1. **Attempt any EIGHT of the following (out of TEN)** **[8×1=8]**

a) What are the advantages of Python? **(Chapter – 1)**

Solution:

**Note: I have written 5 advantages for 2 marks. For 1 mark you can write 2-3 advantages only**

Here are some of the key advantages of Python:

**Readability and Simplicity:** Python emphasizes readability and clean code, making it easy for developers to write and maintain code.

**Large Standard Library:** Python comes with a comprehensive standard library that provides modules and packages for a wide range of tasks.

**Community and Support:** Python has a large and active community of developers. This community contributes to the language's growth, shares knowledge, and provides support through forums, mailing lists, and other online platforms.

**Diverse Application Domains:** Python is suitable for a wide range of applications, from web development and data science to artificial intelligence and automation.

**Open Source and Free:** Python is an open-source language, and its interpreter is freely available.

b) List out main differences between list & tuple. **(Chapter – 3)**

Solution:

**Note: I have written 4 differences for 2 marks. For 1 mark you can write 1-2 advantages only**

| **Key Difference** | **List** | **Tuple** |
|---|---|---|
| Mutability | Lists are mutable, i.e. we can modify their elements by adding, removing, or changing items after the list is created. | Tuples are immutable, i.e. they are created, their elements cannot be changed or modified. We can't add, remove, or modify elements in a tuple. |
| Syntax | Lists are created using square brackets []. **Example:** mylist = [1, 2, 3]. | Tuples are created using parentheses (). **Example:** mytuple = (1, 2, 3). |
| Performance | Due to their mutability, lists may require more memory and can be slightly slower than tuples in certain operations. | Tuples are generally more memory-efficient and can be faster in certain situations because of their immutability. |
| Methods | Lists have more built-in methods compared to tuples because of their mutability. Methods like append(), extend(), remove(), and pop() can be used on lists. | Tuples have fewer built-in methods since they are immutable. Common operations include count() and index(). |

c) Python is a scripting language. Comment. **(Chapter – 1)**

Solution:

**Yes**, Python is scripting, general-purpose, high-level, and interpreted programming language. It also provides the object-oriented programming approach.

The scripting languages are interpreted language instead of a compiled language. The scripting language is referred to perform the task based on automating a repeated task. It includes same types of steps while implementing the procedure or program. It reduces time and cuts the costs further.

The name of few scripting languages is Perl, Visual Basic, JavaScript, Python, Unix Shell Scripts, ECMAScript, and Bash etc.

d) Demonstrate set with example. **(Chapter – 3)**

Solution:

**Note: I have written output of Example for 2-3 marks. For 1 mark write only example and even comment is not required for 1 mark.**

Set is an unordered collection of unique elements. Sets are defined by enclosing a comma-separated list of elements inside curly braces `{}`.

**Example:**

```
myset = {1,2,3,4,5}
print("Set:",myset)# Displaying the set
myset.add(6)       # Adding elements to the set
myset.add(3)       # Adding a duplicate element
                   #(no effect since sets only contain unique elements)
print("Modified Set:", myset)      # Displaying the modified set
myset.remove(2)                    # Removing an element from the set
print("Set after removal:", myset) # Displaying the set after removal
```

**Output:**

```
Set: {1, 2, 3, 4, 5}
Modified Set: {1, 2, 3, 4, 5, 6}
Set after removal: {1, 3, 4, 5, 6}
```

e) What is dictionary? Give example. **(Chapter – 3)**

Solution:

**Note: I have written output of Example for 2-3 marks. For 1 mark write only example and even comment is not required for 1 mark.**

Dictionary is a collection of key-value pairs. It is a mutable, unordered, and iterable data type. Each key in a dictionary must be unique, and the values can be of any data type, including other dictionaries. Dictionaries are defined using curly braces `{}`, and the key-value pairs are separated by colons `:`.

