Module 7) Python – Collections, functions and Modules

Accessing List

Theory:

Understanding how to create and access elements in a list.

To create a list, you define a variable and enclose your items—which can be strings, integers, or other data types—inside square brackets separated by commas, such as my_list = [10, 20, 30]. To access specific elements within that list, you use indexing by placing the position number of the item in square brackets immediately following the list name. In most programming languages like Python, indexing starts at zero, meaning my_list[0] retrieves the first element, my_list[1] retrieves the second, and so on. You can also use negative indexing, where -1 refers to the last item in the list, allowing for flexible data retrieval regardless of the list's total length.

Indexing in lists (positive and negative indexing).

Indexing in a list allows you to locate specific elements using their numerical position, starting with positive indexing which begins at 0 for the first item and increases from left to right. Negative indexing provides a convenient way to access elements from the end of the list, where -1 represents the last item, -2 the second to last, and so on. This dual system ensures you can quickly reference data regardless of whether you know the exact length of the collection. By placing the index number inside square brackets after the list name, you can retrieve, modify, or delete specific entries efficiently. Understanding both directions is essential for navigating data structures and performing operations like slicing or reversing sequences.

Slicing a list: accessing a range of elements.

Slicing a list involves using a colon within square brackets to define a range of elements, typically following the syntax [start:stop]. The first index is inclusive, while the stop index is exclusive, meaning the slice extracts every item from the beginning point up to, but not including, the end point. You can also include a third parameter called a step to skip items, such as [0:10:2] to select every second element in that range. Omitting the start index defaults to the beginning of the list, while omitting the stop index captures everything until the very end. This technique is a powerful tool for creating sub-lists or reversing sequences without altering the original data structure.

Lab:

Write a Python program to create a list with elements of multiple data types (integers, strings, floats, etc.).

```
mixed_list = [10, "hello", 3.14, True, [1, 2]]
print(mixed_list)
```

Write a Python program to access elements at different index positions.

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]

print(fruits[0])   # first element
print(fruits[2])   # third element
print(fruits[-1])  # last element
print(fruits[-3])  # third from last
```

Practical Examples:

1. Write a Python program to create a list of multiple data type elements.

```
my_data = [25, "python", 9.81, False]
print(my_data)
```

2. Write a Python program to find the length of a list using the len() function.

```
items = ["laptop", "mouse", "keyboard", "monitor"]
list_length = len(items)
print(list_length)
```

5. Accessing Tuples

Theory:

Accessing tuple elements using positive and negative indexing.

Tuple elements are accessed by placing the index number inside square brackets, functioning identically to list indexing but applied to immutable sequences. Positive indexing starts at 0 for the first element and moves forward, allowing you to pinpoint specific data from the beginning of the tuple. Negative indexing starts at -1 for the last element and moves backward, which is particularly useful for retrieving items relative to the end without knowing the total length. Since tuples are ordered, each item maintains a fixed position that can be referenced quickly for data retrieval. This system of dual indexing provides a flexible and efficient way to read stored values while ensuring the underlying data remains protected from accidental modification.

Slicing a tuple to access ranges of elements.

Slicing a tuple allows you to extract a specific portion of the sequence by defining a range with a colon, such as my_tuple[start:stop]. The resulting slice creates a new tuple containing all elements from the start index up to, but not including, the stop index.

Lab:

Write a Python program to access values between index 1 and 5 in a tuple.

```
numbers = (0, 10, 20, 30, 40, 50, 60)
result = numbers[1:5]
print(result)
```

Write a Python program to access alternate values between index 1 and 5 in a tuple.

```
data = (0, 10, 20, 30, 40, 50, 60)
result = data[1:5:2]
print(result)
```

Practical Examples:

11) Write a Python program to access values between index 1 and 5 in

a tuple.

```
my_tuple = (10, 20, 30, 40, 50, 60, 70)
sub_tuple = my_tuple[1:5]
print(sub_tuple)
```

12) Write a Python program to access the value from the last index in a tuple.

