**Python – Full Stack Assignment**

**Module-7 Python – Collections, functions and Modules:-**

* **Accessing List :-**

1. **Understanding how to create and access elements in a list.**

Accessing List Elements. Elements in a list can be accessed using indexing. Python indexes start at 0, so a[0] will access the first element, while negative indexing allows us to access elements from the end of the list. Like index -1 represents the last elements of list.

1. **Indexing in lists (positive and negative indexing)..**

Positive Indexing: Starts from 0 and goes up to n-1 (where n is the length of the sequence). Negative Indexing: Starts from -1 for the last element and goes up to -n for the first element.

1. **Slicing a list: accessing a range of elements.**

To access a range of elements in a list, you must slice it. One option is to use the simple slicing operator, i.e. colon (:) With this operator, you can specify where to start slicing, where to stop slicing, and the step. List slicing produces a new list from an existing one.

* **List Operations :-**

1. **Common list operations: concatenation, repetition, membership.**

* Concatenation operator (+) The (+) operator is used to add to two lists.
* Repetition operator (\*) Like string, (\*) operator replicates the string number of specified times. ...
* List Slicing in Python. List slicing returns a slice or part of the list from the given index range x to y. ...
* Membership Operator (in, not in).

.

1. **Understanding list methods like append(), insert(), remove(), pop().**

Here's an explanation of common list methods in Python: append(), insert(), remove(), and pop(). These methods are used to modify lists by adding, inserting, or removing elements.

* append(element): Adds an element to the end of the list.

my\_list = [1, 2, 3]  
 my\_list.append(4)  
 print(my\_list) # Output: [1, 2, 3, 4]

* insert(index, element): Inserts an element at a specific index in the list.

my\_list = [1, 2, 3]  
 my\_list.insert(1, 5) *# Insert 5 at index 1*  
 print(my\_list) # Output: [1, 5, 2, 3]

* remove(element): Removes the first occurrence of an element from the list.

my\_list = [1, 2, 3, 2]  
 my\_list.remove(2) *# Removes the first 2*  
 print(my\_list) # Output: [1, 3, 2]

* pop(index): Removes and returns the element at the specified index. If no index is provided, it removes and returns the last element.

my\_list = [1, 2, 3]  
 popped\_element = my\_list.pop(1) *# Removes and returns element at index 1*  
 print(my\_list) *# Output: [1, 3]*  
 print(popped\_element) *# Output: 2*  
  
 my\_list = [1, 2, 3]  
 last\_element = my\_list.pop() *# Removes and returns the last element*  
 print(my\_list) *# Output: [1, 2]*  
 print(last\_element) # Output: 3

* **Tuple:-**

1. **Introduction to tuples, immutability.**

Python Tuple is a collection of objects separated by commas. A tuple is similar to a Python list in terms of indexing, nested objects, and repetition but the main difference between both is Python tuple is immutable, unlike the Python list which is mutable.

Tuples are immutable. Some Characteristics of Tuples in Python.

* Like Lists, tuples are ordered and we can access their elements using their index values
* We cannot update items to a tuple once it is created.
* Tuples cannot be appended or extended.
* We cannot remove items from a tuple once it is created.

1. **Creating and accessing elements in a tuple.**

**Creating Tuples:-**

mixed\_type\_tuple = (1, "hello", 3.14, True)

**Accessing Tuple Elements:-**

my\_tuple = ("apple", "banana", "cherry")  
  
*# Accessing elements by index (starting from 0)*  
first\_element = my\_tuple[0] *# "apple"*  
second\_element = my\_tuple[1] *# "banana"*  
third\_element = my\_tuple[2] *# "cherry"*  
  
*# Accessing elements from the end using negative indexing*  
last\_element = my\_tuple[-1] *# "cherry"*  
second\_last\_element = my\_tuple[-2] *# "banana"*  
  
*# Slicing tuples (creating a new tuple from a subset)*  
sliced\_tuple = my\_tuple[1:3] *# ("banana", "cherry")*

1. **Basic operations with tuples: concatenation, repetition, membership.**

Concatenation of Python Tuples:-

To Concatenation of Python Tuples, we will use plus operators(+).

tup1 = (0, 1, 2, 3)

tup2 = ('python', 'geek')

# Concatenating above two

print(tup1 + tup2)

