SYLLABUS
PROGRAMMING IN PYTHON
(CIE-332T)
Applicable from the Academic Session 2023-24

Discipline(s) / EAE / OAE	Semester	Group	Sub-group	Paper Code
CSE/IT/CST/ITE	6	PCE	PCE-2	CIE-332T
CSE-IoT/CSE-ICB	6	PC	PC	IOT-320T
EAE	6	IOT-EAE	IOT-EAE-2B	IOT-330T
EAE	6	ICB-EAE	ICB-EAE-2B	IOT-330T

Marking Scheme:

1. Teachers Continuous Evaluation: 25 marks
2. Term end Theory Examinations: 75 marks

Instructions for paper setter:

1. There should be 9 questions in the term end examinations question paper.
2. The first (Ist) question should be compulsory and cover the entire syllabus. This question should be objective, single line answers or short answer type question of total 15 marks.
3. Apart from question 1 which is compulsory, rest of the paper shall consist of 4 units as per the syllabus. Every unit shall have two questions covering the corresponding unit of the syllabus. However, the student shall be asked to attempt only one of the two questions in the unit. Individual questions may contain upto 5 sub-parts / sub-questions. Each Unit shall have a marks weightage of 15.

UNIT I

Introduction, Python Basics: Entering Expressions into the Interactive Shell, The Integer, Floating-Point, and String Data Types, String Concatenation and Replication, Storing Values in Variables, Your First Program, Dissecting Your Program. Flow control: Boolean Values, Comparison Operators, Boolean Operators, Mixing Boolean and Comparison Operators, Elements of Flow Control, Program Execution, Flow Control Statements, Importing Modules, Ending a Program Early with sys.exit().

UNIT II

Functions: def Statements with Parameters, Return Values and return Statements, The None Value, Keyword Arguments and print(), Local and Global Scope, The global Statement, Exception Handling. Lists: The List Data Type, Working with Lists, Augmented Assignment Operators, Methods. Dictionaries and Structuring Data: The Dictionary Data Type, Pretty Printing, Using Data Structures to Model Real-World Things. Manipulating Strings - Working with Strings, Useful String Methods.

UNIT III

Reading and Writing Files: Files and File Paths, The os.path Module, The File Reading/ Writing Process, Saving Variables with the shelve Module, Saving Variables with the

pprint.pformat() Function. Organizing Files: The shutil Module, Walking a Directory Tree, Compressing Files with the zipfile Module.

UNIT IV

Web Scraping: Project: MAPIT.PY with the web browser Module, Downloading Files from the Web with the requests Module, Saving Downloaded Files to the Hard Drive, HTML.

From Academic Sessions 2023-24 Onwards

SIXTH SEMESTER [B.TECH]

PROGRAMMING IN PYTHON [CIE-332T]

UNIT - I

Q.1. What are the key features of Python?

Ans. (a) Python is an **interpreted** language. That means that, unlike languages like C and its variants, Python does not need to be compiled before it is run. Other interpreted languages include PHP and Ruby.

(b) Python is **dynamically typed**, this means that you don't need to state the types of variables when you declare them or anything like that. You can do things like $x = 111$ and then x = "I'm a string" without error

(c) Python is well suited to **object orientated programming** in that it allows the definition of classes along with composition and inheritance. Python does not have access specifiers (like C++'s public, private).

(d) In Python, **functions** are **first-class objects**. This means that they can be assigned to variables, returned from other functions and passed into functions. Classes are also first class objects.

(e) Writing Python code is quick but running it is often slower than compiled languages. Fortunately, Python allows the inclusion of C-based extensions so bottlenecks can be optimized away and often are. The numpy package is a good example of this, it's really quite quick because a lot of the number-crunching it does isn't actually done by Python.

(f) Python finds **use in many spheres:** web applications, automation, scientific modeling, big data applications and many more. It's also often used as "glue" code to get other languages and components to play nice.

Q.2. What type of language is python? Programming or scripting?

Ans. Python is capable of scripting, but in general sense, it is considered as a general-purpose programming language.

Q.3. Python an interpreted language, Explain.

Ans. An interpreted language is any programming language which is not in machine-level code before runtime. Therefore, Python is an interpreted language.

Q.4. What is pep 8?

Ans. PEP stands for **Python Enhancement Proposal**. It is a set of rules that specify how to format Python code for maximum readability.

Q.5.What are the benefits of using Python?

Ans. The benefits of using python are:

1. Easy to use: Python is a high-level programming language that is easy to use, read, write and learn.

2. Interpreted language: Since python is interpreted language, it executes the code line by line and stops if an error occurs in any line.

3. Dynamically typed: the developer does not assign data types to variables at the time of coding. It automatically gets assigned during execution.

4. Free and open-source: Python is free to use and distribute. It is open source.

5. Extensive support for libraries: Python has vast libraries that contain almost any function needed. It also further provides the facility to import other packages using

Python Package Manager(pip).

6. Portable: Python programs can run on any platform without requiring any change.

7. The data structures used in python are user friendly.

8. It provides more functionality with less coding.

Q.6. What are different types of Data Types?

Ans. int: Integer which stores only integer values ex: $x = 1$

float: Float stores decimal points values along with integer values ex: $x = 1.1$

Strings: Stores string type values ex: x = "hello"

Boolean: Store True or False ex: x = True

List: Is an array which is mutable and can store heterogeneous values. Ex: [1,2,3,4,5]

Tuple: Is an array which is immutable and can store heterogeneous values. Ex: (1,2,3,4,5)

Dictionary: Is map type of data type which stores key and value type of variable. The keys cannot be mutable objects like lists. Ex: {1:6, 2:3, 4:5}

Q.7. What are Literals in Python and explain about different Literals.

Ans. A literal in python source code represents a fixed value for primitive data types. There are 5 types of literals in python-

1. String literals: A string literal is created by assigning some text enclosed in single or double quotes to a variable. To create multiline literals, assign the multiline text enclosed in triple quotes. Eg.name="Tanya"

2. A character literal: It is created by assigning a single character enclosed in double quotes. Eg. a = 't'

3. Numeric literals include numeric values that can be either integer, floating point value, or a complex number. Eg. a = 50

4. Boolean literals: These can be 2 values- either True or False.

5. Literal Collections: These are of 4 types-

(a) List collections-Eg. a = [1,2,3,'Amit']

(b) Tuple literals- Eg. a = (5,6,7,8)

(c) Dictionary literals- Eg. dict = {1: 'apple', 2: 'mango, 3: 'banana''}

(d) Set literals- Eg. {"Tanya", "Rohit", "Mohan"}

6. Special literal: Python has 1 special literal None which is used to return a null variable.

Q.8 . How is memory managed in Python?

Ans. Memory is managed in Python in the following ways:

1. Memory management in python is managed by **Python private heap space.** All Python objects and data structures are located in a private heap. The programmer does not have access to this private heap. The python interpreter takes care of this instead.

2. The allocation of heap space for Python objects is done by Python's memory manager. The core API gives access to some tools for the programmer to code.

3. Python also has an inbuilt garbage collector, which recycles all the unused memory and so that it can be made available to the heap space.

Q.9. What is namespace in Python?

Ans. A namespace is a naming system used to make sure that names are unique to avoid naming conflicts.

Q.10. What is PYTHONPATH?

Ans. It is an environment variable which is used when a module is imported. Whenever a module is imported, PYTHONPATH is also looked up to check for the presence of the imported modules in various directories. The interpreter uses it to determine which module to load.

Q.11. What are python modules? Name some commonly used built-in modules in Python?

Ans. Python modules are files containing Python code. This code can either be functions classes or variables. A Python module is a .py file containing executable code. Some of the commonly used built-in modules are:

- Os
- Sys
- Math
- Random
- Data time
- JSON

Q.12.What are local variables and global variables in Python?

Ans. Global Variables:

Variables declared outside a function or in global space are called global variables. These variables can be accessed by any function in the program.

Local Variables:

Any variable declared inside a function is known as a local variable. This variable is present in the local space and not in the global space.

Example:

```
1. a = 2
2. def add():
3. b = 3
4. c = a + b
5. print(c)
6. add()
```

Output: 5

When you try to access the local variable outside the function add(), it will throw an error.

Q.13. Is python case sensitive?

Ans. Yes. Python is a case sensitive language.

Q.14.What is type conversion in Python?

Ans. Type conversion refers to the conversion of one data type into another.

int() – converts any data type into integer type

float() – converts any data type into float type

ord() – converts characters into integer

hex() – converts integers to hexadecimal

oct() – converts integer to octal

tuple() – This function is used to convert to a tuple.

set() – This function returns the type after converting to set.

list() – This function is used to convert any data type to a list type.

dict() – This function is used to convert a tuple of order (key, value) into a dictionary.

str() – Used to convert integer into a string.

complex(real,imag) – This function converts real numbers to complex(real, imag) number.

Q.15. How to install Python on Windows and set path variable?

Ans. To install Python on Windows, follow the below steps:

- Install python from this link: https://www.python.org/downloads/
- After this, install it on your PC. Look for the location where PYTHON has been installed on your PC using the following command on your command prompt: cmd python.
- Then go to advanced system settings and add a new variable and name it as PYTHON_NAME and paste the copied path.
- Look for the path variable, select its value and select 'edit'.
- Add a semicolon towards the end of the value if it's not present and then type %PYTHON_HOME%

Q.16. Is indentation required in python?

Ans. Indentation is necessary for Python. It specifies a block of code. All code within loops, classes, functions, etc is specified within an indented block. It is usually done using four space characters. If your code is not indented necessarily, it will not execute accurately and will throw errors as well.

Q.17 How to concatenate 2 strings?

Ans. Concatenating 2 string can be done using the addition operator "+" ex:

x = "hello"

y = " world"

result = x + y

Q.18 What will be the result of the following code:

Code:

```
print("Five" * "Three")
```

(a) FiveThree

(b) Error

(c) ThreeThreeThree

(d) FiveFiveFive

Ans. Error: Because string cannot be multiplied by a string, it can only be multiplied with an int

Q.19 What will be the result of the following code:

Code:

```
print("Five" * 2.0)
```

(a) Error

(f) **Five Five**
(c) **Five Five Five Five**
(d) **Five 2.0**

Ans. Error: Because string cannot be multiplied by a float, it can only be multiplied with an int

Q.20. How to fix this code so that the answer comes out as 2 + 1 = 3

Code:

ans = 3

print("2 + 1 = " + ans)

Ans. Possible answers all of them are correct:

print(f"2 + 1 = {ans}")

print("2 + 1 = {}".format(ans))

print("2 + 1 = " + str(ans))

Q.21 What is the difference between Python Arrays and lists?

Ans. Arrays and lists, in Python, have the same way of storing data. But, arrays can hold only a single data type elements whereas lists can hold any data type elements.

Example:

```
1. import array as arr
2. My_Array=arr.array('i',[1,2,3,4])
3. My_list=[1,'abc',1.20]
4. print(My_Array)
5. print(My_list)
```

Output: array('i', [1, 2, 3, 4]) [1, 'abc', 1.2]

Q.22 . What are functions in Python?

Ans. A function is a block of code which is executed only when it is called. To define a Python function, the **def** keyword is used.

Example:

```
1. def Newfunc():
2. print("Hi, Welcome to Edureka")
3. Newfunc(); #calling the function
```

Output: Hi, Welcome to Edureka

Q. 23. What is __init__?

Ans. __init__ is a method or constructor in Python. This method is automatically called to allocate memory when a new object/ instance of a class is created. All classes have the __init__ method.

Here is an example of how to use it.

```
1. class Employee:
2. def__init__(self, name, age,salary):
3. self.name = name
4. self.age = age
5. self. salary = 20000
6. E1 = Employee("XYZ", 23, 20000)
```

```
7.# E1 is the instance of class Employee.
8. #__init__ allocates memory for E1.
9. print(E1.name)
10. print(E1.age)
11. print(E1.salary)
```

Output:

XYZ

23

20000

Q.24 .What is a lambda function?

