SYLLABUS

	TOPIC
1.	**Python :** Features of Python, Environmental setup, Installation and tools required for running, Basic Types Variable types and operators : Assigning values to variables Multiple Assignments Standard Data Types Set Map Single line comments using Multi-line comments using triple quote, Data Type Conversion Operators, Types of Operator, Conditional statement, Looping statements with else-Pass-Break continue.
2.	**Number and List :** Accessing values in List-Delete, update List element-Basic List operations-Indexing, Slicing and Matrices Built in methods and Functions for List-Accessing values in Tuple-Delete, List element-Basic Tuple operations Indexing, Slicing and Matrices Built in methods and Functions for Tuple.
3.	**Dictionary and Function :** Accessing values in Dictionary-Updating Dictionary-Deleting Dictionary – elements-Properties of Dictionary keys-Built in Dictionary Functions and Methods Defining Function-Calling function- Pass by reference vs value Function Arguments-Required arguments-Keyword arguments-Default arguments-Variable-length arguments Recursion.
4.	**Modules and Packages :** The Time Module and its functions-Calendar modules and its functions-Other modules and Functions Sum and Difference time and date Import From import statement From import statement Executing modules, Local functions-Reload function Packages in Python.
5.	**Exception Handling :** Exception handling and assertions-Standard Exceptions-Assertions in Python-Handling an exception-Except clause with no exception-Except Clause with multiple exception-Try-Finally Clause-Argument of an Exception Raising an Exception.

1. ***What is python***

The Python
executes the co
instructions you
that the comp
compiler, which
execution, the i
at a time. T
print("Hello, wo
The inter
perform the ins
world!" on th
beneficial beca
you write the
program. How
halfway throu
that point, an
until the error

2. ◆ ***List the***

◆ ***Determ***
Python

Data ty
data items.
operations c
everything i
are actually
these classe
Follow
Python :

(1) Nume
(2) Seque
(3) Boole
(4) Set
(5) Dictio

1. *What is python interpreter?* **(2023-24)**

The Python interpreter is a program that reads and executes the code you write. Python script is a set of instructions you've written, and translates it into a form that the computer's hardware can execute. Unlike a compiler, which translates the entire program before execution, the interpreter translates the program one line at a time. To illustrate, a simple Python script: print("Hello, world!")

The interpreter reads the line and immediately perform the instruction, which is to display the text "Hello, world!" on the screen. This line-by-line execution is beneficial because it allows you to test parts of your code as you write them, without needing to translate the entire program. However, it also means that if there is an error halfway through your script, the interpreter will stop at that point, and the rest of the code will not be executed until the error is fixed.

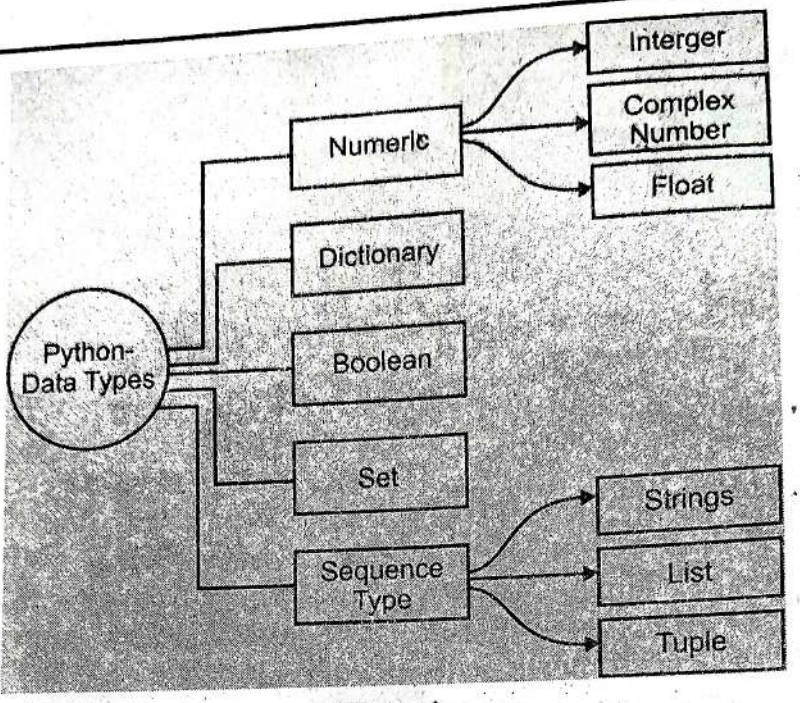
2. ◆ ***List the standard data types in python.*** **(2023-24)**

◆ ***Determine various data types available in Python.***

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

Following are the standard or built-in data type of Python :

(1) Numeric
(2) Sequence Type
(3) Boolean
(4) Set
(5) Dictionary



(Figure)

Numeric

Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data-type. Python provides the type() function to know the data-type of the variable. Similarly, the isinstance() function is used to check an object belongs to a particular class.

Python Supports Three Types of Numeric Data

(1) **Int :** Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to int

(2) **Float :** Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.

(3) **Complex :** A complex number contains an ordered pair, i.e., $x + iy$ where x and y denote the real and imaginary parts, respectively. The complex numbers like $2.14j$, $2.0 + 2.3j$, etc.

Sequence Type

String : The string can be defined as the sequence of characters represented in the quotation marks. In Python, we can use single, double, or triple quotes to define a string.

String h
since Python
perform opera
(1) In the c
to conca
python"
(2) The ope
the ope
List : Pythor
list can conta
the list are s
square brack
We ca
list. The con
(*) works v
working wit
Tuple : A
lists, tuple
different da
with a com
A tup
modify the
Dictionar
pair of iter
where eac
primitive
object.
The
comma (,)
Boolean
and False
statemen
can be re
false can
Set : Py
type. It i
has uniq
undefine
element.
set(), or

Interger
Complex Number
Float
Strings
List
Tuple

String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.

(1) In the case of string handling, the operator + is used to concatenate two strings as the operation "hello"+" python" returns "hello python".

(2) The operator * is known as a repetition operator as the operation "Python" *2 returns 'Python Python'.

List : Python Lists are similar to arrays in C. However, the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets[ ].

We can use slice operators to access the data of the list. The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

Tuple : A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Dictionary : Dictionary is an unordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type, whereas value is an arbitrary Python object.

The items in the dictionary are separated with the comma (,) and enclosed in the curly braces {}.

Boolean : Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false. It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'.

Set : Python Set is the unordered collection of the data type. It is iterable, mutable(can modify after creation), and has unique elements. In set, the order of the elements is undefined; it may return the changed sequence of the element. The set is created by using a built-in function set(), or a sequence of elements is passed in the curly

braces and separated by the comma. It can contain various types of values.

3. *Give the features of python set.* **(2023-24)**

Set : Sets are used to store multiple items in a single variable. Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage. A set is a collection which is unordered, unchangeable*, and unindexed.

Set Items : Set items are unordered, unchangeable, & don't allow duplicate values.

Unordered : Unordered means that the items in a set do not have a defined order. Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Unchangeable : Set items are unchangeable, meaning that we cannot change the items after the set has been created. Once a set is created, you cannot change its items, but you can remove items and add new items.

Duplicates Not Allowed : Sets cannot have two items with the same value.

4. *Write a program to print all prime numbers from 1 to 100.* **(2023-24)**

Python program to print all prime numbers from 1 to 100

```
    i = 2
while i <= 100:
  j = 2
  while j < 100:
    if i%j == 0:
      break
    j += 1
  if i == j:
    print(i,end=',')
  i += 1
```

Output :

2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,

5. *What is loop*

the sum of t

Python
loops in Pyth
requirements
basic functi
condition-che
understand t

This is

$1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}$

Program/So

Program to fi

$1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}$

shown below

```
n=int(inp
sum1=0
for i in ra
    sum1
print("Th
```

Output

Ca

Ca

Program E

(1) User m
of.

(2) The su

(3) The for
the nu

(4) The nu
continu
terms.

(5) Then t

5. *What is looping in python? Write a program to print the sum of the following series* $1+\frac{1}{2}+\frac{1}{3}.....+\frac{1}{n}$.
(2023-24)

Python programming language provides two types of loops in Python – For loop and While loop to handle looping requirements. While all the ways provide similar basic functionality, they differ in their syntax and condition-checking time look at Python loops and understand their working with the help of examples.

This is a Python Program to find the sum of series :

$$1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\cdots.....+\frac{1}{N}$$

Program/Source Code: Here is source code of the Python Program to find the sum of series : $1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\cdots.....+\frac{1}{N}$. The program output is also shown below.

```
n=int(input("Enter the number of terms: "))
sum1=0
for i in range(1,n+1):
   sum1=sum1+(1/i)
print("The sum of series is",round(sum1,2))
```

Output :

Case 1 :

```
Enter the number of terms: 7
 of series is 2.59
```

Case 2 :

```
Enter the number of terms: 15
The sum of series is 3.32
```

Program Explanation :

(1) User must enter the number of terms to find the sum of.

(2) The sum variable is initialized to 0.

(3) The for loop is used to find the sum of the series and the number is incremented for each iteration.

(4) The numbers are added to the sum variable and this continues till the value of i reaches the number of terms.

(5) Then the sum of the series is printed.

6. *Briefly explain about Python IDLE.* **(2022-23)**

Python IDLE (Integrated Development and Learning Environment) is an interactive development environment for Python programming language. It comes pre-installed with Python distribution and provides an easy-to-use interface for writing, executing, and debugging Python code.

Some of the Key Features of Python IDLE are

(1) **Interactive Shell :** It provides a Python shell, where you can enter Python commands and see the output immediately. This is useful for testing small code snippets and experimenting with the language.

(2) **Code Editor :** It has a built-in code editor, where you can write and edit Python scripts. The editor provides features such as syntax highlighting, auto-completion, and indentation, which make writing Python code easier.

(3) **Debugger :** It comes with a debugger that allows you to debug Python code. You can set breakpoints, step through code, and inspect variables to find and fix bugs in your code.

(4) **Documentation :** It provides easy access to Python documentation and help files.

Overall, Python IDLE is a useful tool for learning and experimenting with Python programming, as well as developing small to medium-sized Python projects.

7. *Explain about Logical operator and Boolean expression.* **(2022-23)**

Logical operators are special symbols in programming that are used to combine or manipulate Boolean expressions. Boolean expressions are expressions that evaluate to either True or False, and they are fundamental to decision-making in programming.

The Three Logical Operators are

(1) **AND Operator :** Denoted by and. It returns True if both the operands are True, otherwise it returns False.

(2) **OR Operator :** Denoted by or. It returns True if at least one of the operands is True, otherwise it returns False.

(3) NOT Operator : Denoted by not. It reverses the truth value of its operand. If the operand is True, then it returns False, and if the operand is False, then it returns True.

Boolean expressions are formed using comparison operators (such as ==, <, >, etc.), and can be combined using logical operators. For example, consider the following boolean expressions :

```
x = 5
y = 10
z = 15
# Boolean expressions
x >y  # False
y >x  # True
z == x + y. # True
```

These boolean expressions can be combined using logical operators to form more complex expressions. For example :

```
# Using logical AND operator
x > y and y <z  # False

# Using logical OR operator
x > y or y <z  # True

# Using logical NOT operator
not (x >y)  # True
```

8. *Write a program to check a given number is prime or not. Number is given by user.* **(2022-23)**

Program in Python that checks whether a given number is prime or not :

```
num = int(input("Enter a number: "))  # take input from user

# prime numbers are greater than 1
if num> 1:
   # check for factors
   for i in range(2, int(num/2)+1):
      if (num % i) == 0:
print(num, "is not a prime number")
         break
   else:
print(num, "is a prime number")

# if input number is less than or equal to 1, it is not prime
else:
print(num, "is not a prime number")
```

9. *Write a program to print Armstrong numbers from 100 to 200.* **(2022-23)**

Program in Python that prints the Armstrong numbers between 100 and 200:

```
for num in range(100, 201):
   # order of number
   order = len(str(num))

   # initialize sum
   sum = 0

   # find the sum of the cube of each digit
   temp = num
   while temp > 0:
      digit = temp % 10
      sum += digit ** order
      temp //= 10

   # check if the number is Armstrong
   if num == sum:
      print(num)
```

10. *What is generator in python? Explain with example.* **(2022-23)**

In Python, a generator is a special type of iterator that allows you to iterate over a sequence of values without having to create the entire sequence in memory at once. Instead, a generator produces the values on-the-fly as you iterate over it.

Generators are defined using a special type of function called a generator function. A generator function is similar to a regular function, but it uses the yield keyword instead of the return keyword to produce a sequence of values. When you call a generator function, it returns a generator object, which you can then use to iterate over the sequence of values.

Here's an example of a generator function that generates the first n numbers in the Fibonacci sequence :

```
def fibonacci(n) :
   a, b = 0, 1
   for i in range(n):
      yield a
```

$a, b = b, a + b$

When you call this function with a value of n, it returns a generator object that you can use to iterate over the first n numbers in the Fibonacci sequence. For example :

```
>>> fib = fibonacci(5)
>>> for i in fib:
...     print(i)
...
0
1
1
2
3
```

Notice that we are using a for loop to iterate over the values produced by the generator. Each time we call yield in the generator function, it produces a new value in the sequence. When the for loop is finished, the generator function is automatically closed and any resources it was using are released.

Generators are useful when you need to work with large sequences of values that cannot be easily stored in memory all at once. By producing values on-the-fly as you iterate over them, generators allow you to work with large sequences of data efficiently and effectively.

11. ◆ ***Explain the need for continue, break and pass statements. Write a program in python where these three statements are implemented.*** **(2022-23)**

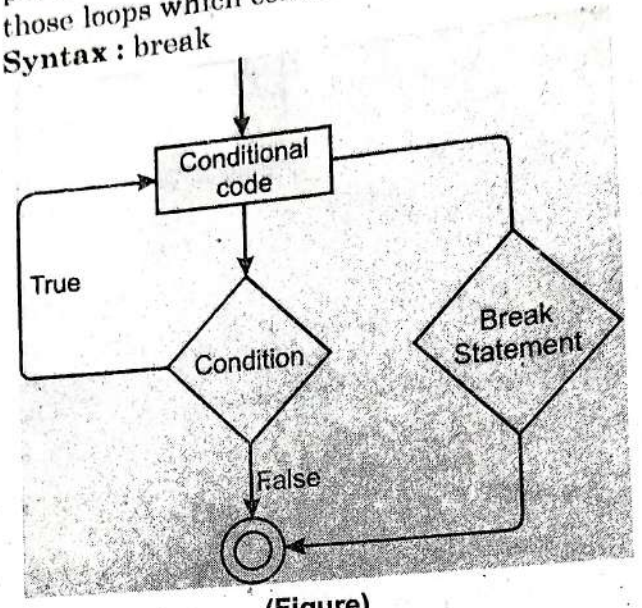
◆ ***Explain break, continue and pass in Python?***

Break, pass, and continue statements are provided in Python to handle instances where you need to escape a loop fully when an external condition is triggered or when you want to bypass a section of the loop and begin the next iteration. These statements can also help you gain better control of your loop.

(1) **Break Statement in Python :** The break statement is used to terminate the loop or statement in which it is present. After that, the control will pass to the statements that are present after the break

statement, if available. If the break statement is present in the nested loop, then it terminates only those loops which contains break statement.

Syntax : break



(Figure)

Example :

```
# Python program to demonstrate
# break statement
# Python program to
# demonstrate break statement
 nums = [7, 2, 3, 1, 5, 4, 6, 8, 9]
count = 0
while count < 7:
  print(nums[count])
  count += 1
  if nums[count] == 4:
    break
        print("End")
```

Output :

```
7
2
3
1
5
4
END
```

(2) Pass Statement in Python : As the name suggests pass statement simply does nothing. The pass statement in Python is used when a statement is required syntactically but you do not want any

command or code to execute. It is like null operation, as nothing will happen is it is executed. Pass statement can also be used for writing empty loops. Pass is also used for empty control statement, function and classes.

Syntax : pass

```
# Python program to demonstrate
# pass statement


s = "geeks"

# Empty loop
for i in s:
   # No error will be raised
   pass

# Empty function
def fun():
   pass

# No error will be raised
fun()

# Pass statement
for i in s:
   if i == 'k':
      Print ('Pass executed')
      pass
   print(i)
```

Output :

```
g
e
e
Pass executed
k
s
```

(3) Continue Statement in Python : Continue is also a loop control statement just like the break statement. continue statement is opposite to that of break statement, instead of terminating the loop, it forces to execute the next iteration of the loop.

As the name suggests the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the

continue statement will be skipped and the next iteration of the loop will begin.

Syntax : continue

Example :

```
# python program to
# demonstrate continue
# statement

# loop from 1 to 10
for i in range(1, 11):

    # If i is equals to 6,
    # Continue to next iteration
    # Without printing
    if i == 6:
        continue
    else:
        # Otherwise print the value
        # of i
        print(i, end = " ")
```

Output :

1 2 3 4 5 7 8 9 10

12. *What are the applications of Python? (In Details)*

Python supports cross-platform operating systems which makes building applications with it all the more convenient. Some of the globally known applications such as YouTube, BitTorrent, DropBox, etc. use Python to achieve their functionality.

(1) **Web Development :** Python can be used to make web-applications at a rapid rate. It is because of the frameworks Python uses to create these applications. There is common-backend logic that goes into making these frameworks and a number of libraries that can help integrate protocols such as HTTPS, FTP, SSL etc. and even help in the processing of JSON, XML E-Mail and so much more. Some of the most well-known frameworks are Django, Flask, Pyramid. Why use a framework? The security, scalability convenience that they provide is commendable if we compare it to starting the development of a website from scratch.

