**Module - 14**

**Python – Collections, functions and Modules**

1. **Accessing List**

* **Understanding how to create and access elements in a list**.
* In Python, a**list**is a built-in dynamic sized array (automatically grows and shrinks). We can store all types of items (including another list) in a list
* List can contain duplicate items.
* List in Python are Mutable. Hence, we can modify, replace or delete the items.
* List are ordered. It maintain the order of elements based on how they are added.
* Accessing items in List can be done directly using their position (index), starting from 0.
* **Creating a List :**

# List of integers

a = [1, 2, 3, 4, 5]

# List of strings

b = ['apple', 'banana', 'cherry']

# Mixed data types

c = [1, 'hello', 3.14, True]

print(a) #O/P : [1,2,3,4,5]

print(b) #O/P : [‘apple’, ’banana’, ‘cherry’]

print(c) #O/P : [1, ‘hello’, 3.14, True]

* **using list( ) constructor :**

We can also create a list by passing an **iterable** (like a [**string**](https://www.geeksforgeeks.org/python-string/)**,**[**tuple**](https://www.geeksforgeeks.org/tuples-in-python/), or another **list)** to [**list()** function](https://www.geeksforgeeks.org/list-constructor-in-python/).

# From a tuple

a = list((1, 2, 3, 'apple', 4.5))

print(a) #O/P :[1, 2, 3, ‘apple’, 4.5]

* **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.

a = [10, 20, 30, 40, 50]

# Access first element

print(a[0]) #O/P : 10

# Access last element

print(a[-1]) #O/P : 50

* **Indexing in lists (positive and negative indexing).**
* Negative indexes in Python are a powerful feature that allows us to access elements in a list from the end instead of the beginning.
* Negative indexes, start from -1, where -1 represents the last element in the list, -2 represents the second-to-last element, and so forth. This means that the negative index -n corresponds to the nth element from the end of the list.

L1 = [10, 20, ”apple”, 34.6, 86, “banana”]

print(L1[-1]) #O/P : banana

print(L1[-3]) #O/P : 34.6

* In Python Positive indexes, lists are zero-indexed, meaning that the first element is at index 0, the second element at index 1, and so on.

L1 = [10, 20, ”apple”, 34.6, 86, “banana”]

print(L1[1]) #O/P : 10

print(L1[4]) #O/P : 34.6

print(L1[2]) #O/P : 20

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

In Python, "slicing a list" means extracting a specific portion of a list by specifying a range of indices using the colon (:) operator, allowing you to access only a subset of elements within the list without modifying the original list.

syntax **:**

list\_name[start:stop:step]

EX :

my\_list = ["apple", "banana", "cherry", "mango", "grape"]

# Get elements from index 1 to 3 (inclusive)

sliced\_list = my\_list[1:4]

print(sliced\_list) # Output: ['banana', 'cherry', 'mango']

# Get every other element starting from index 0

sliced\_list = my\_list[::2]

print(sliced\_list) # Output: ['apple', 'cherry', 'grape']

# Get the last two elements

sliced\_list = my\_list[-2:]

print(sliced\_list) # Output: ['mango', 'grape']

1. **List Operations**

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

You can join two or more lists using the + operator.

EX :

list1 = [1, 2, 3]

list2 = [4, 5, 6]

result = list1 + list2 # Concatenation

print(result) # Output: [1, 2, 3, 4, 5, 6]

* **Repetition**

You can repeat a list multiple times using the \* operator.

Ex :

list1 = [1, 2, 3]

result = list1 \* 3 # Repetition

print(result) # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]

* **Membership**

You can check whether an element exists in a list using in and not in.

Ex :

list1 = [1, 2, 3, 4, 5]

print(3 in list1) # Output: True

print(6 in list1) # Output: False

print(6 not in list1) # Output: True

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

The append() method adds a single item to the end of the list.

Ex :

numbers = [1, 2, 3]

numbers.append(4) # Adds 4 to the end

print(numbers) # Output: [1, 2, 3, 4]

* **insert( ) :**

The insert(index, element) method inserts an element at a specific position.

Ex :

numbers = [1, 2, 4]

numbers.insert(2, 3) # Inserts 3 at index 2

print(numbers) # Output: [1, 2, 3, 4]

* **remove( ) :**

The remove(element) method removes the first occurrence of a specified value.

