**Module – 14**

**Collections, functions and Modules**

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

**Ans. In Python, a list is created using square brackets [] and can hold any data type.**

**For example:**

**Fruits = [“apple”, “mango”, “grapes”, “pineapple”]**

**Print(fruits[0]) # output = “apple”**

**Print(fruits[1]) #output = “mango”**

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

**Ans. Positive indexing starts from 0, moving left to right.**

**Item = [“pen”, “notebook”, “eraser”, “pencil”]**

**Print(item[0]) #output = “pen”**

**Print(item[2] #output = “eraser”**

**Negative indexing starts from -1, moving right to left**

**Print(item[-1]) #output = “pencil”**

**Print(item[-3]) #output = “notebook”**

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

**Ans. Slicing lets you extracts a portion of a list using list[start:end]. It grabs from start up to but not including end.**

**Colors = [“red”, “green”, “blue”, “yellow”, “purple”]**

**Print(colors[1:4]) #[‘green’, ‘blue’, ‘yellow’]**

* **Start at index 1 – green**
* **Ends just before index 4 – so “purple” is not included.**

**List[start:end:step]**

* **Start : where to begin (default is 0)**
* **End: where to stop (exclusive)**
* **Step: how many items to skip (default is 0).**

**Example:**

**Positive indexing:**

**Print(colors[0:5:2]) # [‘red’, ‘blue’, ‘purple’]**

**Negative indexing:**

**Print(colors[-4:-1]) #[‘green’, ‘blue’, ‘yellow’]**

**Print(colors[:3]) # first 3 items**

**Print(colors[2:]) # from index 2 to end**

**Print(colors[:]) # whole list (a copy!)**

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

**Ans. Concatenation: combing two or more lists into one.**

**List1 = [1,2,3]**

**List2 = [4,5,6]**

**Result = list1 + list2**

**Print(result) # [1,2,3,4,5,6]**

**Repetition: repeating the elements in a list a certain number of times.**

**List1 = [a,b]**

**Result = list1 \* 3 # output: [a,b,a,b,a,b]**

**Membership: checking if an item is present in the list.**

**List1 = [1,2,3]**

**Print(2 in list1) # output: True**

**Print(5 not in list1) # output: True**

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

**Ans. Append() : adds a item to the end of the list.**

**Fruits = [‘apple’, ‘banana’]**

**Fruits.append(mango)**

**Print(fruits) #output: [‘apple’, ‘banana’, ‘mango’]**

**Insert: insert a value at a specific index.**

**Fruits = [‘apple’, ‘banana’]**

**Fruits.insert(1, ‘orange’) # inserts before index 1**

**Print(fruits) #output : [‘apple’, ‘orange’,’banana’]**

**Remove(): remove the first occurrence of a value.**

**Fruits = [‘apple’, ‘banana’, ‘apple’]**

**Fruits.remove(‘apple’)**

**Print(fruits) # output: [‘banana’, ‘apple’]**

**Pop(): removes and returns the items at the index. If no index is provided, removes the last item.**

**Fruits = [‘apple’, ‘banana’, ‘mango’]**

**Item = fruits.pop(1)**

**Print(item) #output: ‘banana’**

**Print(fruits) # output: [‘apple’, ‘mango’]**

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