(2) **Game Development :** Python is also used in the development of interactive games. There are libraries

such as PySoy which is a 3D game engine supporting Python 3, PyGame which provides functionality and a library for game development. Games such as Civilization-IV, Disney's Toontown Online, Vega Strike etc. have been built using Python.

(3) **Machine Learning and Artificial Intelligence :** Machine Learning and Artificial Intelligence are the talks of the town as they yield the most promising careers for the future. We make the computer learn based on past experiences through the data stored or better yet, create algorithms which makes the computer learn by itself. The programming language that mostly everyone chooses? It's Python. Why? Support for these domains with the libraries that exist already such as Pandas, Scikit-Learn, NumPy and so many more. Learn the algorithm, use the library and you have your solution to the problem. It is that simple. But if you want to go the hardcore way, you can design your own code which yields a better solution, which still is much easier when we compare it to other languages.

(4) **Data Science and Data Visualization :** Data is money if you know how to extract relevant information which can help you take calculated risks and increase profits. You study the data you have, perform operations and extract the information required. Libraries such as Pandas, NumPy help you in extracting information. You can even visualize the data libraries such as Matplotlib, Seaborn, which are helpful in plotting graphs and much more. This is what Python offers you to become a Data Scientist.

(5) **Desktop GUI :** We use Python to program desktop applications. It provides the Tkinter library that can be used to develop user interfaces. There are some other useful toolkits such as the wxWidgets, Kivy, PYQT that can be used to create applications on several platforms. You can start out with creating simple applications such as Calculators, To-Do apps and go ahead and create much more complicated applications.

(6) **Web Scraping Applications :** Python is a savior when it comes to pull a large amount of data from

websites which can then be helpful in various real-world processes such as price comparison, job listings, research and development and much more.

(7) **Business Applications :** Business Applications are different than our normal applications covering domains such as e-commerce, ERP and many more. They require applications which are scalable, extensible and easily readable and Python provides us with all these features. Platforms such as Tryton is available to develop such business applications.

(8) **CAD Applications :** Computer-Aided Designing is quite challenging to make as many things have to be taken care of. Objects and their representation, functions are just the tip of the iceberg when it comes to something like this. Python makes this simple too and the most well-known application for CAD is Fandango.

(9) **Embedded Applications :** Python is based on C which means that it can be used to create Embedded C software for embedded applications. This helps us to perform higher-level applications on smaller devices which can compute Python.

13. *What is python and its features?*

Python is a dynamic, high level, free open source and interpreted programming language. It supports object-oriented programming as well as procedural oriented programming. In Python, we don't need to declare the type of variable because it is a dynamically typed language.

For example, $x = 10$ Here, x can be anything such as String, int, etc.

Python Features

(1) **Simple and Easy to Learn :** Python is simple and easy to learn, read and write.

(2) **Free and Open-Source :** Python is a free and open-source software which means that a user can edit, modify or reuse the software's source code. This gives the programmers an opportunity to improve the program functionality by modifying it.

(3) **Interpreted Language :** Python is an interpreted language, which means when we execute a python

progr
line a
suitab

(4) **Pyth**
comm
for ea

(5) **Port**
Linu

(6) **Obje**
Orie
mean
beha

(7) **Supp**
Pyth
well

(8) **Flex**
libra
and

14. *What is*

Pyth
operation
variable
Python O

(1) Pyth
(2) Pyth
(3) Pyth
(4) Pyth
(5) Pyth
(6) Pyth
(7) Pyth

(1) **Ari**
arit
basi

(a)
(b)
(c)
(d)
(e)
(f)
(g)

program, the interpreter executes the code line by line at a time. This makes debugging easy and thus is suitable for beginners.

(4) **Python is Interactive :** Interactive mode is a command-line shell that gives an immediate response for each statement.

(5) **Portable :** Python supports many platforms like Linux, Windows, MacOS, and Solaris.

(6) **Object-Oriented :** Python supports the Object-Oriented technique of programming which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

(7) **Supports Different Programming Models :** Python supports procedure-oriented programming as well as object-oriented programming.

(8) **Flexible :** Python code can invoke C and C++ libraries and can be called from and C++ programs, and can integrate with Java and .NET components.

14. *What is Python Operator?*

Python operator is a symbol that performs an operation on one or more operands. An operand is a variable or a value on which we perform the operation.

Python Operator falls into 7 categories:

(1) Python Arithmetic Operator
(2) Python Relational Operator
(3) Python Assignment Operator
(4) Python Logical Operator
(5) Python Membership Operator
(6) Python Identity Operator
(7) Python Bitwise Operator

(1) **Arithmetic Operators in Python :** These Python arithmetic operators include Python operators for basic mathematical operations.

(a) Addition (+)
(b) Subtraction (-)
(c) Multiplication (*)
(d) Division (/)
(e) Exponentiation (**)
(f) Floor Division (//)
(g) Modulus (%)

(2) **Python Relational Operator :** Relational Python Operator carries out the comparison between operands. they tell us whether an operand is greater than the other, lesser, equal, or a combination of those.
(a) Less than (<)
(b) Greater than (>)
(c) Less than or equal to (<=)
(d) Greater than or equal to (>=)
(e) Equal to (= =)
(f) Not equal to (!=)

(3) **Python Assignment Operator :** Python assignment operator assigns a value to a variable. It may manipulate the value by a factor before assigning it. We have 8 assignment operators- one plain, and seven for the 7 arithmetic python operators.
(a) Assign (=)
(b) Add and Assign (+=)
(c) Subtract and Assign (-=)
(d) Divide and Assign (/=)
(e) Multiply and Assign (*=)
(f) Modulus and Assign (%=)
(g) Exponent and Assign (**=)
(h) Floor-Divide and Assign (//=)

(4) **Python Logical Operator :** These are conjunctions that you can use to combine more than one condition. We have three Python logical operator and, or, and not that come under python operators.
(a) and Operator in Python
(b) or Operator in Python
(c) not Operator in Python

(5) **Membership Python Operator :** These operators test whether a value is a member of a sequence. The sequence may be a list, a string, or a tuple. We have two membership python operators- 'in' and 'not in'.
(a) in Operator in Python
(b) not in Operator in Python

(6) **Python Identity Operator :** These operators test if the two operands share an identity. We have two identity operators- 'is' and 'is not'.
(a) is Operator in Python
(b) is not Operator in Python

(7) **Python B**
operate bit
(a) Binar
(b) Binar
(c) Binar
(d) Binar
(e) Binar
(f) Binar

15. *Why we use*

The loopi
easy ones. It e
that instead
we can repeat
For example,
numbers then
times, we can
iterations.

Advantages
(1) It provid
(2) Using lo
again ar
(3) Using l
data str

There are t
(1) **For Lo**
need to
conditio
per-test
number
(2) **While**
scenari
iteratio
execute
specifie
a pre-t
(3) **Do-wh**
until
called
to exec
progra

rator : Relational Python
the comparison between
ther an operand is greater
qual, or a combination of

(<=)
to (>=)

rator : Python assignment
e to a variable. It may
factor before assigning it.
one plain, and
ators.

ese are conjunctions
ore than one condition.
al operator and, or, and
perators.

ator : These operators
nber of a sequence. The
ng, or a tuple. We have
tors- 'in' and 'not in'.

ese operators test if
entity. We have two
ot'.

(7) **Python Bitwise Operator :** On the operands, these operate bit by bit.
- (a) Binary AND (&) Operator in Python
- (b) Binary OR (|) Operator in Python
- (c) Binary XOR (^) Operator in Python
- (d) Binary One's Complement (~) in Python
- (e) Binary Left-Shift (<<) Operator in Python
- (f) Binary Right-Shift (>>) in Python

15. *Why we use loops in python?*

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the print statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantages of Loops

(1) It provides code re-usability.
(2) Using loops, we do not need to write the same code again and again.
(3) Using loops, we can traverse over the elements of data structures (array or linked lists).

There are the Following Loop Statements in Python

(1) **For Loop :** The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance.

(2) **While Loop :** The while loop is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

(3) **Do-while Loop :** The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

16. How To Convert A List Into Other Data Types?

Sometimes, we don't use lists as is. Instead, we have to convert them to other types.

Turn A List Into A String

We can use the ' '.join() method which combines all elements into one and returns as a string.

Programming :

```
weekdays = ['sun','mon','tue','wed','thu','fri','sat']
listAsString = ' '.join(weekdays)
print(listAsString)
```

output :

```
sun mon tue wed thu fri sat
```

Turn A List Into A Tuple

Call Python's tuple() function for converting a list into a tuple. This function takes the list as its argument. But remember, we can't change the list after turning it into a tuple because it becomes immutable.

Programming :

```
weekdays = ['sun','mon','tue','wed','thu','fri','sat']
listAsTuple = tuple(weekdays)
print(listAsTuple)
```

output :

```
('sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat')
```

Turn A List Into A Set

Converting a list to a set causes two side-effects.

Set doesn't allow duplicate entries, so the conversion will remove any such item if found.

A set is an ordered collection, so the order of list items would also change.

However, we can use the set() function to convert a list to a set.

Programming :

```
weekdays = ['sun','mon','tue','wed','thu','fri','sat','sun','tue']
listAsSet = set(weekdays)
print(listAsSet)
```

output :

```
set(['wed', 'sun', 'thu', 'tue', 'mon', 'fri', 'sat'])
```

Turn A List Into A Dictionary

In a dictionary, each item represents a key-value pair. So converting a list isn't as straight forward as it were for other data types.

Passing the tuples into the dict() function would finally turn them into a dictionary.

Programming :

```
weekdays = ['sun','mon','tue','wed','thu','fri']
listAsDict = dict(zip(weekdays[0::2], weekdays[1::2]))
print(listAsDict)
output:
{'sun': 'mon', 'thu': 'fri', 'tue': 'wed'}
```

17. *What do you mean by Python literals?*

A literal is a simple and direct form of expressing a value. Literals reflect the primitive type options available in that language. Integers, floating-point numbers, Booleans, and character strings are some of the most common forms of literal.

The Following Literals are Supported by Python : Literals in Python relate to the data that is kept in a variable or constant. There are several types of literals present in Python

String Literals : It's a sequence of characters wrapped in a set of codes. Depending on the number of quotations used, there can be single, double, or triple strings. Single characters enclosed by single or double quotations are known as character literals.

Numeric Literals : These are unchangeable numbers that may be divided into three types: integer, float, and complex.

Boolean Literals : True or False, which signify '1' and '0,' respectively, can be assigned to them.

Special Literals : It's used to categorise fields that have not been generated. 'None' is the value that is used to represent it.

String literals : "halo" , '12345'
Int literals : 0,1,2,-1,-2
Long literals : 89675L
Float literals : 3.14
Complex literals : 12j
Boolean literals : True or False
Special literals : None
Unicode literals : u"hello"
List literals : [], [5, 6, 7]
Tuple literals : (), (9,), (8, 9, 0)
Dict literals : {}, {'x':1}
Set literals : {8, 9, 10}

18. *Comparing between local and global variable in Python?*

A global variable is a variable that is accessible globally. A local variable is one that is only accessible to the current scope, such as temporary variables used in a single function definition.

Example : In the given code

```
q = "I love coffee" # global variable
def f():
   p = "Me Tarzan, You Jane." # local variable
   print p
f()
print q
```

Output :

```
Me Tarzan, You Jane.
I love coffee
```

In the given code, p is a local variable, local to the function f(). q is a global variable accessible anywhere in the module.

19. *Benefits of Python Programming.*

Python is a dynamic-typed language, this means that you don't need to mention the date type of variables during their declaration. It allows to set variables like var1=101 and var2 =" You are an engineer." without any error.

Python supports object orientated programming as you can define classes along with the composition and inheritance. It doesn't use access specifiers like public or private).

Functions in Python are like first-class objects. It suggests you can assign them to variables, return from other methods and pass as arguments.

Developing using Python is quick but running it is often slower than compiled languages. Luckily, Python enables to include the "C" language extensions so you can optimize your scripts.

Python has several usages like web-based applications, test automation, data modeling, big data analytics and much more. Alternatively, you can utilize it as "glue" layer to work with other languages.

20. *How is Python an interpreted language?*

An interpreter takes your code and executes (does) the actions you provide, produces the variables you specify, and performs a lot of behind-the-scenes work to ensure it works smoothly or warns you about issues.

Python is Not an Interpreted or Compiled Language : The implementation's attribute is whether it is interpreted or compiled. Python is a bytecode (a collection of interpreter-readable instructions) that may be interpreted in a variety of ways.

The Source Code is Saved in a Python File : Python generates a set of instructions for a virtual machine from the source code. This intermediate format is known as "bytecode," and it is created by compiling.py source code into .pyc, which is bytecode. This bytecode can then be interpreted by the standard C. Python interpreter or PyPy's JIT (Just in Time compiler).

Python is known as an interpreted language because it uses an interpreter to convert the code you write into a language that your computer's processor can understand. You will later download and utilize the Python interpreter to be able to create Python code and execute it on your own computer when working on a project.

21. *How is Memory managed in Python?*

Memory in Python is managed by Python private heap space. All Python objects and data structures are located in a private heap. This private heap is taken care of by Python Interpreter itself, and a programmer doesn't have access to this private heap.

Python memory manager takes care of the allocation of Python private heap space.

Memory for Python private heap space is made available by Python's in-built garbage collector, which recycles and frees up all the unused memory.

22. *How to remove whitespaces from a string in Python?*

To remove the whitespaces and trailing spaces from the string, Python provides strip([str]) built-in

function. This function returns a copy of the string after removing whitespaces if present. Otherwise returns original string.

Example :

```
string = "  python"
string2 = "    python"
string3.= "      python"
print(string)
print(string2)
print(string3)
print("After stripping all have placed in a sequence:")
print(string.strip())
print(string2.strip())
print(string3.strip())
```

Output :

```
python
  python
    python
After stripping all have placed in a sequence:
python
python
python
```

❑❑

1. *What are the methods that are used in Python Tuple?* **(2023-24)**

In Python, tuples are immutable. Meaning, you cannot change items of a tuple once it is assigned. There are only two tuple methods count() and index() that a tuple object can call.

Python Tuple count() : returns count of the element in the tuple

Python Tuple index() : returns the index of the element in the tuple

2. *Explain Tuples and Unpacking Sequences in Python Data Structure with examples.* **(2023-24)**

Sequence unpacking in python allows you to take objects in a collection and store them in variables for later use. This is particularly useful when a function or method returns a sequence of objects. Now, we will discuss sequence unpacking in python.

A key feature of python is that any sequence can be unpacked into variables by assignment. Consider a list of values corresponding to the attributes of a specific run on the Nike run application. This list will contain the date, pace (minutes), time (minutes), distance (miles) and elevation (feet) for a run :

```
new_run = ['09/01/2020', '10:00', 60, 6, 100]
```

We can unpack this list with appropriately named variables by assignment :

```
date, pace, time, distance, elevation = new_run
```

We can then print the values for these variables to validate that the assignments were correct :

```
print("Date: ", date)
print("Pace: ", pace, 'min')
print("Time: ", time, 'min')
print("Distance: ", distance, 'miles')
print("Elevation: ", elevation, 'feet')
```

The elements in the sequence we are unpacking can be sequences as well.

3. ◆ ***Write the difference between List and Tuple.*** **(2022-23)**

◆ ***Differentiate between the list and tuple.***

Differentiate Between the List and Tuple

List	Tuple
The implication of iterations is Time-consuming	The implication of iterations is comparatively Faster
The list is better for performing operations, such as insertion and deletion.	Tuple data type is appropriate for accessing the elements
Lists consume more memory	Tuple consumes less memory as compared to the list
Lists have several built-in methods	Tuple does not have many built-in methods.
The unexpected changes and errors are more likely to occur	In tuple, it is hard to take place.

4. ***Explain about string data type in python. What is slicing and indexing? Explain about negative indexing. Give four examples of slicing and indexing with negative indexing.*** **(2022-23)**

In Python, a string is a sequence of characters that is enclosed in quotes, either single quotes (') or double quotes ("). Strings are a fundamental data type in Python and are used to represent text and other data that can be represented as a sequence of characters.

Strings are immutable, which means that once a string is created, it cannot be modified in place. However, you can create a new string that is a modified version of the original string using various string manipulation methods.

Indexing and slicing are two common operations used to extract portions of a string. Indexing is used to access individual characters in a string, while slicing is used to extract a portion of a string by specifying a range of indices.

Indexing in Python starts at 0, which means that the first character in a string has index 0, the second character has index 1, and so on. Negative indexing is also supported, which means that the last character in a string

has index -1, the second-to-last character has index -2, and so on.

Here are four examples of slicing and indexing with negative indexing :

```
my_string = "Hello, world!"

# indexing
print(my_string[1])   # prints 'e'
print(my_string[-1])  # prints '!'
print(my_string[7])   # prints 'w'
print(my_string[-9])  # prints 'l'

# slicing
print(my_string[0:5])    # prints 'Hello'
print(my_string[7:])     # prints 'world!'
print(my_string[:5])     # prints 'Hello'
print(my_string[-6:-1]) # prints 'world'
```

In these examples, we first create a string my_string with the value "Hello, world!". We then use indexing to access individual characters of the string, such as my_string[1] which returns the character 'e', and my_string[-1] which returns the last character of the string '!'. We also use slicing to extract portions of the string, such as my_string[0:5] which returns the first five characters of the string 'Hello', and my_string[-6:-1] which returns the second-to-last through last characters of the string 'world'.