Ex :

numbers = [1, 2, 3, 2, 4]

numbers.remove(2) # Removes the first 2

print(numbers) # Output: [1, 3, 2, 4]

* **pop( ):**

The pop(index) method removes an element at a specific index and returns it. If no index is provided, it removes the last element.

Ex :

numbers = [1, 2, 3, 4]

removed\_element = numbers.pop(2) # Removes and returns the element at index 2

print(removed\_element) # Output: 3

print(numbers) # Output: [1, 2, 4]

# Without index (removes the last element)

last\_element = numbers.pop()

print(last\_element) # Output: 4

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

1. **Working with Lists**

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

In Python, you can iterate over a list using different types of loops, such as for loops and while loops.

* **Using a for Loop**

A for loop is the most common and efficient way to iterate over a list.

Ex **:**

numbers = [10, 20, 30, 40, 50]

for num in numbers:

print(num)

#O/P :

10

20

30

40

50

* **Using for Loop with range() and len()**

numbers = [10, 20, 30, 40, 50]

for i in range(len(numbers)):

print(f"Index {i}: {numbers[i]}")

#O/P :

Index 0 : 10

Index 1 : 20

Index 2 : 30

Index 3 : 40

Index 4 : 50

* **Using enumerate() for Index & Value**

The enumerate() function is a more Pythonic way to get both the index and value while iterating.

Ex :

numbers = [10, 20, 30, 40, 50]

for index, value in enumerate(numbers):

print(f"Index {index}: {value}")

#O/P :

Index 0 : 10

Index 1 : 20

Index 2 : 30

Index 3 : 40

Index 4 : 50

* **Using a while Loop**

A while loop can also be used when you need more control over iteration.

Ex :

numbers = [10, 20, 30, 40, 50]

i = 0

while i < len(numbers):

print(numbers[i])

i += 1

#O/P :

10

20

30

40

50

* **Sorting and reversing a list using sort(), sorted(), and reverse().**
* **Using sort() (Modifies the Original List)**

The sort() method sorts a list **in place**, meaning it modifies the original list and does not return a new list. It sorts in **ascending order** by default.

Ex :

numbers = [5, 2, 9, 1, 5, 6]

numbers.sort() # Sorts the list in ascending order

print(numbers) # Output: [1, 2, 5, 5, 6, 9]

#You can sort in **descending order** using reverse=True.

numbers = [5, 2, 9, 1, 5, 6]

numbers.sort(reverse=True) # Sorts in descending order

print(numbers) # Output: [9, 6, 5, 5, 2, 1]

**Sorting with a Custom Key (key parameter)**

You can use the key parameter to sort based on custom criteria

Ex : sorting by length

words = ["apple", "banana", "kiwi", "grape"]

words.sort(key=len) # Sorts by string length

print(words) # Output: ['kiwi', 'apple', ‘grape’, 'banana'].

* **Using sorted() (Returns a New Sorted List)**

The sorted() function **returns a new sorted list** without modifying the original list.

Ex :

numbers = [5, 2, 9, 1, 5, 6]

sorted\_numbers = sorted(numbers) # Returns a new sorted list

print(sorted\_numbers) # Output: [1, 2, 5, 5, 6, 9]

print(numbers) # Original list remains unchanged: [5, 2, 9, 1, 5, 6]

# Sorting in Descending Order with sorted()

numbers = [5, 2, 9, 1, 5, 6]

sorted\_numbers = sorted(numbers, reverse=True)

print(sorted\_numbers) # Output: [9, 6, 5, 5, 2, 1]

#### ****Sorting with a Custom Key****

words = ["apple", "banana", "kiwi", "grape"]

sorted\_words = sorted(words, key=len)

print(sorted\_words) # Output: ['kiwi', 'grape', 'apple', 'banana']

* **Using reverse()**

The reverse() method reverses the list in place. It modifies the original list.It does not sort, just reverses the order

Ex :

numbers = [1, 2, 3, 4, 5]

numbers.reverse()

print(numbers) # Output: [5, 4, 3, 2, 1]

# Using [::-1] (Returns a Reversed Copy)

You can use **list slicing** to create a reversed copy without modifying the original list.

