Module 14) Python – Collections, functions and Modules

Accessing List Theory:

• Understanding how to create and access elements in a list.

A **list** in Python is an ordered collection of items (which can be of any data type).

**Creating a List:**

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

**Accessing Elements:**

Use **index numbers**, starting from 0.

• Indexing in lists (positive and negative indexing).

Indexing refers to the position of elements in a list.

#### Positive Indexing:

* Starts from the beginning of the list.
* The first element is at index 0, the second at 1, and so on.

Example:

fruits = ["apple", "banana", "cherry"]

print(fruits[1]) # Output: banana

#### Negative Indexing:

* Starts from the end of the list.
* The last element is at index -1, the second last at -2, and so on.

Example:

print(fruits[-1]) # Output: cherry

• Slicing a list: accessing a range of elements.

Slicing is used to retrieve a subset or range of elements from a list.

#### Syntax:

list[start:stop]

* start: The index to begin the slice (inclusive).
* stop: The index to end the slice (exclusive).

Example:

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

print(fruits[1:4]) # Output: ['banana', 'cherry', 'date']

2. List Operations Theory:

• Common list operations: concatenation, repetition, membership.

| **Operation** | **Symbol** | **Description** |
| --- | --- | --- |
| Concatenation | + | Joins two or more lists |
| Repetition | \* | Repeats the list multiple times |
| Membership | in / not in | Checks if an element exists in the list |

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

| **Method** | **Function** | **Modifies List?** | **Returns Value?** |
| --- | --- | --- | --- |
| append() | Adds an item to the end | Yes | No |
| insert() | Adds an item at a given index | Yes | No |
| remove() | Removes the first matching item | Yes | No |
| pop() | Removes item by index (default last) | Yes | Yes |

3. Working with Lists Theory:

• Iterating over a list using loops.

| **Loop Type** | **Best Used When...** |
| --- | --- |
| for item in list | You only need values (simple and clean) |
| for i in range(len(list)) | You need **index and value** |
| while loop | More control over iteration steps or conditions |

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

| **Method** | **Action** | **Modifies Original List?** | **Sorts Data?** |
| --- | --- | --- | --- |
| sort() | Sorts the list in place | Yes | Yes |
| sorted() | Returns a new sorted list | No | Yes |
| reverse() | Reverses element order | Yes | No |

• Basic list manipulations: addition, deletion, updating, and slicing.

| **Operation** | **Method(s)** | **Description** |
| --- | --- | --- |
| Addition | append(), insert(), extend() | Add new elements to the list |
| Deletion | remove(), pop(), del | Remove elements from the list |
| Updating | Index assignment (list[i] = x) | Change the value of existing elements |
| Slicing | list[start:stop] | Access or modify a range of elements |

4. Tuple Theory:

• Introduction to tuples, immutability.

**A tuple is a collection data type in Python that is:**

* Ordered (elements have a defined position)
* Immutable (cannot be changed after creation)
* Allows duplicate values
* Defined using parentheses ()

**Creating a Tuple** : tuple\_name = (item1, item2, item3)

**Immutability :**

Immutability means that once a tuple is created, its elements cannot be changed, added, or removed.

### Why Use Tuples?

* Data protection: Tuples are used when data should not change.
* Faster than lists (because they are immutable).
* Often used as keys in dictionaries or for fixed data structures.

• Creating and accessing elements in a tuple.

### 1. Creating a Tuple

A **tuple** is created by placing elements inside **parentheses ()**, separated by commas.

#### Syntax: tuple\_name = (element1, element2, element3, ...)

### 2. Accessing Elements in a Tuple

Just like lists, tuple elements are accessed using **indexing**. Indexes start from **0**.

#### Syntax: tuple\_name[index]

• Basic operations with tuples: concatenation, repetition, membership.

| **Operation** | **Symbol / Keyword** | **Description** |
| --- | --- | --- |
| Concatenation | + | Combines two or more tuples |
| Repetition | \* | Repeats the tuple multiple times |
| Membership | in, not in | Checks if an element exists in tuple |

5. Accessing Tuples Theory:

• Accessing tuple elements using positive and negative indexing.