**Example:**

```
my_dict = {
    "name": "Kunal",
    "age": 20,
    "class": "NSG Academy",
}
print("Dictionary:", my_dict)            # Displaying the dictionary
my_dict["age"] = 22                      # Modifying values
print("Modified Dictionary:", my_dict)   # Displaying the modified
                                                    dictionary
my_dict["gender"] = "Male"               # Adding a new key-value pair
print("Dictionary with New Key-Value Pair:", my_dict) # Displaying the
                                                    dictionary after addition
removed_item = my_dict.pop("class")      # Removing a key-value pair
print("Dictionary after Removing 'class':", my_dict) # Displaying the
                                                    dictionary after removal
print("Removed Item:", removed_item)
```

**Output:**

```
Dictionary: {'name': 'Kunal', 'age': 20, 'class': 'NSG Academy'}
Modified Dictionary: {'name': 'Kunal', 'age': 22, 'class': 'NSG
Academy'}
Dictionary with New Key-Value Pair: {'name': 'Kunal', 'age': 22,
'class': 'NSG Academy', 'gender': 'Male'}
Dictionary after Removing 'class': {'name': 'Kunal', 'age': 22,
'gender': 'Male'}
Removed Item: NSG Academy
```

f) What is regEx? Give example. **(Chapter – 4)**

Solution:

**Note: I have written 2 different examples for understanding. For 1 mark write any one Ex.**

Regex, short for regular expression, is a powerful tool for pattern matching and manipulation of strings. In Python, the `re` module provides support for regular expressions. Regular expressions allow us to search for, match, and manipulate strings based on specific patterns.

**Example1: Matching a pattern in a string**

```
import re
text = "Hello, my email is sagrawal@nsgacademy.com and my friend's
email is mjain@nsgacademy.com."

# Pattern for matching email addresses, it is a simple representation
and may not cover all possible email address variations.
pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
```

```
# Use re.findall() to find all matches in the text
email_addresses = re.findall(pattern, text)

# Display the matched email addresses
print("Email Addresses:", email_addresses)
```

**Example 2: Searching and replacing using regex**

```
sentence = "The cat and the hat are on the mat."

# Pattern for matching "cat" and replace it with "dog"
pattern_to_replace = r'\bcat\b'
replacement = 'dog'

# Use re.sub() to replace matches in the text
modified_sentence = re.sub(pattern_to_replace, replacement, sentence)

# Display the modified sentence
print("Modified Sentence:", modified_sentence)
```

**Output of both examples:**

```
Email Addresses: ['sagrawal@nsgacademy.com', 'mjain@nsgacademy.com']
Modified Sentence: The dog and the hat are on the mat.
```

g) What is user defined Module? Give example. **(Chapter – 4)**

Solution:

**Note: I have also written how to use user-defined module in example for better understanding.**

User defined module is a file containing Python definitions and statements that can be reused in other python scripts or modules. A module can define functions, classes, and variables. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

**Example:**

Let's say we have a file named `my_module.py` with the following content:

**# my_module.py**

```
def greet(name):
    return "Hello, {name}!"
def add_numbers(a, b):
    return a + b
```

Now, in another Python script, we can import and use the functions defined in `my_module.py`:

```
# main_script.py
import my_module                        # Import the user-defined module
message = my_module.greet("NSG Academy") # Use the functions of module
print(message)
result = my_module.add_numbers(10, 20)   # Use the functions of module
print(result)
```

**Output:** When we run `main_script.py`, it will output:

```
Hello, NSG Academy!
30
```

h) Python is case sensitive language. Comment. **(Chapter – 1)**

Solution:

**Yes**, Python is a case-sensitive language. This means that it distinguishes between uppercase and lowercase letters in variable names, function names, and other identifiers. For example, the variables `myVariable`, `MyVariable`, and `myvariable` would be treated as three distinct variables in Python.

**Example:**

```
myVariable = 32
MyVariable = "NSG"
myvariable = [1, 2, 3]
print(myVariable)   # Output: 32
print(MyVariable)   # Output: NSG
print(myvariable)   # Output: [1, 2, 3]
```

i) What is dry run in Python? **(Chapter – 1)**

Solution:

A dry run is the process of a programmer manually working through their code to trace the value of variables. There is no software involved in this process.

It is used to check whether the code we have written works properly or not. It can be done using paper/whiteboard and pen.

It is very helpful to understand how the code flows from start to end and how the values of variable changes during the execution of program.

j) What is lambda function? Give example. **(Chapter – 3)**

Solution:

Lambda function is a concise way to create anonymous or unnamed functions. Lambda functions are often used for short-lived operations where a full function definition is not necessary.

**Syntax:**

```
lambda arguments: expression
```

This function can have any number of arguments but only one expression, which is evaluated and returned.