Ans. An anonymous function is known as a lambda function. This function can have any number of parameters but, can have just one statement.

Example:

```
1. a = lambda x, y : x + y
2. print(a(5, 6))
```

Output: 11

Q.25. What is self in Python?

Ans. Self is an instance or an object of a class. In Python, this is explicitly included as the first parameter. However, this is not the case in Java where it's optional. It helps to differentiate between the methods and attributes of a class with local variables.

The self variable in the init method refers to the newly created object while in other methods, it refers to the object whose method was called.

Q.26. How does break, continue and pass work?

Ans.

Break	Allows loop termination when some condition is met and the control is transferred to the next statement.
Continue	Allows skipping some part of a loop when some specific condition is met and the control is transferred to the beginning of the loop
Pass	Used when you need some block of code syntactically, but you want to skip its execution. This is basically a null operation. Nothing happens when this is executed.

Q.27. What does [::-1} do?

Ans. [::–1] is used to reverse the order of an array or a sequence.

For example:

```
1. importarray as arr
2. My_Array=arr.array('i',[1,2,3,4,5])
3. My_Array[::-1]
```

Output: array('i', [5, 4, 3, 2, 1])

[::-1] reprints a reversed copy of ordered data structures such as an array or a list the original array or list remains unchanged.

Q.28. How can you randomize the items of a list in place in Python?

Ans. Consider the example shown below:

```
1. from random import shuffle
```

2. x = ['Keep', 'The', 'Blue', 'Flag', 'Flying', 'High']

3. shuffle(x)

4. print(x)

The output of the following code is as below.

['Flying', 'Keep', 'Blue', 'High', 'The', 'Flag']

Q.34. What are python iterators?

Ans. Iterators are objects which can be traversed though or iterated upon.

Q.29. How can you generate random numbers in Python?

Ans. Random module is the standard module that is used to generate a random number. The method is defined as:

1. import random
2. random.random

The statement random.random() method return the floating-point number that is in the range of [0, 1). The function generates random float numbers. The methods that are used with the random class are the bound methods of the hidden instances. The instances of the Random can be done to show the multi-threading programs that creates a different instance of individual threads. The other random generators that are used in this are:

1. randrange (a, b): It chooses an integer and define the range in-between [a, b). It returns the elements by selecting it randomly from the range that is specified. It doesn't build a range object.

2. uniform(a, b): It chooses a floating point number that is defined in the range of [a,b).Iyt returns the floating point number

3. normalvariate(mean, sdev): It is used for the normal distribution where the mu is a mean and the sdev is a sigma that is used for standard deviation.

4. The Random class that is used and instantiated creates independent multiple random number generators.

Q.30. What is the difference between range & xrange?

Ans. For the most part, xrange and range are the exact same in terms of functionality. They both provide a way to generate a list of integers for you to use, however you please. The only difference is that range returns a Python list object and x range returns an xrange object.

This means that xrange doesn't actually generate a static list at run-time like range does. It creates the values as you need them with a special technique called yielding. This technique is used with a type of object known as generators. That means that if you have a really gigantic range you'd like to generate a list for, say one billion, xrange is the function to use.

This is especially true if you have a really memory sensitive system such as a cell phone that you are working with, as range will use as much memory as it can to create your array of integers, which can result in a Memory Error and crash your program. It's a memory hungry beast.

Q.31. How do you write comments in python?

Ans. Comments in Python start with a # character. However, alternatively at times, commenting is done using docstrings (strings enclosed within triple quotes).

Example:

1. <span data-mce-type="bookmark"style="display: inline-block; width: 0px; overflow: hidden; line-height: 0;"class="mce_SELRES_end">

2. <pre>#Comments in Python start like this

3. print("Comments in Python start with a #")

Output: Comments in Python start with a #

Q.32. How will you capitalize the first letter of string?

Ans. In Python, the capitalize() method capitalizes the first letter of a string. If the string already consists of a capital letter at the beginning, then, it returns the original string.

Q.33. How will you convert a string to all lowercase?

Ans. To convert a string to lowercase, lower() function can be used.

Example:

1. stg = 'ABCD'
2. print(stg.lower())

Output: abcd

Q.34. How to comment multiple lines in python?

Ans. Multi-line comments appear in more than one line. All the lines to be commented are to be prefixed by a #. You can also a very good shortcut method to comment multiple lines. All you need to do is hold the ctrl key and left click in every place wherever you want to include a # character and type a # just once. This will comment all the lines where you introduced your cursor.

Q.35.What are docstrings in Python?

Ans. Docstrings are not actually comments, but, they are documentation strings. These docstrings are within triple quotes. They are not assigned to any variable and therefore, at times, serve the purpose of comments as well.

Example:

1. """
2. Using docstring as a comment.
3. This code divides 2 numbers
4. """
5. x = 8
6. y = 4
7. z = x/y
8. print(z)

Output: 2.0

Q.35. What is the purpose of 'is', 'not' and 'in' operators?

Ans. Operators are special functions. They take one or more values and produce a corresponding result.

is: returns true when 2 operands are true (Example: "a" is 'a')

not: returns the inverse of the boolean value

in: checks if some element is present in some sequence

Q.36. What is the usage of help() and dir() function in Python?

Ans. Help() and dir() both functions are accessible from the Python interpreter and

used for viewing a consolidated dump of built-in functions.

1. Help() function: The help() function is used to display the documentation string and also facilitates you to see the help related to modules, keywords, attributes, etc.

2. Dir() function: The dir() function is used to display the defined symbols.

Q.37. Whenever Python exits, why isn't all the memory de-allocated?

Ans. 1. Whenever Python exits, especially those Python modules which are having circular references to other objects or the objects that are referenced from the global namespaces are not always de-allocated or freed.

2. It is impossible to de-allocate those portions of memory that are reserved by the C library.

3. On exit, because of having its own efficient clean up mechanism, Python would try to de-allocate/destroy every other object.

Q.38. What is a dictionary in Python?

Ans. The built-in datatypes in Python is called dictionary. It defines one-to-one relationship between keys and values. Dictionaries contain pair of keys and their corresponding values. Dictionaries are indexed by keys.

Let's take an example:

The following example contains some keys. Country, Capital & PM. Their corresponding values are India, Delhi and Modi respectively.

```
1 dict = {'Country':'India','Capital':'Delhi','PM':'Modi'}
1 print dict [Country]
```

Output: India

```
1. print dict[Capital]
```

Output: Delhi

```
1. print dict[PM]
```

Output: Modi

Q.39. How to use if, else and if elif else flow controls?

Ans. The usage of if else is like this:

```
If <condition>:
        # do something
else:
    # do something else
```

Example:

```
If x == 1:
        print ("x is 1")
    Else:
        print("x is not 1")
```

On similar lines if elif else works like this:

```
If <condition 1>:
        # do something
elIf<condition 2>:
        # do something
else:
```

```
# do something
```

Example

```
If x == 1:
        print ("x is 1")
If x == 2:
        print ("x is 2")
    Else:
        print("x is neither 1 nor 2")
```

Q.40. What are the six comparison operators?

Ans. (a) == Equal to

(b) ! = Not equal to

(c) < Less than

(d) > Greater than

(e) < = Less than or equal to

(f) > = Greater than or equal to.

Q.41. Consider you have a project Calculator inside which you have 2 directories, main and utils. If utils has 2 files utils1.py and utils2.py. If you wanted to import a function foo() from utils2.py into main.py which is inside the main folder, how would you do it?

Ans. There are 2 possible ways to do it.

1. From calculator.utils.utils2 import foo
2. From ..utils.utils2 import foo

Q.42. Create a function named "calculator", which accepts 3 variables, "func_type", "a" and "b". The possible options for func_type are "plus", "minus", "multiply" and "divide" ; you are allowed to ignore any other inputs that may come into func_type variables. The possible values into "a" and "b" are int and float types. Please provide appropriate error messages.

Ans.

```
def calculator(func_type, a, b):
 If func_type == "plus":
            return a + b
 eliffunc_type == "minus":
            return a - b
eliffunc_type == "multiply":
            return a * b
 eliffunc_type == "divide":
        return a / b if b != 0 else "cannot divide anything with Zero"
        # This can also be handled without the ternary operator.
        #  you can use if else as well.
else:
        return "this input is not supported"
        # They told us to ignore the other func_type values but
            # they also asked us to provide appropriate error messages
            # This is ambiguous, so it would be safer to add the message.
```

Q.43. What do the following expressions evaluate to?

(a) (5 > 1) and (13 != 20)

(b) not (51 > 44)

(c) (5 > 4) or (3 = = 5)

(d) not ((5 > 4) or (3 = = 5))

(e) (True and True) or (True = = False)

(f) (False) or (not True)

(g) 37 // 17

(h) 5 * 6 / 2.0 + 1

Ans. (a) (5 > 1) and (13 ! = 20) => True and True => True

(b) not (51 > 44) => not True => False

(c) (5 > 4) or (3 = = 5) => True and False => False

(d) not ((57> 8) or (3 != 5)) => not (True or True) => not True => False

(e) (True and True) or (True = = False) => True and False => False

(f) (False) or (not True) => False or False => False

(g) 37 // 17 =>floor(37/17) it can also be written as int(37/17) => 2

(h) 5 * 6 / 2.0 + 1 => use BODMAS => 5 * 3.0 + 1 => 15.0 + 1 => 16.0

Q.44. What is the difference between break and continue? Explain with detailed examples

Ans. break:

There is a shortcut to getting the program execution to break out of a loop's clause early. If the execution reaches a break statement, it immediately exits the oop's clause. In code, a break statement simply contains the break keyword.

```
For i in range(1000):
If i> 10:
Break
```

This loop will break when i gets the value greater than 10

Continue:

Like break statements, continue statements are used inside loops. When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition.

Example:

```
For i in range(5):
If i == 3:
        Continue
print(i)
```

The result will be 0,1,2,4. When i = = 3, the control goes back to the top of the loop and looks at the next value for i.

Q.45. Write a short program that prints the numbers 10 to 100 using a for loop. Then write an equivalent program that prints the numbers 10 to 100 using a while loop.

Ans. For *i* in range(10, 101):

print(i)

```
i = 10
While i != 101:
print(i)
i += 1
```

Q.46. What is sys.exit()? How is it used? How is it different from break?

Ans. Sys.exit() is used to exit the whole program meanwhile break is used to exit a loop such as for loop or while loop. It is possible to use sys.exit() anywhere in the program whereas break is limited to be inside loops. Writing break outside of a loop will throw error.

Example:

```
For i in range(10):
If i = = 5:
        sys.exit()
print(i, end=", ")
print('Program Ended')
```

Output of this program will be 1, 2, 3, 4, 5,

The program ends the moment sys.exit() is read.

```
For i in range(10):
If i = = 5:
        break
print(i, end=", ")
print('Program Ended')
```

Output of this program will be 1, 2, 3, 4, 5, 'Program Ended

Unlike sys.exit(), when the program reaches break, only the control flow exits the for loop and moves to the next executable statement which prints program ends.

UNIT - II

Q.1. What are keyword arguments?

Ans. In a usual function call you use positional arguments, that means the position in which the argument values are sent decides which value will be stored in which function parameter, for example:

Def foo(a, b, c, d):

do something

If we call this function like this:

foo(1 , 2 , 3 , 4)

Due to the positions 1 will be stored in a, 2 in b, 3 in c and 4 in d.

Now if we wanted to send the argument value of c and d first then b and then a, here keyword arguments helps us. Keyword arguments are created when you mention that the argument value is meant for a specific function parameter by mentioning the name of the function parameter before it, added bonus : when you use this the positions do not matter.

foo(c = 1 , d = 2 , a = 3 , b = 4) ☑

These are keyword arguments. A point to note is that positional argument **cannot come after** the keyword arguments it means something like this is not possible:

foo(a = 1 , b = 2 , 3 , 4) ×

But this will work:

foo(1 , 2 , d = 3 , c = 4) ☑

Also, mentioning 2 keyword arguments with the same keyword will throw an error:

foo(1 , 2 , c = 3 , d = 4, c = 5) ×

Also, giving keyword argument for a parameter for which positional argument is already given will result in error:

foo(1 , 2 , c = 3, d = 4 , b = 5) ×

(b) already has value 2 in it due to the positional argument used so giving a value 5 using a keyword argument throws error.

