

Example

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

    def display (self):
        print("ID:",self.id,"\nName:",self.name)

emp1 = Employee("ABC",101)
emp2 = Employee("PQR",102)
#accessing display() method to print employee 1 information
emp1.display();
#accessing display() method to print employee 2 information
emp2.display();
```

Out Put

```
ID: 101
Name: ABC
ID: 102
Name: PQR
```

- **Deleting Attributes or Object of Class:**

  Any attribute of an object can be deleted anytime, using the **"del"** statement. It is a kind of **destructor** in python.

Example

```
class Employee:
    def __init__(self,name,id):
        self.id = id;
        self.name = name;
    def display (self):
        print("ID:",self.id,"\nName:",self.name)

emp1 = Employee("ABC",101)
emp2 = Employee("PQR",102)

#accessing display() method to print employee 1 information
emp1.display()
#accessing display() method to print employee 2 information
emp2.display()

del emp1
emp1.display()
```

Out Put

```
ID: 101
Name: ABC
ID: 102
Name: PQR
Traceback (most recent call last):
  File "C:\01.py", line 17, in <module>
    emp1.display()
NameError: name 'emp1' is not defined
```

➢ Inheritance in Python :

Inheritance is one of the mechanisms to achieve the same. In inheritance, a class (usually called **superclass**) is inherited by another class (usually called **subclass**). The subclass adds some attributes to superclass. Below is a sample Python program to show how inheritance is implemented in Python.

A class can inherit attributes and behavior methods from another class, called the superclass. A class which inherits from a superclass is called a subclass, also called **heir class** or **child class**. Superclasses are sometimes called ancestors as well. There exists a hierarchy relationship between classes.

Syntax :

```
class BaseClass:
      Body of base class

class DerivedClass(BaseClass):
      Body of derived class
```

Example

```
class Person:

    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def Name(self):
        return "Name: "+self.firstname + " " + self.lastname

class Employee(Person):

    def __init__(self, first, last, staffnum):
        Person.__init__(self,first, last)
        self.staffnumber = staffnum

    def GetEmployee(self):
        return self.Name() + ", \nEmployee ID:" +  self.staffnumber

x = Person("FName", "LName")
y = Employee("FName", "LName", "101")

print(x.Name())
print(y.GetEmployee())
```

Out Put

```
Name: FName LName
Name: FName LName,
Employee ID:101
```

➢ Encapsulation and Information hiding :

The terms encapsulation and abstraction (also data hiding) are often used as synonyms. They are nearly synonymous, i.e. abstraction is achieved though encapsulation. Data hiding and encapsulation are the same concept, so it's correct to use them as synonyms.

Encapsulation is the mechanism for restricting the access to some of an object's components; this means that the internal representation of an object can't be seen from outside of the objects definition. Access to this data is typically only achieved through special methods: **Getters and Setters**. By using solely **`get()`** and **`set()`** methods, we can make sure that the internal data cannot be accidentally set into an inconsistent or invalid state.

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, Private attributes denoted using underscore as prefix i.e single **“ _ “** or double **“ __“**.

Example

```
class Person:
  def __init__(self):
    self.name = 'Mr. ABC'
    self.__lastname = 'PQR'

  def PrintName(self):
    return self.name +' ' + self.__lastname

#Outside class
P = Person()

print("Full Name:",P.PrintName())

print("First Name:",P.name)

#AttributeError: 'Person' object has no attribute '__lastname'
print("Last Name:",P.__lastname)
```

Out Put

```
Full Name: Mr. ABC PQR
First Name: Mr. ABC
Traceback (most recent call last):
  File "C:\Encapsulation.py", line 14, in <module>
    print("Last Name:",P.__lastname)
AttributeError: 'Person' object has no attribute '__lastname'
```

➢ Search Algorithms in Python :

Searching is a very basic necessity when you store data in different data structures. The simplest approach is to go across every element in the data structure and match it with the value you are searching for. This is known as "**Linear Search**". It is inefficient and rarely used, but creating a program for it gives an idea about how we can implement some advanced search algorithms.

Basically there are two types of Searching Algorithms in Python.

1. Linear Search 2. Interpolation Search

1. Linear Search :

In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data structure.

This technique iterates over the sequence, one item at a time, until the specific item is found or all items have been examined. In Python, a target item can be found in a sequence using the in operator.

Example

```
def myLinearSearch(arr, n, x):
    for i in range (0, n):
        if (arr[i] == x):
            return i;
    return -1;
arr = [10, 20, 30, 40, 50];
print("Values : ",arr)
result = myLinearSearch(arr, len(arr), int(input("Enter Value: ")))
if(result == -1):
    print("Element is not present in this Array")
else:
    print("Element is present at index: ", result);
```

Out Put

```
Values :  [10, 20, 30, 40, 50]
Enter Value: 20
Element is present at index:  1
```