5. *Elaborate on the common inbuilt-methods available for list with examples.*

Some most common methods on list are :

(1) append() : The append() method adds an item to the end of the list.

Syntax : list.append(item)

Parameter : item - an item (number, string, list etc.) to be added at the end of the list

Example :

```
currencies = ['Dollar', 'Euro', 'Pound']

# append 'Yen' to the list
currencies.append('Yen')

print(currencies)

# Output : ['Dollar', 'Euro', 'Pound', 'Yen']
```

(2) **clear() :** The clear() method removes all items from the list.
Syntax : list.clear()
Parameter : The clear() method doesn't take any parameters.
Example :

```
# Defining a list
list = [{1, 2}, ('a'), ['1.1', '2.2']]

# clearing the list
list.clear()

print('List:', list)
```

(3) **copy() :** The copy() method returns a shadow copy of the list.
Syntax : new_list = list.copy()
Parameter : The copy() method doesn't take any parameters.
Example :

```
# mixed list
prime_numbers = [2, 3, 5]

# copying a list
numbers = prime_numbers.copy()

print('Copied List:', numbers)

# Output: Copied List: [2, 3, 5]
```

(4) **count() :** The count() method returns the number of times the specified element appears in the list.
Syntax : list.count(element)
Parameter : element - the element to be counted
Example :

```
# create a list
numbers = [2, 3, 5, 2, 11, 2, 7]

# check the count of 2
count = numbers.count(2)

print('Count of 2:', count)

# Output: Count of 2: 3
```

(5) **index() :** The index() method returns the index of the specified element in the list.

Syntax : list.index(element, start, end)

Parameter : element - the element to be searched
start (optional) - start searching from this index
end (optional) - search the element up to this index

Example :

```
animals = ['cat', 'dog', 'rabbit', 'horse']

# get the index of 'dog'
index = animals.index('dog')


print(index)

# Output: 1
```

(6) **remove() :** The remove() method removes the first matching element (which is passed as an argument) from the list.

Syntax : list.remove(element)

Parameter : The remove() method takes a single element as an argument and removes it from the list.

If the element doesn't exist, it throws ValueError: list.remove(x): x not in list exception.

Example :

```
# create a list
prime_numbers = [2, 3, 5, 7, 9, 11]

# remove 9 from the list
prime_numbers.remove(9)

# Updated prime_numbers List
print('Updated List: ', prime_numbers)

# Output: Updated List:  [2, 3, 5, 7, 11]
```

(7) **sort() :** The sort() method sorts the items of a list in ascending or descending order.

Syntax : list.sort(key=..., reverse=...)

Parameter : By default, sort() doesn't require any extra parameters. However, it has two optional parameters:

reverse : If True, the sorted list is reversed (or sorted in Descending order)

key : function. that serves as a key for the sort comparison

Example :

```
prime_numbers = [11, 3, 7, 5, 2]

#sorting the list in ascending order
prime_numbers.sort()

print(prime_numbers)
```

Output : [2, 3, 5, 7, 11]

6. *Explain about matrix and it's common methods (provided by numpy) using examples.*

Two matrices are initialized by value

```
x = numpy.array([[1, 2], [4, 5]])
y = numpy.array([[7, 8], [9, 10]])
```

add() : add elements of two matrices.

```
#  add()is used to add matrices
print ("Addition of two matrices: ")
print (numpy.add(x,y))
```

```
Addition of two matrices:
[[ 8 10]
[13 15]]
```

subtract() : subtract elements of two matrices.

```
# subtract()is used to subtract matrices
print ("Subtraction of two matrices : ")
print (numpy.subtract(x,y))
```

```
Subtraction of two matrices :
[[-6 -6]
[-5 -5]]
```

divide() : divide elements of two matrices.

```
# divide()is used to divide matrices
print ("Matrix Division : ")
print (numpy.divide(x,y))
```

```
Matrix Division :
[[0.14285714 0.25 ]
[0.44444444 0.5 ]]
```

multiply() : multiply elements of two matrices.

```
print ("Multiplication of two matrices: ")
print (numpy.multiply(x,y))
```

```
Multiplication of two matrices:
```



```
[[ 7 16]
[36 50]]
```

dot() : It per
wise multipli

```
print ("Tl
print (nu
The pro
[[25 28]
[73 82]]
```

sqrt() : squ

```
print ("
print (
squar
[[1. 1
[2. 2.
```

sum(x,a
argumen
the colu

```
pri
pri
pri
pri
pr
pr
```

The sur

```
3
T
[
```

"T" :

```
[[ 7 16]
[36 50]]
```

dot() : It performs matrix multiplication, does not element wise multiplication.

```
print ("The product of two matrices : ")
print (numpy.dot(x,y))
The product of two matrices :
[[25 28]
[73 82]]
```

sqrt() : square root of each element of matrix.

```
print ("square root is : ")
print (numpy.sqrt(x))
square root is :
[[1. 1.41421356]
[2. 2.23606798]]
```

sum(x,axis) : add to all the elements in matrix. Second argument is optional, it is used when we want to compute the column sum if axis is 0 and row sum if axis is 1.

```
print ("The summation of elements : ")
print (numpy.sum(y))
print ("The column wise summation  : ")
print (numpy.sum(y,axis=0))
print ("The row wise summation: ")
print (numpy.sum(y,axis=1))
The summation of elements :
34
The column wise summation :
[16 18]
The row wise summation:
[15 19]
```

"T" : It performs transpose of the specified matrix.

```
# using "T" to transpose the matrix
print ("Matrix transposition : ")
print (x.T)
Matrix transposition :
[[1 4]
[2 5]]
```

[B.8]

7. *Explain about the tuple and it's operations.*

Python Tuple is a collection of objects separated by commas. In some ways, a tuple is similar to a list in terms of indexing, nested objects, and repetition but a tuple is immutable, unlike lists which are mutable.

Creating Tuples in Python
To create a tuple we will use ()

Example
```
var = ("this", "is", "it")
print(var)
('this', 'is', 'it')
```

(1) Accessing Values in Tuples in Python : This can be done using positive or negative Index

Using square brackets we can get the values from tuples in Python.

```
var = ("this", "is", "it")
print("Value in Var[0] = ", var[0])
print("Value in Var[1] = ", var[1])
print("Value in Var[-2] = ", var[-2])
print("Value in Var[-1] = ", var[-1])

Value in Var[0] =  this
Value in Var[1] =  is
Value in Var[-2] = is
Value in Var[-1] =  it
```

(2) Concatenation of Tuples in Python : To concatenate the Python tuple we will use plus operators(+).

```
# Code for concatenating 2 tuples
tuple1 = (0, 1, 2, 3)
tuple2 = ('python', 'geek')
# Concatenating above two
print(tuple1 + tuple2)
(0, 1, 2, 3, 'python', 'geek')
```

(3) Nesting of Tuples in Python :

```
# Code for creating nested tuples

tuple1 = (0, 1, 2, 3)
```

```
tuple2 = ('python', 'geek')
tuple3 = (tuple1, tuple2)
print(tuple3)
((0, 1, 2, 3), ('python', 'geek'))
```

(4) Repetition Tuples in Python :

```
# Code to create a tuple with repetition

tuple3 = ('python',)*3
print(tuple3)
```

(5) Slicing Tuples :

```
# code to test slicing

tuple1 = (0 ,1, 2, 3)
print(tuple1[1:])
print(tuple1[::-1])
print(tuple1[2:4])

(1, 2, 3)
(3, 2, 1, 0)
(2, 3)
```

(6) Tuples are immutable so update and delete is not allowed

```
# code to test that tuples are immutable

tuple1 = (0, 1, 2, 3)
tuple1[0] = 4
print(tuple1)

# Code for deleting a tuple

tuple3 = ( 0, 1)
del tuple3
print(tuple3)
```

TypeError: 'tuple' object does not support item assignment

8. *Explain in detail the slicing functionality for list.*

In Python, list slicing is a common practice and it is the most used technique for programmers to solve efficient problems. Consider a python list, In-order to access a range of elements in a list, you need to slice a list. One way to do this is to use the simple slicing operator i.e. colon(:)

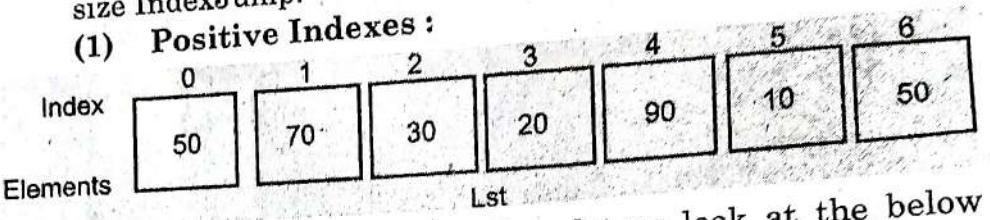
With this operator, one can specify where to start the slicing, where to end, and specify the step. List slicing returns a new list from the existing list.

Syntax :

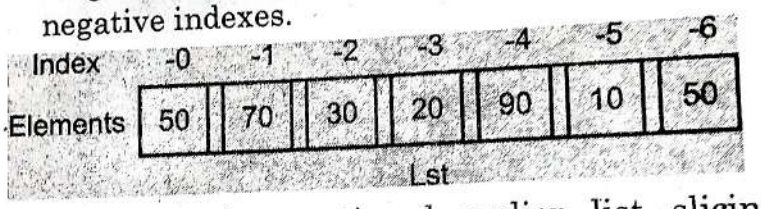
List[Initial : End : IndexJump]

If Lst is a list, then the above expression returns the portion of the list from index Initial to index End, at a step size IndexJump.

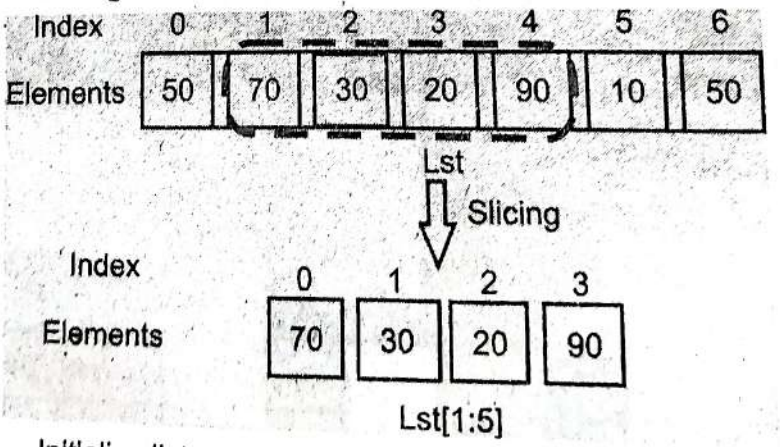
(1) Positive Indexes :



(2) Negative Indexes : Now, let us look at the below diagram which illustrates a list along with its negative indexes.



(3) Slicing : As mentioned earlier list slicing is a common practice in Python and can be used both with positive indexes as well as negative indexes. The below diagram illustrates the technique of list s licing :




```
Initialize list
Lst = [50, 70, 30, 20, 90, 10, 50]

Display list
print(Lst[1:5])
```

9. *Write the python code for printing the matrix in spiral.*

```
def printSpiralOrder(mat):

    # base case
    if not mat or not len(mat):
        return

    top = left = 0
    bottom = len(mat) - 1
    right = len(mat[0]) - 1

    while True:
        if left > right:
            break

        # print top row
        for i in range(left, right + 1):
            print(mat[top][i], end=' ')
        top = top + 1

        if top > bottom:
            break

        # print right column
        for i in range(top, bottom + 1):
            print(mat[i][right], end=' ')
        right = right - 1

        if left > right:
            break

        # print bottom row
        for i in range(right, left - 1, -1):
            print(mat[bottom][i], end=' ')
        bottom = bottom - 1

        if top > bottom:
            break

        # print left column
        for i in range(bottom, top - 1, -1):
            print(mat[i][left], end=' ')
        left = left + 1


if __name__ == '__main__':
```

```
mat = [
   [1, 2, 3, 4, 5],
   [16, 17, 18, 19, 6],
   [15, 24, 25, 20, 7],
   [14, 23, 22, 21, 8],
   [13, 12, 11, 10, 9]
]

printSpiralOrder(mat)
```

10. *Change all elements of row 'i' and column 'j' in a matrix to 0 if cell '(i, j)' has a value of 0*

Function to change all elements of row '*x*' and column '*y*' to -1

```
def changeRowColumn(mat, M, N, x, y):

    for j in range(N):
        if mat[x][j]:
            mat[x][j] = -1

    for i in range(M):
        if mat[i][y]:
            mat[i][y] = -1


# Function to convert the matrix
def convert(mat):

    # base case
    if not mat or not len(mat):
        return

    # `M × N` matrix
    (M, N) = (len(mat), len(mat[0]))

    # traverse the matrix
    for i in range(M):
        for j in range(N):
            if mat[i][j] == 0:    # cell `(i, j)` has value 0
                # change each non-zero element in row `i` and column `j` to -1
                changeRowColumn(mat, M, N, i, j)

    # traverse the matrix once again and replace cells having
    # value -1 with 0
    for i in range(M):
```

```
            for j in range(N):
                if mat[i][j] == -1:
                    mat[i][j] = 0


if __name__ == '__main__':

    mat = [
        [1, 1, 0, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 0, 1, 1],
       [1, 1, 1, 1, 1],
        [0, 1, 1, 1, 1]
    ]

    # convert the matrix
    convert(mat)

    # print matrix
    for r in mat:
        print(r)
```

11. *How to iterate over multiple lists at the same time? Explain with an example.*

You can zip() lists and then iterate over the zip object. A zip object is an iterator of tuples.

Below we iterate over 3 lists simultaneously and interpolate the values into a string.

```
name = ['Snowball', 'Chewy', 'Bubbles', 'Gruff']
animal = ['Cat', 'Dog', 'Fish', 'Goat']
age = [1, 2, 2, 6]
z = zip(name, animal, age)
z #=> <zip at 0x111081e48>
for name,animal,age in z:
    print("%s the %s is %s" % (name, animal, age))

#=> Snowball the Cat is 1
#=> Chewy the Dog is 2
#=> Bubbles the Fish is 2
#=> Gruff the Goat is 6
```

12. *'The python list is mutable'. What does this statement mean?*

The list data structure in python is mutable. This means we an

Notice in the code below how the value associated with the same identifier in memory has not changed.

```
x = [1]
print(id(x),':',x) #=> 4501046920 : [1]
x.append(5)
x.extend([6,7])
print(id(x),':',x) #=> 4501046920 : [1, 5, 6, 7]
```

13. *Comment on the homogeneity of the list?*

Different types of objects can be mixed together in a list. This means the list data type can be non-homogeneous. For example we can have the following list

```
a = [1,'a',1.0,[]]
a #=> [1, 'a', 1.0, []]
```

14. *What is the difference between append and extend?*

Following discusses the differences between the 2 methods

append()

It adds an object to the end of a list.

```
a = [1,2,3]
a.append(4)
a #=> [1, 2, 3, 4]
```

This also means appending a list adds that whole list as a single element, rather than appending each of its values.

```
a.append([5,6])
a #=> [1, 2, 3, 4, [5, 6]]
```

.extend()

This method adds each value from a 2nd list as its own element. So extending a list with another list combines their values.

```
b = [1,2,3]
b.extend([5,6])
b #=> [1, 2, 3, 5, 6]
```

15. *Do python lists store values or pointers?*

Python lists don't store values themselves. They store pointers to values stored elsewhere in memory. This allows lists to be mutable.

Here we initialize values 1 and 2, then create a list including the values 1 and 2.

```
print( id(1) ) #=> 4438537632
print( id(2) ) #=> 4438537664
a = [1,2,3]
print( id(a) ) #=> 4579953480
print( id(a[0]) ) #=> 4438537632
print( id(a[1]) ) #=> 4438537664
```

Notice how the list has its own memory address. But 1 and 2 in the list point to the same place in memory as the 1 and 2 we previously defined.

16. *What does "del" keyword do in regards to list?*

'del' removes an item from a list given its index.

Here we'll remove the value at index 1.

```
a = ['w', 'x', 'y', 'z']
a #=> ['w', 'x', 'y', 'z']
del a[1]
a #=> ['w', 'y', 'z']
```

Notice how del does not return the removed element.

17. *What is the difference between "remove" and "pop" keywords? Explain using a list as an example.*

'remove()' removes the first instance of a matching object. Below we remove the first b.

```
a = ['a', 'a', 'b', 'b', 'c', 'c']
a.remove('b')
a #=> ['a', 'a', 'b', 'c', 'c']
```

.pop() removes an object by its index.

The difference between pop and del is that pop returns the popped element. This allows using a list like a stack.

```
a = ['a', 'a', 'b', 'b', 'c', 'c']
a.pop(4) #=> 'c'
a #=> ['a', 'a', 'b', 'b', 'c']
```

By default, pop removes the last element from a list if an index isn't specified.

18. *Remove elements in a list after and before a specific index using 'slice'.*

Using the slice syntax, we can return a new list with only the elements up to a specific index.

```
li = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,10]
li[:10]
#=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The slice syntax can also return a new list with the values after a specified index.

