Assignment 2

1. Accessing List:-

Understanding how to create and access elements in a list:

1. Creating a List

A list in Python is created using square brackets [], and it can contain elements of different data types.

Ex:-

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

mixed_list = [1, "hello", 3.14, True]
```

2. Accessing by Index

Python uses zero-based indexing, meaning the first element is at index 0.

Ex:-

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

# Accessing elements by index

print(numbers[0])  # Output: 10 (first element)

print(numbers[2])  # Output: 30 (third element)

# Accessing the last element using negative indexing

print(numbers[-1])  # Output: 50

print(numbers[-2])
```

Indexing in lists (positive and negative indexing).

1. Positive Indexing

Positive indexing starts from 0 and increases sequentially.

Ex:-

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

print(fruits[0])  # apple

print(fruits[1])  # banana
```

```
print(fruits[2])  # cherry
print(fruits[3])  # date
print(fruits[4])  # elderberry
```

2. Negative Indexing

Negative indexing allows accessing elements from the end of the list.

Ex:-

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]
print(fruits[-1])  # elderberry
print(fruits[-2])  # date
print(fruits[-3])  # cherry
print(fruits[-4])  # banana
print(fruits[-5])  # apple
```

Slicing a list: accessing a range of elements.

List slicing allows you to extract a subset of elements from a list using a specific range.
The syntax for slicing is:

Ex:-

```
list[start:stop:step]
```

- start (optional) – The index where slicing starts (inclusive).
- stop – The index where slicing stops (exclusive).
- step (optional) – The increment between indices (default is 1).
- If start or stop is omitted, Python uses default values:
- If start is omitted, it defaults to 0 (beginning of the list).
- If stop is omitted, it defaults to the length of the list (end of the list).
- If step is omitted, it defaults to 1 (selects every element in order).

Ex:-

```
numbers = [10, 20, 30, 40, 50, 60, 70]
print(numbers[1:4]) # [20, 30, 40]
print(numbers[:3]) # [10, 20, 30] (default start is 0)
print(numbers[4:]) # [50, 60, 70] (default stop is end of list)
```

2. List Operations

Common list operations: concatenation, repetition, membership.

Python provides various operations to manipulate and work with lists efficiently. Three commonly used operations are:

Concatenation (+) – Merging two or more lists.

Repetition (*) – Duplicating a list multiple times.

Membership (in, not in) – Checking if an element exists in a list.

Let's explore each operation in detail.

1. List Concatenation (+)

Concatenation means joining two or more lists into a single list using the + operator.

Syntax:

```
new_list = list1 + list2
```

Example:

```
list1 = [1, 2, 3]

list2 = [4, 5, 6]

result = list1 + list2

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

list1 = [1, 2]

list2 = [3, 4]

list3 = [5, 6]

result = list1 + list2 + list3

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

2. List Repetition (*)

Repetition allows duplicating a list multiple times using the * operator.

Syntax:

```
new_list = list1 * n  # n is the number of times to repeat
```

Example:

```
numbers = [1, 2, 3]
result = numbers * 3
print(result)  # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Use Cases of Repetition:

Initializing a list with default values:

```
zeros = [0] * 5
print(zeros)  # Output: [0, 0, 0, 0, 0]
empty_list = [""] * 4
print(empty_list)  # Output: ['', '', '', '']
nested_list = [[0] * 3] * 3
nested_list[0][0] = 1
print(nested_list)
# Output: [[1, 0, 0], [1, 0, 0], [1, 0, 0]] (Because the same list is repeated)
```

To create independent copies, use list comprehension:

```
nested_list = [[0] * 3 for _ in range(3)]
nested_list[0][0] = 1
print(nested_list)  # Output: [[1, 0, 0], [0, 0, 0], [0, 0, 0]]
```

3. Membership Operators (in, not in)

The membership operators in and not in check whether a specific element exists in a list.

