Module 14 – Python: Collections, Functions and Modules

---

1) Accessing List

Theory

A **list** in Python is one of the most commonly used data structures. It is an **ordered collection** of elements where we can store different types of data such as integers, strings, floats, or even another list.

- **Creation of a list**:
  A list can be created using square brackets `[]` with elements separated by commas.
  Example:
- `my_list = [1, "Hello", 3.14, True]`
- **Indexing**:
  - Python uses **zero-based indexing**, meaning the first element is at index `0`, second at `1`, and so on.
  - Python also supports **negative indexing**, where `-1` represents the last element, `-2` the second-last, and so on.
    Example:
- `my_list[0]  # first element`
- `my_list[-1] # last element`
- **Slicing**:
  Slicing means extracting a **sublist** from the original list. The syntax is:
- `my_list[start:end:step]`
  - `start`: starting index (inclusive).
  - `end`: ending index (exclusive).
  - `step`: interval between elements (default is 1).

Lists are **mutable**, which means we can modify them (add, remove, or change elements) after creation. This flexibility makes them very powerful in real-world applications.

Programs

```
# Create list with multiple data types
my_list = [10, "Hello", 3.14, True]
print(my_list)

# Access different indexes
print(my_list[0])
print(my_list[-1])
print(my_list[1:3])
```

```
# Find length of list
print(len(my_list))
```

---

2) List Operations

Theory

Python lists support many useful operations:

- **Concatenation (`+`)**: Combines two lists into one.
- **Repetition (`*`)**: Repeats the elements of the list multiple times.
- **Membership (`in`, `not in`)**: Checks if an element exists in the list.

Common list methods:

1. `append(x)` → Adds an element at the end of the list.
2. `insert(i, x)` → Inserts an element at a specific index `i`.
3. `remove(x)` → Removes the first occurrence of the element `x`.
4. `pop(i)` → Removes and returns the element at index `i`. If no index is given, it removes the last element.

These operations make lists versatile for dynamic storage and manipulation of data.

Programs

```
# Append and Insert
my_list = [1, 2, 3]
my_list.append(4)
my_list.insert(1, 10)
print(my_list)

# Pop and Remove
my_list.pop()
my_list.remove(10)
print(my_list)
```

---

3) Working with Lists

Theory

- **Iteration**: Lists can be traversed using loops (like `for` or `while`). This allows us to process each element one by one.
- **Sorting**:
    - `sort()` → Sorts the list in place (modifies the original).

    - `sorted()` → Returns a new sorted list, leaving the original unchanged.
- **Reverse**: The `reverse()` method is used to reverse the order of the elements in the list.

Lists are often used when working with large collections of data that need frequent updates, sorting, and searching.

Programs

```
nums = [3, 1, 4, 2]

# Iteration
for n in nums:
    print(n)

# Sorting
nums.sort()
print(nums)
print(sorted(nums, reverse=True))

# Insert elements using loop
new_list = []
for i in range(5):
    new_list.append(i)
print(new_list)
```

---

4) Tuple

Theory

A **tuple** is similar to a list but with one important difference: **it is immutable**. This means that once a tuple is created, its values cannot be changed (no adding, removing, or updating).

- **Creation**: Tuples are created using parentheses `()`.
- **Accessing**: Tuples support both positive and negative indexing, as well as slicing.
- **Operations supported**:
    - Concatenation (`+`)
    - Repetition (`*`)
    - Membership (`in`)

Because tuples are immutable, they are faster than lists and are often used for **fixed data** like coordinates, days of the week, etc.

Programs

```
# Convert list to tuple
my_list = [1, 2, 3]
my_tuple = tuple(my_list)
print(my_tuple)
```

```
# Tuple with multiple data types
t = (10, "Hi", 3.5, False)
print(t)

# Concatenate tuples
t1 = (1, 2)
t2 = (3, 4)
print(t1 + t2)

# Access first index
print(t[0])
```

---

5) Accessing Tuples

Theory

Like lists, tuples also support **positive and negative indexing**.

- Positive indexing starts at `0`.
- Negative indexing starts at `-1`.

Tuples also support **slicing** for accessing ranges of elements. Slicing works exactly like lists:

```
tuple[start:end:step]
```

Programs

```
t = (1, 2, 3, 4, 5, 6, 7)
print(t[1:6])      # values from index 1 to 5
print(t[1:6:2])    # alternate values
```

---

6) Dictionaries

Theory

A **dictionary** in Python is a collection of key-value pairs. Each key is unique, and it maps to a specific value. Dictionaries are created using curly braces `{}`.

- **Accessing values**: Done using the key. Example: `dict[key]`.
- **Updating values**: Assigning a new value to a key.
- **Deleting elements**: Using `del dict[key]`.
- **Methods**:
    - `keys()` → returns all keys.
    - `values()` → returns all values.
    - `items()` → returns key-value pairs.

Dictionaries are very efficient for lookups because they use **hashing internally**.

Programs

```
# Dictionary with 6 key-value pairs
my_dict = {"a":1, "b":2, "c":3, "d":4, "e":5, "f":6}
print(my_dict)

# Access value using key
print(my_dict["c"])
```

---

7) Working with Dictionaries

Theory

Dictionaries can be updated, iterated, and used for many real-world applications.

- **Updating values**: Change the value associated with a key.
- **Iterating**: We can loop over keys, values, or key-value pairs.
- **Merging lists into a dictionary**: Using `zip()` or a loop.
- **Counting occurrences**: A powerful use of dictionaries is to count how often each item appears in data.

Programs

```
# Update value
d = {"name":"John", "age":20}
d["age"] = 25
print(d)

# Separate keys and values
print(list(d.keys()))
print(list(d.values()))

# Convert two lists into dictionary
keys = ["a", "b", "c"]
values = [1, 2, 3]
new_dict = {}
for i in range(len(keys)):
    new_dict[keys[i]] = values[i]
print(new_dict)

# Count characters in string
text = "banana"
count = {}
for ch in text:
    count[ch] = count.get(ch, 0) + 1
print(count)
```

---

8) Functions

Theory

A **function** is a block of reusable code that performs a specific task. Functions help make programs modular, organized, and easier to debug.

- **Defining functions**: Using the `def` keyword.
- **Parameters**: Input values that are passed to the function.
- **Return values**: Functions can optionally return results using the `return` statement.
- **Types of functions**:
    1. Without parameters, without return
    2. With parameters, without return
    3. Without parameters, with return
    4. With parameters, with return

Lambda functions: Small anonymous functions defined using the `lambda` keyword. These are usually used when a short one-line function is required.

Programs

```
def show(msg):
    print(msg)
show("Hello")

def add(a, b):
    print(a + b)
add(5, 7)

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

add = lambda a, b: a + b
print(add(3, 4))
```

---

9) Modules

Theory

A **module** in Python is simply a file containing Python definitions, functions, and classes.

- **Importing modules**:
    - `import module_name` → imports the module.
    - `from module import function` → imports only a specific function.
- **Standard library modules**: Python comes with many built-in modules like `math`, `random`, `datetime`, etc.

- **Custom modules**: We can create our own module (a `.py` file) and import it into another program.

Modules promote **code reuse** and help organize large projects.

Programs

```
import math
print(math.sqrt(16))
print(math.ceil(4.3))
print(math.floor(4.7))

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