Keyword arguments are generally used for using default/optional parameters.

Q.2. What are positional arguments? How are they different from keyword arguments?

Ans. In a function call when we send in values without mentioning any keywords in front of them, these values are called positional arguments. The word positional is used because the position in which the arguments are sent decides which value will be stored in which function parameter. For example:

Def foo(a, b, c, d):

do something

If we call this function like this:

foo(1 , 2 , 3 , 4)

Due to the positions 1 will be stored in a, 2 in b, 3 in c and 4 in d.

The differences between keyword and positional arguments are:

(a) In using keyword arguments the position doesn't matter, in positional arguments it matters.

(b) Positional arguments are prone to errors in case the position of parameters

changes in the function definition, this doesn't affect the keyword arguments.

(c) Keyword arguments are prone to errors in case the name of the parameters changes in the function definition, this doesn't affect the positional arguments.

Q.3. What are default parameters and how are they different from position and keyword arguments?

Ans. Default parameters are also known as Default arguments so it is easy to confuse them with arguments which are used while calling the function but that is not the case, default parameters are the parameters which are optional and they are used in the function definition not in function calling. Default parameters allow users to not send the argument values for parameters c and d. Which means if the user doesn't send the values, the default values are used and if they send it then the user given values are used.

Example:

Def foo(a, b, c = 3, d = 4):

do something

Here the parameters c and d are default parameters. C has a default value 1 and d has 2.

Default parameters are used when you want to give an option to the user about the functionality of a function. Default parameters five an extra dimension to the function calling, example:

foo(1, 2) ☑

This type of calling is allowed due to the fact that the parameter c and d have default values.

foo(1, 2, 3) ☑

This is also allowed, where c is given value using a positional argument.

foo(1, 2, d = 4) ☑

This is also allowed, where c takes the default value and d takes the value via the keyword argument.

foo(c = 5, d = 6) ×

This is not allowed, since the positional arguments for a and b are missing.

The difference between Default parameters and keyword and positional arguments is the former is a type of parameter which is used while defining a function whereas the other 2 are 2 types of arguments which are used while calling the function.

Q.4. For this function definition. What will be the output of the function calls? If it is an error, mention why.

Code:

def foo(a, b, c, d = 1, e = 2, f = 3):

print("{}, {}, {}, {}, {}, {}".format(a, b, c, d, e, f))

(a) foo(1, 2, d=3, e=4, f = 6, c = 7)

(b) foo(a = 1, 2, 3)

(c) foo(1, 2)

(d) foo(1,2,3)

Ans.

(a) foo(1, 2, d = 3, e = 4, f = 6, c = 7) => 1, 2, 7, 3, 4, 6

(b) foo(a = 1, 2, 3) => Error: position arguments cannot come after keyword arguments

(c) foo(1, 2) => Error, position value for argument c is missing

(d) foo(1,2,3) => 1, 2, 3, 4, 5, 6

Q.5. How many arguments can you return using a return statement in a function? Does the function return anything if we do not use the return statement? What does the function return in case we write "return" without any variable in front of it?

Ans. We can return as many values we want from a function using the return statement using comma separated values like this:

```
def foo(a, b, c, d, e, f):
        return a, b, c, d, e, f
```

If we call this function and store the result in a single value that is possible, the value will have a tuple in it which stores these values, example:

```
ans = foo(1, 2, 3, 4, 5, 6)
print( ans)
```

This will print: (1, 2, 3, 4, 5, 6) which is a print of a tuple

If we write only return in a function the return value is None.

If we do not write a return statement in a function, the return will still be None.

Q.6. What is local and global scope?

Ans. A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call this function, the local variables will not remember the values stored in them from the last time the function was called.

```
def foo():
    # local scope started
    var1 = 0 # local variable
    return
    # local scope ends
```

There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran your program, the variables would remember their values from the last time you ran it.

```
# global scope started
def foo():
    # local scope started
    var1 = 0 # local variable
    return
    # local scope ends
foo()
print("foo executed")
# global scope ends
```

Q.7. What are local and global variables? Explain the things we need to be

careful about while using local and global variables.

Ans. Parameters and variables that are assigned in a called function are said to exist in that function's local scope. Variables that are assigned outside all functions are said to exist in the global scope. A variable that exists in a local scope is called a local variable, while a variable that exists in the global scope is called a global variable. When a scope is destroyed, all the values stored in the scope's variables are forgotten. Here are a few points to keep in point

Here are the things that we need to careful about when using local and global variables:

Local Variables Cannot Be Used in the Global Scope

```
# global scope START
Def foo():
# local scope START
        var1 = 31337
# local scope END
foo()
print(var1) × => Error: var1 is not defined
# global scope END
```

Local Scopes Cannot Use Variables in Other Local Scopes

```
# global scope START
Def foo1():
    # foo1 's local scope START
    var1 = 31337
    # foo1 's local scope END
Def foo2():
    # foo2 's local scope START
    var2 = 31337
    print(var1) × => Error: var1 is not defined
    # foo2 's local scope END
# global scope END
```

Global Variables Can Be Read from a Local Scope

```
# global scope START
Def foo():
    # foo 's local scope START
    print(var1)☑
    # foo 's local scope END
var1 = 200
foo()
print(var1)
# global scope END
```

In case Local and Global Variables with the Same Name, Local is preferred in local and global is preferred in global

```
# global scope START
```

```
Def foo():
        # foo 's local scope START
        print(var1)☑
        # foo 's local scope END
var1 = 200
foo()
print(var1)
# global scope END
```

Changing values of Global variables in local scopes using Global statement

```
# global scope START
Def foo():
        # foo 's local scope START
        global var1 ☑
        Var1 = 'inside local scope' ☑
        # foo 's local scope END
var1 = 'inside global scope'
print(var1)
foo()
print(var1)
# global scope END
```

Output:

inside global scope

inside local scope

The global variable can be changed only when the global statement is used inside the local scope.

Q.8. How do we handle exceptions in python? Give examples.

Ans. We can handle exceptions using the try exception block in python. The try exception block usually has the following type of syntax:

```
try:
        # do something
except Exception as e:
        # print or log the error
        # do something
```

Example:

```
Def foo(var):
try:
        return 10000 / var
except ZeroDivisionError as zde:
print("10000 cannot be divided by 0")
                return
```

Here in case var is ever sent as 0 we will get an error.

If we do not do this sort of exception handling, the whole program will crash when

it reaches the line which will throw an exception.

Q.9. Consider L = [3.14, 'cat', 11, 'cat', True]. What does L.index('cat') evaluate to?

Ans. The .index function in a list finds the value passed and returns its index if present and return -1 if the value is not present. In this case L = [3.14, 'cat', 11, 'cat', True] there are duplicates, in case of duplicates, it sends the index of first occurrence, hence the answer for L.index('cat') would be 1.

Q.10. What is the difference between the append() and insert() list methods?

Ans. The **append** method is a method of a list data type, it accepts a variable and it appends that variable at the end of the list and if the list is empty it assigns it as the first variable of the list. Example:

```
L = [1,2,3]
L.append(4)
print(L)
[1,2,3,4]
```

The **insert** variable is a method of a list data type, it accepts 2 positional arguments, 1st is the index at which the insertion should take place, the second argument accepted is the value that we want to insert into the list. The insert method also supports negative indexes as a first positional argument. Example:

```
L = [1,2,3]
L.insert(1, 50)
print(L)
[1,50,2,3]
L.insert(-1, 60)
print(L)
[1,50,2,3,60]
```

Q. 11. What are python collections?

Ans. There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.

```
mylist = ["apple", "banana", "cherry"]
print(type(mylist))
```

- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.

```
tup =(1, "a", "string", 1 + 2)
print(tup)
print(tup[1])
Output: (1, 'a', 'string', 3)
a
```

- **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.

```
# Python program to demonstrate working of
# Set in Python
# Creating two sets
```

```
set1 = set()
set2 = set()
# Adding elements to set1
for i in range(1, 6):
    set1.add(i)
# Adding elements to set2
foriinrange(3, 8):
    set2.add(i)
print("Set1 = ", set1)
print("Set2 = ", set2)
print("\n")
```

Output:

('Set1 = ', set([1, 2, 3, 4, 5]))

('Set2 = ', set([3, 4, 5, 6, 7]))

- **Dictionary** is a collection which is ordered** and changeable. No duplicate members.

dictionary = {"key name": value}

Q.12. What is the difference between a list and a tuple?

Ans.

List	Tuple
USes bold brackets [ ]	Uses round brackets ()
Is mutable, means values can be changed	Is immutable, means values cannot be changed.
Cannot be used as a key in a dictionary	Can be used as a key in a dictionary

Q.13. What is slicing? Use slicing to get the reverse of a list.

Ans. Slicing is a way given by python to cut a list/tuple from either end and create a new list/tuple. It also supports steps. The step defines what will be the difference between the indexes that are being picked, by default it is set to 1 which means all the indexes will be picked from the list, if we put the step value as 2, only the even indexes will be picked(that is if you are starting from the 0th index). Example:

```
L = [i for i in range(10)] # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Creating a new copy of this list
L_copy = L[ : ]
Slicing the last and the first value of L
L_2 = L[ 1 : -1 ] # [1, 2, 34, 5, 6, 7, 8]
Slicing the last and the first value and step of 2
L_3 = L[ 1: -1 : 2 ] # [1, 3, 5, 7]
Slicing to get reverse of a L
L_reverse = L[ : : -1] # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Q.14. Which of these statements is incorrect? Explain why. (Consider there is a tuple named T and a list named L and a dictionary called D, all of them have 10 items in them)

(a) T[3] = 3

(b) L.remove(1)

(c) T.remove(1)

(d) T.sort()

(e) D[T] = 4

(f) D[L] = 4

(g) T[: 2 : -1]

(h) T[0] + T[1]

Ans. (a) T[3] = 3 : Error: Tuples are immutable. Tuple doesn't support item assignment

(b) L.remove(1) : No error, it will remove the first occurrence of 1

(c) T.remove(1) : Error: Tuples are immutable. Tuple doesn't support item deletion. Tuple doesn't have a method named remove.

(d) T.sort() : Error: Tuples are immutable. Tuple doesn't support item rearrangement. Tuple doesn't have a method named sort.

(e) D[T] = 4 : This is correct, tuple can be a key of a dictionary

(f) D[L] = 4 : This is incorrect, lists are mutable, keys in a dictionary need to be immutable.

(g) T[: 2 : -1] : This is correct, this will result in a new tuple which will store values from tuple.

(h) T[0] + T[1] : Error: Tuples are immutable. Tuple doesn't support item assignment

Q.15. What are dictionaries?

Ans. Dictionaries are built in data type in python, Dictionary closely represents a map. It stores key value pairs. The retrieval of a value, checking if a key exists in a dictionary, and inserting or updating a key value pair is an O(1) operation. Because of its constant time operations it is used widely to optimize the code. It is defined using curly braces like dict_example = {}, the other way to initialize a dictionary is dict_Example = dict()

Q.16. Explain these functions in a dictionary, Keys(), values(), items(), get() and setdefault()

Ans. Keys()

This function of a dictionary gives back a list of keys in the dictionary.

values()

This function of a dictionary gives back a list of values in the dictionary.

items()

This function of a dictionary gives back a list of tuples, each tuple is of length 2, each tuple contains a key value pair present in the dictionary.

get()

This function of a dictionary is used to fetch the value of a key, if the key is not present, it just returns None. It also accepts another argument which if provided will be returned if the key is not present.

Example:

