Python Notes - Week 3

Date:- 03/03/2022

**Errors and Exceptions**- In a program, errors are those issues due to which the execution of the program stops whereas an exception occurs during the execution/runtime of a program which disrupts the normal flow of the program. There are two types of errors which can occur:

a) **Syntax errors**: Example-

| age = 15   if(age >= 18)  print("You are eligible to vote!") # Invalid Syntax Error will be thrown due to the missing colon above |
| --- |

b) **Exceptions (Logical errors)**: Example-

| num\_val = 561   result = num\_val / 0 # ZeroDivisionError gets thrown as we are trying to divide a number by 0. print(result) |
| --- |

-> For all built-in exceptions, we have a base class i.e. "**BaseException**" which is not meant to be directly inherited by user-defined classes. For user-defined classes, 'Exception' is used. This class is responsible for creating a string representation of the exception using str() using the arguments passed. An empty string is returned if there are no arguments. All user-defined exceptions should also be derived from this class.

-> **ArithmeticError**: This class is the base class for those built-in exceptions that are raised for various arithmetic errors such as:

1) OverflowError

2) ZeroDivisionError

3) FloatingPointError

Some common built-in exceptions are:-

-> **ZeroDivisionError** - When a number is divided by 0.

-> **IndexError**- When the wrong index of a list is retrieved.

-> **AssertionError**- It occurs when the assert statement fails

-> **AttributeError**- It occurs when an attribute assignment is failed.

-> **ImportError**- It occurs when an imported module is not found.

-> **KeyError**- It occurs when the key of the dictionary is not found.

-> **NameError**- It occurs when the variable is not defined.

-> **MemoryError**- It occurs when a program runs out of memory.

**Handling Exceptions using Try/Except/Finally**:- We can handle errors using Try/Except/Finally.

-> '**Try**' block consists of the code in which an exception might occur. A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed.

-> Code for handling the exception exists in the '**Except**' block. An except clause may contain multiple exceptions at the same time enclosed in a parenthesis. For example:

| except (RuntimeError, TypeError, NameError):  pass |
| --- |

-> '**Finally**' block consists of that code which will always get executed whether exception is handled or not.

Example:-

| try:   print(56 / 0) # code in which exception occurs   except ZeroDivisionError: # block for handling the exception raised  print("Division by 0 not possible.")   finally:  print("I'm always executed!") |
| --- |

-> The try-except statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.

**Raising Exceptions**- When we want to code for the limitation of certain conditions then we can raise an exception. It allows the programmer to force a specified exception to occur when some condition is not being fulfilled. Consider the following example:-

| try:  amount = 500  if amount < 2500:  raise ValueError("Insufficient Amount. Please add more funds!")  else:  print("Your purchase was successful!")   except ValueError as e:  print(e) |
| --- |

**User-Defined Exceptions**:- We can create our own exceptions by creating a new exception class which is derived from the main 'Exception' class. Consider the following example-

| class testError(Exception):  def \_\_init\_\_(self, value): # Constructor method for initializing the variable  self.value = value    def \_\_str\_\_(self): # \_\_str\_\_ is to print() the value  return(repr(self.value))   try:  raise(testError(11))   except MyError as error: # Value of Exception is stored in error  print('Test Exception Occured: ', error.value) |
| --- |

**Classes**- A class is a collection of objects or a blueprint of objects defining the common attributes and behavior. Well, it logically groups the data in such a way that code reusability becomes easy. A class is defined using a “Class” Keyword. The attributes are data members (class variables and instance variables) and methods which are accessed via dot notation.

**Scope of Variables**:- The part of the program where a variable is accessible is called its scope. There are four major types of variable scope and is the basis for the LEGB rule. LEGB stands for Local -> Enclosing -> Global -> Built-in.

1) Local Scope- A variable defined inside a function, its scope lies ONLY within that particular function. It is accessible from the point at which it is defined until the end of the function and exists for as long as the function is executing. Example:

| def print\_number():  num\_val = 1  print("Number defined is: ", num\_val)  print\_number() # This statement will throw a NameError: name 'num\_val' is not defined print("Number defined is: ", num\_val) |
| --- |

2) Enclosing Scope- This kind of scope comes into play when we have a nested function. In this, variables defined in the outer function can easily be accessed by the inner function but the outer function cannot directly access variables of the inner function as their scope is limited to that inner function. Consider the example below:-

| def outer():  first\_num = 1  def inner():  second\_num = 2  print("first\_num from outer: ", first\_num)  print("second\_num from inner: ", second\_num)    inner()  print("second\_num from inner: ", second\_num) # This statement will throw a NameError: name 'num\_val' is not defined  outer() |
| --- |

3) Global Scope- Whenever a variable is defined outside a function, it is said to be a global variable, and its scope is throughout the program. Example:-

| str\_1 = "Hello"  def greeting\_msg():  str\_1 = "World"  print(str\_1, str\_2 # Hello World)  greeting\_msg() |
| --- |

4) Built-in Scope- All the special reserved keywords fall under this scope. We can call the keywords anywhere within our program without having to define them before use.