Syntax:

```
element in list   # Returns True if element exists
element not in list  # Returns True if element does not exist
```

Example:

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)   # True
print("grape" in fruits)    # False
print("cherry" not in fruits)  # False
```

Use Cases of Membership Operators:

Checking if an item exists before performing an action:

```
fruits = ["apple", "banana", "cherry"]
if "banana" in fruits:
    print("Banana is available!")
```

Using not in to prevent duplicates:

```
numbers = [1, 2, 3, 4, 5]
if 6 not in numbers:
    numbers.append(6)
print(numbers)  # Output: [1, 2, 3, 4, 5, 6]
```

Using membership with loops:

```
my_list = [10, 20, 30, 40]
for num in my_list:
    print(num)
```

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

1. append(item) → Add an Element at the End

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

Syntax:

```
list.append(item)
```

Example:

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'orange']
```

- **Appending a List Inside a List:**

```
numbers = [1, 2, 3]
numbers.append([4, 5])
print(numbers)  # Output: [1, 2, 3, [4, 5]] (Nested list)
```

2. insert(index, item) → Insert an Element at a Specific Position

The insert() method inserts an element at a specified index.

Syntax:

list.insert(index, item)

Example:

```
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "orange")
print(fruits)  # Output: ['apple', 'orange', 'banana', 'cherry']
```

- **Inserting at the Beginning:**

```
numbers = [10, 20, 30]
numbers.insert(0, 5)
print(numbers)  # Output: [5, 10, 20, 30]
```

- **Inserting at an Index Beyond the List Length:**

```
numbers = [1, 2, 3]
numbers.insert(10, 4)  # Large index adds at the end
print(numbers)  # Output: [1, 2, 3, 4]
```

3. remove(item) → Remove a Specific Element

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

Syntax:

list.remove(item)

Example:

```
fruits = ["apple", "banana", "cherry", "banana"]
fruits.remove("banana")
print(fruits)  # Output: ['apple', 'cherry', 'banana']
```

- **Handling Errors with remove()**

```
numbers = [10, 20, 30]
if 40 in numbers:
```

```
    numbers.remove(40)  # Avoids error if item is not found
print(numbers)  # Output: [10, 20, 30]
```

4. pop(index) → Remove and Return an Element

The pop() method removes and returns an element at a specified index.

Syntax:

```
list.pop(index)  # index is optional (default is -1, the last element)
```

Example:

```
fruits = ["apple", "banana", "cherry"]
removed_item = fruits.pop(1)
print(fruits)  # Output: ['apple', 'cherry']
print(removed_item)  # Output: 'banana'
```

- **Removing the Last Element (pop() Without an Index):**

```
numbers = [1, 2, 3, 4]
last_item = numbers.pop()
print(numbers)  # Output: [1, 2, 3]
print(last_item)  # Output: 4
```

- **Popping an Empty List (Avoiding Errors):**

```
numbers = []
if numbers:
    numbers.pop()
else:
    print("List is empty!")
```

3. Working with Lists:-

Iterating over a list using loops.:-

Iteration is the process of accessing each element in a list one by one. In Python, lists are ordered, mutable collections of elements, and we can iterate over them using loops to process their elements efficiently.

Python provides multiple methods to iterate over lists, depending on the use case:

1. Using a for loop – Best for direct element access.
2. Using range() with for loop – Used when index-based access is required.
3. Using a while loop – Suitable for condition-based iteration

1. Iterating with a for Loop

A for loop is the most common and **simplest way** to iterate over a list in Python. It directly accesses each element without needing an index.

Syntax:

```
for element in list_name:
    # Perform operations with element
```

Example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

2. Using range() with for Loop

When we need to access elements using **index values**, we can use the range() function along with len().

Syntax:

```
for i in range(len(list_name)):
    # Access elements using list_name[i]
```

Example:

```
numbers = [10, 20, 30, 40]
for i in range(len(numbers)):
    print(f"Index {i} → {numbers[i]}")
```

3. Iterating with a while Loop

A while loop is useful when **iteration depends on a condition** rather than a fixed number of elements.