```
d = dict()
X = d.get(1)
```

```
print(X)
None
X = d.get(1, "NOT PRESENT")
print(X)
NOT PRESENT
```

setdefault()

This is a function of a dictionary which allows us to set value for a key if the key is not present and if the key is present then it does nothing. It accepts 2 variables, the first 1 is key and the other one is value. It returns the value of the key, if the key is not present it returns the value passed as the argument if the key is present then it returns the value of the key in the dictionary.

It basically is a shorter version of this piece of code:

```
d = dict()
def foo(key, value, d):
    if key' not in d:
    d[key] = value
return d[key]
```

Q.17. What is pretty printing? Why is it used? How to use it?

Ans. Pprint is a module which is a wrapper over the print function. It is tailor made for dictionary printing, if you print a dictionary in python it writes something like this:

```
D = { 1:2, 3:4, 5:[ 1,2,3,4], 6:{1:['a', 'b', 'c']}}
print(D)
{ 1:2, 3:4, 5:[ 1,2,3,4], 6:{1:['a', 'b', 'c']}}
```

Now this seems very confusing, whereas if we use pprint

```
From pprint import pprint
pprint(D)
{
1: 2,
3: 4,
5: [ 1, 2, 3, 4],
6: {
1: ['a', 'b', 'c']
}
}
```

Now in this version of pprint it is very easy to understand what is going on in the dictionary. This is very helpful when there are nested dictionaries.

Q.18. What are the 3 methods to remove whitespaces in a string?

Ans. There are 3 method that a string supports which can remove whitespaces:

1. str.lstip(): This removes the leading whitespaces in a string. Which means the whitespaces at the starting of the string.

2. str.rstrip(): This removes the trailing whitespaces in a string. Which means the whitespaces at the end of the string.

3. str.strip(): This function removes all the white spaces in a string.

Q. 19. How do we use upper and lower functions?

Ans. The upper function is used to make the string all uppercase and the lower function is called to make the string all lowercase. They are called like this:

str.upper()

str.lower()

The main use case is when a user enters things like AbCd where you are expecting abcd, here the lower or upper function is used, to make sure the user input is read without the case of the string being an issue.

Q.20. Explain these 5 is X functions:

(a) Isalpha

(b) Isdecimal

(c) Isalnum

(d) Isspace

(e) Istitle

Ans. (a) Isalpha: returns True if the string consists only of letters and is not blank.

(b) Isdecimal: returns True if the string consists only of numeric characters and is not blank.

(c) Isalnum: returns True if the string consists only of letters and numbers and is not blank.

(d) Isspace: returns True if the string consists only of spaces, tabs, and new-lines and is not blank.

(e) Istitle: returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

Q.21. What is the output of the following code:

(a) "AbcdE".isupper()

(b) "abcdef".islower()

(c) "ABCDE".isupper()

(d) "aBCDe".islower()

(e) "AbcdE".istitle()

(f) "ABCDE".istitle()

Ans. (a) "AbcdE".isupper(): False

(b) "abcdef".islower(): True

(c) "ABCDE".isupper(): True

(d) "aBCDe".islower(): False

(e) "AbcdE".istitle(): False

(f) "ABCDE".istitle(): False

Q.22. Write a function which takes in 3 arguments string A, string prefix, string postfix. You need to send true if the string A has the string prefix at the front of it and has the string postfix at the back of it, if not, then return False.

Ans.
```
Def foo(A, prefix, postfix):
        If A.starts with(prefix) and A.endswith(postfix):
                Return True
        Return False
```

Q.23. Explain the join and split functions of a string.

Ans. The join() method is useful when you have a list of strings that need to be joined together into a single string value. The join() method is called on a string, it gets a list of strings as an argument and returns a string. The returned string is the concatenation of each string in the passed-in list and each string is separated by the string on which join is called. For example:

'A'.join(['B', 'C', 'D'])

Result: BACAD

'A' is the string on which join is called.

['B', 'C', 'D'] is the argument sent into the join function.

Result is : BACAD

B, C and D are separated by A

The split()

method does the opposite: It's called on a string value, it takes a string as an argument and returns a list of Strings. If no argument is provided in the split function, the split function will consider whitespace characters such as the space, tab, or newline characters as the things on which it needs to split the string into a list. example

'A B\nC \tD '.split()

['A', 'B', 'C', 'D']

'MyABCnameABCisABCSimon'.split('ABC')

['My', 'name', 'is', 'Simon']

'My name is Simon'.split('m')

['My na', 'e is Si', 'on']

UNIT - III

Q.1. Write a code to create 2 folders Training and Validation in a directory BASE_DIR.

Ans. os.makedirs(os.path.join(BASE_DIR, 'Training'))

os.makedirs(os.path.join(BASE_DIR, 'Validation'))

Q.2. What is the difference between absolute and relative path

Ans. There are two ways to specify a file path.

An absolute path, which always begins with the root folder

A relative path, which is relative to the program's current working directory

Example:

Absolute path looks like /home/abcd/folder1/test.exe

Now consider a file /home/abcd/folder2/test2.py

If we want the relative path of 1st file from the test2.py it would be ..folder2//test.exe

Q.3. How to get a path of a file with respect to some other directory?

Ans. We can get the relative path of 1 path to another using :

os.path.relpath()

It takes 2 arguments, both are supposed to be valid paths in string format and this function will give you the relative of the 2nd path with respect to the 1st path.example:

os.path.relpath('C:\\Windows', 'C:\\')

'Windows'

os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')

'..\\..\\Windows'

Q.4. Write 2 ways to get the name of the file and its directory.

Ans. (a) Os.path.basename(path) and os.path.dirname(path) => the first function gets the basename and 2nd gets the dir.

(b) os.path.split(path) => This gives a list with 2 values. The first being the dir and 2nd being the base name.

Q.5. Write a function to get the total size of a folder BASE_DIR. BASE_DIR contains many folders, do not forget to add their sizes to the total_size as well. If the BASE_DIR does not exist, return 0..

Ans. Def foo(base_dir):

```
If not os.path.exists(base_dir): # checks if the base_dir exists or not
        return 0
total_size = 0
for folder_name, sub_folders, file_names in os.walk(base_dir):
        # walks the directory. It goes into subfolders directly.
        for file_name in file_names: # iterate through files in the directory
        total_Szie += os.path.getsize(os.path.join(folder_name, file_name))
        # gets the file path using o.path.join
        # gets the size using os.path.getsize
return total_size
```

Q.6. What is the reason to use os.path.join?

Ans. On Windows, paths are written using backslashes (\) as the separator between folder names. OS X and Linux, however, use the forward slash (/) as their path separator. If you want your programs to work on all operating systems, you will have to write your Python scripts to handle both cases. Fortunately, this is simple to do with the os.path.join() function. If you pass it the string values of individual file and folder names in your path, os.path.join() will return a string with a file path using the correct path separators. That is why it's important to use os.path.join to make your program valid for both the OS.

Q.7. How does one read the contents of a file and write things into it?

Ans. To open up a file we need to use the function open

f = open(<path_to_file>, 'r')

To read from the file we need to use the 2 available methods, read or readlines.

file_content = f.read()

lines = f.read_lines()

The difference between read and readlines is simple, read function dumps the whole content into the variable whereas readlines function sends back list of lines in the file content, it is similar to calling a split function on file_content with separator set to /n

To write into a file we need to open the file in write mode:

f = open(<path_to_file>, 'w')

And we can write anything in the file using write function:

f.write("ABCD")

The important point to note here is that the write function doesn't put the end of line character (/n) at the end of every entry to write function. We need to add that manually using f.write("/n").

Q.8. What are the different types of mode you can open your file with?

Ans.(a) 'r' (read): Opens the file for reading. The file pointer is placed at the beginning of the file.

(b) 'w' (write): Opens the file for writing. If the file already exists, it truncates the file to zero length. If the file does not exist, it creates a new file.

(c) 'a' (append): Opens the file for writing, but if the file already exists, it appends the new data to the end of the file.

(d) 'b' (binary mode): This can be added to any of the above modes, and it opens the file in binary mode. For example, 'rb' or 'wb'.

(e) 'x' (exclusive creation): Opens the file for exclusive creation. If the file already exists, the operation will fail.

(f) '+' (read and write): Opens the file for both reading and writing.

These are some of the basic file modes in Python. You can combine them as needed, for example, 'rb' for reading a binary file, or 'a+' for appending and reading.

Q.9. Write code for 2 different ways you can read lines from a file.

To read from the file we need to use the 2 available methods, read or readlines.

file_content = f.read()

lines = f.read_lines()

The difference between read and readlines is simple, read function dumps the whole content into the variable whereas readlines function sends back list of lines in the file content, it is similar to calling a split function on file_content with separator set to /n

Q.10. What is shelve? What are its uses, explain with an example.

Ans. You can save variables in your Python programs to binary shelf files using the shelve module. This way, your program can restore data to variables from the hard drive. The shelve module will let you add Save and Open features to your program. For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run. This is how we store the data into a shelf file

```
import shelve
shelfFile = shelve.open('mydata')
cats = ['Zophie', 'Pooka', 'Simon']
shelfFile['cats'] = cats
shelfFile.close()
```

After running the previous code on Windows, you will see three new files in the current working directory: mydata.bak, mydata.dat, and mydata.dir. On OS X, only a single mydata.db file will be created. These binary files contain the data you stored in your shelf.

The module frees you from worrying about how to store your program's data to a file.

```
shelfFile = shelve.open('mydata')
shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
shelfFile.close()
```

Here, we open the shelf files to check that our data was stored correctly. Entering shelfFile['cats'] returns the same list that we stored earlier, so we know that the list is correctly stored, and we call close().

Q.11. What is pformat? How is it different from shelve?

Ans. pprint.pformat() function accepts an argument which is usually a complex variable such as a dictionary or list of dictionaries and it returns a string which is not only formatted to be easy to read, but it is also syntactically correct Python code.

Say you have a dictionary stored in a variable and you want to save this variable and its contents for future use. Using pprint.pformat() will give you a string that you can write to .py file.

This file will be your very own module that you can import whenever you want to use the variable stored in it. For example:

```
import pprint
cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
fileObj = open('myCats.py', 'w')
fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
fileObj.close()
```

Here, We have a list of dictionaries, stored in a variable cats. We stored the output of pformat into a new py file mycats.py Now this mycats.py can be imported in any python file. Example:

```
import myCats
myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
myCats.cats[0]['name']
'Zophie'
```

The benefit of creating a .py file as opposed to saving variables with the shelve module is that because it is a text file, the contents of the file can be read and modified by anyone with a simple text editor.

For most applications, however, saving data using the shelve module is the preferred way to save variables to a file. Only basic data types such as integers, floats, strings, lists, and dictionaries can be written to a file as simple text. File objects, for example, cannot be encoded as text.

Q.12. What is the difference between shutil.copy and shutil.copytree?

Ans. While shutil.copy() will copy a single file, shutil.copytree() will copy an entire folder and every folder and file contained in it. Calling shutil.copytree(source, destination) will copy the folder at the path source, along with all of its files and subfolders, to the folder at the path destination. The source and destination parameters are both strings. The function returns a string of the path of the copied folder.

```
import shutil, os
os.chdir('C:\\')
shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
'C:\\bacon_backup'
```

The shutil.copytree() call creates a new folder named bacon_backup with the same content as the original bacon folder. You have now safely backed up your bacon folder.

Q.13. How would you rename a file?

Ans. Calling shutil.move(source, destination) will move the file or folder at the path source to the path destination and will return a string of the absolute path of the new location. If destination points to a folder, the source file gets moved into destination and keeps its current filename. Example:

```
import shutil
shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

The destination path can also specify a filename. In the following example, the source file is moved and renamed.

```
shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
'C:\\eggs\\new_bacon.txt'
```

This is how you rename a file.

Q.14. What is the difference between os.unlink, os.rmdir and shutil.rmtree?

Ans. • Calling os.unlink(path) will delete the file at path.

• Calling os.rmdir(path) will delete the folder at path. This folder must be empty of any files or folders.

• Calling shutil.rmtree(path) will remove the folder at path, and all files and folders it contains.

it contains will also be deleted.