**Ans. For loop: best for when you want to access each element directly.**

**Items = [‘pen’, ‘book’, ‘eraser’]**

**For item in items:**

**Print(item)**

**Loop with range: useful when you need index too.**

**Items = [‘pen’, ‘book’, ‘eraser’]**

**For I in range(len(items)):**

**Print(f “Item {i}: {items[i]”})**

**While loop: gives a full control with condition-based iteration.**

**Items = [‘pen’, ‘book’, ‘eraser’]**

**I=0**

**While i< len(items):**

**Print(items[1])**

**I += 1**

1. **Sorting and reversing a list using sort(), sorted(), and reverse().**

**Ans. Sort():Sorts the list *in place* (alters the original list).**

**numbers = [5, 2, 9, 1]**

**numbers.sort()**

**print(numbers) # Output: [1, 2, 5, 9]**

**# modifies the original list.**

**You can also sort in descending order.**

**numbers.sort(reverse=True)**

**print(numbers) # Output: [9, 5, 2, 1]**

**Sorted():Returns a *new sorted list* (original remains unchanged).**

**numbers = [5, 2, 9, 1]**

**sorted\_numbers = sorted(numbers)**

**print(sorted\_numbers) # Output: [1, 2, 5, 9]**

**print(numbers) # Original is untouched: [5, 2, 9, 1]**

**reverse():Reverses the list *in place*—does NOT sort!**

**items = ['a', 'b', 'c']**

**items.reverse()**

**print(items) # Output: ['c', 'b', 'a']**

1. **Basic list manipulations: addition, deletion, updating, and slicing.**

**Ans. Addition: You can add items using append() or insert():**

**my\_list = [1, 2, 3]**

**my\_list.append(4) # Adds to the end**

**my\_list.insert(1, 'new') # Inserts at index 1**

**print(my\_list) # Output: [1, 'new', 2, 3, 4]**

**Deletion: you can delete by value, index And slice.**

**my\_list = [1, 2, 3, 4]**

**my\_list.remove(2) # Removes first occurrence of 2**

**del my\_list[0] # Deletes item at index 0**

**popped = my\_list.pop() # Removes and returns last item**

**print(my\_list) # Output: [3, 4]**

**updating element: Direct assignment lets you change values.**

**my\_list = ['a', 'b', 'c']**

**my\_list[1] = 'z'**

**print(my\_list) # Output: ['a', 'z', 'c']**

**slicing: Slice with [start:stop] or [start:stop:step].**

**my\_list = [10, 20, 30, 40, 50]**

**print(my\_list[1:4]) # Output: [20, 30, 40]**

**print(my\_list[::-1]) # Output: [50, 40, 30, 20, 10]**

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

**Ans. Tuples:**

* **Tuples are like lists, but *immutable*.**
* **Defined using parentheses: my\_tuple = (1, 2, 3)**
* **Can hold mixed data types.**
* **Ideal for fixed collections—like coordinates, or read-only settings.**

**Immutability:**

* **Tuples *cannot* be changed after creation.**
* **No methods like append(), remove(), or pop().**
* **Prevents accidental modification—great for safety-critical data.**
* **example = (10, 20, 30) # example[1] = 99 Raises TypeError**