Syntax:

```
i = 0
while i < len(list_name):
```

```
    # Access list_name[i]
    i += 1
```

Example:

```
numbers = [5, 10, 15, 20]
i = 0
while i < len(numbers):
    print(numbers[i])
    i += 1
```

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

Python provides built-in methods to sort and reverse lists efficiently. The primary functions for these operations are:

1.sort() → Sorts the list in place (modifies the original list).

2.sorted() → Returns a new sorted list (original list remains unchanged).

3.reverse() → Reverses the list in place (modifies the original list).

4.[::-1] (Slicing) → Returns a new reversed list without modifying the original list.

1. Sorting a List with sort() (Modifies the List)

The .sort() method **modifies the original list** in ascending order by default.

Syntax:

```
list_name.sort()
```

Example:

```
numbers = [5, 2, 9, 1, 5, 6]
numbers.sort()
print(numbers)
```

2. Sorting a List with sorted() (Creates a New List)

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

Syntax:

sorted_list = sorted(list_name)

Example:

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

sorted_numbers = sorted(numbers)

print(sorted_numbers)

print(numbers)

3. Reversing a List with reverse() (Modifies the List)

The .reverse() method **reverses the elements of a list in place**.

Syntax:

list_name.reverse()

Example:

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

numbers.reverse()

print(numbers)

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

Python lists are **mutable**, meaning you can **add, remove, update, and slice elements** easily. Below are the key list operations with examples.

1. Adding Elements to a List

Using append() (Adds at the End)

Adds a single element to the end of the list.

numbers = [1, 2, 3]

numbers.append(4)

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

Using insert() (Adds at a Specific Index)

Inserts an element at a specific position.

numbers = [1, 2, 4]

```
numbers.insert(2, 3)  # Insert 3 at index 2
print(numbers)
```

2. Deleting Elements from a List

Using remove() (Deletes by Value)

Removes the first occurrence of a specific element.

```
numbers = [1, 2, 3, 4, 3]
numbers.remove(3)
print(numbers)  # Output: [1, 2, 4, 3]
```

Using pop() (Deletes by Index and Returns It

Removes an element by index and returns it.

```
numbers = [10, 20, 30, 40]
removed = numbers.pop(2)  # Removes element at index 2
print(numbers)  # Output: [10, 20, 40]
print(removed)
```

3. Updating Elements in a List

Lists allow modifying elements by direct assignment.

```
numbers = [10, 20, 30, 40]
numbers[1] = 25  # Updating index 1
print(numbers)  # Output: [10, 25, 30, 40]
```

Updating Multiple Elements:

```
numbers[1:3] = [50, 60]
print(numbers)
```

4. Slicing a List (Extracting a Sublist)

Slicing extracts **a portion of a list** using [start:stop:step].

```
numbers = [10, 20, 30, 40, 50, 60]
print(numbers[1:4])  # Output: [20, 30, 40] (Indexes 1 to 3)
print(numbers[:3])   # Output: [10, 20, 30] (Start from 0)
print(numbers[2:])   # Output: [30, 40, 50, 60] (Till end)
```

```
print(numbers[::2])  # Output: [10, 30, 50] (Every second element)
print(numbers[::-1]) # Output: [60, 50, 40, 30, 20, 10] (Reverse list)
```

4. Tuple:-

Introduction to tuples, immutability.

A **tuple** is an **ordered, immutable** collection of elements in Python. Tuples are similar to lists, but **they cannot be modified (immutable)** after creation.

1. Creating a Tuple

A tuple is created using **parentheses ()**, but commas , define a tuple.

Ex:-

```
# Creating a tuple
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple)  # Output: (1, 2, 3, 4, 5)
```

Creating and accessing elements in a tuple.

1. Creating a Tuple

A **tuple** is created using **parentheses ()**, but it is defined by commas ,.