```
li = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,10]
li[15:]
#=> [16, 17, 18, 19, 10]
```

19. *Write a Python program to sum all the items in a list, without using the sum in-built method.*

```
def sum_list(items):
    sum_numbers = 0
    for x in items:
        sum_numbers += x
    return sum_numbers
print(sum_list([1,2,-8]))
```

20. *Write a Python program to get the largest number from a list.*

```
def max_num_in_list( list ):
    max = list[ 0 ]
    for a in list:
        if a > max:
            max = a
    return max
print(max_num_in_list([1, 2, -8, 0]))
```

❑❑

DICTIONARY AND FUNCTION

1. ***Discuss list and dictionary data structure with example for each. Write a python program to accept n numbers and store them in a list. Then print the list without ODD numbers in it.*** **(2023-24)**

Lists in Python

Lists are just like arrays, declared in other languages. Lists need not be homogeneous always which makes it a most powerful tool in Python. A single list may contain data types like Integers, Strings, as well as Objects. Lists are mutable, and hence, even after their creation.

Example : In this example, we will see how to create a simple list as well as a multidimensional list in Python and access its values using the list index.

```
# Creating a List with
# the use of multiple values
List = ["Pkc", "For", "Pkc"]
print("List containing multiple values: ")
print(List[0])
print(List[2])
# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Pkc', 'For'] , ['Pkc']]
print("\nMulti-Dimensional List: ")
print(List)
Output: List containing multiple values:
Pkc
Pkc
Multi-Dimensional List:
[['Pkc', 'For'], ['Pkc']]
```

Dictionary in Python : Python Dictionary on the other hand is an unordered collection of data values, used to store data values like a map, unlike other Data Types that hold only a single value as an element, Dictionary holds a key:value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon (:) whereas each key is separated by a 'comma'.

Example : In this example, we will see how to create a simple dictionary with similar key types as well as a dictionary with mixed key types.

(1)
```
# Creating a Dictionary
# with Integer Keys
Dict = {1: 'Pkc', 2: 'For', 3: 'Pkc'}
print("Dictionary with the use of Integer Keys: ")
print(Dict)
# Creating a Dictionary
# with Mixed keys
Dict = {'Name': 'Pkc', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

Output :

Dictionary with the use of Integer Keys: {1: 'Pkc', 2: 'For', 3: 'Pkc'}

Dictionary with the use of Mixed Keys: {1: [1, 2, 3, 4], 'Name': 'Pkc'}

(2)
```
#print the list without ODD numbers :
for i in range(1, 11, 2):
print ('This will print only odd numbers:', i)
```

Output :

This will print only odd numbers:
1 This will print only odd numbers:
3 This will print only odd numbers:
5 This will print only odd numbers:
7 This will print only odd numbers:
9 This will print only odd numbers:

2. ◆ ***Explain about functions in python? Explain the type of arguments used in python function.*** **(2023-24)**

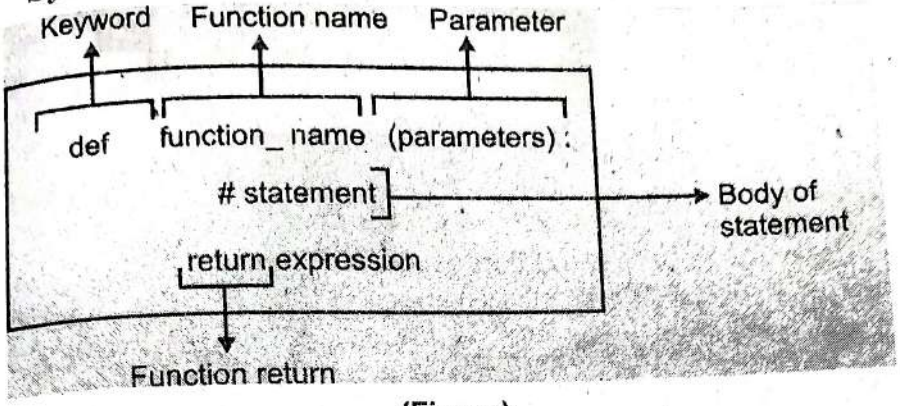
◆ ***Explain about functions in python. Write a python program using functions to find the roots of a quadratic equation.*** **(2022-23)**

◆ ***Define function in Python***

Python Functions is a block of statements that return the specific task.

The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

Syntax : Python Functions



(Figure)

Python Program Using Functions to Find the Roots of a Quadratic Equation

Here's an example of a Python program that uses a function to find the roots of a quadratic equation :

```
import math
    """Compute the roots of an equation of the form
ax^2 + bx + c = 0 ."""
    def quadratic_roots(a, b, c):
        discriminant = b**2 - 4*a*c
        if discriminant < 0:
            return None
        elif discriminant == 0:
            return -b / (2*a)
        else:
            root1 = (-b + math.sqrt(discriminant)) / (2*a)
            root2 = (-b - math.sqrt(discriminant)) / (2*a)
            return root1, root2

    # Example usage
    a = 1
    b = -5
    c = 6

    roots = quadratic_roots(a, b, c)
    if roots is None:
    print("No real roots")
    elifisinstance(roots, float):
    print(f"One real root: {roots}")
    else:
    print(f"Two real roots: {roots[0]}, {roots[1]}")
```

In this example, we first import the math module, which provides functions for mathematical operations like square roots. We then define a function quadratic_roots that takes three parameters a, b, and c, which represent the coefficients of the quadratic equation.

Inside the function, we first compute the discriminant of the quadratic equation using the formula $b^2 - 4ac$. We then use a series of if and elif statements to handle the different cases based on the value of the discriminant. If the discriminant is negative, there are no real roots, so we return None. If the discriminant is zero, there is one real root, so we return that value. If the discriminant is positive, there are two real roots, so we compute and return those values.

Finally, it is demonstrate how to use the quadratic_roots function by setting the coefficients of a quadratic equation $a = 1$, $b = -5$, and $c = 6$, and calling the function with those values. We then check the result and print out a message indicating whether there are no real roots, one real root, or two real roots.

Types of Arguments in Python Functions

Python functions offer several ways to handle arguments :

(1) **Required Arguments:** These are arguments that must be provided when calling the function.

```
def greet(name):
    print("Hello,", name + "!")
greet("Pkc") # Correct call
    # greet() # TypeError: greet() missing 1 required positional argument: name
```

(2) **Default Arguments :** You can assign default values to arguments. If a call omits the argument, the default value is used.

```
def get_sum(num1, num2=10):
    """This function gets the sum of two numbers."""
    sum = num1 + num2
    return sum
    print(get_sum(5)) # Output: 15 (uses default num2=10)
print(get_sum(5, 20)) # Output: 25
```

(3) **Variable-Length Arguments (*args) :** Use *args to accept a variable number of arguments as a tuple.

```
def calculate_average(*numbers):
    """This function calculates the average of any number of numbers."""
    total = sum(numbers)
    average = total / len(numbers)
    return average
print(calculate_average(1, 2, 3)) # Output: 2.0
print(calculate_average(10, 20, 30, 40)) # Output: 25.0
```

(4) Keyword Arguments (kwargs) :** Use **kwargs to accept a variable number of keyword arguments as a dictionary.

```
def user_profile(first_name, last_name, age=None, city=None):
    """This function creates a user profile."""
    profile = {
    "first_name": first_name,
    "last_name": last_name,
    }
    if age:
    profile["age"] = age
    if city:
    profile["city"] = city
    return profile
profile1 = user_profile("Alice", "Smith")
    profile2 = user_profile("Bob", "Johnson", age=30, city="New York") print(profile1) # Output: {'first_name': 'Alice', 'last_name': 'Smith'}
    print(profile2) # Output: {'first_name': 'Bob', 'last_name': 'Johnson', 'age': 30, 'city': 'New York'}
```

By effectively using functions and different argument types, you can create well-organized, reusable, and versatile Python code.

3. *What is Python programming cycle? Mention what are the rules for local and global variable in python. How can you share global variables across modules?* **(2022-23)**

Python programming cycle refers to the process of writing and executing Python code. It typically consists of the following steps :

(1) Planning : This involves defining the problem you want to solve, designing the program structure, and identifying the algorithms and data structures to be used.

(2) **Coding :** This is the process of writing the actual Python code based on the program design.

(3) **Testing :** This involves running the program with sample input data and verifying that the program produces the expected output.

(4) **Debugging :** If there are errors or unexpected results during testing, the code needs to be debugged by identifying and fixing the problem.

(5) **Maintenance :** After the program is working correctly, it may need to be updated or modified over time to accommodate changing requirements or to fix bugs.

Rules for Local and Global Variables in Python are

(1) A local variable is a variable that is defined inside a function. It can only be accessed within the function and is destroyed when the function returns.

(2) A global variable is a variable that is defined outside of any function or class. It can be accessed from anywhere in the program.

(3) If a local variable has the same name as a global variable, the local variable will override the global variable within the scope of the function where it is defined.

(4) If you want to modify a global variable from within a function, you need to use the global keyword to indicate that you are referring to the global variable, not a local variable with the same name.

Here's an example of how to use the global keyword to modify a global variable from within a function :

```
count = 0

def increment():
    global count
    count += 1

increment()
print(count)  # Output: 1
```

Finally, to share global variables across modules, you can use the global keyword as described above, or you can

use the import statement to access the global variables from another module. When you import a module, you can access its global variables by prefixing them with the module name, like this :

```
# module1.py
count = 0

# module2.py
import module1

def increment():
    module1.count += 1

increment()
print(module1.count)  # Output: 1
```

In this example, the module2 module imports the module1 module and then modifies its count global variable by using the module name as a prefix.

4. ◆ ***What is dictionary in python? Write a python program to convert a given dictionary into a list of lists.*** **(2022-23)**
   ◆ ***What is a Python dictionary?***

Python Dictionary

Python Dictionary objects are data types that are enclosed in curly braces '{}' and have key and value pairs and each pair is separate by a comma.

Dictionary is Mapped : Meaning since it has key and value pair, a meaningful key can save a lot of trouble for coders, like using an address key to save all the addresses, an id key for all id's and so on

Python Program to Convert a given Dictionary into a List of Lists : Here's an example of a dictionary :

```
my_dict = {'apple': 3, 'banana': 6, 'orange': 2}
```

To convert a dictionary into a list of lists, we can use a list comprehension that iterates over the dictionary and appends a new list containing the key and value of each item to the resulting list.

Here's the Python code to convert a dictionary into a list of lists :

```
my_dict = {'apple': 3, 'banana': 6, 'orange': 2}
my_list = [[k, v] for k, v in my_dict.items()]
print(my_list)
```

In this example, we first create a dictionary my_dict with three key-value pairs. We then create a list comprehension that iterates over the items in the dictionary using the items() method. For each key-value pair, we append a new list containing the key and value to the resulting list my_list. Finally, we print out my_list, which will be a list of lists containing the key-value pairs of the original dictionary :

```
[['apple', 3], ['banana', 6], ['orange', 2]]
```

Note that the order of the key-value pairs in the resulting list may not be the same as the order in the original dictionary, since dictionaries are unordered collections.

5. *What is lambda function? Explain the features of lambda function. Write a python program to calculate the average value of the numbers in a given tuple of tuples using lambda.* (2022-23)

In Python, a lambda function is a small anonymous function that can have any number of arguments, but can only have one expression. The expression is executed and the result is returned. Lambda functions are typically used as a shorthand for defining simple functions that are only needed once.

Here are Some Key Features of Lambda Functions

(1) Lambda functions are defined using the lambda keyword, followed by a comma-separated list of arguments (if any), and then a single expression that is evaluated and returned.

(2) Lambda functions are often used in combination with higher-order functions, such as map(), filter(), and reduce(), which take other functions as arguments.

(3) Because lambda functions are anonymous, they can be defined inline as part of an expression or function call, without needing to give them a separate name.

Here's an example of a Python program that uses a lambda function to calculate the average value of the numbers in a given tuple of tuples :

```
data = ((1, 2, 3), (4, 5, 6), (7, 8, 9))
Define a lambda function that takes a tuple of numbers and returns their average
average = lambda nums: sum(nums) / len(nums)
Use map() and average() to calculate the average value of each tuple in the data
averages = list(map(average, data))
Print the results :
print(averages)
```

In this example, we start by defining a tuple of tuples called data, which contains three tuples of three numbers each. We then define a lambda function called average that takes a tuple of numbers and returns their average value.

To calculate the average value of each tuple in the data tuple, we use the map() function, which applies the average function to each tuple in the data tuple. We then convert the resulting iterator to a list using the list() function.

Finally, we print the resulting list of average values, which should be [2.0, 5.0, 8.0], indicating that the average value of the first tuple is 2.0, the average value of the second tuple is 5.0, and the average value of the third tuple is 8.0.

6. *Describe Python Dictionary Methods.*

(1) **Copy() :** The copy method is used for copying the contents of one dictionary to another dictionary, So this will include copying of keys and values of the first dict into another given dictionary.

```
Bikes={'Bike_1':'CT100','Bike_2':'HeroHonda','Bike_3':'Yamaha'}
Two_wheelers = Bikes.copy()
    print("All two wheelers in the market: ", Two_wheelers)
```

(2) **get() :** The get() method is used to get the given key's value in a dict variable.

```
Bikes={'Bike_1':'CT100','Bike_2':'HeroHonda','Bike_3':'Yamaha'}
print('The second bike:',Bikes.get('Bike_2'))
```

(3) **Update() :** Two key processes can be achieved by using this update() method. So first, this involves the process of revising an existing key-value pair in the dictionary, whereas the other one is the process of inserting a fresh entry into the dictionary.

```
Bikes={'Bike_1':'CT100','Bike_2':'HeroHonda','Bike_3':'Yamah
a'}
Bikes.update({'Bike_4': 'TVS'})
print("List of bikes in the market after update 1:", Bikes)
print("")
Bikes.update({'Bike_3': 'Hero-Honda'})
print("List of bikes in the market after update 2:", Bikes)
```

(4) **Items() :** For displaying each and every key-value pair of the python dictionary, the method called items() has been used.

```
Bikes={'Bike_1':'CT100','Bike_2':'HeroHonda','Bike_3':'Yamah
a'}
print('All bikes:',Bikes.items())
```

(5) **keyes() :** For displaying the entire set of keys in the dictionary, the keyes() method is used.

```
Bikes={'Bike_1':'CT100','Bike_2':'HeroHonda','Bike_3':'Yamah
a'}
print('All bikes:',Bikes.keys())
```

(6) **sort() :** For ordering or sorting the key value pairs in a dictionary, the sort method has been used.

```
Bikes={'Bike_2':'CT100','Bike_1':'HeroHonda','Bike_3':'Yamah
a'}
print('All bikes:',sorted(Bikes.items()))
```

(7) **values() :** The values method is used to display all the values in the dictionary.

```
Bikes={'Bike_1':'CT100','Bike_2':'HeroHonda','Bike_3':'Yamah
a'}
print('The second bike:',Bikes.values())
```

(8) **len() :** For estimating the count of total key value pairs in a dictionary data type, the len() method can be wisely used.

```
Bikes={'Bike_2':'CT100','Bike_1':'HeroHonda','Bike_3':'Yamah
a'}
print('Overall number of bikes:',len(Bikes))
```

(9) **str() :** For converting a dictionary into a string, the str() method can be used. This process is more of a typecasting kind of method. So it involves converting an item of one data type to a different data type.

```
Bikes={'Bike_2':'CT100','Bike_1':'HeroHonda','Bike_3':'Yamah
a'}
Bikes_str = str(Bikes)
print('Format of Bikes datatype:',type(Bikes_str))
```

7. *Explain Pass by reference vs value in Python?*

When you give function parameters via reference, you're just passing references to values that already exist. When you pass arguments by value, on the other hand, the arguments become independent copies of the original values.

Pass by Value : Any other operation performed will not have any effect on the original value as the argument passed to the function is copied. It only changes the value of the copy created within the function.

Input :

```
def function(int):
  int+=100
  print("Inside function call ",int)

int=100
print("Before function call ",int)
function(int)
print("After function call ",int)
```

Here, a copy of the argument is created, and changes are made to that copy such that the original value is not affected. As a result, the original value is printed after the function call.

Output :

Before function call 100
Inside function call 200
After function call 100

Pass by reference

This method passes a reference to the original arguments, and any operation performed on the parameter affects the actual value. It alters the value in both function and globally.

Input :

```
def function(int):
  int.append('B')print("Inside function call",int)
  int=['A']
```

```
print("Before function call",lnt)
function(lnt)
print("After function call",lnt)
```

When a list is used, its reference is supplied into the function, and any modifications to the list have an effect even after the method has been called.