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

**Ans. Creating a tuple: Just use parentheses () — or leave them out in some cases!**

**my\_tuple = (10, 20, 30)**

**empty\_tuple = ()**

**single\_item = (5,) # Note the trailing comma!**

**mixed = ('Rajkot', 42, True)**

**Accessing tuple element: Use indexing just like lists (but you can't modify the values).**

**print(my\_tuple[0]) # Output: 10**

**print(my\_tuple[-1]) # Output: 30**

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

**Ans. Concatenation:**

**Join two tuples to create a new one.**

**t1 = (1, 2) t2 = (3, 4) result = t1 + t2 print(result) # Output: (1, 2, 3, 4)**

**Returns a new tuple since tuples can't be changed in place.**

**Repetition:**

**Repeat a tuple’s contents multiple times.**

**t = ('A', 'B') print(t \* 3) # Output: ('A', 'B', 'A', 'B', 'A', 'B')**

**Great for generating predictable patterns or test data.**

**Membership:**

**Check if a value exists inside a tuple.**

**t = (10, 20, 30) print(20 in t) # Output: True print(99 not in t) # Output: True**

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

**Ans. Positive Indexing: Starts from 0 for the first element.**

**t = ('apple', 'banana', 'cherry') print(t[0]) # Output: 'apple' print(t[1]) # Output: 'banana'**

**Great when you know the exact position from the start of the tuple.**

**Negative Indexing**

**Starts from -1 for the last element.**

**t = ('apple', 'banana', 'cherry') print(t[-1]) # Output: 'cherry' print(t[-2]) # Output: 'banana'**

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

**Ans. Tuple Slicing Syntax**

**tuple\_name[start:stop:step]**

**start: starting index (inclusive)**

**stop: ending index (exclusive)**

**step: (optional) stride or jump**

**Examples:**

**t = (10, 20, 30, 40, 50, 60) print(t[1:4]) # Output: (20, 30, 40) print(t[:3]) # Output: (10, 20, 30) print(t[::2]) # Output: (10, 30, 50) print(t[::-1]) # Output: (60, 50, 40, 30, 20, 10) — reversed!**

**Slicing always returns a new tuple and never affects the original one.**

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

**Ans. What Is a Dictionary?**

* **A collection of key-value pairs**
* **Unordered (in older versions) and mutable**
* **Created using {} or the dict() constructor**

**student = {'name': 'Bhagirath', 'age': 21, 'grade': 'A'}**

**Each entry is:**

* **Key: A unique identifier (e.g., 'name')**
* **Value: Data associated with the key (e.g., 'Bhagirath')**

**Why Use Dictionaries?**

* **Fast lookups via keys**
* **Ideal for storing structured data (like user profiles or transaction logs)**
* **Easy to update and expand dynamically**

**Accessing Values**

**print(student['name']) # Output: 'Bhagirath'**

**Raises error if key doesn’t exist—use .get() for safety:**

**print(student.get('email', 'Not Provided'))**

**Tuples are great for fixed data—but dictionaries? They’re your tool for building scalable, readable systems. Want to build a CLI user manager or banking app module that leverages dictionaries? I'm ready to scaffold it with you step-by-step.**

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

**Ans. Accessing Elements:**

**Use the key to retrieve its value:**

**data = {'name': 'Bhagirath', 'age': 21} print(data['name']) # Output: 'Bhagirath'**

**Raises KeyError if the key doesn’t exist—use .get() for safe access:**

**print(data.get('email', 'Not Found')) # Output: 'Not Found'**

**Adding Elements:**

**Just assign a new key with a value:**

**data['city'] = 'Rajkot' print(data) # Output includes 'city': 'Rajkot'**

**This works even if the key didn’t exist before.**

**Updating Elements:**

**Reassign the value to an existing key:**

**data['age'] = 22**

**Or use .update() with multiple entries:**

**data.update({'grade': 'A', 'email': 'bhag@example.com'})**

**Deleting Elements:**

* **Use del to remove by key:**

**del data['email']**

* **Use .pop() to remove and *return* the value:**

**removed = data.pop('grade') print(removed) # Output: 'A'**

**Avoid errors by confirming the key exists—or use .pop(key, default).**

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

**Ans. keys()**

**Returns a view of all the dictionary’s keys.**

**info = {'name': 'Bhagirath', 'city': 'Rajkot'} print(info.keys()) # Output: dict\_keys(['name', 'city'])**

**Useful when validating keys or looping over field names.**

**values()**

**Returns a view of all values in the dictionary.**

**print(info.values()) # Output: dict\_values(['Bhagirath', 'Rajkot'])**

**Great for checking what data is stored—like user inputs or transaction totals.**

**items()**

**Returns key-value pairs as tuples.**

**print(info.items()) # Output: dict\_items([('name', 'Bhagirath'), ('city', 'Rajkot')])**

**Perfect for unpacking in loops:**

**for key, value in info.items(): print(f"{key}: {value}")**

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

**Ans. Looping Over Keys:**

**data = {'name': 'Bhagirath', 'city': 'Rajkot'} for key in data: print(key) # Output: name, city**

**Equivalent to: for key in data.keys()**

**Looping Over Values:**

**for value in data.values(): print(value) # Output: Bhagirath, Rajkot**

**Great for analyzing stored info without worrying about labels.**

**Looping Over Key-Value Pairs:**

**for key, value in data.items(): print(f"{key}: {value}")**

**Perfect for formatted output, logs, or summaries.**

**Conditional Looping:**

**for key in data: if key.startswith('c'): print(f"{key} → {data[key]}")**

**Adds power when filtering config options, fields, etc.**

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

**Ans. Using zip()**

**Elegant and compact—pairs items by index.**

**keys = ['name', 'age'] values = ['Bhagirath', 21] merged = dict(zip(keys, values)) print(merged) # Output: {'name': 'Bhagirath', 'age': 21}**