```
my_tuple = ("red", "green", "blue", "yellow")
last_element = my_tuple[-1]
print(last_element)
```

6. Dictionaries

Theory:

Introduction to dictionaries: key-value pairs.

A dictionary is a built-in data structure that stores data in unordered collections of key-value pairs, where each unique key acts as an identifier for its associated value. Unlike lists that use numerical indexing, dictionaries allow you to retrieve information using descriptive keys, such as a name or a code, making data management more intuitive. Each key is separated from its value by a colon, and each pair is separated by commas, all enclosed within curly braces. Because keys must be unique and immutable, they provide a highly efficient way to look up, insert, or delete data even in very large datasets.

Accessing, adding, updating, and deleting dictionary elements.

To access a value in a dictionary, you reference its unique key inside square brackets, such as my_dict["name"], or use the .get() method to avoid errors if the key is missing. Adding a new element or updating an existing one follows the same syntax: simply assign a value to a key, like my_dict["age"] = 25, which either creates the pair or overwrites the current value. To remove elements, the del keyword or the .pop() method can be used to target a specific key for deletion. Dictionaries also provide a .clear() method to remove all entries at once, leaving the dictionary empty.

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

Python dictionaries provide built-in methods to extract specific parts of the data structure for easier iteration and analysis. The .keys() method returns a view object containing all the unique identifiers in the dictionary, which is useful for checking if a specific entry exists. The .values() method retrieves only the data associated with those keys, allowing you to perform calculations or display content without needing the labels. Finally, the .items() method returns pairs of keys

and values as tuples, making it the most efficient way to loop through an entire dictionary when you need both the identifier and its data simultaneously.

Lab:

Write a Python program to create a dictionary with 6 key-value pairs.

```
student_info = {
   "name": "alice",
   "age": 21,
   "major": "computer science",
   "gpa": 3.8,
   "is_enrolled": True,
   "graduation_year": 2026
}
print(student_info)
```

Write a Python program to access values using dictionary keys.

```
car = {
   "brand": "ford",
   "model": "mustang",
   "year": 1964
}
brand_name = car["brand"]
model_name = car["model"]
print(brand_name)
print(model_name)
```

Practical Examples:

13) Write a Python program to create a dictionary of 6 key-value pairs.

```
employee = {
   "id": 101,
   "name": "john",
   "department": "it",
   "position": "developer",
   "salary": 50000,
   "city": "new york"
}
print(employee)
```

14) Write a Python program to access values using keys from a dictionary

```
student = {
   "name": "sophia",
   "grade": "a",
```

```
    "subject": "mathematics"
}

name_val = student["name"]
subject_val = student["subject"]

print(name_val)
print(subject_val)
```

7. Working with Dictionaries

Theory:

Iterating over a dictionary using loops.

Iterating through a dictionary using a for loop allows you to process each entry by targeting keys, values, or both simultaneously. By default, looping directly over a dictionary iterates through its keys, which can then be used to retrieve corresponding values. Using the .items() method is the most common approach as it provides both the key and the value in each iteration, making data manipulation straightforward.

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

The zip() function pairs elements from two lists into tuples, which can be converted directly into a dictionary using the dict() constructor. Alternatively, a for loop can iterate through the list indices to manually assign each key-list element to a corresponding value-list element. Both methods effectively transform parallel data into a structured key-value format for easier access.

Counting occurrences of characters in a string using dictionaries.

To count character frequencies, you can iterate through a string and use each character as a key in a dictionary. For every character encountered, you increment its corresponding value if it already exists or initialize it to 1 if it is new. This creates a complete map showing exactly how many times every character appears within the text.

Lab:

Write a Python program to update a value in a dictionary.

```
user_profile = {
    "username": "coder123",
    "status": "offline",
    "points": 50
}
user_profile["status"] = "online"

print(user_profile)
```

Write a Python program to merge two lists into one dictionary using a loop.

```
keys = ["name", "age", "city"]
values = ["mark", 28, "berlin"]
merged_dict = {}
for i in range(len(keys)):
    merged_dict[keys[i]] = values[i]

print(merged_dict)
```

Practical Examples:

15) Write a Python program to update a value at a particular key in a dictionary.

```
product = {
    "id": 501,
    "name": "laptop",
    "price": 1200
}
product["price"] = 1150
print(product)
```

16) Write a Python program to separate keys and values from a dictionary using keys() and values() methods.

```
student_data = {
    "name": "kevin",
    "course": "python",
    "grade": "a"
}
all_keys = list(student_data.keys())
all_values = list(student_data.values())

print("keys:", all_keys)
print("values:", all_values)
```

17) Write a Python program to convert two lists into one dictionary using a for loop.