**Example:**

```
addition = lambda x, y: x + y

result = addition(5, 3)
print(result)  # Output: 8
```

**2. Attempt any FOUR of the following (out of FIVE)** **[4×2=8]**

a) Write a python program to calculate $X^Y$. **(Chapter – 2)**

Solution:

```
def power(x,y):
   ans = 1
   while y>0:
        ans = ans * x
        y-=1
   return ans

x = int(input('Enter value of X:'))
y = int(input('Enter value of Y:'))
ans = power(x,y)
print('Result of',x,'raise to',y,':',ans)
```

b) Write a python program to accept a number and check whether it is perfect number or not. **(Chapter – 2)**

Solution:

```
def isPerfect(n):
   sum = 0
   for i in range(1,n):
        if n%i == 0 :
             sum += i
   return sum==n

n = int(input('Enter a number:'))
ans = isPerfect(n)
if ans == True:
   print(n,'is a perfect number')
else:
   print(n,'is not a perfect number')
```

c) What is the use of seek() & tell() functions? **(Chapter – 4)**

Solution:

**seek():** The *seek()* method sets the current file position in a file stream and also returns the new position.

**Syntax:**

```
file_object.seek(offset)
```

when,

the offset is 0, it sets the file position at the beginning of the file
the offset is 1, it sets the file position at the current cursor position
the offset is 2, it sets the file position at the end of the file.

**Example:** Using seek() to move the cursor to a specific position

```
with open('example.txt', 'r') as file:
    file.seek(5)  # Move to the 6th byte from the beginning
    content = file.read()
    print(content)
```

**tell():** *tell()* method returns current position of file object. By default, initial position is at the beginning of the file, so the initial value returned by tell() is zero.

**Syntax:**

```
file_object.tell()
```

**Example:** Using tell() to get the current position of the cursor

```
with open('example.txt', 'r') as file:
    content1 = file.read(3)  # Read the first 3 characters
    position = file.tell()   # Get the current position
    print(f"Content1: {content1}")
    print(f"Current Position: {position}")
```

d) Demonstrate list slicing. **(Chapter – 3)**

Solution:

**Note: I have given many examples for better understanding. You can write according to marks.**

List slicing allows us to create a new list by extracting a portion of an existing list.

**Syntax:**

```
list[start:stop:step]
```

where, `start` is the index of the first element, `stop` is the index of the first element not to be included, and `step` is the step size between elements.

**Example:**

```
mylist = [10, 11, 22, 33, 44, 55, 66, 77, 88, 99]
# Basic slicing
subset1 = mylist[2:6]  # Elements from index 2 to 5 (6-1)
subset2 = mylist[4:8]  # Elements from index 4 to 7 (8-1)
subset3 = mylist[:5]   # Elements from the beginning upto index 4(5-1)
subset4 = mylist[7:]   # Elements from index 7 to the end
```

```
print("Original List:", mylist)
print("Subset1:", subset1)
print("Subset2:", subset2)
print("Subset3:", subset3)
print("Subset4:", subset4)

# Slicing with step
subset5 = mylist[1:9:2]  # Elements from index 1 to 8 with a step of 2
subset6 = mylist[::3]    # Every third element in the list
print("Subset5:", subset5)
print("Subset6:", subset6)
```

**Output:**

```
Original List: [10, 11, 22, 33, 44, 55, 66, 77, 88, 99]
Subset1: [22, 33, 44, 55]
Subset2: [44, 55, 66, 77]
Subset3: [10, 11, 22, 33, 44]
Subset4: [77, 88, 99]
Subset5: [11, 33, 55, 77]
Subset6: [10, 33, 66, 99]
```

e) A tuple is ordered collection of items. Comment. **(Chapter – 3)**

Solution:

A tuple is indeed an ordered collection of items. Tuples are similar to lists, but they have a key difference: tuples are immutable, meaning their elements cannot be changed or modified after the tuple is created.Tuples maintain the order of elements, just like lists. The order in which elements are added to a tuple is preserved.

Once a tuple is created, we cannot modify its elements. We cannot add, remove, or change elements in a tuple. This immutability makes tuples suitable for situations where the data should remain constant throughout the program.

Tuples are defined using parentheses `()` and can contain a mix of data types.

Tuples support indexing and slicing operations, similar to lists. We can access individual elements or extract sub-tuples.