Ex :

numbers = [1, 2, 3, 4, 5]

reversed\_numbers = numbers[::-1] # Returns a new reversed list

print(reversed\_numbers) # Output: [5, 4, 3, 2, 1]

print(numbers) # Original list remains unchanged: [1, 2, 3, 4, 5]

* **Basic list manipulations: addition, deletion, updating, and slicing.**
* **Addition to a List**

1. **using append( ) (add at the end of list)**

Ex :

my\_list = [1, 2, 3]

my\_list.append(4)

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

1. **using insert( ) (add at specific index)**

Ex :

my\_list.insert(1, 99)

print(my\_list) # Output: [1, 99, 2,3, 4]

1. **using extend( ) (add multiple element in list)**

Ex:

my\_list.extend([5, 6, 7])

print(my\_list) # Output: [1, 99, 2, 3, 4, 5, 6, 7]

* + - **Deletion from a List**

1. **using remove( value) (remove 1st occurrence of value in list)**

Ex :

my\_list = [1, 99, 2, 3, 4, 5, 6, 7]

my\_list.remove(99)

print(my\_list) # Output: [1, 2, 3, 4, 5, 6, 7]

1. **using pop( index) (remove and returnan element by index)**

Ex :

my\_list = [1, 2, 3, 4, 5, 6, 7]

popped\_value = my\_list.pop(2)

print(popped\_value) # Output: 3

print(my\_list) # Output: [1, 2, 4, 5, 6, 7]

1. **using del (delete by index or full list)**

Ex :

my\_list = [1, 2, 4, 5, 6, 7]

del my\_list[1]

print(my\_list) # Output: [1, 4, 5, 6, 7]

# Delete the entire list

Ex :

del my\_list

# print(my\_list) # This will cause an error since the list no longer exists.

* **Updating a List**
* **Updating an element by index**

Ex :

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

my\_list[1] = 99

print(my\_list) # Output: [10, 99, 30, 40]

* **Updating multiple element using slicing**

Ex :

my\_list[1:3] = [100, 200]

print(my\_list) # Output: [10, 100, 200, 40]

* **Slicing a List**
* **Extract a range [start : end] (end index is excluded)**

Ex :

number = [1, 99, 2, 3, 4, 5, 6]

print(numbers[2:6]) # Output: [2, 3, 4, 5]

* + - * **Extract every n element using step**

Ex :

number = [0, 1, 2, 3, 4, 5, 6, 7, 8]

print(numbers[::2]) # Output: [0, 2, 4, 6, 8] (every second element)

* + - * **Reverse a List using slicing**

Ex :

number = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(numbers[::-1]) # Output: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

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

* A **tuple** is a built-in data structure in Python that is used to store an ordered collection of elements.
* Tuples are similar to lists, but with one key difference: **tuples are immutable** (cannot be changed after creation).
* Tuples are defined by enclosing elements in **parentheses ( )**, separated by commas.
* **Immutability** means that once a tuple is created, its elements **cannot be modified, added, or removed**.

Ex :

# Creating a tuple

my\_tuple = (1, 2, 3, 4, 5)

print(my\_tuple) # Output: (1, 2, 3, 4, 5)

# Tuple with different data types

mixed\_tuple = (1, "Hello", 3.14, True)

print(mixed\_tuple) # Output: (1, 'Hello', 3.14, True)

# Single-element tuple (must include a comma)

single\_element\_tuple = (42,)

print(single\_element\_tuple) # Output: (42,)

* + **Creating and accessing elements in a tuple.**
* **Creating a tuple :**
* A **tuple** is created by enclosing elements in **parentheses ( )**, separated by commas.

Ex :

# Creating a tuple with multiple elements

fruits = ("apple", "banana", "cherry")

print(fruits) # Output: ('apple', 'banana', 'cherry')

# Tuple with different data types

mixed\_tuple = (10, "Python", 3.14, True)

print(mixed\_tuple) # Output: (10, 'Python', 3.14, True)

# Creating an empty tuple

empty\_tuple = ()

print(empty\_tuple) # Output: ()

# Creating a single-element tuple (MUST include a comma)

single\_tuple = ("hello",)

print(single\_tuple) # Output: ('hello',)

* **Accessing element in tuple :**
* Tuple elements are accessed using **indexing** (starting from 0).
* Negative indexes allow access from the **end** of the tuple.

Ex:

# Accessing elements by index

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

print(colors[0]) # Output: red

print(colors[1]) # Output: green

print(colors[2]) # Output: blue

# Using negative indexing

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

print(colors[-2]) # Output: green

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

* + **Basic operations with tuples: concatenation, repetition, membership.**
* **Concatenation :**
* You can **join two or more tuples** using the + operator.
* concatenation creates new tuple, it does not modify the original tuple.