```
keys = ["fruit", "color", "taste"]
values = ["apple", "red", "sweet"]
result_dict = {}
for i in range(len(keys)):
    result_dict[keys[i]] = values[i]
print(result_dict)
```

18) Write a Python program to count how many times each character appears in a string.

```
text = "programming"
char_count = {}
for char in text:
    char_count[char] = char_count.get(char, 0) + 1
print(char_count)
```

8. Functions

Theory:

Defining functions in Python.

In Python, you define a function using the def keyword followed by the function name and parentheses for optional parameters. This allows you to group a block of code into a reusable unit that executes only when specifically called by its name. Functions help organize programs by breaking complex tasks into smaller, manageable parts that can return specific results using the return statement.

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

Python functions are categorized by how they handle inputs and outputs, allowing for flexibility in how code is structured. A function without parameters and without a return value simply executes a static block of code, while one with parameters accepts external data to perform dynamic tasks

Anonymous functions (lambda functions).

Lambda functions are small, anonymous functions in Python defined using the lambda keyword instead of the standard def syntax. They are restricted to a single expression and are typically used for short-term tasks where a full function definition would be unnecessary

Lab:

Write a Python program to create a function that takes a string as input and prints it.

```
def display_message(text):
    print(text)
display_message("hello, welcome to python functions!")
```

Write a Python program to create a calculator using functions.

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b
```

```
def divide(a, b):
    if b == 0:
        return "error: division by zero"
    return a / b

num1 = float(input("enter first number: "))
num2 = float(input("enter second number: "))
op = input("enter operator (+, -, *, /): ")

if op == "+":
    print(add(num1, num2))
elif op == "-":
    print(subtract(num1, num2))
elif op == "*":
    print(multiply(num1, num2))
elif op == "/":
    print(divide(num1, num2))
else:
    print("invalid operator")
```

Practical Examples:

19) Write a Python program to print a string using a function.

```
def print_text(message):
    print(message)
print_text("hello world")
```

20) Write a Python program to create a parameterized function that takes two arguments and prints their sum.

```
def add_numbers(a, b):
    print(a + b)
add_numbers(10, 20)
```

21) Write a Python program to create a lambda function with one expression.

```
square = lambda x: x * x
print(square(5))
```

22) Write a Python program to create a lambda function with two expressions

```
multi_expression = lambda a, b: (print(f"adding {a} and {b}"), a + b)
result = multi_expression(5, 10)
print(f"result: {result[1]}")
```

9. Modules

Theory:

Introduction to Python modules and importing modules.

Python modules are files containing Python code—including functions, classes, and variables—that you can reuse across different programs. By organizing code into modules, you make your projects more modular, readable, and easier to maintain. To use the contents of a module in your current script, you use the import keyword, which loads the specified module into your program's memory

Standard library modules: math, random.

The math module provides access to mathematical functions like sqrt(), pow(), and trigonometric operations for precise numerical calculations. The random module allows you to generate pseudo-random numbers, shuffle sequences, or select random elements from a list. Both are part of Python's standard library, meaning they are built-in and ready to use by simply adding an import statement at the start of your script.

Creating custom modules.

Creating a custom module in Python involves saving a script with a .py extension containing functions, classes, or variables you want to reuse. For example, if you save a file named mymodule.py with specific logic, that file name becomes the module name you can reference in other scripts. This practice helps in organizing large projects by separating logic into distinct, manageable files.

Lab:

Write a Python program to import the math module and use functions like sqrt(), ceil(), floor().

```
import math
number = 15.7
print(math.sqrt(number))
print(math.ceil(number))
print(math.floor(number))
```

Write a Python program to generate random numbers using the random module.

```
import random
print(random.random())
print(random.randint(1, 100))
print(random.randrange(0, 10, 2))
```

Practical Examples:

23) Write a Python program to demonstrate the use of functions from the math module.

```
import math

print(math.sqrt(64))
print(math.pow(2, 3))
print(math.ceil(4.2))
print(math.floor(4.8))
print(math.factorial(5))
print(math.pi)
```

24) Write a Python program to generate random numbers between 1 and 100 using the random module.

```
import random

print(random.randint(1, 100))
```