Output :

```
Before function call ['A']
Inside function call ['A', 'B']
After function call ['A', 'B']
```

8. *Explain*args and **kwargs in python?*

*args is used when we are not sure about the number of arguments to be passed to a function.

```
def add(*num):
    sum = 0
    for val in num:
        sum = val+sum
    print(sum)
add(4,5)
add(7,4,6)
add(10,34,23)
```

```
9
17
57
```

**kwargs is used when we want to pass a dictionary as an argument to a function.

```
def intro(**data):
    print("\nData type of argument:",type(data))
    for key, value in data.items():
        print("{} is {}".format(key,value))
intro(name="alex",Age=22, Phone=1234567890)
intro(name="louis",Email="a@gmail.com",Country="Wakanda",
Age=25)
```

```
Data type of argument: <class 'dict'>
name is alex
Age is 22
Phone is 1234567890

Data type of argument: <class 'dict'>
name is louis
Email is a@gmail.com
Country is Wakanda
Age is 25
```


9. *Explai*
(1) *Ra*
(2) *Lo*
(3) *M*
(4) *Fi*
(5) *R*

Range
from t
There
to defi

Lam
func
argu
calle

Ma
app
obj

els

'oc
Fi
it

9. ***Explain with an example?***
   (1) ***Range function***
   (2) ***Lambda function***
   (3) ***Map Function***
   (4) ***Filter***
   (5) ***Reduce***

Range : range function returns a number of sequences from the starting point to an endpoint. range(start, end) There is one more third parameter which is the step used to define steps in the range.

```
for i in range(5):  ## number
  print(i)
> 0,1,2,3,4
for i in range(1,5):  ##(start,end)
  print(i)
> 1,2,3,4
for i in range(0,5,2):  ## (start,end,step)
  print(i)
>0,2,4
```

Lambda Function : Lambda function is a single line function with no name, which can have n number of arguments but it can only have on expression. It is also called an anonymous function.

```
a = lambda x,y : x+y
print(a(5,6))
> 11
```

Map Function : a map function returns a map object after applying a certain function to each item of the iterable object.

```
Lst = [1,2,3,4,5,6,7,8,9,10]
def even_or_odd(num)
If num%2==0:
Return "even"
else :
Return "odd"
List(map(even_or_odd, lst))
---------------------------------------
['odd', 'even', 'odd', 'even', 'odd', 'even', 'odd', 'even', 'odd', 'even']
```

Filter : a filter function used to filter values from an iterable based on some condition.

```
lst = [1,2,3,4,5,6,7,8,9,10] ##iterable
def even(num):
    if num%2==0:
        return num
list(filter(even,lst))  ## filter all even numbers
```

```
[2, 4, 6, 8, 10]
```

Reduce : The reduce () function accepts a function and a sequence and returns a single value after calculation.

```
from functools import reduce
    a = lambda x,y:x+y
    print(reduce(a,[1,2,3,4]))
     10
```

10. *Explain with an example?*
- **(1) *Enumerate***
- **(2) *Split Function***
- **(3) *Rstrip() Function***
- **(4) *Zip Function***

Enumerate : the enumerate() function is used to add a counter to an iterable object. enumerate() function accepts sequential indexes starting from zero.

For example,

```
    subjects = ('Python', 'Java', 'C++')
for i, subject in enumerate(subjects):
    print(i, subject)
```

Output:

```
    0 Python
    1 Java
    2 C++
```

Split Function : The split() function can be used to separate large strings into smaller ones or rather sub-strings. We can either use a separator for the split else it will use the space as one by default.

For example,

```
    str_name = 'alice bob charlie'
    str_name.split(" ")
    str_name.split()
```

The output will be -

```
    ['alice', 'bob', 'charlie']
    ['alice', 'bob', 'charlie']
```

Rstrip() Function : Rstrip() function can be used to duplicate the string but it leaves out the whitespace

characters from the end. This function excludes the characters from the right end as per the argument value i.e, a string mentioning the group of characters to get excluded.

The signature of the rstrip() is:

```
str.rstrip([char sequence/pre>
```

For example,

```
str_name = 'Python'
The white spaces are excluded
print(str_name.rstrip())
```

Zip Function : The zip function takes iterables, aggregates them in a tuple, and returns it. The syntax of the zip() function is zip(*iterables)

For example,

```
numbers = [1, 2, 3]
string = ['one', 'two', 'three']
result = zip(numbers,string)
print(set(result))
The output will be -
{(3, 'three'), (2, 'two'), (1, 'one')}
```

11. *How would you check if a key exists in a Python dictionary?*

To see if a key exists in a Python dictionary. There are two built-in functions in Python :

has_key() : The has_key method returns true if a given key is available in the dictionary; otherwise, it returns false.

```
Fruits = {'a': "Apple", 'b':"Banana", 'c':"Carrot"}
key_to_lookup = 'a'
if Fruits.has_key(key_to_lookup):
  print "Key exists"
else:
  print "Key does not exist"
```

Output :

```
Key exists
```

if-in statement : This approach uses the if-in statement to check whether or not a given key exists in the dictionary.

```
Fruits = {'a': "Apple", 'b':"Banana", 'c':"Carrot"}
key_to_lookup = 'a'
if key_to_lookup in Fruits:
  print "Key exists"
```

```
    else:
      print "Key does not exist"
```

Output

```
    Key exists
```

12. *How would you sort a dictionary in Python?*

Sorted() syntax : This method takes one mandatory and two optional arguments :

Data (Mandatory) : The data to be sorted. We will pass the data we retrieved using one of the above methods.

Key (Optional) : A function (or criteria) based on which we would like to sort the list. For example, the criteria could be sorting strings based on their length or any other arbitrary function. This function is applied to every element of the list and the resulting data is sorted. Leaving it empty will sort the list based on its original values.

Reverse (Optional) : Setting the third parameter as true will sort the list in descending order. Leaving these empty sorts in ascending order.

We can sort this type of data by either the key or the value and this is done by using the sorted() function.

First, we need to know how to retrieve data from a dictionary to be passed on to this function.

There are three basic ways to get data from a dictionary :

Dictionary.keys() : Returns only the keys in an arbitrary order.

```
dict = {}
dict['1'] = 'apple'
dict['3'] = 'orange'
dict['2'] = 'pango'
lst = dict.keys()
Sorted by key
print("Sorted by key: ", sorted(lst))
```

Output :

```
Sorted by key:  ['1', '2', '3']
```

Dictionary.values() : Returns a list of values.

```
dict = {}
dict['1'] = 'apple'
dict['3'] = 'orange'
dict['2'] = 'pango'
lst = dict.values()
```

```
#Sorted by value
print("Sorted by value: ", sorted(lst))
```

Output :

Sorted by value: ['apple', 'orange', 'pango']

Dictionary.items() : Returns all of the data as a list of key-value pairs.

```
dict = {}
dict['1'] = 'apple'
dict['3'] = 'orange'
dict['2'] = 'strawberry'
lst = dict.items()
# Sorted by key
print("Sorted by key: ", sorted(lst, key = lambda x : x[0]))
#Sorted by value
print("Sorted by value: ", sorted(lst, key = lambda x : x[1]))
Output
Sorted by key : [('1', 'apple'), ('2', 'strawberry'), ('3', 'orange')]
Sorted by value : [('1', 'apple'), ('3', 'orange'), ('2', 'strawberry')]
```

13. *Implement breadth first search (BFS) in Python with explanation.*

```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}

visited = [] # List to keep track of visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)

  while queue:
    s = queue.pop(0)
    print (s, end = " ")

    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.append(neighbour)
```

```
        queue.append(neighbour)

# Driver Code
bfs(visited, graph, 'A')
```

Explanation :

Lines 3-10 : The illustrated graph is represented using an adjacency list. An easy way to do this in Python is to use a dictionary data structure, where each vertex has a stored list of its adjacent nodes.

Line 12 : visited is a list that is used to keep track of visited nodes.

Line 13 : queue is a list that is used to keep track of nodes currently in the queue.

Line 29 : The arguments of the bfs function are the visited list, the graph in the form of a dictionary, and the starting node A.

Lines 15-26 : bfs follows the algorithm described above :

(1) It checks and appends the starting node to the visited list and the queue.

(2) Then, while the queue contains elements, it keeps taking out nodes from the queue, appends the neighbors of that node to the queue if they are unvisited, and marks them as visited.

(3) This continues until the queue is empty.

14. *Implement depth first search (DFS) in Python,*

```
# Using a Python dictionary to act as an adjacency list
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}

visited = set() # Set to keep track of visited nodes.

def dfs(visited, graph, node):
  if node not in visited:
    print (node)
    visited.add(node)
```

```
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

    # Driver Code
    dfs(visited, graph, 'A')
```

15. *Explain with example.*
(1) *Required Arguments*
(2) *Keyword Arguments*
(3) *Default Arguments*
(4) *variable length Arguments*

Required Arguments

Required arguments, as the name suggests, are those that are required to be passed to the function at the time of function call. Failing to do so will result in an error.

In the simplest terms, required arguments are exactly the opposite of default function arguments.

Let us try to understand this with the help of an example.

```
def add_nums(num1, num2=12):
    print(num1 + num2)

add_nums(num1=11, num2=13)  # Output: 24

# no value for default argument
add_nums(num1=11)  # Output: 23

# no value for required argument
add_nums(num2=13) # Will throw an error
Output
24
23
TypeError: add_nums() missing 1 required positional argument: 'num1'
    add_nums(num2=13) # Will throw an error
Line 10 in <module> (Solution.py)
```

In the example given above, line 10 will throw the following error.

```
TypeError: add_nums() missing 1 required positional argument: 'num1'
    add_nums(num2=13) # Will throw an error
Line 10 in <module> (Solution.py)
```

It happened since we did not provide any value for the required argument num1. On the other hand,

not providing a value for num2 in line 7 will not result in any error, as the function assumes the default value for num2.

Keyword Arguments

Keyword arguments (or named arguments) are values that, when passed into a function, are identifiable by specific parameter names. A keyword argument is preceded by a parameter and the assignment operator, = .

Keyword arguments can be linked to dictionaries in that they map a value to a keyword.

Example :

```
def team(name, project):
    print(name, "is working on an", project)

team(project = "Edpresso", name = 'FemCode')
```

Output :

```
FemCode is working on an Edpresso
```

As you can see, we had the same output from both codes although, when calling the function, the arguments in each code had different positions.

With keyword arguments, as long as you assign a value to the parameter, the positions of the arguments do not matter:

Default Arguments

Python has a different way of representing syntax and default values for function arguments. Default values indicate that the function argument will take that value if no argument value is passed during the function call. The default value is assigned by using the assignment(=) operator of the form keywordname=value.

Let's understand this through a function student. The function student contains 3-arguments out of which 2 arguments are assigned with default values. So, the function student accepts one required argument (firstname), and rest two arguments are optional.

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
print(firstname, lastname, 'studies in', standard, 'Standard')
```

Variable-length Arguments

Variable-length arguments, abbreviated as varargs, are defined as arguments that can also accept an unlimited amount of data as input. The developer doesn't have to

wrap the data in a list or in any other sequence while using them.

There are two types of variable-length arguments in Python :

(1) Non Keyworded Arguments denoted as (*args)

(2) Keyworded Arguments denoted as (**kwargs)

Non Keyworded Arguments denoted as (*args) :

(a) The * in *args indicates that we can pass variable number of arguments in a function in Python.

(b) We can pass any number of arguments during the function call.

Example :

```
#program to demonstrate *args in python
def my_func(*argp):
   for i in argp:
#printing each element from the list
      print(i)
#passing the parameters in function
my_func("Let","us","study","Data
Science","and","Blockchain")
```

Output :

```
Let
us
study
Data Science
and
Blockchain
```

Keyworded Arguments denoted as (kwargs) :**

(a) The double star indicates that any number of keyworded arguments can be passed in a function.

(b) It is analogous to a dictionary where each key is mapped to a value.

```
#program to demonstrate **kwargs in python
   def my_func(**kwargs):
      for k,v in kwargs.items():
         print("%s=%s"%(k,v))
   #passing the parameters in function
my_func(a_key="Let",b_key="us",c_key="study",d_key="Data
Science",e_key="and",f_key="Blockchain")
```

Output :

```
a_key=Let
```

```
b_key=us
c_key=study
d_key=Data Science
e_key=and
f_key=Blockchain
```

16. *What are the characteristics of Python Dictionaries?*

The 4 main characteristics of a dictionary are:

(1) **Dictionaries are Unordered :** The dictionary elements key-value pairs are not in ordered form.

(2) **Dictionary Keys are Case Sensitive :** The same key name but with different case are treated as different keys in Python dictionaries.

(3) **No Duplicate Key is Allowed :** When duplicate keys encountered during assignment the last assignment wins.

(4) **Keys Must be Immutable :** We can use strings numbers or tuples as dictionary keys but something like ['key'] is not allowed.

17. *Write two differences between List and Dictionary.*

List

List are just like the arrays, declared in other languages. Lists need not be homogeneous always which makes it a most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

Dictionary

Dictionary in Python on the other hand is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds key: value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon: whereas each key is separated by a 'comma'.

18. *How to get list of parameters name from a function in Python?*

To extract the number and names of the arguments from a function or function[something] to return ("arg1", "arg2"), we use the inspect module.

The given code is written as follows using inspect module to find the parameters inside the functions a Method and foo.

Example :

```
import inspect
def aMethod(arg1, arg2): pass
print(inspect.getargspec(aMethod))
def foo(a,b,c=4, *arglist, **keywords): pass
print(inspect.getargspec(foo))
```

Output :

```
ArgSpec(args=['arg1', 'arg2'], varargs=None, keywords=None,
defaults=None)
ArgSpec(args=['a', 'b', 'c'], varargs='arglist',
keywords='keywords', defaults=(4,))
```

19. *Explain built-in Dictionary Functions in Python.*

Listed below are all the Python functions that can be used with the Dictionary type.

(1) **dict.clear() :** Removes all the key-value pairs from the dictionary.

(2) **dict.copy() :** Returns a shallow copy of the dictionary.

(3) **dict.fromkeys() :** Creates a new dictionary from the given iterable (string, list, set, tuple) as keys and with the specified value.

(4) **dict.get() :** Returns the value of the specified key.

(5) **dict.items() :** Returns a dictionary view object that provides a dynamic view of dictionary elements as a list of key-value pairs. This view object changes when the dictionary changes.

(6) **dict.keys() :** Returns a dictionary view object that contains the list of keys of the dictionary.

(7) **dict.pop() :** Removes the key and return its value. If a key does not exist in the dictionary, then

returns the default value if specified, else throws a KeyError.

(8) **dict.popitem()** : Removes and return a tuple of (key, value) pair from the dictionary. Pairs are returned in Last in First Out (LIFO) order.

(9) **dict.setdefault()** : Returns the value of the specified key in the dictionary. If the key not found, then it adds the key with the specified default value. If the default value is not specified then it set None value.

(10) **dict.update()** : Updates the dictionary with the key-value pairs from another dictionary or another iterable such as tuple having key-value pairs.

(11) **dict.values()** : Returns the dictionary view object that provides a dynamic view of all the values in the dictionary. This view object changes when the dictionary changes.

20. *Difference between Method and Function in Python?*

Python Function

Simply put, a function is a series of steps executed as a single unit and "encapsulated" under a single name. It is a group of statements capable of performing some tasks.

A function in a program can take arguments (that is, the data to operate on) and can optionally return some data after performing the intended task. Creating a function is basically giving your computer well-defined instructions to perform a task.

Python Method

Objects in object-oriented programming have attributes, i.e., data values within them. But how do we access these attribute values? We access them through methods.

Method Represent the Behavior of an Object : A method is a piece of code that is associated with an object and operates upon the data of that object. In most respects, a method is identical to a function.

Except for two major differences :

(1) It is associated with an object and we call it 'on' that object.

(2) It operates on the data which is contained within the class.

21. *How to define and call a function in Python?*

Function in Python is defined by the "def" statement followed by the function name and parentheses (())

Example :

Let us define a function by using the command" def func1():" and call the function. The output of the function will be "I am learning Python function".

```
def func1():
print ("I am learning Python Function")        #function definition
func1()                                          # function call
```

The function print func1() calls our def func1(): and print the command " I am learning Python function None."

There are set of rules in Python to define a function.

(1) Any args or input parameters should be placed within these parentheses

(2) The function first statement can be an optional statement- docstring or the documentation string of the function

(3) The code within every function starts with a colon (:) and should be indented (space)

(4) The statement return (expression) exits a function, optionally passing back a value to the caller. A return statement with no args is the same as return None.

22. *What Is Recursion? Why we use in python?*

A recursive definition is one in which the defined term appears in the definition itself. Self-referential situations often crop up in real life, even if they aren't immediately recognizable as such. For example, suppose you wanted to describe the set of people that make up your ancestors.

Your ancestors = (your parents) + (your parents'ansetor)

When you call a function in Python, the interpreter creates a new local namespace so that names defined within that function don't collide with identical names defined elsewhere. One function can call another, and even if they both define objects with the same name, it all works out fine because those objects exist in separate namespaces.

23. *What are the advantages and disadvantages of using recursion?*

Advantages of Using Recursion

(1) A complicated function can be split down into smaller sub-problems utilizing recursion.

(2) Sequence creation is simpler through recursion than utilizing any nested iteration.

(3) Recursive functions render the code look simple and effective.

Disadvantages of Using Recursion

(1) A lot of memory and time is taken through recursive calls which makes it expensive for use.

(2) Recursive functions are challenging to debug.

(3) The reasoning behind recursion can sometimes be tough to think through.

❑❑

MODULES AND PACKAGES

1. ◆ *What is docstring?* (2023-24)
◆ *What is docstring in Python?*

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods. As you can see, even for a relatively simple function, documenting using comments quickly makes it unpleasant and difficult to read. So, to solve this, the docstring was introduced. A docstring is simply a multi-line string, that is not assigned to anything. It is specified in source code that is used to document a specific segment of code. Unlike conventional source code comments, the docstring should describe what the function does, not how.

All modules should normally have docstrings , and all functions and classes exported by a module should also have docstrings. Public methods (including the __init__ constructor) should also have docstrings. A package may be documented in the module docstring of the __init__.py file in the package directory.

One-line Docstrings

One-liners are for Really Obvious Cases : They should really fit on one line . Depending on the complexity of the function, method, or class being written, a one-line docstring may be perfectly appropriate. These are generally used for really obvious cases, such as:

```
def sum(x, y):
 """Returns arg1 value add to arg2 value."""
 return a+b
print sum.__doc__
```

Output:

```
Returns arg1 value add to arg2 value.
```

In larger or more complex projects however, it is often a good idea to give more information about a function, what it does, any exceptions it may raise, what it returns, or relevant details about the parameters. For more detailed documentation of code a popular style is the one used for the Numpy project, often called Numpy style docstrings.