**Repetition Python Tuples**

We can create a tuple of multiple same elements from a single element in that tuple.

tup = ('python',)\*3

print(tup)

* **Accessing Tuples:-**

1. **Accessing tuple elements using positive and negative indexing.**

**Positive:-**

thistuple = ("apple", "banana", "cherry")  
print(thistuple[1])

**Negative:-**

thistuple = ("apple", "banana", "cherry")  
print(thistuple[-1])

1. **Slicing a tuple to access ranges of elements.**

thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(thistuple[2:5])

* **Dictionaries:-**

1. **Introduction to dictionaries: key-value pairs.**

A dictionary in Python is created with key-value pairs, where each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces {}. An empty dictionary without any items is written with just two curly braces, like this:.

1. **Accessing, adding, updating, and deleting dictionary elements.**

* Accessing Dictionary Items:-

d = { "name": "Alice", 1: "Python", (1, 2): [1,2,4] }

# Access using key

print(d["name"])

# Access using get()

print(d.get("name"))

* Adding and Updating Dictionary Items:-

d = {1: 'Geeks', 2: 'For', 3: 'Geeks'}

# Adding a new key-value pair

d["age"] = 22

# Updating an existing value

d[1] = "Python dict"

print(d)

* Removing Dictionary Items:-

d = {1: 'Geeks', 2: 'For', 3: 'Geeks', 'age':22}

# Using del to remove an item

del d["age"]

print(d)

# Using pop() to remove an item and return the value

val = d.pop(1)

print(val)

# Using pop item to removes and returns

# the last key-value pair.

key, val = d.popitem()

print(f"Key: {key}, Value: {val}")

# Clear all items from the dictionary

d.clear()

print(d)

1. **Dictionary methods like keys(), values(), and items().**

* Description of the dictionary methods keys(), values(), and items() in Python:
* keys(): This method returns a view object that displays a list of all the keys in the dictionary, in the order of insertion. If the dictionary is modified, the view object will reflect those changes.

Python

my\_dict = {"a": 1, "b": 2, "c": 3}  
 keys\_view = my\_dict.keys()  
 print(keys\_view) *# Output: dict\_keys(['a', 'b', 'c'])*  
  
 my\_dict["d"] = 4  
 print(keys\_view) # Output: dict\_keys(['a', 'b', 'c', 'd'])

* values(): This method returns a view object that displays a list of all the values in the dictionary, in the order of insertion. Like keys(), it reflects changes to the dictionary.

Python

my\_dict = {"a": 1, "b": 2, "c": 3}  
 values\_view = my\_dict.values()  
 print(values\_view) *# Output: dict\_values([1, 2, 3])*  
  
 my\_dict["a"] = 10  
 print(values\_view) # Output: dict\_values([10, 2, 3])

* items(): This method returns a view object that displays a list of all key-value pairs as tuples, in the order of insertion. It also reflects changes to the dictionary.

Python

my\_dict = {"a": 1, "b": 2, "c": 3}  
 items\_view = my\_dict.items()  
 print(items\_view) *# Output: dict\_items([('a', 1), ('b', 2), ('c', 3)])*  
  
 del my\_dict["b"]  
 print(items\_view) # Output: dict\_items([('a', 1), ('c', 3)])

These view objects are dynamic, meaning they update automatically when the dictionary changes. If a static list is needed, the view object can be converted to a list using list().

* **Working with Dictionaries:-**

1. **Iterating over a dictionary using loops.**

A for loop can iterate through a dictionary's keys directly. To access the values, use the key as an index. The .items() method allows direct iteration over key-value pairs.

Python

my\_dict = {'a': 1, 'b': 2, 'c': 3}  
  
*# Iterate through keys*  
for key in my\_dict:  
 print(key, my\_dict[key])  
  
*# Iterate through key-value pairs*  
for key, value in my\_dict.items():  
 print(key, value)

1. **Merging two lists into a dictionary using loops or zip().**

The zip() function pairs elements from two lists, which can then be converted into a dictionary using the dict() constructor.