Ex :

tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)

# Concatenating tuples

result = tuple1 + tuple2

print(result) # Output: (1, 2, 3, 4, 5, 6)

* **Repetition :**
* You can **repeat a tuple multiple times** using the \* operator.
* creates new tuple, it does not modify the original tuple.

Ex :

numbers = (7, 8, 9)

# Repeating the tuple 3 times

repeated\_tuple = numbers \* 3

print(repeated\_tuple) # Output: (7, 8, 9, 7, 8, 9, 7, 8, 9)

* **Membership : (in and not in)**
* The in keyword checks if an element **exists in the tuple**.

Ex :

fruits = ("apple", "banana", "cherry")

# Checking membership

print("banana" in fruits) # Output: True

print("grape" in fruits) # Output: False

print("mango" not in fruits) # Output: True

1. **Accessing tuple**

* **Accessing tuple elements using positive and negative indexing.**
* Tuple elements are accessed using **indexing** (starting from 0).
* Negative indexes allow access from the **end** of the tuple.

Ex:

# Accessing elements by index

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

print(colors[0]) # Output: red

print(colors[1]) # Output: green

print(colors[2]) # Output: blue

# Using negative indexing

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

print(colors[-2]) # Output: green

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

* **Slicing a tuple to access ranges of elements.**
* You can extract multiple elements using **slicing.**

Ex :

numbers = (10, 20, 30, 40, 50, 60)

# Extracting a sub-tuple

print(numbers[1:4]) # Output: (20, 30, 40)

# Omitting start index (default is 0)

print(numbers[:3]) # Output: (10, 20, 30)

# Omitting end index (default is till the end)

print(numbers[3:]) # Output: (40, 50, 60)

# Using step value

print(numbers[::2]) # Output: (10, 30, 50)

print(numbers[::-1]) # Output: (60, 50, 40, 30, 20, 10) (Reversing the tuple)

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

* A **dictionary** in Python is an mutable,**unordered collection** of **key-value pairs**. It is used to store data in a way that allows **fast lookup** using keys.
* Dictionaries are created using **curly braces {}**, with each key-value pair separated by a **colon :**.
* **Keys must be unique and immutable** (e.g., strings, numbers, tuples).
* **Values can be of any data type**, including lists, tuples, and even other dictionaries.

Ex : Create dictionaries

# Creating a dictionary

student = {

"name": "Alice",

"age": 20,

"course": "Computer Science"

}

print(student)

# Output: {'name': 'Alice', 'age': 20, 'course': 'Computer Science'}

* + **Accessing, adding, updating, and deleting dictionary elements.**
* **Accessing Dictionaries element :**
* You can access values using **keys(using [ ], get ( ))**.

Ex : using [ ] bracket

student = {"name": "Alice", "age": 20, "course": "Computer Science"}

# Accessing values using keys

print(student["name"]) # Output: Alice

print(student["age"]) # Output: 20

Ex :using get method

# Using get() to avoid KeyError

print(student.get("name")) # Output: Alice

print(student.get("gender")) # Output: None

print(student.get("gender", "Not Found")) # Output: Not Found

* + - **Adding element to a Dictionaries :**
      * You can add new key-value pairs by assigning a value to a new key.

Ex :

student = {"name": "Alice", "age": 20, "course": "Computer Science"}

# Adding a new key-value pair

student["gender"] = "Female"

print(student)

# Output: {'name': 'Alice', 'age': 20, 'course': 'Computer Science', 'gender': 'Female'}

* **Updating Dictionaries element :**
* You can update an existing key's value.

Ex :

student = {"name": "Alice", "age": 20, "course": "Computer Science"}

# Updating an existing value

student["age"] = 21

print(student)

# Output: {'name': 'Alice', 'age': 21, 'course': 'Computer Science', 'gender': 'Female'}

* + - * The .update() method allows updating multiple values at once.