```
example
def generic_socket(param1, param2):
    """
    Summary generic_socket.
    Extended description of generic_socket.
    Parameters
    ----------
    param1 : int
        Description of param1 (port)
    param2 : str
        Description of param2 (ipaddress)
    Returns
    -------
    int
        Description of return value
    """
    return 42
```

2. ◆ ***What is module and package in Python?*** **(2023-24,2022-23)**

◆ ***What are modules and packages in Python?***

Python packages and Python modules are two mechanisms that allow for modular programming in Python.

Modules

Modules, in general, are simply Python files with a .py extension and can have a set of functions, classes, or variables defined and implemented. They can be imported and initialized once using the import statement. If partial functionality is needed, import the requisite classes or functions using from foo import bar.

Packages

Packages allow for hierarchial structuring of the module namespace using dot notation. As, modules help avoid clashes between global variable names, in a similar manner, packages help avoid clashes between module names.

Creating a package is easy since it makes use of the system's inherent file structure. So just stuff the modules into a folder and there you have it, the folder name as the package name. Importing a module or its contents from this package requires the package name as prefix to the module name joined by a dot.

3. *How to print today date?* (2023-24)

In Python, Date and Time are not data types of their own, but a module named DateTime can be imported to work with the date as well as time. Datetime module comes built into Python, so there is no need to install it externally. The DateTime module provides some functions to get the current date as well as time.

Get the current date using date.today()

The today() method of date class under DateTime module returns a date object which contains the value of Today's date.

Syntax : date.today()

```
# Import date class from datetime module
from datetime import date
# Returns the current local date
today = date.today()
print("Today date is: ", today)
```

Output :

```
Today date is: 2024-30-03
```

4. ◆ *How to create and import a module in python? Explain in detail with example.* (2023-24)
◆ *Explain import module in Python?*

Creating a Modules

Modules are a fundamental concept in Python for code organization and reusability. Here's a detailed explanation with examples :

Creating a Module :

(1) Create a Python File :

(a) Use a text editor or an IDE to create a new file.

(b) Give it a descriptive name ending with the .py extension (e.g., my_functions.py).

(2) Write your Code :

(a) Inside the file, define functions, variables, or classes you want to reuse.

Example :

```
def greet(name):
"""This function greets a person."""
print("Hello,", name + "!")
```

```
def calculate_area(length, width):
    """This function calculates the area of a rectangle."""
    area = length * width
    return area
```

Import a Module in Python

Import in python is similar to #include header_file in C/C++. Python modules can get access to code from another module by importing the file/function using import. The import statement is the most common way of invoking the import machinery, but it is not the only way.

import module_name : When the import is used, it searches for the module initially in the local scope by calling __import__ () function. The value returned by the function is then reflected in the output of the initial code.

```
import math
pie = math.pi
print("The value of pi is : ",pie)
```

Output

```
The value of pi is : ', 3.141592653589793
```

import module_name.member_name : In the above code module, math is imported, and its variables can be accessed by considering it to be a class and pi as its object.

The value of pi is returned by __import__(). pi as a whole can be imported into our initial code, rather than importing the whole module.

```
from math import pi
# Note that in the above example,
# we used math.pi. Here we have used
# pi directly.
print(pi)
```

from module_name import * : In the above code module, math is not imported, rather just pi has been imported as a variable.

All the functions and constants can be imported using *.

```
from math import *
print(pi)
print(factorial(6))
```

Output

```
3.14159265359
720
```

Example :

```
from my_functions import greet, calculate_area
    greet("Alice") # Output: Hello, Alice! area =
```

):
rea of a rectangle."""

include header_file in
s to code from another
in using import. The
n way of invoking the
ly way.
e import is used. it
the local scope by
alue returned by the
t of the initial code.

e code module,
n imported as


```
calculate_area(4, 2)
print("Area:", area) # Output: Area: 8
```

Note :

(1) Modules help organize code and prevent naming conflicts.

(2) You can create multiple functions, variables, or classes within a single module.

(3) The import statement makes the module's elements available in your script.

(4) Choose the appropriate import method based on whether you need the entire module or just specific elements.

(5) **Module Location :** If the module is not in the same directory as your script, you'll need to adjust the import statement accordingly (e.g., using relative or absolute paths).

(6) **Packages :** For larger projects, you can create a hierarchy of modules using directories and init .py files to form packages.

5. *What is Date module? Explain five functions with examples used in Date module.* **(2023-24)**

The Datetime Module in Python : The datetime module offers a rich set of classes and functions for working with dates, times, timedeltas, and timezones in Python. It's essential for tasks like :

(1) **Parsing Dates and Times :** Converting strings representing dates and times into datetime objects.

(2) **Formatting Dates and Times :** Creating human-readable strings from datetime objects in various formats.

(3) **Date and Time Arithmetic :** Performing calculations on dates, times, and durations.

(4) **Extracting Date and Time Components :** Accessing individual components like year, month, day, hour, minute, etc., from datetime objects.

(5) **Timezone Handling :** Working with timezones to account for different locations around the world.

Five Important Functions in Datetime :

(1) date.today():

Returns the current local date as a date object.

Example :

```
import datetime
today = datetime.date.today()
print("Today's date:", today) # Output: Today's date: 2024-03-30
```

(2) date.fromisoformat(date_string) :

Creates a date object from a string formatted in YYYY-MM-DD (ISO format). Example:

```
birthday_str = "2001-10-26"
birthday = datetime.date.fromisoformat(birthday_str)
print("Birthday:", birthday) # Output: Birthday: 2001-10-26
```

(3) date.year, date.month, date.day() :

Access individual components (year, month, day) from a date object.

Example :

```
print("Year:", birthday.year) # Output: Year: 2001
print("Month:", birthday.month) # Output: Month: 10
print("Day:", birthday.day) # Output: Day: 26
```

(4) date.replace() :

Creates a new date object with modified components.

Example :

```
ew_birthday = birthday.replace(year=2025)
print("New birthday (modified year):", new_birthday) # Output: New birthday (modified year): 2025-10-26
```

(5) datetime.timedelta() :

(a) Represents a duration (time difference) between two dates or times.

(b) You can use it for calculations like finding the date a certain number of days in the future.

Example :

```
time_delta = datetime.timedelta(days=30)
future_date = today + time_delta
print("Date 30 days from today:", future_date)
```

 # Output: Date 30 days from today: 2024-04-29

6. *Why do we need a date class in python?* (2022-23)

The date class in Python is part of the datetime module and is used to represent and work with dates. It provides a way to store, manipulate, and perform

calculations with dates in a consistent and reliable manner. Here are a few reasons why the date class is useful :

(1) **Date Representation :** The date class allows you to represent dates accurately. It stores the year, month, and day components separately, ensuring that you have a standardized format for dates.

(2) **Date Arithmetic :** The date class provides methods to perform arithmetic operations on dates. You can calculate differences between dates, add or subtract a certain number of days, and compare dates to determine their relative ordering.

(3) **Date Formatting :** The date class allows you to format dates in various ways, such as converting them to strings in a specific format. This is helpful when you need to display dates in a human-readable format or when you want to convert them to a specific string representation for storage or communication purposes.

(4) **Date Parsing :** The date class also provides methods for parsing strings and converting them into date objects. This is useful when you need to extract dates from user input or when you're working with dates in a string format.

(5) **Date Validation :** The date class performs validation checks to ensure that the specified date is valid. It takes into account factors like leap years, varying month lengths, and valid year ranges, preventing you from working with incorrect or nonsensical dates.

(6) **Date Manipulation :** The date class allows you to manipulate dates easily. You can extract specific components (year, month, day) from a date, replace individual components with new values, and create new dates based on existing ones.

Overall, the date class in Python provides a comprehensive set of functionalities for working with dates, making it easier to handle date-related operations accurately and efficiently in your code.

[D.8]

7. ◆ ***What is time module? Explain five functions of time module with example. Write a Python program to subtract five days from current date.*** **(2022-23)**

◆ ***Explain the Time Module and its function with example.***

The time module follows the "EPOCH" convention which refers to the point where the time starts. In Unix system "EPOCH" time started from 1 January, 12:00 am, 1970 to year 2038.

The epoch is the point where the time starts and is platform-dependent. On Windows and most Unix systems, the epoch is January 1, 1970, 00:00:00 (UTC), and leap seconds are not counted towards the time in seconds since the epoch. To check what the epoch is on a given platform we can use time.gmtime(0).

Functions in Python Time Module

(1) time.time() function : The time () is the main function of the time module. It measures the number of seconds since the epoch as a floating-point value.

Example :

```
import time
start = time.time()
print ("Time elapsed on working...")
time.sleep(0.9)
end = time.time()
print ("Time consumed in working: ", end - start)
```

Output

```
Time elapsed on working...
Time consumed in working: 0.9219651222229004
```

(2) time.clock() function : The time.clock() function return the processor time. It is used for performance testing/benchmarking.

Example :

```
import time
template = 'time()# {:0.2f}, clock()# {:0.2f}'
print(template.format(time.time(), time.clock()))
for i in range(5, 0, -1):
  print('---Sleeping for: ', i, 'sec.')
time.sleep(i)
print(template.format(time.time(), time.clock())
```

```
    )
Output
    time()# 1553263728.08, clock()# 0.00
        ---Sleeping for: 5 sec.
    time()# 1553263733.14, clock()# 5.06
        ---Sleeping for: 4 sec.
    time()# 1553263737.25, clock()# 9.17
        ---Sleeping for: 3 sec.
    time()# 1553263740.30, clock()# 12.22
        ---Sleeping for: 2 sec.
    time()# 1553263742.36, clock()# 14.28
    ---Sleeping for: 1 sec.
    time()# 1553263743.42, clock()# 15.34
```

(3) time.ctime() function : time.ctime() function takes the time in "seconds since the epoch" as input and translates into a human readable string value as per the local time. If no argument is passed, it returns the current time.

Example ;

```
import time
print('The current local time is :', time.ctime())
newtime = time.ctime() + 60
print('60 secs from now :', time.ctime(newtime))
```

Output :

```
The current local time is : Fri Mar 22 19:43:11 2019
60 secs from now : Fri Mar 22 19:44:11 2019
```

(4) time.sleep() function : time.sleep() function halts the execution of the current thread for the specified number of seconds. Pass a floating point value as input to get more precise sleep time.

The sleep() function can be used in situation where we need to wait for a file to finish closing or let a database commit to happen.

Example :

```
import time
# using ctime() to display present time
print ("Time starts from : ",end="")
print (time.ctime())
# using sleep() to suspend execution
print ('Waiting for 5 sec.')
time.sleep(5)
# using ctime() to show present time
print ("Time ends at : ",end="")
print (time.ctime())
```

```
Output :
    Time starts from : Fri Mar 22 20:00:00 2019
    Waiting for 5 sec.
    Time ends at : Fri Mar 22 20:00:05 2019
```

(5) **time.struct_time class :** The time.struct_time is the only data structure present in the time module. It has a named tuple interface and is accessible via index or the attribute name.

Example :

```
import time
print(' Current local time:', time.ctime())
t = time.localtime()
print('Day of month:', t.tm_mday)
print('Day of week :', t.tm_wday)
print('Day of year :', t.tm_yday)
```

Output :

```
Current local time: Fri Mar 22 20:10:25 2019
Day of month: 22
Day of week : 4
Day of year : 81
```

(6) **time.strftime() function :** This function takes a tuple or struct_time in the second argument and converts to a string as per the format specified in the first argument.

Example :

```
import time
now = time.localtime(time.time())
print("Current date time is: ",time.asctime(now))
print(time.strftime("%y/%m/%d %H:%M", now))
print(time.strftime("%a %b %d", now))
print(time.strftime("%c", now))
print(time.strftime("%I %p", now))
print(time.strftime("%Y-%m-%d %H:%M:%S %Z", now))
```

Output :

```
Current date time is: Fri Mar 22 20:13:43 2019
19/03/22 20:13
Fri Mar 22
Fri Mar 22 20:13:43 2019
08 PM
2019-03-22 20:13:43 Indla Standard Time
```

Python Program to Subtract Five Days from Current Date : Here's a Python program to subtract five days from the current date:

```
import datetime
```

```
current_date = datetime.date.today()
print('Current date:', current_date)

delta = datetime.timedelta(days=5)
new_date = current_date - delta
print('New date:', new_date)
```

Output :

```
Current date: 2023-05-13
New date: 2023-05-08
```

In this program, we use the datetime module to work with dates and time intervals. We first get the current date using datetime.date.today(), and then create a datetime.timedelta object representing a time interval of five days. We subtract this interval from the current date using the - operator, and store the result in new_date. Finally, we print both the current and new dates for comparison.

8. *What is Modular Programming?*

Modular programming is the practice of segmenting a single, complicated coding task into multiple, simpler, easier-to-manage sub-tasks. We call these subtasks modules. Therefore, we can build a bigger program by assembling different modules that act like building blocks.

Modularizing our code in a big application has a lot of benefits.

(1) **Simplification :** A module often concentrates on one comparatively small area of the overall problem instead of the full task. We will have a more manageable design problem to think about if we are only concentrating on one module. Program development is now simpler and much less vulnerable to mistakes.

(2) **Flexibility :** Modules are frequently used to establish conceptual separations between various problem areas. It is less likely that changes to one module would influence other portions of the program if modules are constructed in a fashion that reduces interconnectedness. (We might even be capable of

editing a module despite being familiar with the program beyond it.) It increases the likelihood that a group of numerous developers will be able to collaborate on a big project.

(3) **Reusability :** Functions created in a particular module may be readily accessed by different sections of the assignment (through a suitably established api). As a result, duplicate code is no longer necessary.

9. *How do you create your own package in Python? Explain with example.*

To create a package in Python, we need to follow these three simple steps :

(1) First, we create a directory and give it a package name, preferably related to its operation.

(2) Then we put the classes and the required functions in it.

(3) Finally, we create an __init__.py file inside the directory, to let Python know that the directory is a package.

Example :

Let's create a package named Cars and build three modules in it namely, Bmw, Audi and Nissan.

(1) First, we create a directory and name its Cars.

(2) Then we need to create modules. To do this we need to create a file with the name Bmw.py and create its content by putting this code into it.

```
# Python code to illustrate the Module
class Bmw:
# First we create a constructor for this class
# and add members to it, here models
def __init__(self):
self.models = ['i8', 'x1', 'x5', 'x6']

# A normal print function
def outModels(self):
print('These are the available models for BMW')
for model in self.models:
print('\t%s ' % model)
```

Then we create another file with the name Audi.py and add the similar type of code to it with different members.

```
# Python code to illustrate the Module
class Audi:
# First we create a constructor for this class
# and add members to it, here models
def __init__(self):
self.models = ['q7', 'a6', 'a8', 'a3']

# A normal print function
def outModels(self):
print('These are the available models for Audi')
for model in self.models:
print('\t%s ' % model)
```

Then we create another file with the name Nissan.py and add the similar type of code to it with different members.

```
# Python code to illustrate the Module
class Nissan:
# First we create a constructor for this class
# and add members to it, here models
def __init__(self):
self.models = ['altima', '370z', 'cube', 'rogue']

# A normal print function
def outModels(self):
print('These are the available models for Nissan')
for model in self.models:
print('\t%s ' % model)
```

(3) Finally, we create the __init__.py file. This file will be placed inside Car's directory and can be left blank or we can put this initialization code into it

Now, let's use the package that we created. To do this make a sample.py file in the same directory where Cars package is located and add the following code to it:

```
# Import classes from your brand new package
from Cars import Bmw
from Cars import Audi
from Cars import Nissan

# Create an object of Bmw class & call its method
ModBMW = Bmw()
```

```
ModBMW.outModels()

# Create an object of Audi class & call its method
ModAudI = Audi()
ModAudI.outModels()

# Create an object of Nissan class & call its method
ModNissan = Nissan()
ModNissan.outModels()
```

10. *How can I import a module from the different directory?*

When we try to import a python module, it's looked into the current directory and the PATH variable location. So if your python file is not present in these locations, then you will get ModuleNotFoundError. The solution is to import sys module and then append the required directory to its path variable.

Below code shows the error when we try to import from a different directory and how I am fixing it by adding its directory to the path variable.

```
>>> import test123
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'test123'
>>> import sys
>>> sys.path.append('/Users/pankaj/temp')
>>> import test123
>>> test123.x
10
>>> test123.foo()
foo
```

11. *Why we should avoid using import star in Python?*

Using import * in python programs is considered a bad habit because this way you are polluting your namespace, the import * statement imports all the functions and classes into your own namespace, which may clash with the functions you define or functions of other libraries that you import. Also it becomes very difficult at

some times to say from which library does a particular function came from. The risk of overriding the variables/functions etc always persist with the import * practice.

Below are some points about why import * should not be used :

(1) Code Readability

(2) It is always remains a mystery what is imported and cannot be found easily from which module a certain thing was imported that result in low code readability.

(3) Polluting the namespace, import * imports all the functions and classes in your own namespace that may clash with the function and classes you define or function and classes of other libraries that you may import.

(4) Concrete possibility of hiding bugs

(5) Tools like pyflakes can't be used to statically detect errors in the source code.

12. *Explain the Calendar Module and its function with example.*

Python defines an inbuilt module calendar that handles operations related to the calendar.

The calendar module allows output calendars like the program and provides additional useful functions related to the calendar. Functions and classes defined in the Calendar module use an idealized calendar, the current Gregorian calendar extended indefinitely in both directions. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention).

Example : Display the Calendar of a given month.