Q.15. How to open a zip file and read its content without extracting the files?

Ans. To open a zip file read its content we need to use zipfile.ZipFile to open the file and to read the file we can use .namelist() on the opened zip file to get all the names of the files in it, to select a particular file we would use.getinfo() on the zip file and send the file name as an argument. Then we can read the contents of the file. The other way would be to extract everything from the zip file and read the contents.

Going with the first approach here is the code on how to do it. Let's suppose the zip file we need to open is named as example.zip and it is placed in a folder with path BASE_DIR

```
import zipfile, os
os.chdir(BASE_DIR) # move to the folder with example.zip
exampleZip = zipfile.ZipFile('example.zip') # opening the zip file in memory
exampleZip.namelist() # gets the file and folder names in the zip file
```

['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg'] => output of the namelist function.

```
# assuming we want to read the spam.txt
with exampleZip.open('spam.txt') as spamFile:
    content = spamFile.read()
# do something with the content.
exampleZip.close()
```

Q.16. Write a code to extract the contents of a zip file in a folder.

Ans. To extract the content of a zip file we need to use extractall() method on an opened zip file. We can also send a path to this function and it will extract all the content into that path.

Example:

```
import zipfile, os
os.chdir(base_dir) # move to the folder with example.zip
exampleZip = zipfile.ZipFile('example.zip') # open the zip file
exampleZip.extractall() # extract the zip file
exampleZip.close() # close the zip file
```

Q.17. How to get the compressed and actual size of a file in a zip file.

Ans. To get the actual size and the compressed sizes we need to use getinfo() to select a file and on that file we need to call it's attributes "file_size" to get the actual size and "compress_size" to get the compressed file size.

Example:

```
import zipfile, os
os.chdir('C:\\') # move to the folder with example.zip
exampleZip = zipfile.ZipFile('example.zip')
spamInfo = exampleZip.getinfo(abcd.txt') # selects one particular file in the zip file
spamInfo.file_size 13908 # gets the original size of the file in the zip file
spamInfo.compress_size 3828 # gets the compressed size of the file in zip file
exampleZip.close()
```

Q.18. Write a code to create a zip file in a folder.

Ans. To create a zip file we need to open a zipfile in write mode and then call the write function with compress type set to zip_deflated. Example:

```
import zipfile
newZip = zipfile.ZipFile('new.zip', 'w') # open new file with write mode
newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED) # write files into the zip file
newZip.close() # close the file
```

UNIT-IV

Q.1. What is web scraping?

Ans. Web scraping is an automatic method to obtain large amounts of data from websites. Most of this data is unstructured data in an HTML format which is then converted into structured data in a spreadsheet or a database so that it can be used in various applications. There are many different ways to perform web scraping to obtain data from websites. These include using online services, particular API's or even creating your code for web scraping from scratch. Many large websites, like Google, Twitter, Facebook, StackOverflow, etc. have API's that allow you to access their data in a structured format. This is the best option, but there are other sites that don't allow users to access large amounts of data in a structured form or they are simply not that technologically advanced. In that situation, it's best to use Web Scraping to scrape the website for data.

Web scraping requires two parts, namely the **crawler** and the **scraper**. The crawler is an artificial intelligence algorithm that browses the web to search for the particular data required by following the links across the internet. The scraper, on the other hand, is a specific tool created to extract data from the website. The design of the scraper can vary greatly according to the complexity and scope of the project so that it can quickly and accurately extract the data.

Q.2 How Web Scrapers Work?

Ans. Web Scrapers can extract all the data on particular sites or the specific data that a user wants. Ideally, it's best if you specify the data you want so that the web scraper only extracts that data quickly. For example, you might want to scrape an Amazon page for the types of juicers available, but you might only want the data about the models of different juicers and not the customer reviews.

So, when a web scraper needs to scrape a site, first the URLs are provided. Then it loads all the HTML code for those sites and a more advanced scraper might even extract all the CSS and Javascript elements as well. Then the scraper obtains the required data from this HTML code and outputs this data in the format specified by the user. Mostly, this is in the form of an Excel spreadsheet or a CSV file, but the data can also be saved in other formats, such as a JSON file.

Q.3 Different Types of Web Scrapers.

Ans. Web Scrapers can be divided on the basis of many different criteria, including Self-built or Pre-built Web Scrapers, Browser extension or Software Web Scrapers, and Cloud or Local Web Scrapers.

You can have **Self-built Web Scrapers** but that requires advanced knowledge of programming. And if you want more features in your Web Scraper, then you need even more knowledge. On the other hand, pre-built **Web Scrapers** are previously created scrapers that you can download and run easily. These also have more advanced options that you can customize.

Browser extensions Web Scrapers are extensions that can be added to your browser. These are easy to run as they are integrated with your browser, but at the same time, they are also limited because of this. Any advanced features that are outside the scope of your browser are impossible to run on Browser extension Web Scrapers.

But **Software Web Scrapers** don't have these limitations as they can be downloaded and installed on your computer. These are more complex than Browser web scrapers, but they also have advanced features that are not limited by the scope of your browser.

Local Web Scrapers – on the other hand, which is usually provided by the company that sells the scraper tool. These allow your computer to function as the crawler, the computer to manipulate and organize the scraped data from websites.

Cloud Web Scrapers – on the other hand, runs on your computer using local resources. Since the Web scraper requires many CPU or RAM, their own computer will become slow and not be able to perform other tasks.

Q 4 Why is Python a popular programming language for Web Scraping?

Ans – Python seems to be in fashion these days. It is the most popular language for web scraping as it can handle most of the processes easily. It also has a variety of libraries that were created specifically for Web Scraping. **Scrapy** is a very popular open source web crawling framework that is written in Python. It is ideal for web scraping as well as extracting data using APIs. **Beautiful soup** is another Python library that is highly suitable for Web Scraping. It creates a parse tree that can be used to extract data from HTML on a website. Beautiful soup also has multiple features for navigation, searching, and modifying these parse trees.

Q 5 What is Web Scraping used for?

Ans – Web Scraping has multiple applications across various industries. Let's look at some of these now.

1. Price Monitoring

Web Scraping can be used by companies to scrape the product data for their products and competing products as well as see how it impacts their pricing strategies. Companies can use this data to fix the optimal pricing for their products so that they can obtain maximum revenue.

2. Market Research

Web scraping can be used for market research by companies. High quality web scraped data obtained in large volumes can be very helpful for companies in analyzing consumer trends and understanding which direction the company should move in the future.

3. News Monitoring

Web scraping news sites can provide detailed reports on the current news to a company. This is even more essential for companies that are frequently in the news or that depend on daily news for their day-to-day functioning. After all, news reports can make or break a company in a single day.

4. Sentiment Analysis

If companies want to understand the general sentiment for their products among their consumers, then Sentiment Analysis is a must. Companies can use web scraping to collect data from social media websites such as Facebook and Twitter as to what the general sentiment about their products is. This will help them in creating products that people desire and moving ahead of their competition.

5. Email Marketing

Companies can also use Web scraping for email marketing. They can collect Email IDs from various sites using web scraping and then send bulk promotional and marketing Emails to all the people owning those Email IDs.

Q 7 What are the different Python web scraping libraries?

Ans – Python is popular for web scraping owing to the abundance of third party libraries that can scrape complex HTML structures, parse text, and interact with HTML

form. Here, we've listed some top Python web scraping libraries.

Urllib3 is a powerful HTTP client library for Python. This makes it easy to perform HTTP requests programmatically. It handles HTTP headers, retries, redirects, and other low-level details, making it an excellent library for web scraping. It also supports SSL verification, connection pooling, and proxying.

BeautifulSoup allows you to parse HTML and XML documents. Using API, you can easily navigate through the HTML document tree and extract tags, meta titles, attributes, text, and other content. BeautifulSoup is also known for its robust error handling.

MechanicalSoup automates the interaction between a web browser and a website efficiently. It provides a high-level API for web scraping that simulates human behavior. With MechanicalSoup, you can interact with HTML forms, click buttons, and interact with elements like a real user.

Requests is a simple yet powerful Python library for making HTTP requests. It is designed to be easy to use and intuitive, with a clean and consistent API. With Requests, you can easily send GET and POST requests, and handle cookies, authentication, and other HTTP features. It is also widely used in web scraping due to its simplicity and ease of use.

Selenium allows you to automate web browsers such as Chrome, Firefox, and Safari and simulate human interaction with websites. You can click buttons, fill out forms, scroll pages, and perform other actions. It is also used for testing web applications and automating repetitive tasks.

Pandas allow storing and manipulating data in various formats, including CSV, Excel, JSON, and SQL databases. Using Pandas, you can easily clean, transform, and analyze data extracted from websites.

Q.8 Write a program to download a file from a url.

Ans. To get something from a url we need to use requests.get function.

Example:

```
import requests
res = requests.get('http://www.gutenberg.org/cache/epub/1112/pg1112.txt')
res.raise_for_status()
playFile = open('RomeoAndJuliet.txt', 'wb')
for chunk in res.iter_content(100000):
playFile.write(chunk)
playFile.close()
```

Q.9 How to scrape data from websites using Python?

Ans. Let's take a look at the step-by-step process of using Python to scrape website data.

Step 1: Choose the Website and Webpage URL

The first step is to select the website you want to scrape. For this particular tutorial, let's scrape https://www.imdb.com/. We will try to extract data on the top-rated movies on the website.

Step 2: Inspect the website

Now the next step is to understand the website structure. Understand what the attributes of the elements that are of your interest are. Right-click on the website to

select "Inspect". This will open the HTML code. Use the inspector tool to see the name of all the elements to use in the code.

Note these elements' class names and ids as they will be used in the Python code.

Step 3: Installing the important libraries

Python has several web scraping libraries. We will use the following libraries:

requests - for making HTTP requests to the website

BeautifulSoup - for parsing the HTML code

pandas - for storing the scraped data in a data frame

time - for adding a delay between requests to avoid overwhelming the website with requests

Install the libraries using the following command

pip install requests beautifulsoup4 pandas time

Step 4: Write the Python code

Now, it's time to write the main python code. The code will perform the following steps:

Using requests to send an HTTP GET request

Using BeautifulSoup to parse the HTML code

Extracting the required data from the HTML code

Store the information in a pandas dataframe

Add a delay between requests to avoid overwhelming the website with requests

Here's the Python code to scrape the top-rated movies from IMDb:

```
import requests
from bs4 import BeautifulSoup
import pandas as pd
import time
# URL of the website to scrape
url = "https://www.imdb.com/chart/top"
# Send an HTTP GET request to the website
response = requests.get(url)
# Parse the HTML code using BeautifulSoup
soup = BeautifulSoup(response.content, 'html.parser')
# Extract the relevant information from the HTML code
movies = []
for row in soup.select('tbody.lister-list tr'):
title = row.find('td', class_='titleColumn').find('a').get_text()
year = row.find('td', class_='titleColumn').find('span', class_='secondaryInfo').get_
text()[1:-1]
rating = row.find('td', class_='ratingColumnimdbRating').find('strong').get_text()
movies.append([title, year, rating])
# Store the information in a pandas dataframe
df = pd.DataFrame(movies, columns=['Title', 'Year', 'Rating'])
# Add a delay between requests to avoid overwhelming the website with requests
```

```
time.sleep(1)
```