### 1. Positive Indexing

* Starts from **0** for the **first element**.
* Each next element increases the index by 1.

**Syntax: tuple\_name[index]**

colors = ("red", "green", "blue", "yellow")

print(colors[0]) # Output: red

print(colors[2]) # Output: blue

### 2. Negative Indexing

* Starts from **-1** for the **last element**.
* Each previous element decreases the index by 1 (goes more negative).

**Syntax: tuple\_name[-index]**

colors = ("red", "green", "blue", "yellow")

print(colors[-1]) # Output: yellow

print(colors[-3]) # Output: green

• Slicing a tuple to access ranges of elements.

| **Slice Notation** | **Result** | **Description** |
| --- | --- | --- |
| tuple[start:stop] | Subset from start to stop-1 | Standard slice |
| tuple[:stop] | From beginning up to stop-1 | Starts from the first element |
| tuple[start:] | From start to the end | Goes till the last element |
| tuple[:] | Whole tuple | Copy of the entire tuple |
| Negative indices | Works from the end | Useful for slicing backwards |

6. Dictionaries Theory:

• Introduction to dictionaries: key-value pairs.

A **dictionary** is a **collection data type** in Python used to store data in **key-value pairs**.

* Each **key** is unique and is used to access its associated **value**.
* Dictionaries are **unordered** (prior to Python 3.7), but from Python 3.7+ they preserve insertion order.
* Dictionaries are **mutable** — you can add, update, or delete items.

**Key-Value Pairs**

* **Key:** Acts like an **identifier** or **index** (usually a string, number, or tuple).
* **Value:** The **data** or **information** associated with the key (can be any data type).

**Syntax:** ]

dictionary\_name = {

key1: value1,

key2: value2,

key3: value3

}

• Accessing, adding, updating, and deleting dictionary elements.

| **Operation** | **Syntax** | **Description** |
| --- | --- | --- |
| Access | dict[key] or dict.get(key) | Retrieve value by key |
| Add | dict[new\_key] = value | Add a new key-value pair |
| Update | dict[key] = new\_value | Change value for an existing key |
| Delete | del dict[key] or dict.pop(key) | Remove key-value pair from dictionary |

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

### 1. keys()

* Returns a **view object** containing all the **keys** in the dictionary.
* Can be converted to a list if needed.

### 2. values()

* Returns a **view object** containing all the **values** in the dictionary.

### 3. items()

* Returns a **view object** containing all **key-value pairs** as tuples (key, value).

7. Working with Dictionaries Theory:

• Iterating over a dictionary using loops.

| **Loop Type** | **Syntax** | **Description** |
| --- | --- | --- |
| Keys | for key in dict: | Loop through keys |
| Values | for value in dict.values(): | Loop through values |
| Key-Value Pairs | for key, val in dict.items(): | Loop through keys and values |

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

### 1. Using a Loop

* Iterate over the lists using an index.
* Assign elements from one list as keys and from the other as values.

**Example:**

keys = ["name", "age", "city"]

values = ["Alice", 25, "New York"]

result = {}

for i in range(len(keys)):

result[keys[i]] = values[i]

print(result)

# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

**2. Using the zip() Function**

* zip() pairs elements from both lists.
* Convert the zipped pairs directly into a dictionary.

**Example:**

keys = ["name", "age", "city"]

values = ["Alice", 25, "New York"]

result = dict(zip(keys, values))

print(result)

# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

• Counting occurrences of characters in a string using dictionaries

 Create an empty dictionary.

 Loop through each character in the string.

 For each character:

* If it’s already a key in the dictionary, increment its value by 1.
* Otherwise, add the character as a key with value 1.

8. Functions Theory:

• Defining functions in Python.