```
# Python program to display calendar of
# given month of the year
# import module
import calendar
yy = 2017
mm = 11
# display the calendar
print(calendar.month(yy, mm))
```

(1) **class calendar.Calendar** : The calendar class creates a Calendar object. A Calendar object provides several methods that can be used for preparing the calendar data for formatting. This class doesn't do any formatting itself. This is the job of subclasses. Calendar class allows the calculations for various tasks based on date, month, and year. Calendar class provides the following methods:

iterweekdays() : Returns an iterator for the week day numbers that will be used for one week

itermonthdates() : Returns an iterator for the month (1–12) in the year

itermonthdays() : Returns an iterator of a specified month and a year

itermonthdays2() : Method is used to get an iterator for the month in the year similar to itermonthdates(): -Days returned will be tuples consisting of a day of the month number and a week day number.

itermonthdays3() : Returns an iterator for the month in the year similar to itermonthdates(), but not restricted by the datetime.date range. Days returned will be tuples consisting of a year, a month and a day of the month numbers.

itermonthdays4() : Returns an iterator for the month in the year similar to itermonthdates(), but not restricted by the datetime.date range. Days returned will be tuples consisting of a year, a month, a day of the month, and a day of the week numbers.

monthdatescalendar(): Used to get a list of the weeks in the month of the year as full weeks

monthdays2calendar() : Used to get a list of the weeks in the month of the year as full weeks

monthdayscalendar : Used to get a list of the weeks in the month of the year as full weeks

yeardatescalendar() : Used to get a list of the weeks in the month of the year as full weeks

yeardays2calendar() : Used to get the data for specified year. Entries in the week lists are tuples of day numbers and weekday numbers

yeardayscalendar() : Used to get the data for specified year. Entries in the week lists are day numbers

(2) **class calendar.TextCalendar :** TextCalendar class can be used to generate plain text calendars. TextCalendar class in Python allows you to edit the calendar and use it as per your requirement.

formatmonth() : Method is used to get month's calendar in a multi-line string

prmonth() : Method is used to print a month's calendar as returned by formatmonth()

formatyear() : Method is used to get m-column calendar for an entire year as a multi-line string

pryear() : Method is used to print the calendar for an entire year as returned by formatmonth()

(3) **class calendar.HTMLCalendar :** HTMLCalendar class can be used to generate HTML calendars. HTMLCalendar class in Python allows you to edit the calendar and use as per your requirement.

formatmonth() : Method is used to get month's calendar as an HTML table

formatyear() : Method is used to get year's calendar as an HTML table.

formatyearpage() : Method is used to get year's calendar as a complete HTML page

(4) **Simple TextCalendar class :** For simple text calendars calendar module provides the following functions:

setfirstweekday() : Function sets the day start number of weeks

firstweekday() : Function returns the first week day number. By default, 0 (Monday)

isleap() : Function checks if year mentioned in argument is leap or not

leapdays() : Function returns the number of leap days between the specified years in arguments

weekday () : Function returns the week day number (0 is Monday) of the date specified in its arguments

weekheader() : Returns a header containing abbreviated weekday names

monthrange() : Function returns two integers, first, the starting day number of week (0 as monday), second, the number of days in the month

monthcalendar() : Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month are represented by zeros

prmonth() : Function also prints the month of specific year but there is no need of "print" operation to execute this

month () : Function prints the month of a specific year mentioned in arguments

prcal() : Function also prints the calendar of specific year but there is no need of "print" operation to execute this

calendar () : Function displays the year, width of characters, no. of lines per week and column separations.

13. *Explain the Random Module and its function with example.*

Python Random module is an in-built module of Python which is used to generate random numbers. These are pseudo-random numbers means these are not truly random. This module can be used to perform random actions such as generating random numbers, print random a value for a list or string, etc.

Example : Printing a random value from a list

```
# Import random
import random
# Prints a random value from the list
list1 = [1, 2, 3, 4, 5, 6]
print(random.choice(list1))
```

Output :

```
2
```

List of all the Functions in Random Module

seed() : Initialize the random number generator

getstate() : Returns an object with the current internal state of the random number generator

setstate() : Used to restore the state of the random number generator back to the specified state

getrandbits() : Return an integer with a specified number of bits
randrange() : Returns a random number within the range
randint() : Returns a random integer within the range
choice() : Returns a random item from a list, tuple, or string
choices() : Returns multiple random elements from the list with replacement
sample() : Returns a particular length list of items chosen from the sequence
random() : Generate random floating numbers
uniform() : Return random floating number between two numbers both inclusive
triangular() : Return a random floating point number within a range with a bias towards one extreme
betavariate() : Return a random floating point number with beta distribution
expovariate() : Return a random floating point number with exponential distribution
gammavariate() : Return a random floating point number with gamma distribution
gauss() : Return a random floating point number with Gaussian distribution
lognormvariate() : Return a random floating point number with log-normal distribution
normalvariate() : Return a random floating point number with normal distribution
vonmisesvariate() : Return a random floating point number with von Mises distribution or circular normal distribution
paretovariate() : Return a random floating point number with Pareto distribution
weibullvariate() : Return a random floating point number with Weibull distribution

14. *What is the difference between a module, a package, and a library?*

A module is a Python file that's intended to be imported into scripts or other modules. It can contain functions, classes, and global variables.

A package is a collection of modules that are grouped together inside a folder to provide consistent functionality. Packages can be imported just like modules. They usually have a __init__.py file in them that tells the Python interpreter to process them as such.

A library is a collection of packages.

15. *Write definition of a method ZeroEnding(SCORES) to add all those values in the list of SCORES, which are ending with zero (0) and display the sum.*

If the SCORES contain [200,456,300,100,234,678]
The sum should be displayed as 600

```
def ZeroEnding (SCORES) :
    SZero = 0
    for i in SCORES :
        if i % 10 == 0:
            SZero += i
    print ("sum of numbers ending with zero : ", SZero)
```

Output :

```
>>> ZeroEnding ([10, 20, 30, 40])
sum of numbers ending with zero :  100
>>>
>>> ZeroEnding ([100, 25, 51, 20, 35, 50, 60, 120, 300, 555,
450, 999, 100])
sum of numbers ending with zero :  1200
```

16. *What are Python namespaces? Why are they used?*

A namespace in Python ensures that object names in a program are unique and can be used without any conflict. Python implements these namespaces as dictionaries with 'name as key' mapped to a corresponding 'object as value'. This allows for multiple namespaces to use the same name and map it to a separate object. A few examples of namespaces are as follows :

Local Namespace includes local names inside a function. the namespace is temporarily created for a function call and gets cleared when the function returns.

Global Namespace includes names from various imported packages/ modules that are being used in the current project. This namespace is created when the package is imported in the script and lasts until the execution of the script.

Built-in Namespace includes built-in functions of core Python and built-in names for various types of exceptions.

The lifecycle of a namespace depends upon the scope of objects they are mapped to. If the scope of an object ends, the lifecycle of that namespace comes to an end. Hence, it isn't possible to access inner namespace objects from an outer namespace.

17. *How many types of modules are there in Python?*

Modules in Python can be of two types :

(1) Built-in Modules.

(2) User-defined Modules.

(1) **Built-in Modules in Python :** One of the many superpowers of Python is that it comes with a "rich standard library". This rich standard library contains lots of built-in modules. Hence, it provides a lot of reusable code.

To name a few, Python contains modules like "os", "sys", "datetime", "random".

You can import and use any of the built-in modules whenever you like in your program.

(2) **User-Defined Modules in Python :** Another superpower of Python is that it lets you take things in your own hands. You can create your own functions and classes, put them inside modules and voila! You can now include hundreds of lines of code into any program just by writing a simple import statement.

18. *Difference between import and from import in python?*

When we use import <module> command, all the defined functions and variables created in the module are

now available to the program that imported it. The objects can be called by using the module's name as prefix before them. e.g.

```
import math
    print(math.sin(3.14)) # prefix math is being used
```

When we use from <module> import statements, only the asked objects are imported to the programme. No prefix is required to use them. e.g. from math import sin

❑❑

EXCEPTION HANDLING

1. *Write a simple program which illustrates Handling Exceptions.* (2023-24)

Python Exception Handling

Exceptions abnormally terminate the execution of a program. Since exceptions abnormally terminate the execution of a program, it is important to handle exceptions. In Python, we use the try...except block to handle exceptions.

Python try__except Block : The try...except block is used to handle exceptions in Python. Here's the syntax of try...except block :

```
try:
    # code that may cause exception
except:
    # code to run when exception occurs
```

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by except block. When an exception occurs, it is caught by the except block. The except block cannot be used without the try block.

Example : Exception Handling Using try...except

```
try:
    numerator = 10
    denominator = 0
    result = numerator/denominator
    print(result)
except:
    print("Error: Denominator cannot be 0.")
# Output: Error: Denominator cannot be 0.
```

Run Code : In the example, we are trying to divide a number by 0. Here, this code generates an exception.

To handle the exception, we have put the code, result = numerator/denominator inside the try block. Now when an exception occurs, the rest of the code inside the try block is skipped. The except block catches the exception and statements inside the except block are executed.

2. *Illustrate *args and ** kwargs parameters in Python programming language with an example.* (2023-24)

*args and **kwargs in Python

In Python, *args and **kwargs are special parameters used to handle variable-length arguments passed to functions. They provide flexibility in how you define and call functions.

(1) args (Variable-Length Positional Arguments) :

(a) *args allows you to accept a variable number of positional arguments as a tuple within the function.

(b) You can then iterate over the arguments or access them by index within the function.

Example :

```
def calculate_average(*numbers):
    """This function calculates the average of any number of numbers."""
total = sum(numbers)
average = total / len(numbers)
return average
print(calculate_average(1, 2, 3))  # Output : 2.0
print(calculate_average(10, 20, 30, 40)) # Output: 25.0
```

Here :

(i) The calculate_average function takes *numbers as an argument.

(ii) When the function is called, any number of arguments passed become a tuple stored in numbers.

(iii) The function iterates over numbers to calculate the average.

(2) kwargs (Variable-Length Keyword Arguments) :

(a) **kwargs allows you to accept a variable number of keyword arguments as a dictionary within the function.

(b) You can then access the arguments by their keyword names within the function.

Example :

```
def user_profile(first_name, last_name, age=None, city=None):
"""This function creates a user profile."""
profile = {
 "first_name": first_name,
"last_name": last_name,
}
 if age:
profile["age"] = age
```

```
if city:
profile["city"] = city
return profile
profile1 = user_profile("Alice", "Smith")
profile2 = user_profile("Bob", "Johnson", age=30,
city="New York")
print(profile1) # Output: {'first_name': 'Alice', 'last_name':
'Smith'}
print(profile2) # Output: {'first_name': 'Bob', 'last_name':
'Johnson', 'age': 30, 'city': 'New York'}
```

Here :

(i) The user_profile function takes first_name, last_name, age (default=None), and city (default=None) as arguments.

(ii) age and city have default values, so they are optional.

(iii) When the function is called with keywords, they become key-value pairs in the kwargs dictionary.

(iv) The function creates a profile dictionary based on the provided arguments.

(3) *args and **kwargs Key Points :

(a) Use *args when you don't know the exact number of positional arguments in advance.

(b) Use **kwargs when you want to accept optional keyword arguments with variable names.

(c) You can combine *args and **kwargs in a single function definition, but *args should come before **kwargs. By understanding *args and **kwargs, you can write more flexible and adaptable functions in Python.

3. *Describe the need for catching exceptions using try and except statements also explain multiple except blocks used in exception handling with example.* (2023-24)

Exception handling is a crucial mechanism in Python to gracefully manage errors that might occur during program execution. These errors, known as exceptions, can arise from various scenarios :

(1) User Input Errors : When the user enters unexpected or invalid data (e.g., non-numeric input for a calculation).

(2) **File I/O Errors :** When issues occur while reading from or writing to files (e.g., file not found, permission denied).

(3) **Network Errors :** When problems encounter during network operations (e.g., timeout, connection refused).

(4) **Logical Errors :** When bugs in your own code lead to incorrect calculations or unexpected behavior (e.g., division by zero, accessing an invalid index).

If you don't handle exceptions, your program will abruptly terminate with an error message. This can be :

(1) **Uninformative :** The error message might not be clear to the user about what went wrong.

(2) **Disruptive :** The program crash could leave the user frustrated and unable to complete their task.

Try-Except Statements : The try-except block is the foundation for exception handling in Python. It allows you to :

(1) **Execute code in the try block :** This code might potentially raise exceptions.

(2) **Catch exceptions in the except block(s) :** If an exception occurs within the try block, control is transferred to an appropriate except block that matches the exception type. You can then provide a user-friendly error message or take corrective actions.

Example : Division by Zero

```
def divide(num1, num2):
    try:
    result = num1 / num2
    print("Result:", result)
    except ZeroDivisionError: # Catch specific exception
    print("Error: Division by zero is not allowed.")
divide(10, 2) # Output: Result: 5.0
divide(10, 0) # Output: Error: Division by zero is not allowed.
```

Multiple Except Blocks : You can have multiple except blocks to handle different types of exceptions :

```
def get_file_data(filename):
    try:
    with open(filename, "r") as file:
    data = file.read()
    print("File data:", data)
    except FileNotFoundError:
    print("Error: File not found.")
    except PermissionError:
    print("Error: You don't have permission to access the file.")
```

In this example :

(1) If FileNotFoundError occurs (file not found), the first except block handles it.

(2) If PermissionError occurs (permission denied), the second except block is triggered.

Benefits of Exception Handling :

(1) **Robust Programs :** Programs become more resilient to unexpected errors, preventing crashes and improving user experience.

(2) **Meaningful Error Messages :** Provide informative messages to the user, guiding them on how to rectify the issue.

(3) **Maintainability :** Code that handles exceptions is easier to understand and maintain, especially as you're codebase grows.

By effectively using try-except statements with multiple blocks, you can create well-structured, error-tolerant Python applications that gracefully deal with exceptions, leading to a more positive user experience.

4. *How can you raise an exception in python? Explain with example.* **(2022-23)**

In Python, exceptions are used to indicate that an error or unexpected condition has occurred during program execution. When an exception is raised, it signals that something has gone wrong and the normal flow of program execution cannot continue.

You can raise an exception in Python using the raise statement. The raise statement is followed by an exception type and an optional error message. For example :

```
def divide(x, y):
    if y == 0:
        raise ZeroDivisionError("division by zero")
    return x / y

try:
    result = divide(10, 0)
except ZeroDivisionError as e:
    print(e)
```

In this example, the divide function raises a ZeroDivisionError exception if the second argument (y) is zero. The try block catches the exception and prints the error message, which in this case is "division by zero".

You can also define your own custom exception types by creating a new class that inherits from the Exception base class. For example :

```
class MyCustomError(Exception):
  pass

def my_function():
  raise MyCustomError("something went wrong")

try:
my_function()
except MyCustomError as e:
  print(e)
```

In this example, we define a new exception type called MyCustomError that inherits from the Exception base class. The my_function function raises this exception when something goes wrong. The try block catches the exception and prints the error message, which in this case is "something went wrong".

5. ◆ ***What is exception? Explain about try, except, else and finally blocks used in exception handling. Write a user defined exception using exception class.*** **(2022-23)**

◆ ***Why use Exceptions?***

Exception : Exception can be helpful in case you know a code which can produce error in the program. You can put that code under exception and run an error free syntax to execute your program. This helps in preventing the error from any block which is capable of producing errors. Since exception can be fatal error, one should use exception syntax to avoid the hurdles.

Python itself contains a number of different exceptions. The reasons can vary as per the exception.

Here, we are listing some of the most common exceptions in Python.

ImportError : When any object to be imported into the program fails.

Indexerror : The list being used in the code contains an out of range number.

NameError : An unknown variable is used in the program. If the program do not contains any defined by the user and the used variable is also not pre defined in the Python, hence this error comes into play.

SyntaxError : The code cannot be parsed properly. Hence it is necessary to take various precautionary measures while writing the code.

TypeError : An inappropriate type of function has been used for the value.

ValueError : A function has been used with correct type however the value for the function is irrelevant.

These errors are some of the most common ones and are required to be used, in case any sort of exceptions are required to be used.

Python also has assertion feature which can be used for raising exceptions. This feature tests an expression, Python tests it. If the result comes up false then it raises an exception. The assert statement is used for assertion. Assert feature improves the productivity of the exceptions in Python.

Try, Except, Else and Finally Blocks Used in Exception Using Exception Class : The basic syntax for a try-except block is :

```
try :
    # code that may raise an exception
except ExceptionType:
    # code to handle the exception
```

In this syntax, ExceptionType is the type of exception that the code in the try block may raise. If an exception of that type is raised, the code in the except block will be executed.

Here's an example:

```
try:
    x = 1 / 0
except ZeroDivisionError:
print('Cannot divide by zero')
```

Output :

```
Cannot divide by zero
```

In this example, the code in the try block attempts to divide 1 by 0, which raises a ZeroDivisionError. Since the except block handles that type of exception, the code in that block is executed and the program continues to run.

In addition to the try and except blocks, there are also else and finally blocks that can be used in exception handling.

The else block is executed if the code in the try block completes without raising an exception. Here's an example :

