UNIT 2: OOP USING PYTHON

Compiled by: Kruti Kotak, Christ College, Rajkot

What is Exception?

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an exception

If you have some suspicious code that may raise an exception, you can defend your program by placing 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.

Syntax

Here is simple syntax of try....except...else blocks −

```
try:
   You do your operations here;
   ......................
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.
```

Here are few important points about the above-mentioned syntax −

- 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.

List of Standard Exceptions –

Sr.No.	Exception Name & Description
1	**Exception** Base class for all exceptions
2	**StopIteration** Raised when the next() method of an iterator does not point to any object.
3	**SystemExit** Raised by the sys.exit() function
4	**StandardError**

	Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError** Base class for all errors that occur for numeric calculation.
6	**OverflowError** Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError** Raised when a floating point calculation fails.
8	**ZeroDivisionError** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError** Raised in case of failure of the Assert statement.
10	**AttributeError** Raised in case of failure of attribute reference or assignment.
11	**EOFError** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError** Raised when an import statement fails.
13	**KeyboardInterrupt** Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	**LookupError** Base class for all lookup errors.
15	**IndexError** Raised when an index is not found in a sequence
16	**KeyError** Raised when the specified key is not found in the dictionary
17	**NameError** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError** Base class for all exceptions that occur outside the Python environment.
20	**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.
21	**IOError** Raised for operating system-related errors.
22	**SyntaxError** Raised when there is an error in Python syntax.
23	**IndentationError** Raised when indentation is not specified properly.

24	**SystemError** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	**SystemExit** Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	**TypeError** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError** Raised when a generated error does not fall into any category.
29	**NotImplementedError** Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Example

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all −

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

This produces the following result −

```
Written content in the file successfully
```

Example

This example tries to open a file where you do not have write permission, so it raises an exception −

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
```

This produces the following result −

```
Error: can't find file or read data
```

The except Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows −

```
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.
```

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The except Clause with Multiple Exceptions

You can also use the same except statement to handle multiple exceptions as follows −

```
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.
```

The try-finally Clause

You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this −

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

You cannot use else clause as well along with a finally clause.
Example

```
try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print "Going to close the file"
      fh.close()
except IOError:
   print "Error: can\'t find file or read data"
```

When an exception is thrown in the try block, the execution immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the exceptstatements if present in the next higher layer of the try-except statement.

Argument of an Exception

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows −

```
try:
   You do your operations here;
   ......................
except ExceptionType as e:
   You can print value of Argument here “e.args”...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.
This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.
Example
Following is an example for a single exception −

```
# Define a function here.
def temp_convert(var):
   try:
      return int(var)
   except ValueError as ve:
      print( "The argument does not contain numbers\n", ve.args)
```

```
    # Call above function here.
    temp_convert("xyz");
```

This produces the following result −

```
    The argument does not contain numbers
    invalid literal for int() with base 10: 'xyz'
```

example2:

```
    import sys
    def tmp_cnv(var):
       try:
          return int(var)
       except:
          print("error in converting\n",sys.exc_info())

    tmp_cnv("xy")
```

output:

```
    error in converting
     (<class 'ValueError'>, ValueError("invalid literal for int() with base 10: 'xy'"),
    <traceback object at 0x038D2E68>)
```

Example3:

```
    def tmp_cnv(var):
       try:
          return int(var)
       except Exception as e:
             print("error in converting:\n " + str(e))
             # print("error in converting:\n " + e.args)
tmp_cnv("xy")
```

Output:

```
    error in converting:
     ("invalid literal for int() with base 10: 'xy'",)
```

Example4:

```
    import traceback
    def tmp_cnv(var):
       try:
          return int(var)
       except:
          print(traceback.format_exc())
```

```
tmp_cnv("xy")
```

output:

```
Traceback (most recent call last):
  File "C:\Users\VAIBHAV\Desktop\e.py", line 4, in tmp_cnv
    return int(var)
ValueError: invalid literal for int() with base 10: 'xy'
```

Raising an Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the raise statement is as follows.

Syntax

```
raise [Exception [, args [, traceback]]]
```

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows −

```
def functionName( level ):
  if level < 1:
    raise "Invalid level!", level
    # The code below to this would not be executed
    # if we raise the exception
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows −

```
try:
  Business Logic here...
except "Invalid level!":
  Exception handling here...
else:
  Rest of the code here...
```

Python assert Statement

Python has built-in assert statement to use assertion condition in the program. assertstatement has a condition or expression which is supposed to be always true. If the condition is false assert halts the program and gives an AssertionError.

Syntax for using Assert in Pyhton:

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

In Python we can use assert statement in two ways as mentioned above.
assert statement has a condition and if the condition is not satisfied the program will stop and give AssertionError.
assert statement can also have a condition and a optional error message. If the condition is not satisfied assert stops the program and gives AssertionError along with the error message.
Let's take an example, where we have a function which will calculate the average of the values passed by the user and the value should not be an empty list. We will use assertstatement to check the parameter and if the length is of the passed list is zero, program halts.

Example 1: Using assert without Error Message

```
def avg(marks):
    assert len(marks) != 0
    return sum(marks)/len(marks)

mark1 = []
print("Average of mark1:",avg(mark1))
```

When we run the above program, the output will be:

```
AssertionError
```

We got an error as we passed an empty list mark1 to assert statement, the condition became false and assert stops the program and give AssertionError.
Now let's pass another list which will satisfy the assert condition and see what will be our output.

Example 2: Using assert with error message

```
def avg(marks):
    assert len(marks) != 0,"List is empty."
    return sum(marks)/len(marks)

mark2 = [55,88,78,90,79]
print("Average of mark2:",avg(mark2))

mark1 = []
print("Average of mark1:",avg(mark1))
```

When we run the above program, the output will be:

```
Average of mark2: 78.0
AssertionError: List is empty.
```

We passed a non-empty list mark2 and also an empty list mark1 to the avg() function and we got output for mark2 list but after that we got an error AssertionError: List is empty.
The assert condition was satisfied by the mark2 list and program to continue to run. However, mark1 doesn't satisfy the condition and gives an AssertionError.

Introduction to OOPs in Python

Python is a multi-paradigm programming language. Meaning, it supports different programming approach.
One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).
An object has two characteristics:

- attributes
- behavior

Let's take an example:
Parrot is an object,

- name, age, color are attributes
- singing, dancing are behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).
In Python, the concept of OOP follows some basic principles:

Inheritance	A process of using details from a new class without modifying existing class.
Encapsulation	Hiding the private details of a class from other objects.

Creating Classes

The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon as follows −

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

The class has a documentation string, which can be accessed via ClassName.__doc__.
The class_suite consists of all the component statements defining class members, data attributes and functions.

Example
Following is the example of a simple Python class −

```
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
```

- The variable empCount is a class variable whose value is shared among all instances of a this class. This can be accessed as Employee.empCount from inside the class or outside the class.
- The first method __init__() is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is self. Python adds the self argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts.

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows −

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Now, putting all the concepts together −

```
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

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

When the above code is executed, it produces the following result −

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time −

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions −

- The getattr(obj, name[, default]) − to access the attribute of object.
- The hasattr(obj,name) − to check if an attribute exists or not.
- The setattr(obj,name,value) − to set an attribute. If attribute does not exist, then it would be created.
- The delattr(obj, name) − to delete an attribute.

```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```

Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you 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 Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

When the above code is executed, it produces the following result −

```
Calling child constructor
Calling child method
Calling parent method
```

```
Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows −

```
class A:        # define your class A
.....

class B:        # define your class B
.....

class C(A, B):   # subclass of A and B
.....
```

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

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

Python - Searching Algorithms

Searching is a very basic necessity when you store data in different data structures. The simplest appraoch 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.

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.

```
def linear_search(values, search_for):
   search_at = 0
   search_res = False

# Match the value with each data element
   while search_at < len(values) and search_res is False:
      if values[search_at] == search_for:
         search_res = True
      else:
         search_at = search_at + 1
```

```
    return search_res

l = [64, 34, 25, 12, 22, 11, 90]
print(linear_search(l, 12))
print(linear_search(l, 91))
```

When the above code is executed, it produces the following result −

```
True
False
```

Binary search

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

```
def binary_search(item_list,item):
    first = 0
    last = len(item_list)-1
    found = False
    while( first<=last and not found):
        mid = (first + last)//2
        if item_list[mid] == item :
            found = True
        else:
            if item < item_list[mid]:
                last = mid – 1
            else:
                first = mid + 1
    return found
print(binary_search([1,2,3,5,8], 6))
print(binary_search([1,2,3,5,8], 5))
```

Python - Sorting Algorithms

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.

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.

```
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 = [19,2,31,45,6,11,121,27]
bubblesort(list)
print(list)
```

When the above code is executed, it produces the following result −

```
[2, 6, 11, 19, 27, 31, 45, 121]
```

Insertion Sort

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

```
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 = [19,2,31,45,30,11,121,27]
insertion_sort(list)
print(list)
```

When the above code is executed, it produces the following result −

```
[2, 11, 19, 27, 30, 31, 45, 121]
```

Merge Sort

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

```
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 = [64, 34, 25, 12, 22, 11, 90]
print(merge_sort(unsorted_list))
```

When the above code is executed, it produces the following result −

```
[11, 12, 22, 25, 34, 64, 90]
```

Shell Sort

Shell Sort involves sorting elements which are away from ech other. We sort a large sublist of a given list and go on reducing the size of the list until all elements are sorted. The below program finds the gap by equating it to half of the length of the list size and then starts sorting all elements in it. Then we keep resetting the gap until the entire list is sorted.

```
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 = [19,2,31,45,30,11,121,27]
shellSort(list)
print(list)
```

When the above code is executed, it produces the following result −

```
[2, 11, 19, 27, 30, 31, 45, 121]
```

Selection Sort

In selection sort we start by finding the minimum value in a given list and move it to a sorted list. Then we 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.

```
def selection_sort(input_list):
   for idx in range(len(input_list)):
      min_idx = idx
      for j in range( idx +1, len(input_list)):
         if input_list[min_idx] > input_list[j]:
            min_idx = j
# Swap the minimum value with the compared value
      input_list[idx], input_list[min_idx] = input_list[min_idx], input_list[idx]
l = [19,2,31,45,30,11,121,27]
selection_sort(l)
print(l)
```

When the above code is executed, it produces the following result −

```
[2, 11, 19, 27, 30, 31, 45, 121]
```

Python - Hash Table

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.

So we see the implementation of hash table by using the dictionary data types as below.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

```
# Declare a dictionary
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

# Accessing the dictionary with its key
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

When the above code is executed, it produces the following result −

```
dict['Name']:  Zara
dict['Age']:  7
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example −

```
# Declare a dictionary
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry
print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

When the above code is executed, it produces the following result −

```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation. To explicitly remove an entire dictionary, just use the del statement. −

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
dict.clear();     # remove all entries in dict
del dict ;        # delete entire dictionary
print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

This produces the following result. Note that an exception is raised because after del dict dictionary does not exist any more −

```
dict['Age']:
```

```
Traceback (most recent call last):
  File "test.py", line 8, in
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```