Ex :

student = {'name': 'Alice', 'age': 21, 'course': 'Computer Science', 'gender': 'Female'}

student.update({"age": 22, "course": "Data Science"})

print(student)

# Output: {'name': 'Alice', 'age': 22, 'course': 'Data Science', 'gender': 'Female'}

* **Deleting element from a dictionaries :**

Ex : using del keyword

student= {'name': 'Alice', 'age': 21, 'course': 'Computer Science', 'gender': 'Female'}

# Deleting a specific key

del student["course"]

print(student)

# Output: {'name': 'Alice', 'age': 22, 'gender': 'Female'}

Ex : Using .pop( )

student = {'name': 'Alice', 'age': 21, 'gender': 'Female'}

# Removing a key and getting its value

removed\_value = student.pop("age")

print(removed\_value) # Output: 22

print(student)

# Output: {'name': 'Alice', 'gender': 'Female'}

* Removes the **last inserted key-value pair** .

Ex :

student["city"] = "New York" # Adding a new key

print(student.popitem())

# Output: ('city', 'New York') (Removes and returns the last inserted item)

* **clearing all element(using .clear( )) :**

Ex :

student.clear()

print(student) # Output: {}

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

**keys ( ) method :**

* The .keys() method returns a **view object** containing all the keys in the dictionary.

Ex :

student = {"name": "Alice", "age": 20, "course": "Computer Science"}

# Getting all keys

print(student.keys())

# Output: dict\_keys(['name', 'age', 'course'])

**values ( ) method :**

* The .values() method returns a **view object** of all values in the dictionary.

Ex :

student = {"name": "Alice", "age": 20, "course": "Computer Science"}

# Getting all values

print(student.values())

# Output: dict\_values(['Alice', 20, 'Computer Science'])

# Converting to a list

print(list(student.values()))

# Output: ['Alice', 20, 'Computer Science']

**items ( ) method :**

* The .items() method returns a **view object** with key-value pairs as tuples.

Ex ;

student = {"name": "Alice", "age": 20, "course": "Computer Science"}

# Getting all key-value pairs

print(student.items())

# Output: dict\_items([('name', 'Alice'), ('age', 20), ('course', 'Computer Science')])

1. **Working with dictionaries**

* **Iterating over a dictionary using loops.**
* Dictionaries store data as **key-value pairs**, and we often need to iterate over them. Python provides different ways to **loop through dictionaries** efficiently.
* **Iterating over Keys :**

**Ex :**

student = {"name": "Alice", "age": 20, "course": "Computer Science"}

# Iterating over keys

for key in student:

print(key)

O/P :

name

age

course

* **Iterating over values :**

To loop over **only values**, use the .values() method.

Ex :

student = {"name": "Alice", "age": 20, "course": "Computer Science"}

for value in student.values():

print(value)

O/P :

Alice

20

Computer Science

* **Iterating over key-value pair :**

To access both **keys and values** at the same time, use the .items() method.

Ex :

student = {"name": "Alice", "age": 20, "course": "Computer Science"}

for key, value in student.items():

print(f"{key}: {value}")

O/P :

name: Alice

age: 20

course: Computer Science

* + - **using enumerate ( ) with dictionaries :**

If you need an **index while iterating**, use enumerate().

Ex :

student = {"name": "Alice", "age": 20, "course": "Computer Science"}

for index, (key, value) in enumerate(student.items()):

print(f"{index}. {key} → {value}")

O/P :

0. name → Alice

1. age → 20

2. course → Computer Science

* **Merging two lists into a dictionary using loops or zip().**
* Sometimes, we have two separate lists:
* One list containing **keys**
* Another list containing **values**  
  We can combine them into a dictionary using **loops** or the **zip() function**.
* **Using a for loop** :

We can use a loop to iterate over both lists and create a dictionary.

Ensure both lists are of the same length to avoid IndexError.

Ex :

# Two lists

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

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

# Creating a dictionary using a loop

result\_dict = {}

for i in range(len(keys)):

result\_dict[keys[i]] = values[i]

print(result\_dict)

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

* **Using zip ( ) :**

The zip() function pairs corresponding elements from both lists and converts them into a dictionary.

Ex :

# Two lists

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

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

# Using zip() to merge lists into a dictionary

result\_dict = dict(zip(keys, values))

print(result\_dict)

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

* If the lists have different lengths, zip() will stop at the shortest one.

Ex :

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

values = ["Alice", 25] # Missing one value

result\_dict = dict(zip(keys, values))

print(result\_dict)

# Output: {'name': 'Alice', 'age': 25}

* If you want missing keys to have a default value, use zip\_longest from itertools.