```
try:
    x = 1 / 2
except ZeroDivisionError:
print('Cannot divide by zero')
else:
print('Result:', x)
```

Output :

Result: 0.5

In this example, the code in the try block successfully divides 1 by 2, so the code in the else block is executed and the result is printed.

The finally block is always executed, regardless of whether an exception was raised or not. Here's an example :

```
try:
    x = 1 / 0
except ZeroDivisionError:
print('Cannot divide by zero')
else:
print('Result:', x)
finally:
  print('Done')
```

Output :

Cannot divide by zero
Done

In this example, an exception is raised and the code in the except block is executed. After that, the finally block is executed to perform any necessary cleanup tasks.

Finally, to create a user-defined exception in Python, we can define a new exception class that inherits from the built-in Exception class. Here's an example:

```
class MyException(Exception):
  pass

try:
  raise MyException('Something went wrong')
except MyException as e:
print('Caught exception:', e)
```

Output :

Caught Exception : Something went wrong

In this example, we define a new exception class called MyException, and then raise an instance of that class with a custom error message. The except block catches that exception and prints the error message.

6. *When are the following built-in exceptions raised? Give examples to support your answers.*

(1) *ImportError*
(2) *IOError*
(3) *NameError*
(4) *Zero DivisionError*

(1) ImportError : It is raised when the requested module definition is not found.

Example :

```
>>> import maths
Traceback (most recent call last):
    File " " , line 1, in
    import maths
ModuleNotFoundError: No module named
'maths'
>>> import randoms
Traceback (most recent call last):
File " ", line 1, in
import randoms
ModuleNotFoundError: No module named
'randoms'
```

Note : import maths #Raise an ImportError, due to maths module does not exists.

(2) IO Error : It is raised when the file specified in a program statement cannot be opened.

Example :

```
>>> f = open("abc.txt", 'r')
Traceback (most recent call last):
File " ", line 1, in
f = open('abc.txt', 'r')
FileNotFoundError: [Errno 2]
No such file or directory: 'abc.txt'
```

(3) NameError : It is raised when a local or global variable name is not defined.

Example :

```
>>> print (name)
Traceback (most recent call last):
File " ", line 1, In
print(name)
NameError: name 'name' is not defined
```

(4) Zero Division Error : It is raised when the denominator in a division operation is zero.

Example :

```
>>>
10/0
```

```
Traceback (most recent call last):
File " ", line 1, in
10/0
ZeroDivisionError: division by zero
```

7. *Explain exception handling mechanism in python?*

Exception handling is managed by the following 4 keywords :

(1) try
(2) catch
(3) finally
(4) throw

(1) **Python Try Statement :** A try statement includes keyword try, followed by a colon (:) and a suite of code in which exceptions may occur. It has one or more clauses.

During the execution of the try statement, if no exceptions occurred then, the interpreter ignores the exception handlers for that specific try statement.

In case, if any exception occurs in a try suite, the try suite expires and program control transfers to the matching except handler following the try suite.

Syntax :

```
try:
statement(s)
```

(2) **The Catch Statement :** Catch blocks take one argument at a time, which is the type of exception that it is likely to catch. These arguments may range from a specific type of exception which can be varied to a catch-all category of exceptions.

Rules for Catch Block :

(a) You can define a catch block by using the keyword catch
(b) Catch Exception parameter is always enclosed in parentheses
(c) It always represents the type of exception that catch block handles.
(d) An exception handling code is written between two {} curly braces.
(e) You can place multiple catch block within a single try block.
(f) You can use a catch block only after the try block.

(g) All the catch block should be ordered from subclass to superclass exception.

Example:

```
try
}
catch (ArrayIndexOutOfBoundsException e) {
System.err.println("Caught first " + e.getMessage()); }
catch (IOException e) {
System.err.println("Caught second " + e.getMessage());
}
```

(3) **Finally Statement in Python :** Finally block always executes irrespective of an exception being thrown or not. The final keyword allows you to create a block of code that follows a try-catch block.

Finally, clause is optional. It is intended to define clean-up actions which should be that executed in all conditions.

```
try:
   raise KeyboardInterrupt
finally:
   print 'welcome, world!'
```

Output

```
Welcome, world!
KeyboardInterrupt
```

Finally, clause is executed before try statement.

(4) **Raise Statement in Python :** The raise statement specifies an argument which initializes the exception object. Here, a comma follows the exception name, and argument or tuple of the argument that follows the comma.

Syntax :

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

In this syntax, the argument is optional, and at the time of execution, the exception argument value is always none.

8. *How to handling an Exception in Python with example.*

To handle an exception in python, try command is used. And there are certain instruction to use it,

(1) The try clause is executed including a statement between try and except.

(2) If no exception, the except clause is skipped and try statement is executed.

(3) If exception occurred, statement under except if matches the exception then clause is executed else try statement is executed

(4) If an exception occurs which does not match the exception named in the except clause, outer try statements are followed. If no handler is found, it is an unhandled exception and execution stops.

(5) The block of the code that can cause an exception can be avoided by placing that block in try: block. The block in the question can then be used under except exception.

Example :

```
>>> while True:
try:
   num1 = 7
   num2 = 0
   print (num1 / num2)
   print("Done calculation")
except ZeroDivisonError:
   print("Infinity")
```

The code signifies another example of handling the exception. Here the exception is ZeroDivisionError.

Output : Infinity

The Except Clause with No Exceptions : Though it is not considered like a good practice, but yet if it has to be done, the syntax will be

```
try:
print 'foo'+'qux'+ 7
except:
print' There is error'
You get the output
There is error
```

This type of coding is not usually preferred by programmers because it is not considered as a good coding practice. The except clause is mainly important in case of exceptions. The program has to be given certain exceptions for the smooth execution of the code. It is the main function of the except clause.

The python program containing except clause with no exception is susceptible to errors which can break the flow of the execution of the program. It is a safe practice to recognize any block susceptible to exceptions and using except clause in the case.

Example :

```
try:
  word = "python"
  print(word / 0)
except:
  print("An error occurred")
```

As you can see in the above code, there are no exceptions whatsoever. Moreover the function described in the above code is incorrect. Hence, as per the instructions, we got the following output.

Output :

An error occured

The Except Clause with Multiple Exceptions : For multiple exceptions, Python interpreter finds a matching exception, then it will execute the code written under except clause.

```
Except(Exception1, Exception2,...ExceptionN)
```

Example :

```
import sys
try:
d = 8
d = d + '5'
except(TypeError, SyntaxError)as e:
print sys.exc_info()
We get output as shown
(, TypeError("unsupported operand type(s) for
+: 'int' and 'str'",), )
```

The usage of multiple exceptions is a very handy process for the programmers. Since, the real world execution requires some aspects which can be in deviation with analytical aspect, the exceptions are handy to use. Python being particularly used for the quick development projects, prevents exceptions while saving the errors at the same time.

The try-finally Clause : There are certain guidelines using this command, either define an "except" or a "finally" clause with every try block. Also, else and finally clauses cannot be used together. Finally, clause is particularly useful when the programmer is aware with the outcome of the program.

Generally, it is seen that the else clause is helpful for the programmers with not much expertise in the programming. But when it comes to error correction and termination, this clause is particularly helpful.

For example:

```
try:
  f = open("test.txt",encoding = 'utf-8')
  # perform file operations
finally:
  f.close()
```

"Finally" is a clause to ensure any code to run without any consequences. Final statement is always to be used at the bottom of 'try' or 'except' statement. The code within a final statement always runs after the code has been executed, within try po even in 'except' blocks.

9. *Write a code to accept two numbers and display the quotient. Appropriate exception should be raised if the user enters the second number (denominator) as zero (0). Using*
(1) *use the raise statement?*
(2) *Use the assert?*

(1) The raise statement can be used to throw an exception.

The syntax of raise statement is: raise exception-name [(optional argument)]

The argument is generally a string that is displayed when the exception is raised.

Code to accept two numbers and display the quotient.

```
n = int(input("Enter Number 1 :))
m = int(input("Enter Number 2 : ))
if m == 0:
raise ZeroDivisionError
else:
print("Quotient : ,n / m)
```

(2) An assert statement in Python is used to test an expression in the program code. If the result after testing comes false, then the exception is raised. This statement is generally used in the beginning of the function or after a function call to check for valid input.

The syntax for assert statement is: assert Expressions arguments)

On encountering an assert statement, Python evaluates the expression given immediately after the assert keyword. If this expression is false, an AssertionError exception is raised which can be handled like any other exception.

```
n = int(input("Enter Number 1 :))
m = int(input("Enter Number 2 : ))
assert (m == 0), "Oops, Zero Division Error
.....
print ("Quotient: ", n/m)
```

Output -1:

```
Enter Number 1:10
Enter Number 2:2
Traceback (most recent call last):
File "D:/Python Prog/ExceptionHandling/DivideByZero.py", line 4, in
assert (m == 0), "Opps, .... ZeroDivisionError
AssertionError: Opps..... ZeroDivisionError
```

10. *Define the following:*
(1) *Exception Handling*
(2) *Throwing an exception*
(3) *Catching an exception*

(1) Exception Handling : Writing additional code in a program to give proper messages or instructions to the user on encountering an exception, called exception handling.

```
try:
    statements
except Exception Name:
    statements for handling exception
```

(2) Throwing an Exception : Throwing an exception means raising an exception.

Each time an error is detected in a program, the Python interpreter raises (throws) an exception. Exception handlers are designed to execute when a specific exception is raised.

Programmers can also forcefully raise exceptions in a program using the raise and assert statements. Once an exception is raised, no further statement in the current block of code is executed.

```
raise NameError
assert conditon, "message
```

(3) Catching an Exception : Catching an exception means handling of an exception by exception handlers. An exception is said to be caught when a code that is designed to handle a particular exception is executed. Exceptions, if any, are caught in the try block and handled in the except block.

```
try:
    statements
except Exception_Name:
    statements for handling exception
```

11. *You have learnt how to use math module in Class XI. Write a code where you use the wrong number of arguments for a method (say sqrt() or pow(). Use the exception handling process to catch the ValueError exception.*

Code to show the wrong number of arguments with exception handling

```
import math
try:
print(math.sqrt(25,6))
except TypeError:
print("wrong number of arguments used in sqrt() )
finally:
print("okay, Correct it)
```

Output:

```
Wrong number of arguments used in sqrt()
Okay, Correct it
```

12. *"Every syntax error is an exception but every exception cannot be a syntax error." Justify the statement.*

Exception is an error which occurs at Run Time, due to Syntax Errors, Run Time Errors or Logical Errors. In Python Exceptions are triggered automatically. It can be called forcefully also by using the code.

Syntax Error means errors occurs at compile time, due to not followed the rules of Python Programming Language. These errors are also known as parsing errors. On encountering a syntax error, the interpreter does not execute the program unless we rectify the errors, save and rerun the program. When a syntax error is encountered while working in shell mode, Python displays the name of the error and a small description about the error.

Because Exception can be occurred due to Syntax Errors, Logical Error and Run Time Errors, while Syntax errors occurs only due to syntax. So that "Every syntax error is an exception but every exception cannot be a syntax error."

13. What are some advantages of exceptions?

Traditional error detection and handling techniques often lead to spaghetti code hard to maintain and difficult to read. However, exceptions enable us to separate the core logic of our application from the details of what to do when something unexpected happens.

Also, since the JVM searches backward through the call stack to find any methods interested in handling a particular exception; we gain the ability to propagate an error up in the call stack without writing additional code.

Also, because all exceptions thrown in a program are objects, they can be grouped or categorized based on its class hierarchy. This allows us to catch a group exceptions in a single exception handler by specifying the exception's superclass in the catch block.

14. Difference between Error vs. Exceptions.

Error	Exceptions
All errors in Python are the unchecked type.	Exceptions include both checked and unchecked type.
Errors occur at run time which unknown to the compiler.	Exceptions can be recover by handling them with the help of try-catch blocks.
Errors are mostly caused by the environment in which an application is running.	The application itself causes exceptions.
Examples: OutofMemoryError	Examples: Checked Exceptions, SQL exception, NullPointerException,etc.

15. What is Assertion in Python?

Assertion in Python or a Python Assert Statement is one that asserts (or tests the trueness of) a condition in your code. This is also a Boolean expression that confirms the Boolean output of a condition.

Simply the Boolean statement checks the conditions applied by the user and then returns true or False. If it returns true, the program does nothing and moves to the next line of code and executes it.

However, if it's false, the program stops and displays an error. It is a debugging code hence saves a carbon copy of the indented code with the main code as well.

16. *Where is Python Assertion Used?*

Python assertion used as

(1) As a debugger in python code.
(2) Helps the coder in locating errors.
(3) Helps in copying file indexes in the library.
(4) It helps in the indirect indentation of input.
(5) Provides a shallow copy of the indented code.
(6) It removes the bugs and saves the file as original in the main source code with the help of the library.
(7) As a debugger, it generates a carbon code while interpreting the main code, which is helpful in crossing the codes.

Syntax of Python Assert

```
assert condition, error_message(optional)
```

17. *Argument of an Exception in Python describe?*

Arguments provide a useful way of analyzing exceptions with the interactive environment of Python. If the programmer is having a number of arguments which are to be used in the program, then the programmer can help in more effective results with 'except' clause. Here:

```
def print_sum_twice(a,b):
    print(a + b)
    print(a + b)
print_sum_twice(4,5)
```

In this program, the program has instructed the python print the sum twice. There are two conditions that have been specified in the code. Please note that the arguments are enclosed in the parenthesis.

Output :

```
>>>
9
9
```

□□

BCA

(SEM. III) EXAMINATION, 2022-23

BCA – 3001 : PYTHON PROGRAMMING

Time : 2 Hours Maximum Marks : 75

Note : This paper consists of three Sections A, B and C. Carefully read the instructions of each Section in solving the questions paper. Candidates have to write their answers in the given answer-copy only. No separate answer copy (B-Copy) Will be provided.

SECTION – A
(Short Answer Type Questions)

Note: All questions are compulsory. Answer the following questions as short answer type questions. Each question carries 5 marks.

1. (A) Briefly explain about Python IDLE.
   (B) Explain about Logical operator and Boolean expression.
   (C) Write a program to check a given number is prime or not. Number is given by user.
   (D) Write a program to print Armstrong numbers from 100 to 200.
   (E) Write the difference between List and Tuple.
   (F) What is module and package in python?
   (G) What is generator in python? Explain with example.
   (H) Why do we need a date class in python?
   (I) How can you raise an exception in python? Explain with example.

SECTION – B
(Long Answer Type Questions)

Note: This section contains four questions from which one question is to be answered as long question. Each question carries 15 marks.

2. What is Python programming cycle? Mention what are the rules for local and global variable in python. How can you share global variables across modules?

Or

3. Explain the need for continue, break and pass statements. Write a program in python where these three statements are implemented.

Or

4. What is dictionary in python? Write a python program to convert a given dictionary into a list of lists.

Or

5. Explain about string data type in python. What is slicing and indexing? Explain about negative indexing. Give four examples of slicing and indexing with negative indexing.

SECTION – C
(Long Answer Type Question)

Note: This section contains four questions from which one question is to be answered as long question. Each question carries 15 marks.

6. Explain about function in python. Write a python program using functions to find the roots of a quadratic equation.

Or

7. What is lambda function? Explain the features of lambda function. Write a python program to calculate the average value of the numbers in a given tuple of tuples using lambda.

Or

8. What is time module? Explain five functions of time module with example. Write a Python program to subtract five days from current date.

Or

9. What is exception? Explain about try, except, else and finally blocks used in exception using exception handling. Write a user defined exception using exception class.

❑❑

BCA

(SEM. III) EXAMINATION, 2023-24

BCA – 3001 : PYTHON PROGRAMMING

Time : Two Hours *Maximum Marks : 75*

Note : This paper consists of three Section A, B and C. Carefully read the instructions of each Section in solving the question paper. Candidates have to write their answers in the given answer-copy only. No separate answer – copy (B copy) will be provided.

SECTION – A
(Short Answer Type Questions)

Note : *All* questions are compulsory. Answer the following questions as short answer type questions. Each question carries 5 marks.

1. (A) What is python interpreter?
   (B) List the standard data types in python.
   (C) Give the features of python set.
   (D) Write a program to print all prime numbers from 1 to 100.
   (E) What is docstring?
   (F) What are the methods that are used in Python Tuple?
   (G) What is module and package in Python?
   (H) Write a simple program which illustrates Handling Exceptions.
   (I) How to print today date?

SECTION – B
(Long Answer Type Questions)

Note : This section contains four questions from which one question is to be answered as long question. Each question carries 15 marks.

2. Discuss list and dictionary data structure with example for each. Write a python program to accept n numbers and store them in a list. Then print the list without ODD numbers in it.

Or

3. Explain Tuples and Unpacking Sequences in Python Data Structure with examples.

Or

4. What is looping in python? Write a program to print the sum of the following series $1+\frac{1}{2}+\frac{1}{3}.....+\frac{1}{n}$.

Or

5. Illustrate *args and ** kwargs parameters in Python programming language with an example.

[F A]

SECTION – C
(Long Answer Type Questions)

Note : This section contains four questions from which one question is to be answered as long question. Each question carries 15 marks.

6. Explain about functions in python? Explain the type of arguments used in python function.

Or

7. How to create and import a module in python? Explain in detail with example.

Or

8. What is Date module? Explain five functions with examples used in Date module.

Or

9. Describe the need for catching exceptions using try and except statements also explain multiple except blocks used in exception handling with example.

❑❑