A **function** is a reusable block of code designed to perform a specific task. Functions help to:

* Organize code into logical sections
* Avoid repetition
* Improve readability and maintainability

### How to Define a Function

* Use the def keyword.
* Followed by the **function name** and parentheses ().
* Inside parentheses, you can specify **parameters** (optional).
* The function body is indented below the definition.
* Use the return statement to send back a result (optional).

Sytex:

def function\_name(parameters):

# Function body

# Code to execute

return value # Optional

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

| **Function Type** | **Parameters** | **Return Value** | **Example** |
| --- | --- | --- | --- |
| Without Parameters, Without Return | No | No | def greet(): print("Hi") |
| With Parameters, Without Return | Yes | No | def greet(name): print(name) |
| Without Parameters, With Return | No | Yes | def get\_pi(): return 3.14 |
| With Parameters, With Return | Yes | Yes | def add(a, b): return a + b |

• Anonymous functions (lambda functions).

 Lambda functions are **small, anonymous** functions defined without a name.

 They are useful for simple, one-line functions.

 Syntax is more concise than regular functions.

 Typically used where a function is needed temporarily (e.g., as an argument to another function).

Sytex:

lambda arguments: expression

 arguments: Input parameters (can be zero or more).

 expression: A single expression evaluated and returned automatically.

9. Modules Theory:

• Introduction to Python modules and importing modules.

 A **module** is a file containing Python code — functions, variables, classes — that can be reused in other Python programs.

 Modules help organize code into separate files for better structure and reusability.

 Python comes with many **built-in modules**, and you can also create your own.

### Why Use Modules?

* To **reuse code** without rewriting it.
* To **organize** large programs into manageable parts.
* To use **pre-built functionality** (like math functions, file handling, etc.).

### Importing Modules

To use a module, you need to **import** it into your program.

Sytex: import module\_name

| **Concept** | **Description** | **Example** |
| --- | --- | --- |
| Module | A Python file with reusable code | math.py (built-in module) |
| Import | Bring module into your program | import math |
| Access | Use module’s functions or variables | math.sqrt(16) |
| Import specific items | Import only parts of a module | from math import sqrt |
| Alias | Rename module on import | import math as m |

• Standard library modules: math, random.

### 1. math Module

* Provides mathematical functions and constants.
* Useful for performing advanced mathematical operations.

| **Function/Constant** | **Description** | **Example** |
| --- | --- | --- |
| math.sqrt(x) | Returns the square root of x | math.sqrt(16) → 4.0 |
| math.pow(x, y) | Returns x raised to the power y | math.pow(2, 3) → 8.0 |
| math.ceil(x) | Returns smallest integer ≥ x | math.ceil(4.2) → 5 |
| math.floor(x) | Returns largest integer ≤ x | math.floor(4.8) → 4 |
| math.pi | The constant π (approx. 3.14159) | math.pi |
| math.sin(x), math.cos(x) | Trigonometric functions (angle in radians) | math.sin(math.pi/2) → 1.0 |

### . random Module

* Used to generate **random numbers**.
* Useful for simulations, games, and testing.

| **Function** | **Description** | **Example** |
| --- | --- | --- |
| random.random() | Returns a random float in [0.0, 1.0) | random.random() |
| random.randint(a, b) | Returns a random integer between a and b (inclusive) | random.randint(1, 10) |
| random.choice(seq) | Returns a random element from a sequence | random.choice(['red', 'blue', 'green']) |
| random.shuffle(list) | Shuffles a list in place | random.shuffle(my\_list) |

• Creating custom modules.

* A **custom module** is a Python file you create containing functions, classes, or variables.
* It allows you to organize your code into reusable parts.
* You can import and use this module in other Python scripts.

**How to Create a Custom ModuleCreate a Python file** with a .py extension.

1. Define functions, classes, or variables inside this file.
2. Save the file with a meaningful name (e.g., mymodule.py).