```
Values :  [10, 20, 30, 40, 50]
Enter Value: 200
Element is not present in this Array
```

2. Interpolation Search :

This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed. Initially, the probe position is the position of the middle most item of the collection. If a match occurs, then the index of the item is returned. If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. Otherwise, the item is searched in the ***subarray*** to the left of the middle item. This process continues on the sub-array as well until the size of ***subarray*** reduces to zero.

The Interpolation Search is an improvement over Binary Search for instances, where the values in a sorted array are uniformly distributed. Binary Search always goes to the middle element to check. On the other hand, interpolation search may go to different locations according to the value of the key being searched. For example, if the value of the key is closer to the last element, interpolation search is likely to start search toward the end side.

Example

```
# If x is present in arr[0..n-1], then returns
# index of it, else returns -1
def interpolationSearch(arr, n, x):
    # Find indexs of two corners
    lo = 0
    hi = (n - 1)
    # Since array is sorted, an element present
    # in array must be in range defined by corner
    while lo <= hi and x >= arr[lo] and x <= arr[hi]:
        # Probing the position with keeping
        # uniform distribution in mind.
        pos  = lo + int(((float(hi - lo) /
            ( arr[hi] - arr[lo])) * ( x - arr[lo])))
        # Condition of target found
        if arr[pos] == x:
            return pos
        # If x is larger, x is in upper part
        if arr[pos] < x:
            lo = pos + 1;
        # If x is smaller, x is in lower part
        else:
            hi = pos - 1;
    return -1

# Array of items in which search will be conducted
arr = [10, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 33, 35, 42, 47]
print("Values : ",arr)
index = interpolationSearch(arr, len(arr), int(input("Enter Value: ")))
if index != -1:
    print ("Element found at index :", index)
else:
    print ("Element not found")
```

Out Put

```
Values :  [10, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 33, 35, 42, 47]
Enter Value: 16
Element found at index : 3
```

```
Values :  [10, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 33, 35, 42, 47]
Enter Value: 50
Element not found
```

➢ Sorting Algorithms in Python :

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

There are FIVE sorting algorithms in python.

1. Bubble Sort ***2. Merge Sort*** ***3. Insertion Sort***

4. Shell Sort ***5. Selection Sort***

1. Bubble Sort :

It is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. So, Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Logical Example :

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since **5 > 1**.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since **5 > 4**

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since **5 > 2**

Now, since these elements are already in order (**8 > 5**), algorithm does not swap them.

(1 4 2 5 8) –> (1 4 2 5 8),

Second Pass:

(1 4 2 5 8) –> (1 4 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since **4 > 2**

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

Example

```
def bubblesort(list):
# Swap the elements to arrange in order
    for iter_num in range(len(list)-1,0,-1):
        for idx in range(iter_num):
            if list[idx]>list[idx+1]:
                temp = list[idx]
                list[idx] = list[idx+1]
                list[idx+1] = temp
list = [19,12,31,45,16,14,120,29]
print("\nBefore Sorting :\n",list)
bubblesort(list)
print("\nAfter Sorting :\n",list)
```

Out Put

```
Before Sorting :
 [19, 12, 31, 45, 16, 14, 120, 29]
After Sorting :
 [12, 14, 16, 19, 29, 31, 45, 120]
```

2. Merge Sort :

Merge Sort is a **Divide and Conquer algorithm**. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The **`merge()`** function is used for merging two halves. The **`merge(arr, l, m, r)`** is key process that assumes that **`arr[l..m] and arr[m+1..r]`** are sorted and merges the two sorted sub-arrays into one. So, Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Example

```
def merge_sort(unsorted_list):
    if len(unsorted_list) <= 1:
        return unsorted_list

# Find the middle point and devide it
    middle = len(unsorted_list) // 2
    left_list = unsorted_list[:middle]
    right_list = unsorted_list[middle:]
    left_list = merge_sort(left_list)
    right_list = merge_sort(right_list)
    return list(merge(left_list, right_list))
```

```
# Merge the sorted halves
def merge(left_half,right_half):
    res = []
    while len(left_half) != 0 and len(right_half) != 0:
        if left_half[0] < right_half[0]:
            res.append(left_half[0])
            left_half.remove(left_half[0])
        else:
            res.append(right_half[0])
            right_half.remove(right_half[0])
    if len(left_half) == 0:
        res = res + right_half
    else:
        res = res + left_half
    return res


unsorted_list = [19,12,31,45,16,14,120,29]

print("\nBefore Sorting :\n",unsorted_list)