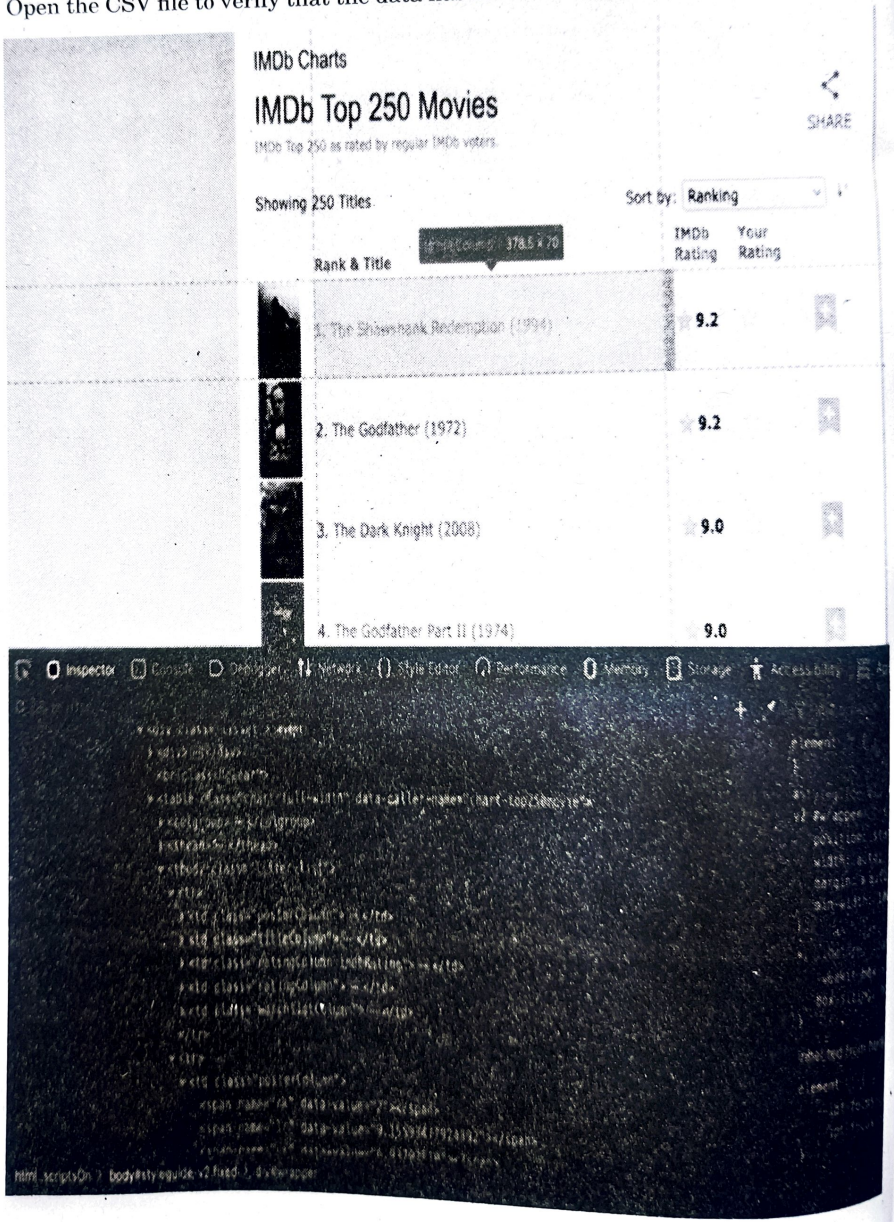
Step 5: Exporting the extracted data

Now, let's export the data as a CSV file. We will use the pandas library.

```
# Export the data to a CSV file
df.to_csv('top-rated-movies.csv', index=False)
```

Step 6: Verify the extracted data

Open the CSV file to verify that the data has been successfully scraped and stored.



SYLLABUS
PYTHON PROGRAMMING
(MCA-106)
Applicable from the Academic Session 2023-24

UNIT I

Conceptual Introduction: History, Features and Future of Python, Installation, Interactive Shell, Program Structure, Identifiers, Keywords, Escape Sequences, Data-Types, Variables, Assignments, Immutable Variables, Operators and Operands, Precedence of Operators, Type-Conversion Functions, Short-circuit vs.Lazy Evaluation, Input and Output Functions, Comments, Command-Line arguments. **Control Flow:** Conditional Statements: if, if-else, if-elif-else, Nested Conditionals, Loops: for, while, break, and continue statements. **Functions:** Defining, Calling and Types of Functions, Arguments and Return Values, Formal vs. Actual Arguments, Scope and Lifetime, Keyword Arguments, Default Arguments, Decorators, Iterators and Generators, Recursion. **Modules:** Importing Modules, Math and Random Module, Packages and Composition. **[No. of Hrs. 10]**

UNIT II

String Manipulations: Basic functions of Strings, Subscript Operator, Indexing, Slicing and Immutable Strings. **Data Structures:** Lists, Tuples, and Dictionaries, Basic List Operations, List Slicing, List Methods, Cloning Lists, Mutability, Searching and Sorting Lists, Tuples: Tuple Assignment, Tuple as Return Value, Dictionary Literals, Adding and Removing Items, Accessing and Replacing Values, Traversing Dictionaries, Sorting Dictionaries, Dictionary Operations, Sets Operations. **File Handling:** Text Files-Writing and Reading Operations, Creating and Reading a Formatted File, Manipulating Files and Directories, Closing Files. **[No. of Hrs. 11]**

UNIT III

Object Oriented Programming: Classes, Objects, Attributes and Methods, Access Specifiers, Constructors, Static Methods, Data Hiding, Inheritance, Polymorphism, Operator Overloading, Abstract Classes. **Threads:** Multi-Threading, Life-Cycle of a Thread, Synchronization using Locks and Semaphores. **Exception Handling:** Exception Class Hierarchy, Except clause, Try, Finally clause, User-Defined Exceptions, Assertions. **[No. of Hrs. 12]**

UNIT IV

Advanced Python: Lambda and List Comprehensions, Map, Reduce and Filter Functions **NumPy Library:** Introduction to NumPy, Creation of One-Dimensional Arrays, Reshaping of an Array, Element-wise Operations, Aggregate Operations, Array Indexing, Array Slicing, Insert Row/Columns, Append Row/Columns, Array Manipulation Operations, Multi-Dimensional Arrays. **Pandas Library:** Data Preparation and Pre-Processing, Series, Querying a Series, Data-Frames, DataFrame Indexing and Loading, Querying a Data Frame, Indexing Data Frames, Missing Values. **Data Visualization:** Graphs in Python: Bar charts, Pie-charts, Scatter plots, Multiple plots, Subplots, Legends, Changing figure Size, Styling plots using Matplotlib Library. **GUI Programming:** Creating User-interface, GUI Widgets with Tkinter, Creating Layouts, Check Box, Radio Buttons, List Box, Menus, Menus Options, Dialog Boxes **Database Access:** Database Connectivity Operations: Create, Insert, Select, Delete, Drop, Update, Joins. **[No. of Hrs. 12]**

END TERM EXAMINATION [JULY-2023]
SECOND SEMESTER [MCA]
PYTHON PROGRAMMING [MCA-106]

Time: 3 Hrs. **Max. Marks: 75**

Note: Attempt five questions in all including Q .No. 1 is compulsory. Select one question from each unit.

Q.1. Answer all the following questions briefly: **(2.5 × 10 = 25)**

Q.1. (a) Write the syntax of for loop statement.

Ans. In Python, the for loop is used to iterate over a sequence (such as a list, tuple, string, or range) or other iterable objects. The basic syntax of a for loop in Python is as follows:

```
for variable in iterable:
    # Code to be executed in each iteration
    # ...
# Code outside the loop
```

Here, variable is a user-defined variable that takes on the value of each element in the iterable during each iteration of the loop. The indented block of code below the for statement is executed for each element in the iteration.

For example, iterating over a list:

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

This loop will iterate over each element in the fruits list, and in each iteration, the fruit variable will take on the value of the current element, which is then printed.

You can also use the range() function to generate a sequence of numbers and iterate over it:

```
for i in range(5):
    print(i)
```

This loop will print numbers from 0 to 4.

Q.1. (b) Illustrate membership operator.

Ans. Membership operators are used to test whether a value is a member of a sequence or not. The two main membership operators are in and not in. These operators return True if the specified value is found in the sequence and False otherwise.

For example;

```
# Illustrating the 'in' membership operator
fruits = ['apple', 'banana', 'cherry']
# Check if 'banana' is in the list
if 'banana' in fruits:
    print('Yes, banana is in the list.')
else:
    print('No, banana is not in the list.')
# Check if 'orange' is not in the list
```

```
if 'orange' not in fruits:
    print('Yes, orange is not in the list.')
else:
    print('No, orange is in the list.')
```

In this example, the in operator is used to check whether 'banana' is in the fruits list and whether 'orange' is not in the list.

Q.1. (c) Discuss importance of docstring.

Ans. Docstrings (documentation strings) in Python are important for several reasons. They serve as a form of inline documentation and are used to document the purpose and usage of functions, classes, methods, and modules. Here are some key reasons why docstrings are important in Python:

1. Documentation: Docstrings provide a convenient way to document code. They allow developers to describe the purpose of a function or class, explain the parameters, and provide examples of usage. Proper documentation helps other developers (and often yourself in the future) understand how to use the code correctly.

2. Readability: Well-documented code is more readable and maintainable. Docstrings make it easier for others (or yourself) to quickly understand the functionality of a particular piece of code without having to dive into the implementation details.

3. Automated Documentation Tools: Docstrings are used by automated documentation tools like Sphinx, which can generate documentation in various formats (HTML, PDF, etc.). This is particularly useful for creating documentation for larger projects or libraries.

4. Interactive Help: Docstrings can be accessed interactively in the Python interpreter using the help() function. This provides users with quick access to documentation while working in an interactive environment.

5. Testing: Docstrings often include examples and use cases for functions and classes. These examples can serve as test cases, helping to ensure that the code behaves as expected. Automated testing tools can also use docstrings for testing purposes.

6. IDE Support: Many integrated development environments (IDEs) use docstrings to provide context-aware help and autocompletion suggestions. Properly documented code enhances the developer experience within these environments.

7. Code Quality: Including docstrings is considered good coding practice. It reflects a commitment to writing self-explanatory and well-documented code. This can contribute to overall code quality and maintainability.

Q.1. (d) Demonstrate any two operations on strings with example.

Ans. Here are two common operations on strings in Python:

1. Concatenation:

Concatenation is the process of combining two or more strings into a single string. This can be done using the + operator or by using the str.join() method.

Example using + operator:

```
string1 = "Hello"
string2 = "World"
# Concatenation using +
result = string1 + " " + string2
print(result)
```

2. String Slicing:

String slicing is the process of extracting a portion (substring) from a string. It done using the slicing syntax string[start:stop], where start is the index where the sl begins, and stop is the index where the slice ends (the character at this index is n included).

```
original_string = "Python Programming"
# Extracting a substring using string slicing
substring = original_string[7:18]
print(substring)
```

Q.1. (e) Make use of comprehension to create a dictionary. Give suitab example.

Ans. Comprehensions in Python provide a concise way to create collections li lists, dictionaries, or sets. Here's an example of using a dictionary comprehension create a dictionary:

```
# Example: Creating a dictionary of squares using a dictionary comprehension
# Using a for loop to create the dictionary
squares_dict = {}
for num in range(1, 6):
    squares_dict[num] = num**2
print("Dictionary created using a for loop:", squares_dict)
# Using a dictionary comprehension for the same task
squares_dict_comprehension = {num: num**2 for num in range(1, 6)}
print("Dictionary created using a comprehension:", squares_dict_comprehension
```

Output:

Dictionary created using a for loop: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

Dictionary created using a comprehension: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

In this example, we are creating a dictionary where the keys are numbers from 1 5, and the values are the squares of those numbers. The first part of the code achiev this using a traditional for loop, while the second part achieves the same result using dictionary comprehension.

Q.1. (f) Briefly explain seek() and tell () methods.

Ans. In Python, the seek() and tell() methods are related to file handling and a used with file objects.

1. seek(offset, whence) Method:

- The seek() method is used to change the current position of the file cursor with a file.
- It takes two arguments:

→ **offset:** The number of bytes to move the cursor. A positive offset moves th cursor forward, and a negative offset moves it backward.

→ **whence:** It specifies the reference position for the offset. It can take one of thre values:

- 0 (default): The beginning of the file.
- 1: The current file position.
- 2: The end of the file.