**Great when both lists are the same length.**

**Using a Loop:**

**Gives you flexibility and error handling.**

**keys = ['name', 'age'] values = ['Bhagirath', 21] merged = {} for i in range(len(keys)): merged[keys[i]] = values[i]**

**Add a check if len(keys) != len(values) for robustness.**

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

**Ans. Basic Approach:**

**text = "hello world" char\_count = {} for char in text: if char in char\_count: char\_count[char] += 1 else: char\_count[char] = 1 print(char\_count)**

**Each character becomes a key, and its count becomes the value.**

**With dict.get() Simplification**

**text = "hello world" char\_count = {} for char in text: char\_count[char] = char\_count.get(char, 0) + 1**

**Avoids the if-else block with cleaner logic.**

**Using collections.Counter (Optional Shortcut)**

**from collections import Counter char\_count = Counter("hello world")**

**Returns a dictionary-like object that handles counting automatically.**

1. **Defining functions in Python.**

**Ans. Syntax**

**def function\_name(parameters):**

**# function body**

**Return value**

**Example:**

**def greet(name):**

**return f"Hello, {name}!"**

**Functions let you organize, reuse, and modularize your logic**

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

**Ans. Function Types**

| **Function Type** | **Syntax Example** | **Use Case** |
| --- | --- | --- |
| **With parameters, with return** | **def add(a, b): return a + b** | **Calculations, data processing** |
| **With parameters, no return** | **def greet(name): print(f"Hi {name}")** | **Displaying output, logging** |
| **No parameters, with return** | **def get\_pi(): return 3.14** | **Constants, default values** |
| **No parameters, no return** | **def show\_welcome(): print("Welcome!")** | **Setup steps, simple CLI prompts** |

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

**Ans. A lambda function is just a compact, anonymous way of writing a function in Python. Instead of using def, you write lambda, give it some inputs, and immediately say what it returns—all in one line.**

**square = lambda x: x \* x**

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

**Ans. What’s a Module?**

**A module is a Python file (.py) containing reusable code—functions, variables, classes. It helps you organize code logically and keep things modular.**

**Importing a Module**

**import math print(math.sqrt(16)) # Output: 4.0**

**You can also use:**

* **from math import sqrt**
* **import my\_custom\_module**

**Why Use Modules?**

* **Keeps code clean and maintainable**
* **Promotes reusability across projects**
* **Let’s you split logic into functional units (e.g., storage, operations, main)**

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

**Ans. math Module**

**Provides advanced mathematical functions beyond basic arithmetic.**

**import math print(math.sqrt(25)) # Square root - 5.0 print(math.pow(2, 3)) # Power - 8.0 print(math.pi) # Pi constant - 3.14159... print(math.floor(2.7)) # Rounds down - 2**

**Great for financial apps, geometry, or scientific tools.**

**random Module**

**Used for generating pseudo-random numbers and choices.**

**import random print(random.randint(1, 10)) # Random int from 1 to 10 print(random.choice(['a', 'b'])) # Random pick from a list print(random.shuffle(my\_list)) # Shuffles a list in-place**

1. **Creating custom modules.**

**Ans. What’s a Custom Module?**

**Your own .py file containing reusable code—functions, variables, or classes—that can be imported into other Python files.**

**How to Create One**

**File: greetings.py def say\_hello(name): return f"Hello, {name}!"**

**How to Use It:**

**File: main.py import greetings print(greetings.say\_hello("Bhagirath")) # Output: Hello, Bhagirath!**

**Make sure both files are in the same folder, or set up a proper module path.**