**Example:**

```
mytuple = (1, 'apple', 3.14, True)

print(mytuple[0])  # Output: 1
print(mytuple[1])  # Output: 'apple'

subset = mytuple[1:3]
print(subset)  # Output: ('apple', 3.14)
```

**3. Attempt any TWO of the following (out of THREE)** **[2×4=8]**

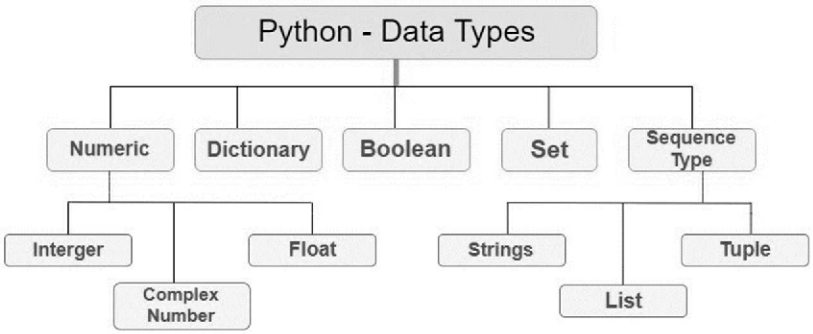
a) Write a short note on datatypes in Python. **(Chapter – 1)**

Solution:

In Python, data types represent the type of value that a variable can hold. Python is a dynamically-typed language, meaning that we don't need to explicitly declare the data type of a variable; it is inferred at runtime.

Since everything is an object in Python programming, data types are actually classes and variables are instances (object) of these classes. The following are the standard or built-in data types in Python:

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




6) Other Types (bytearray, bytes)

**Numeric Types:** It represents the data that has a numeric value. A numeric value can be an integer, a floating number or even a complex number

integer: Integer type, represents whole numbers (e.g., 5, -10).

float: Floating-point type, represents decimal numbers (e.g., 3.14, -0.5).

complex: It is specified as (real part) + (imaginary part)j. (e.g., -3+4j)

**Example:**

```
integer_var = 10
float_var = 3.14
complex_var = 3 + 4j
```

**Boolean Type:**

bool: Boolean type, represents True or False.

**Example:**

```
flag = True
status = False
```

**Sequence Types:**

str: String type, represents text (e.g., "Hello, Python!").

list: List type, an ordered and mutable collection of items.

tuple: Tuple type, an ordered and immutable collection of items.

**Example:**

```
string_var = "Hello, Python!"
list_var = [1, 2, 3, 'apple']
tuple_var = (4, 5, 6, 'orange')
```

**Set Type:**

set: Set type, an unordered and mutable collection of unique items.

**Example:**

```
set_var = {11,33,33,44} # Note :Duplicates are automatically removed
```

**Dictionary Type:**

dict: Dictionary type, an unordered collection of key-value pairs.

**Example:**

```
dict_var = {'name': Kunal, 'age': 22, 'class': 'NSG Academy'}
```

**Other Types:**

Python supports additional types such as byte sequences (`bytes`, `bytearray`), and others.

**Example:**

```
bytes_var = b'hello'
bytearray_var = bytearray([65, 66, 67])
```

b) Write a short note on exception handling. **(Chapter – 4)**

Solution:

An exception can be defined as an abnormal condition in a program which may occur during execution.

Whenever an exception occurs, it will interrupt the flow of the program and program will get aborted i.e. improper termination. Such exceptions can be handled in python by using *try* block and *except* block.

The code that can raise exceptions are kept inside *try* block and the code that handle the exception are written within *except* block.

There may be more than one *except* block after try block. There may be optional *else* block and *finally* block in exception handling mechanism.

**Syntax:**

```
try:
      #block of code(s)
except Exception1:
      #block of code(s)
except Exception2:
      #block of code(s)
else:
      #this code executes if no except block is executed
finally:
      #must executable block of code(s)
```

**Working:**

If no exception occurs, all except blocks will get skipped and program continues further.

If an exception occurs within try clause, the control immediately passes to an appropriate except block and after handling exception program continues further.

We can keep some codes in else block which will be executed if no exception occurs.

And the code which we want to be executed compulsory, can be kept in finally block. We can use a finally block along with try block only.

c) What is a module? What is package? Explain with example. **(Chapter – 4)**

Solution:

**Module**:

Module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended.

Modules allow us to organize code logically by splitting it into separate files.

Each module can define functions, variables, and classes that can be used in other Python scripts or modules.

**Package**:

A package is a way of organizing related modules into a directory hierarchy. It includes a special file called `__init__.py` to indicate that the directory should be treated as a package.