- After using seek(), the next read or write operation occurs from the new curs position.

```
# Open a file for reading
file = open("example.txt", "r")
# Move the cursor to the 10th byte from the beginning of the file
file.seek(10, 0)
# Read and print the content from the current cursor position
content = file.read()
print(content)
# Close the file
file.close()
```

2. tell() Method:

- The tell() method is used to get the current position of the file cursor within a file.
- It does not take any arguments.
- It returns an integer representing the current offset from the beginning of the file.

Example:

```
# Open a file for reading
file = open("example.txt", "r")
# Read the first 5 bytes from the file
content = file.read(5)
print("Content read:", content)
# Get the current cursor position using tell()
position = file.tell()
print("Current position:", position)
# Close the file
file.close()
```

Q.1. (g) Discuss indexing and slicing with example.

Ans. Indexing in Python refers to accessing individual elements of a sequence, such as a string or a list, using their position or index. Indexing starts from 0 for the first element. Negative indices can be used to access elements from the end, where -1 corresponds to the last element.

For example;

```
# Indexing in a string
text = "Hello, World!"
first_char = text[0]    # 'H'
third_char = text[2]    # 'l'
last_char = text[-1]    # '!'
```

Slicing is the process of extracting a portion (substring or subsequence) from a sequence. It is done using the slicing syntax **start:stop**, where start is the index where the slice begins, and **stop** is the index where the slice ends (the character at this index is not included).

For example;

```
# Slicing a list
numbers = [0, 1, 2, 3, 4, 5]
subset = numbers[2:5]  # [2, 3, 4]
```

```
s = "Hello, World!"
word = s[7:12]
substring = word[2:4]
print(substring)
```

Q.... (i) Differentiate between exception and error.

Ans. **Error**

- An error, often referred to as a runtime error or a runtime exception, is a more general term that describes any unexpected or undesirable situation that occurs during the execution of a program.
- Errors can be classified into various categories, such as syntax errors, logical errors, and runtime errors.
- Syntax errors occur during the parsing phase when the code violates the language's syntax rules and prevents the program from running. Logical errors are mistakes in the program's logic that lead to incorrect results. Runtime errors occur during program execution, and they can be caused by a variety of issues, such as division by zero or attempting to access an index that is out of bounds.

2. Exception:

- An exception, on the other hand, specifically refers to a runtime error that is raised during the execution of a Python program and disrupts the normal flow of the program.
- Exceptions are a mechanism in Python for handling errors in a more controlled and structured way. When an exception occurs, the normal flow of the program is interrupted, and the control is transferred to the nearest suitable exception-handling block.
- Examples of exceptions in Python include ZeroDivisionError, IndexError, TypeError, and FileNotFoundError, among others.

Q.... (ii) Discuss grid() method of tkinter.

Ans. In Tkinter, which is the standard GUI (Graphical User Interface) toolkit for Python, the grid() method is used to organize widgets in a table-like structure. This method is associated with the geometry manager known as the grid manager.

The grid() method is applied to widgets (such as frames, labels, buttons, etc.) to specify the row and column in which the widget should be placed within a grid. It allows you to create a more structured layout for your GUI by arranging widgets in rows and columns.

Here is the basic syntax of the grid() method:

widget.grid(options)

The options parameter consists of a set of configuration options that determine the placement and behavior of the widget within the grid.

Q.... (j) Compare and contrast append() and insert() methods of numpy library.

Ans. In the context of the NumPy library in Python, both **append()** and **insert()** methods are used to add elements to arrays, but they have different functionalities.

numpy.append()

The **numpy.append()** method is used to append elements to the end of an array along a specified axis. It creates a new array with the combined elements of the original array and the elements provided for appending.

Syntax:

numpy.append(arr, values, axis = None)

- **arr:** Input array.
- **values:** Values to be appended.
- **axis:** The axis along which values are appended. If not specified, the array is flattened before appending.

numpy.insert()

The **numpy.insert()** method is used to insert elements along a specified axis before a specified index. It creates a new array with the inserted elements.

Syntax:

numpy.insert(arr, obj, values, axis=None)

- **arr:** Input array.
- **obj:** Object that defines the index or indices before which values are inserted.
- **values:** Values to be inserted.
- **axis:** The axis along which values are inserted. If not specified, the array is flattened before inserting.

Comparison:

- **Functionality:**
  - **numpy.append():** Appends elements to the end of the array.
  - **numpy.insert():** Inserts elements before a specified index.
- **Usage:**
  - **numpy.append():** Commonly used to add elements at the end of the array.
  - **numpy.insert():** Used when you want to insert elements at specific positions in the array.
- **Indexing:**
  - **numpy.append():** Does not require specifying an index; appends elements to the end.
  - **numpy.insert():** Requires specifying the index before which elements are inserted.
- **Return Value:**
  - Both methods return a new array with the appended or inserted elements; the original array remains unchanged.

In summary, **numpy.append()** is typically used to add elements at the end of an array, while **numpy.insert()** is used to insert elements at specific positions in the array. The choice between them depends on the specific requirements of your task.

UNIT - I

Q.2. (a) Explain the concept of scope and lifetime of variables in Python Programming language with a suitable example. **(6.5)**

Ans. In Python, the concepts of scope and lifetime are fundamental to understanding how variables are accessed and how long they exist during the execution of a program.

Scope:

Scope refers to the region of the code where a variable can be accessed or modified. In Python, there are mainly two types of scope:

1. **Local Scope:**
   - Variables defined within a function have a local scope.
   - They are accessible only within that specific function.
   - They are not visible to other functions or the global scope.
2. **Global Scope:**

- Variables defined outside of any function have a global scope.
- They are accessible throughout the entire program.
- Global variables can be accessed by functions, but to modify them inside a function, you need to use the global keyword.

Lifetime:

Lifetime refers to the duration for which a variable exists in memory during the execution of a program. Variables have a lifetime based on their scope:

1. **Local Lifetime:**
   - Variables in local scope are created when the function is called and destroyed when the function exits.
   - They exist only as long as the function is being executed.
2. **Global Lifetime:**
   - Global variables are created when the program starts and persist until the program terminates.
   - They exist for the entire duration of the program.

For example:

```
# Global variable
global_var = 10
def example_function():
    # Local variable
    local_var = 5
    print f"Inside the function: global_var = {global_var}, local_var = {local_var}"
# Accessing global variable
print f"Outside the function: global_var = {global_var}"
# Calling the function
example_function()
# Attempting to access the local variable outside the function (will result in an error)
# print f"Outside the function: local_var = {local_var}"
```

In this example:

- **global_var** is a global variable, accessible both outside and inside the function.
- **local_var** is a local variable, accessible only inside the **example_function**.
- When the program is executed, you can see that **global_var** is accessible from both inside and outside the function, while **local_var** is only accessible inside the function.

Attempting to access **local_var** outside the function would result in an error because it is not defined in the global scope.

Q.2. (b) Write a python program to find roots of quadratic equation. **(6)**

Ans. To find the roots of a quadratic equation ax2 + bx + c = 0, you can use the quadratic formula:

x = 2a + b ± b2 - 4ac

Here's a Python program that takes user input for the coefficients *a*, *b*, and *c* and calculates the roots of the quadratic equation:

```
import math
def find_roots(a, b, c):
    # Calculate the discriminant
```

```
discriminant = b**2 - 4*a*c
# Check if the discriminant is positive, negative, or zero
if discriminant > 0:
    root1 = (-b + math.sqrt(discriminant)) / (2*a)
    root2 = (-b - math.sqrt(discriminant)) / (2*a)
print(f"The roots are real and distinct: {root1} and {root2}")
elif discriminant == 0:
    root = -b / (2*a)
    print(f"The root is real and equal: {root}")
    else:
    real_part = -b / (2*a)
    imaginary_part = math.sqrt(abs(discriminant)) / (2*a)
    print(f"The roots are complex: {real_part} + {imaginary_part}j and {real_part}
    - {imaginary_part}j")
# Get user input for coefficients
a = float(input("Enter the coefficient a: "))
b = float(input("Enter the coefficient b: "))
c = float(input("Enter the coefficient c: "))
# Call the function to find roots
find_roots(a, b, c)
```

This program uses the math.sqrt() function to calculate the square root and handles cases where the discriminant is positive, zero, or negative (real and distinct roots, real and equal roots, or complex roots, respectively).

Q.3. (a) Write short note on short circuit evaluation performed by python when evaluating combined comparisons. Support your answer by giving suitable example. **(6.5)**

Ans. Short circuit evaluation is a technique used by programming languages, including Python, to optimize logical expressions by avoiding unnecessary evaluations. In Python, when evaluating combined comparisons using logical operators (such as andandor), the interpreter stops evaluating as soon as the result is determined.

Short Circuit Evaluation with and:

In the case of the and operator, if the first operand is False, the entire expression is False, and there is no need to evaluate the second operand. If the first operand is True, the second operand is then evaluated to determine the final result.

Example:

```
x = 5
y = 10
result = (x > 0) and (y < 20)
print(result)  # Output: True
```

In this example, since the first operand **(x > 0)** is **True**, the second operand (y < 20) is evaluated, and the final result is **True**.

Short Circuit Evaluation with or:

In the case of the or operator, if the first operand is True, the entire expression is True, and there is no need to evaluate the second operand. If the first operand is False, the second operand is then evaluated to determine the final result.

```
x = 5
```

- Example: result = None

Each of these data types serves a specific purpose, and choosing the appropriate data type is important for efficient storage and manipulation of data in a program. Python is a dynamically typed language, which means that the interpreter determines the data type of a variable during runtime.

UNIT - II

Q.4. (a) Demonstrate any four methods of Dictionary. **(6.5)**

Ans. Here are examples of four commonly used methods of dictionaries in Python:

1. get() Method:

- The get() method returns the value for the specified key. If the key is not found, it returns a default value (which is None by default).

```
# Dictionary
student = {'name': 'Alice', 'age': 25, 'grade': 'A'}
# Using get() to retrieve the value for the key 'age'
age = student.get('age')
print("Age:", age)
# Using get() with a default value
gender = student.get('gender', 'Not specified')
print("Gender:", gender)
```

2. keys() Method:

- The keys() method returns a view of all the keys in the dictionary.

```
# Dictionary
student = {'name': 'Alice', 'age': 25, 'grade': 'A'}
# Using keys() to get all keys
all_keys = student.keys()
print("All Keys:", all_keys)
```

3. values() Method:

The values() method returns a view of all the values in the dictionary.

```
# Dictionary
student = {'name': 'Alice', 'age': 25, 'grade': 'A'}
# Using values() to get all values
all_values = student.values()
print("All Values:", all_values)
```

4. items() Method:

The items() method returns a view of all key-value pairs as tuples.

```
# Dictionary
student = {'name': 'Alice', 'age': 25, 'grade': 'A'}
# Using items() to get all key-value pairs
all_items = student.items()
print("All Items:", all_items)
```

Q.4. (b) Write Python program to perform a linear search for a given number in the list and report Success or Failure. **(6)**

Ans. A

```
def linear_search(arr, target):
    for i, num in enumerate(arr):
        if num == target:
```

```
        return True, i # Return True and the index if found
    return False, -1 # Return False and -1 if not found

# Example list
numbers = [12, 34, 56, 78, 90, 23, 45, 67]
# Number to search
target_number = int(input("Enter the number to search: "))
# Perform linear search
found, index = linear_search(numbers, target_number)
# Report success or failure
if found:
    print(f"The number {target_number} is found at index {index}.")
else:
    print(f"The number {target_number} is not found in the list.")
```

Q.5. (a) Illustrate the difference between fully qualified path and relative path with an example. **(6.5)**

Ans. In file systems, a fully qualified path and a relative path are two ways to specify the location of a file or directory.

Fully Qualified Path:

A fully qualified path provides the complete and exact location of a file or directory starting from the root directory. It includes all the necessary information to uniquely identify the file's location.

/home/user/documents/example.txt

In this example, / represents the root directory, and the file example.txt is located in the documents directory, which is inside the user directory, which, in turn, is inside the root directory.