Python

keys = ['a', 'b', 'c']  
values = [1, 2, 3]  
  
*# Merge lists into a dictionary*  
my\_dict = dict(zip(keys, values))  
print(my\_dict)

1. **Counting occurrences of characters in a string using dictionaries.**

A dictionary can store character counts. Iterate through the string, updating the count for each character.

my\_string = "hello"  
char\_counts = {}  
  
*# Count character occurrences*  
for char in my\_string:  
 char\_counts[char] = char\_counts.get(char, 0) + 1  
print(char\_counts)

* **Functions:-**

1. **Defining functions in Python.**

A for loop can iterate through a dictionary's keys directly. To access the values, use the key as an index. The .items() method allows direct iteration over key-value pairs.

Python

my\_dict = {'a': 1, 'b': 2, 'c': 3}  
  
*# Iterate through keys*  
for key in my\_dict:  
 print(key, my\_dict[key])  
  
*# Iterate through key-value pairs*  
for key, value in my\_dict.items():  
 print(key, value)

1. **Different types of functions: with/without parameters, with/without return values.**

* Functions without parameters and without return values: These functions perform a specific task without requiring any input and do not produce any output that can be used elsewhere.

Python

def greet():  
 print("Hello!")  
  
 greet() # Output: Hello!

* Functions with parameters but without return values: These functions accept input values (arguments) to perform operations but do not return any specific value. They might modify data or produce side effects like printing.

Python

def print\_sum(a, b):  
 print(a + b)  
  
 print\_sum(5, 3) # Output: 8

* Functions without parameters but with return values: These functions do not accept any input but produce an output value based on their internal logic or state.

Python

def get\_current\_time():  
 import datetime  
 return datetime.datetime.now()  
  
 now = get\_current\_time()  
 print(now) # Output: (current date and time)

* Functions with parameters and with return values: These functions accept input values, perform calculations or operations using them, and return a result.

Python

def multiply(x, y):  
 return x \* y  
  
 product = multiply(7, 6)  
 print(product) # Output: 42

1. **Anonymous functions (lambda functions).**

Anonymous functions, often called lambda functions, are small, unnamed functions that are defined and used in a single line. They are particularly useful for creating functions that are used only once, such as in the arguments of higher-order functions like map, filter, and sorted.

Syntax (Python):

lambda arguments : expression

Examples:

Squaring a number.

square = lambda x : x\*\*2  
 print(square(5)) # Output: 25

* **Modules :-**

1. **Introduction to Python modules and importing modules.**

Python Module is a file that contains built-in functions, classes, its and variables. There are many Python modules, each with its specific work.

In this article, we will cover all about Python modules, such as How to create our own simple module, Import Python modules, From statements in Python, we can use the alias to rename the module, etc.

1. **Standard library modules: math, random.**

The Python standard library offers several modules, with math and random being among the most useful for mathematical operations and generating pseudo-random numbers, respectively. The math module provides access to standard mathematical functions like trigonometric, logarithmic, and power functions, as well as mathematical constants like pi and e. The random module, on the other hand, provides functions for generating random numbers, including integers, floating-point numbers, and selecting random elements from sequences.

* math module:
  + Provides access to a wide range of mathematical functions, including trigonometric functions (sin, cos, tan, etc.), logarithmic functions (log, log10, etc.), exponential functions (exp, pow, etc.), and more.
  + Includes mathematical constants like math.pi and math.e.
  + Example: math.sqrt(16) returns 4.0.
* random module:
  + Provides functions for generating pseudo-random numbers.
  + Functions include:
    - random.randint(a, b): Returns a random integer between a and b (inclusive).
    - random.random(): Returns a random floating-point number between 0.0 and 1.0.
    - random.uniform(a, b): Returns a random floating-point number between a and b.
    - random.choice(sequence): Returns a random element from the given sequence.
    - random.shuffle(sequence): Shuffles the elements of the given sequence in place.
    - random.sample(population, k): Returns a list of k unique elements randomly chosen from the population.
  + Example: random.randint(1, 10) returns a random integer between 1 and 10.

1. **Creating custom modules.**

 module is simply a Python file with a .py extension that can be imported inside another Python program. The name of the Python file becomes the module name. The module contains definitions and implementation of classes, variables, and functions that can be used inside another program. **Example:** Let's create a simple module named GFG.

*''' GFG.py '''*

*# Python program to create a module*

*# Defining a function*

def Geeks():

print("GeeksforGeeks")

*# Defining a variable*

location = "Noida"

for char in my\_string:  
 char\_counts[char] = char\_counts.get(char, 0) + 1  
print(char\_counts)