Ex :

from itertools import zip\_longest

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

values = ["Alice", 25]

result\_dict = dict(zip\_longest(keys, values, fillvalue="Unknown"))

print(result\_dict)

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

* **Counting occurrences of characters in a string using dictionaries**
* **Using simple For loop :**

Ex :

def count\_chars(s):

char\_count = {} # Empty dictionary to store character counts

for char in s:

char\_count[char] = char\_count.get(char, 0) + 1 # Increment count

return char\_count

# Example usage

text = "hello world"

result = count\_chars(text)

print(result)

O/P :

{'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}

* **Using Collection.Counter (Best for long text)**

Ex :

from collections import Counter

text = "hello world"

char\_count = Counter(text)

print(char\_count)

O/P :

Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})

* **Using defaultdict (Avoid keyerror)**

Ex:

from collections import defaultdict

def count\_chars(s):

char\_count = defaultdict(int) # Default value of 0 for missing keys

for char in s:

char\_count[char] += 1

return dict(char\_count) # Convert back to a normal dictionary

text = "hello world"

print(count\_chars(text))

O/P:

1. **Function**

* **Defining functions in Python.**
* **functions** are blocks of reusable code that perform a specific task. They help in organizing code and improving reusability.
* A function is defined using the **def** keyword.

Ex **:**

def greet():

print("Hello, welcome to Python!")

# Calling the function

greet()

O/P :

Hello, welcome to Python!

* **Different types of functions: with/without parameters, with/without return values.**
* Functions in Python can be categorized based on **parameters (input)** and **return values (output).**
* **Function Without Parameter and Without Return type**
* A simple function that **does not take any arguments** and **does not return a value**.
* Used when you just want to execute a block of code without needing input or returning anything.

Ex :

def greet():

print("Hello, welcome to Python!")

# Calling the function

greet()

O/P :

Hello, welcome to Python!

* **Function With Parameter and Without Return type**
* A function that **accepts parameters** but **does not return a value**.
* Useful for printing or performing actions based on inputs but without needing to return anything.

Ex :

def greet(name):

print(f"Hello, {name}!")

greet("Alice")

greet("Bob")

O/P:

Hello, Alice!

Hello, Bob!

* **Function Without Parameter and With Return type**
  + - * + A function that **does not take parameters** but **returns a value**.
        + Used when a function needs to return a constant or computed value.

Ex :

def get\_pi():

return 3.14159

pi\_value = get\_pi()

print("Value of Pi:", pi\_value)

O/P :

Value of Pi: 3.14159

* **Function With Parameter and With Return type**
* A function that **accepts parameters** and **returns a computed result**.
* Used for mathematical operations, data processing, etc.

Ex :

def add(a, b):

return a + b

result = add(5, 3)

print("Sum:", result)

O/P:

Sum: 8

* **Anonymous functions (lambda functions).**
* A **one-line function** that doesn’t need a def statement.
* Used for short, simple functions.

Ex :

add = lambda x, y: x + y

print(add(3, 5))

O/P:

8

1. **Modules**

* **Introduction to Python modules and importing modules.**
* A **module** in Python is a file that contains Python code, including functions, classes, and variables. Modules help in organizing code, making it reusable and manageable.
* A module can be:
* A file with a .py extension containing Python code.
* A built-in module (like math, random, etc.).
* A third-party module installed using pip (e.g., numpy, pandas).
* **Importing Modules in python :**
* Python provides different ways to import and use modules:
* **Imporing an Entire module**
* Here, the math module is imported, and we use math.sqrt() to calculate the square root.

Ex :

import math

print(math.sqrt(25)) # Output: 5.0

* **Importing Specific Functions from a Module**
* Here, only sqrt and pow are imported from math, so we can use them directly without math. prefix.

Ex :

from math import sqrt, pow

print(sqrt(25)) # Output: 5.0

print(pow(2, 3)) # Output: 8.0

* **Importing a Module with an Alias**

Using an alias (as np) makes it easier to use a module with a shorter name.