**Practice Programs**

Q.1 Python program that accepts a hyphen-separated sequence of words as input and prints the words in a hyphen-separated sequence after sorting them alphabetically.

Sol:-

| def sort\_words(str\_val):  sorted\_seq = []  for val in sorted(str\_val.split("-")):  sorted\_seq.append(val)  print('-'.join(sorted\_seq))   str\_val = "Violet-Blue-Orange-Pink" sort\_words(str\_val) |
| --- |

Output:

| Blue-Orange-Pink-Violet |
| --- |

Q.2 Python program to remove specific words from a given list using lambda.

Sol:-

| list\_items = []  for i in range(5):  list\_items.append(input("Enter any element: "))  words\_remove = []  for i in range(2):  words\_remove.append(input("Enter element you want to remove: "))  final\_list = list(filter(lambda word: word not in words\_remove, list\_items)) print(final\_list) |
| --- |

Output:

| Enter any element: Dog Enter any element: Cat Enter any element: Rat Enter any element: Lion Enter any element: Snake Enter element you want to remove: Rat Enter element you want to remove: Snake ['Dog', 'Cat', 'Lion'] |
| --- |

Q.3 Python class to find a pair of elements (indices of the two numbers) from a given array whose sum equals a specific target number.

Sol:-

| class elementsPair:  def find\_pairs(self, list\_items):  pairs = list()  global target\_sum # Using global variable inside a function with 'global' keyword  target\_sum = 70   for i in range(len(list\_items)):  for j in range(i + 1, len(list\_items)):  if list\_items[i] + list\_items[j] == target\_sum:  pairs.append((list\_items[i], list\_items[j]))   print(list(set(pairs)))   list\_items = [10, 20, 10, 40, 50, 60, 70] target\_sum = 60 # global variable not passed as a parameter inside the function  ep = elementsPair() ep.find\_pairs(list\_items) |
| --- |

Output:

| [(10, 60), (20, 50)] |
| --- |

Q.4 Python program to sort a list of tuples and dictionaries using Lambda

Sol:-

| roll\_list = [('Jack', 76), ('Emma', 78), ('Adam', 77), ('Simon', 79)] mobile\_list = [{'make': 'Nokia', 'model': '216', 'color': 'Black'},  {'make': 'Samsung', 'model': '7', 'color': 'Blue'},  {'make': 'Mi Max', 'model': '2', 'color': 'Gold'}]  roll\_list.sort(key=lambda a: a[1]) mobile\_list.sort(key=lambda val: val['color'], reverse=True)  print(roll\_list) print(mobile\_list) |
| --- |

Output:

| [(10, 60), (20, 50)] |
| --- |

Q.5 Python class which has two methods get\_String and print\_String. get\_String accept a string from the user and print\_String print the string in upper case.

Sol:-

| class strInput():  str\_1 = ""   def get\_string(self):  self.str\_1 = input("Enter a string: ")   def print\_string(self):  print(self.str\_1.upper())   str\_1 = strInput() str\_1.get\_string() str\_1.print\_string() |
| --- |

Output:

| Enter a string: Hello worlD HELLO WORLD |
| --- |

Q.6 Python function that checks whether a passed string is palindrome or not.

Sol:-

| class palindromeString:  def check\_string(self, str\_val):  if str\_val.lower() == str\_val.lower()[::-1]:  print("Palindrome String!")  else:  print("Not a Palindrome String!")   str\_val = "Radar" ps = palindromeString() ps.check\_string(str\_val) |
| --- |

Output:

| Palindrome String! |
| --- |

Q.7 Python program to find a list of integers with exactly two occurrences of nineteen and at least three occurrences of five.

Sol:-

| list\_items = [19, 19, 15, 5, 3, 5, 5, 2]  occur\_times\_19 = list\_items.count(19) occur\_times\_5 = list\_items.count(5)  if occur\_times\_19 == 2 and occur\_times\_5 >= 3:  print("Condition Satisfied") else:  print("Condition Not Satisfied") |
| --- |

Output:

| Condition Satisfied |
| --- |