Packages help in organizing larger codebases by grouping related modules together.

Suppose we have the following directory structure:

```
my_project/
|-- main_script.py
|-- my_package/
|   |-- __init__.py
|   |-- module1.py
|   |-- module2.py
```

**Module Example :**

```
# module1.py
def greet(name):
    return f"Hello, {name}! This is module1."

# main_script.py
# Importing the greet function from module1
from my_package.module1 import greet

# Using the function
message = greet("NSG Academy")
print(message)
```

**Package Example:**

```
# module2.py

def square(x):
    return x ** 2

def cube(x):
    return x ** 3


# main_script.py

# Importing the square function from module2
from my_package.module2 import square

result = square(5) # Using the function
print(result)
```

In this example, `my_package` is a package containing two modules (`module1` and `module2`). The `__init__.py` file indicates that `my_package` is a package.

In `main_script.py`, we import the `greet` function from `module1` and the `square` function from `module2`. This allows us to use functions defined in different modules within the same package.

**4. Attempt any TWO of the following (out of THREE) [2×4=8]**

a) Write a recursive function in Python to display addition of digits in single digit. **(Chapter – 3)**

Solution:

```
def sod(n):
   if n<10:
        sum = n
   else:
        sum = sod(n%10 + sod(n//10))
   return sum

n = int(input('Enter a number:'))
ans = sod(n)
print('Addition of digits in single digit',ans)
```

b) Write a program in python to accept 'n' integers in a list, compute & display addition of all squares of theses integers. **(Chapter – 3)**

Solution:

```
mylist = []
n = int(input('Enter how many integers:'))
print('Enter',n,'integers:')
for i in range(n):
   x = int(input())
   mylist.append(x)

print("List of integers : " + str(mylist))
ans = sum([x**2 for x in mylist])
print("The sum of squares of these integers : " + str(ans))
```

c) Write a Python program to count all occurrences of "India" and "Country" in a text file "pledge.txt" **(Chapter – 4)**

Solution:

```
import string
word1 = "India"
word2 = "Country"
cnt1 = 0
cnt2 = 0
with open("pledge.txt", 'r') as myfile:
   for line in myfile:
        # Remove the leading spaces and newline character
        line = line.strip()

        # Remove the punctuation marks from the line
        line = line.translate(line.maketrans("", "", string.punctuation))

        # Split the line into words
        words = line.split(" ")

        for w in words:
             if(w == word1):
                  cnt1 = cnt1 + 1
             if(w == word2):
                  cnt2 = cnt2 + 1

print("Occurrences of the word", word1, ":", cnt1)
print("Occurrences of the word", word2, ":", cnt2)
```

**5. Attempt any ONE of the following (out of TWO)** **[1×3=3]**

a) What is the output of following code: **(Chapter – 3)**

```
X = 5
def f1():
    global X
    X = 4
def f2(a,b):
    global X
    return a+b+X
f1()
total = f2(1,2)
print(total)
```

Solution:

**Output:**

7

**Explanation:**

In given code X is assigned the value of 5.

But when *f1()* is called, it will modify the global variable X because *f1()* uses the *global* keyword to indicate that it will modify the global variable X and X is set to 4.

Function *f2(1,2)* is called with arguments 1 and 2. Inside f2(), it adds the parameters *a* and *b* along with the global variable X (which is now 4) and returns the result.

Result of *f2(1,2)* will be assigned to the variable *total.*

And then **value of *total* is printed i.e., 7**

b) What is the output of following code: **(Chapter – 3)**

```
def f(X):
    def f1(a,b):
            print("hello")
            if(b==0):
                    print("NO")
                    return
            return f(a,b)
    return f1
@f
def f(a,b):
    return a%b
f(4,0)
```

Solution:

**Output:**

```
hello
NO
```

**Explanation:**

The f(X) function takes a parameter X but doesn't use it.

Inside f(X), a nested function f1(a, b) is defined. It prints *hello* and checks if *b* is equal to 0. If it is, it prints *NO* and returns; otherwise, it calls the outer function f(a, b).

The @f decorator is used on the function f(a, b). This means that the function f(a, b) is decorated with the f function. In this case, it means that f = f(X)(f).

The decorated f function is then called with arguments 4 and 0 (f(4, 0)).

Output is **hello NO** because the inner function f1(a, b) prints *hello*, checks if *b* is 0 (which is true in this case), prints *NO*, and then returns without calling the outer function f(a, b).