Ex :

import numpy as np

arr = np.array([1, 2, 3])

print(arr)

import numpy as np

arr = np.array([1, 2, 3])

print(arr)

* **Creating and Importing a Custom Module**
* Create a file named my\_module.py with the following code:

Ex :

def greet(name):

return f"Hello, {name}!"

pi\_value = 3.14

* + - Now, import it in another Python file:

Ex :

import my\_module

print(my\_module.greet("Alice")) # Output: Hello, Alice!

print(my\_module.pi\_value) # Output: 3.14

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

Python provides a rich **Standard Library** with built-in modules that help perform various tasks. Two commonly used modules are:

* + - 1. **math module –** provides mathematical functions**.**
* The math module contains many mathematical operations like square root, trigonometry, logarithms, and constants.
* import math(importing math module)
* **Common function in math module:**

|  |  |  |
| --- | --- | --- |
| **Function** | **Description** | **Example** |
| math.sqrt(x) | Square root of x | math.sqrt(25) → 5.0 |
| math.pow(x, y) | x raised to power y | math.pow(2, 3) → 8.0 |
| math.ceil(x) | Rounds x up | math.ceil(4.2) → 5 |
| math.floor(x) | Rounds x down | math.floor(4.8) → 4 |
| math.factorial(x) | Factorial of x | math.factorial(5) → 120 |
| math.log(x, base) | Logarithm of x | math.log(8, 2) → 3.0 |
| math.sin(x), math.cos(x), math.tan(x) | Trigonometric functions | math.sin(math.radians(30)) → 0.5 |
| math.pi | Constant π (3.1416...) | math.pi |
| math.e | Constant e (2.718...) | math.e |

Ex :

import math

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

print(math.pow(2, 5)) # Output: 32.0

print(math.ceil(4.3)) # Output: 5

print(math.factorial(5)) # Output: 120

print(math.sin(math.radians(30)))

# Output: 0.5

* + - 1. **random module** – generate the random numberand select random element
* The random module provides functions to generate random numbers and make random selections.
* import ramdom(importing random module)
* **Common function in random module**:

|  |  |  |
| --- | --- | --- |
| Function | Description | Example |
| random.random() | Returns a random float between 0.0 and 1.0 | random.random() → 0.6578 |
| |  | | --- | | random.randint(a, b) |  |  | | --- | |  | | Returns a random integer between a and b | random.randint(1, 10) → 7 |
| random.uniform(a, b) | Returns a random float between a and b | random.uniform(1, 10) → 4.56 |
| |  | | --- | | random.choice(seq) |  |  | | --- | |  | | Picks a random element from a sequence | random.choice(['apple', 'banana', 'cherry']) → 'banana' |
| random.choices(seq, k=n) | Picks n random elements (with replacement) | random.choices([1, 2, 3], k=2) → [2, 3] |
| random.sample(seq, k=n) | Picks n unique random elements (without replacement) | random.sample([1, 2, 3, 4], k=2) → [3, 1] |
| random.shuffle(seq) | Shuffles a sequence in place | random.shuffle(my\_list) |

Ex :

import random

print(random.random()) # Output: 0.8274 (random float)

print(random.randint(1, 10)) # Output: 7 (random integer between 1 and 10)

print(random.uniform(1, 10)) # Output: 4.23 (random float)

print(random.choice(['red', 'blue', 'green'])) # Output: 'blue'

numbers = [1, 2, 3, 4, 5]

random.shuffle(numbers)

print(numbers)

# Output: [3, 5, 2, 1, 4] (shuffled list)

* **Creating custom modules.**
* In Python, you can create your own **custom modules** to organize and reuse code efficiently. A module is simply a .py file containing functions, variables, and classes that can be imported into other Python scripts.

**1.Creating a Custom Module**

* Create a python file.
* Define functions, variables, or classes in it.
* Import and use it in another script.

Ex :

# my\_module.py

def greet(name):

"""Function to greet a person"""

return f"Hello, {name}!"

def add(a, b):

"""Function to add two numbers"""

return a + b

pi\_value = 3.14159 # Variable

**2.Importing and using the custom module:**

Once you've created module,you can import and use it.

Ex:

import my\_module # Importing the custom module

# Using functions and variables from the module

print(my\_module.greet("Alice")) # Output: Hello, Alice!

print(my\_module.add(10, 5)) # Output: 15

print(my\_module.pi\_value) # Output: 3.14159

**3.Importing specific function or variable :**

Ex :

from my\_module import greet, add

print(greet("Bob")) # Output: Hello, Bob!

print(add(4, 6)) # Output: 10

**4.Storing module in different folder (packages):**

* If your module is inside a folder (e.g., utils/), you need an **\_\_init\_\_.py** file in the folder.
* Folder Structure :

**project/**

**│── main.py**

**│── utils/**

**│ │── \_\_init\_\_.py**

**│ │── my\_module.py**

Then, in main.py, you can import it like this:

from utils import my\_module

print(my\_module.greet("David")) # Output: Hello, David!