print("\nAfter Sorting :\n",merge_sort(unsorted_list))
```

Out Put

Before Sorting :

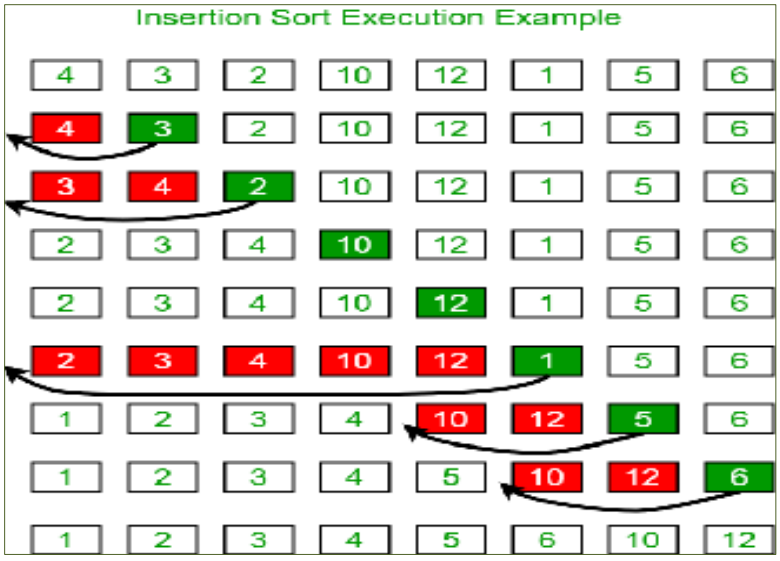
[19, 12, 31, 45, 16, 14, 120, 29]

After Sorting :

[12, 14, 16, 19, 29, 31, 45, 120]

3. Insertion Sort :

Insertion sort involves finding the right place for a given element in a sorted list. So in beginning it compares the first two elements and sort them by comparing them. Then picks the third element and find its proper position among the previous two sorted elements. This gradually goes on adding more elements to the already sorted list by putting them in their proper position.



Example

```
def insertion_sort(InputList):
    for i in range(1, len(InputList)):
        j = i-1
        nxt_element = InputList[i]

# Compare the current element with next one

        while (InputList[j] > nxt_element) and (j >= 0):
            InputList[j+1] = InputList[j]
            j=j-1
        InputList[j+1] = nxt_element

list = [19,12,31,45,16,14,120,29]
print("\nBefore Sorting :\n",list)

insertion_sort(list)

print("\nAfter Sorting :\n",list)
```

Out Put

```
Before Sorting :
 [19, 12, 31, 45, 16, 14, 120, 29]

After Sorting :
 [12, 14, 16, 19, 29, 31, 45, 120]
```

4. Shell Sort :

Shell Sort is mainly a variation of ***Insertion Sort***. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of ***ShellSort*** is to allow exchange of far items. In ***ShellSort***, we make the array ***h-sorted*** for a large value of ***h***. We keep reducing the value of ***h*** until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element is sorted.

Example

```
def shellSort(input_list):
    gap = len(input_list) // 2
    while gap > 0:
        for i in range(gap, len(input_list)):
            temp = input_list[i]
            j = i

# Sort the sub list for this gap

            while j >= gap and input_list[j - gap] > temp:
                input_list[j] = input_list[j - gap]
                j = j-gap
            input_list[j] = temp

# Reduce the gap for the next element

        gap = gap//2
list = [19,12,31,45,16,14,120,29]
print("\nBefore Sorting :\n",list)
shellSort(list)
print("\nAfter Sorting :\n",list)
```

Out Put

```
Before Sorting :
 [19, 12, 31, 45, 16, 14, 120, 29]
After Sorting :
 [12, 14, 16, 19, 29, 31, 45, 120]
```

5. Selection Sort :

Selection sort starts by finding the minimum value in a given list and move it to a sorted list. Then repeat the process for each of the remaining elements in the unsorted list. The next element entering the sorted list is compared with the existing elements and placed at its correct position. So at the end all the elements from the unsorted list are sorted.

Example

```
list = [19,12,31,45,16,14,120,29]
print("\nBefore Sorting :\n",list)
# Pass through all array elements
for i in range(len(list)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(list)):
        if list[min_idx] > list[j]:
            min_idx = j
    # Swap the found minimum element with
    # the first element
    list[i], list[min_idx] = list[min_idx], list[i]

print("\nAfter Sorting :\n",list)
```

Out Put

```
Before Sorting :
 [19, 12, 31, 45, 16, 14, 120, 29]
After Sorting :
 [12, 14, 16, 19, 29, 31, 45, 120]
```

➢ Hashtable in Python :

Hash tables are a type of data structure in which the address or the index value of the data element is generated from a hash function. That makes accessing the data faster as the index value behaves as a key for the data value. In other words Hash table stores key-value pairs but the key is generated through a hashing function. So the search and insertion function of a data element becomes much faster as the key values themselves become the index of the array which stores the data.

In Python, the Dictionary data types represent the implementation of hash tables.

The Keys in the dictionary satisfy the following requirements.

- The keys of the dictionary are hashable i.e. the are generated by hashing function which generates unique result for each unique value supplied to the hash function.
- The order of data elements in a dictionary is not fixed.