Relative Path:

A relative path specifies the location of a file or directory relative to the current working directory. It does not start from the root directory but instead describes the path from the current location.

Example:

../images/photo.jpg

In this example, .. refers to the parent directory of the current working directory, and the file **photo.jpg** is located inside the **images** directory, which is a sibling directory to the current working directory.

Q.5. (b) "Lists are Heterogeneous". Support the statement with an example. **(6)**

Ans. Yes, lists in Python are heterogeneous, meaning they can contain elements of different data types. Unlike arrays in some other programming languages, Python lists can hold a mix of integers, floats, strings, and other data types.

Here's an example of a heterogeneous list:

```
heterogeneous_list = [1, 2.5, "hello", True, [3, 4, 5]]
# Print the entire list
print("Heterogeneous List:", heterogeneous_list)
# Accessing individual elements
print("Element 1:", heterogeneous_list[0])      # Integer
print("Element 2:", heterogeneous_list[1])      # Float
print("Element 3:", heterogeneous_list[2])      # String
```

```
print("Element 4:", heterogeneous_list[3])      # Boolean
print("Element 5:", heterogeneous_list[4])      # List
print("Element 5, Sub-element 1:", heterogeneous_list[4][0])  # Accessing a sub-element in the list
```

In this example, **heterogeneous_list** contains elements of various types:

- **1** is an integer.
- **2.5** is a floating-point number.
- **"hello"** is a string.
- **True** is a boolean.
- **[3, 4, 5]** is a nested list.

UNIT - III

Q.6. (a) Write a class Square that finds the square of a number. Throw an exception if instead of the number, user enters a character. (6.5)

Ans.

```
def square(number):
    try:
        result = float(number) ** 2
        return result
    except ValueError:
        raise ValueError("Invalid input. Please enter a valid number.")
# Example usage
user_input = input("Enter a number to find its square: ")
try:
    result = square(user_input)
print(f"The square of {user_input} is: {result}")
except ValueError as e:
print(f"Error: {e}")
```

- The **square** function attempts to convert the input to a float and then squares it.
- If the conversion to float fails (i.e., the user entered a non-numeric value), a **ValueError** is raised with a custom error message.
- In the example usage, the user is prompted to enter a number.
- The **square** function is called, and if successful, the result is printed.
- If an exception is caught (in this case, a **ValueError**), an error message is printed.

Q.6. (b) Discuss polymorphism as implemented in Python programming language with an example. Write Python Program to Create a Class Called as Complex and Implement _ add _ () Method to Add Two Complex Numbers. Display the result by overloading the + Operator. (6)

Ans. Polymorphism in Python allows objects of different types to be treated as objects of a common type. It is achieved through method overloading or operator overloading. Method overloading allows a class to have multiple methods having the same name, but different signatures. Operator overloading, on the other hand, involves defining how operators behave for user-defined objects.

In Python, operator overloading is commonly used to implement polymorphism. Let's create a class called **Complex** and implement the **__add__** method to overload the + operator for adding two complex numbers:

```
class Complex:
```

```
    def __init__(self, real, imag):
    self.real = real
    self.imag = imag
    def __add__(self, other):
    # Overloading the + operator to add two complex numbers
    real_part = self.real + other.real
    imag_part = self.imag + other.imag
    return Complex(real_part, imag_part)
    def __str__(self):
      # String representation of the complex number
      return f"{self.real} + {self.imag}j"
# Creating instances of the Complex class
complex1 = Complex(2, 3)
complex2 = Complex(1, 4)
# Adding two complex numbers using the overloaded + operator
result = complex1 + complex2
# Displaying the result
print(f"Result: {result}")
```

In this example, the Complex class has a constructor __init__ to initialize the real and imaginary parts of a complex number. The __add__ method is defined to overload the + operator, allowing two instances of the Complex class to be added together. The __str__ method is implemented to provide a string representation of the complex number when it is printed.

Q.7. (a) Discuss thread synchronization in python. Also discuss event object. **(6.5)**

Ans. Thread synchronization is crucial in multithreading applications to ensure that multiple threads can safely access shared resources without causing data corruption or race conditions. In Python, we can achieve thread synchronization using various mechanisms, and one of them is through the use of locks. Additionally, event objects can be employed to coordinate the activities of multiple threads.

Thread Synchronization with Locks:

A lock, also known as a mutex (short for mutual exclusion), is a synchronization primitive used to protect shared resources. It allows only one thread to acquire the lock at a time, ensuring that critical sections of code are executed by only one thread at any given moment. Python provides the threading.Lock class for this purpose.

Here's a simple example demonstrating the use of a lock for synchronization:

```
import threading
# Shared resource
counter = 0
# Creating a lock
lock = threading.Lock()
def increment_counter():
  global counter
  for _ in range(100000):
    # Acquiring the lock before accessing the shared resource
    with lock:
      counter += 1
```

```
    # Releasing the lock
# Creating two threads
thread1 = threading.Thread(target=increment_counter)
thread2 = threading.Thread(target=increment_counter)
# Starting the threads
thread1.start()
thread2.start()
# Waiting for both threads to finish
thread1.join()
thread2.join()
# Displaying the final counter value
print("Final Counter:", counter)
```

In this example, the **increment_counter** function increments a shared counter within a loop. The with lock statement ensures that only one thread can acquire the lock at a time, preventing race conditions.

Event Object:

An event is a synchronization primitive that allows one or more threads to wait until a particular event occurs. The **threading. Event** class in Python provides a simple way to communicate between threads using a flag that can be set or cleared. Threads can wait for the event to be set or cleared, and other threads can set or clear the event.

Here's a basic example:

```
import threading
import time
# Creating an event object
event = threading.Event()
def wait_for_event():
print("Thread A is waiting for the event.")
event.wait()  # Block and wait for the event to be set
print("Thread A received the event.")
def set_event():
print("Thread B is sleeping for 2 seconds and then setting the event.")
time.sleep(2)
event.set()  # Set the event, waking up any waiting threads
# Creating two threads
thread_a = threading.Thread(target=wait_for_event)
thread_b = threading.Thread(target=set_event)
# Starting the threads
thread_a.start()
thread_b.start()
# Waiting for both threads to finish
thread_a.join()
thread_b.join()
```

In this example, **Thread A** is waiting for the event using **event.wait()**, and **Thread B** sets the event after a short delay using **event.set()**. The **event.wait()** call in **Thread A** will block until the event is set by **Thread B**. This can be useful for coordinating activities between different threads.

Q.7. (b) Write a program of bank account to create new account, display account, deposit amount and withdraw using class and function. **(6)**

Ans. class BankAccount:

```
    def __init__(self, account_number, account_holder, balance=0.0):
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = balance

    def display_account_info(self):
        print(f"Account Number: {self.account_number}")
        print(f"Account Holder: {self.account_holder}")
        print(f"Balance: $({self.balance:.2f}")
    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"Deposited $({amount:.2f}. New balance: $({self.balance:.2f}")
        else:
            print("Invalid deposit amount. Please enter a positive value.")
    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew $({amount:.2f}. New balance: $({self.balance:.2f}")
        else:
            print("Invalid withdrawal amount or insufficient funds.")
# Example usage of the BankAccount class
# Create a new account
account1 = BankAccount(account_number="123456789", account_holder="John
Doe")
# Display account information
account1.display_account_info()
# Deposit into the account
account1.deposit(1000)
# Withdraw from the account
account1.withdraw(500)
# Display updated account information
account1.display_account_info()
```

This program defines a BankAccount class with methods for displaying account information (display_account_info), depositing funds (deposit), and withdrawing funds (withdraw). The __init__ method is used to initialize the account with an account number, account holder name, and an optional initial balance.

UNIT - IV

Q.8. (a) Mention the different types of Data Structures in Pandas. Demonstrate how missing data is handled in Pandas. **(6.5)**

Ans. Pandas, a powerful data manipulation library in Python, provides several types of data structures. The two primary structures are:

1. Series:

- A one-dimensional array-like object that can hold any data type.
- It is similar to a column in a spreadsheet or a single variable in statistics.
- Created using the pd.Series() constructor.

```
import pandas as pd
# Creating a Series
series_data = pd.Series([1, 3, 5, np.nan, 6, 8])
```

```
print(series_data)
```

2. DataFrame:

A two-dimensional table of data with rows and columns.It is similar to a spreadsheet or a SQL table.Created using the pd.DataFrame() constructor.

```
import pandas as pd
import numpy as np
# Creating a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, np.nan],
        'Salary': [50000, 60000, 75000]}
df = pd.DataFrame(data)
print(df)
```

Now, let's discuss how missing data is handled in pandas:

Pandas uses the NaN (Not a Number) value to represent missing or undefined data. It provides several functions and methods for working with missing data:

1. Checking for Missing Data:

- **isnull()** and **notnull()** functions can be used to detect missing values.

```
import pandas as pd
import numpy as np
data = {'Name': ['Alice', 'Bob', np.nan],
        'Age': [25, 30, np.nan],
        'Salary': [50000, 60000, 75000]}
df = pd.DataFrame(data)
# Check for missing values
print(df.isnull())
print(df.notnull())
```

2. Handling Missing Data:

- **dropna():** Removes rows with missing values.
- **fillna():** Fills in or imputes missing values.

```
import pandas as pd
import numpy as np
data = {'Name': ['Alice', 'Bob', np.nan],
        'Age': [25, 30, np.nan],
        'Salary': [50000, 60000, 75000]}
df = pd.DataFrame(data)
# Drop rows with missing values
df_dropped = df.dropna()
print("DataFrame after dropping rows with missing values:")
print(df_dropped)
# Fill missing values with a specific value (e.g., mean)
mean_age = df['Age'].mean()
df_filled = df.fillna({'Age': mean_age})
print("\nDataFrame after filling missing values:")
print(df_filled)
```

Q.8. (b) Explain show(), figure(),plot(), savefig(), xlabel(), ylable() with suitable example. **(6)**

Ans. The functions you mentioned are commonly used in the context of data visualization using the **matplotlib** library in Python. Here's an explanation of each function along with a suitable example:

```
numbers = [1, 2, 3, 4, 5]
evens = filter(lambda x: x % 2 == 0, numbers)
print(list(evens))  # Output: [2, 4]
```

3. reduce() Function:

- reduce() applies a binary function to the items of an iterable cumulatively, from left to right, so as to reduce the iterable to a single accumulated value.

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 120 (1 * 2 * 3 * 4 * 5)
```

Q.9. (b) With an example, discuss different components of tkinter module. Write a GUI application with a button labeled "GoodBye. When the button is clicked, the window closes. **(6.5)**

Ans. tkinter is a standard GUI (Graphical User Interface) toolkit in Python. It provides various components to create desktop applications with graphical user interfaces. Below is an example that demonstrates the use of **tkinter** to create a simple GUI application with a button that closes the window when clicked.

```
import tkinter as tk
def on_button_click():
root.destroy()  # Close the main window
# Create the main window
root = tk.Tk()
root.title("Goodbye App")
# Create a button widget
goodbye_button = tk.Button(root, text="Goodbye", command=on_button_click)
# Pack the button into the main window
goodbye_button.pack(pady=20)
# Start the main event loop
root.mainloop()
```

Explanation of the components:

1. Tk() - Main Window:

- tk.Tk() creates the main window of the GUI application. This window is the top-level window where other widgets are placed.

2. Button Widget:

- tk.Button() creates a button widget.
- The text parameter sets the text displayed on the button.
- The command parameter specifies the function to be called when the button is clicked.

3. pack() Method:

- pack() is a geometry manager that organizes widgets in blocks before placing them in the parent widget.
- In this example, goodbye_button.pack(pady=20) packs the button into the main window with a vertical padding of 20 pixels.

4. Button Click Function:

- on_button_click() is a function that will be called when the button is clicked.
- Inside this function, root.destroy() is used to close the main window.

5. mainloop():

- root.mainloop() starts the main event loop. The program will continuously listen for user interactions until the main window is closed.