Examples

```
# Creating a tuple with multiple elements
numbers = (10, 20, 30, 40)
print(numbers)  # Output: (10, 20, 30, 40)
# Tuple with mixed data types
mixed_tuple = (1, "Hello", 3.14, True)
print(mixed_tuple)  # Output: (1, 'Hello', 3.14, True)
# Tuple without parentheses (Tuple Packing)
packed_tuple = 10, 20, 30
print(packed_tuple)  # Output: (10, 20, 30)
# Single element tuple (Must include a comma)
single_tuple = (5,)
print(single_tuple)  # Output: (5,)
```

2. Accessing Elements in a Tuple

Tuples support **indexing and slicing**, just like lists.

Accessing Elements by Index

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

print(fruits[0])   # Output: apple

print(fruits[1])   # Output: banana

print(fruits[2])   # Output: cherry
```

Basic operations with tuples: concatenation, repetition, membership.

1. Tuple Concatenation (+ Operator)

Concatenation means joining two or more tuples to create a new tuple. Tuples are **immutable**, so a new tuple is created instead of modifying an existing one.

Example:

```
tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)

result = tuple1 + tuple2

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

2. Tuple Repetition (* Operator)

Repetition means duplicating the elements of a tuple multiple times.

Example:

```
tuple1 = ('A', 'B', 'C')

result = tuple1 * 3

print(result)

# Output: ('A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C')
```

3. Membership Operators (in, not in)

These operators check if a specific element exists in a tuple.

Example:

```
tuple1 = (10, 20, 30, 40, 50)

print(20 in tuple1)      # Output: True
```

```
print(100 in tuple1)     # Output: False
print(50 not in tuple1)  # Output: False
print(100 not in tuple1) # Output: True
```

5. Accessing Tuples:-

Accessing tuple elements using positive and negative indexing.

Since tuples are ordered collections, you can access their elements using **indexing**. Python supports both **positive indexing** (left to right) and **negative indexing** (right to left).

1. Positive Indexing (Left to Right)

- Index starts from 0 for the first element.
- Each subsequent element increases the index by 1.

Example:

```
tuple1 = ('apple', 'banana', 'cherry', 'date')


print(tuple1[0])  # Output: apple
print(tuple1[1])  # Output: banana
print(tuple1[2])  # Output: cherry
print(tuple1[3])  # Output: date
```

2. Negative Indexing (Right to Left)

- The last element has an index of -1.
- Each previous element moves further left with decreasing negative values.

Example:

```
print(tuple1[-1])  # Output: date
print(tuple1[-2])  # Output: cherry
print(tuple1[-3])  # Output: banana
print(tuple1[-4])  # Output: apple
```

Slicing a tuple to access ranges of elements

Tuple **slicing** allows you to extract a subset of elements from a tuple using a specific range.

Tuple Slicing Syntax

```
tuple[start:stop:step]
```

- **start** → The index where slicing begins (inclusive). Default is 0.
- **stop** → The index where slicing ends (exclusive). Default is len(tuple).
- **step** → The increment between elements. Default is 1.

1. Slicing with Positive Indexing

Slicing a tuple from **left to right** using positive indexes.

Example:

```
tuple1 = ('A', 'B', 'C', 'D', 'E', 'F', 'G')
print(tuple1[1:4])  # Output: ('B', 'C', 'D')
print(tuple1[:3])   # Output: ('A', 'B', 'C')  # Default start (0)
print(tuple1[3:])   # Output: ('D', 'E', 'F', 'G')  # Default stop (end)
print(tuple1[:])    # Output: ('A', 'B', 'C', 'D', 'E', 'F', 'G')  # Full copy
```

2. Slicing with Negative Indexing

You can slice using **negative indices** to count from the right.

Example:

```
print(tuple1[-5:-2])  # Output: ('C', 'D', 'E')
print(tuple1[:-3])    # Output: ('A', 'B', 'C', 'D')  # Excludes last 3
print(tuple1[-3:])    # Output: ('E', 'F', 'G')  # Last 3 elements
```

6. Dictionaries:-

Introduction to dictionaries: key-value pairs.

A **dictionary** in Python is a data structure that stores data in **key-value pairs**. It is an unordered, mutable (modifiable), and indexed collection that allows fast lookups, insertions, and deletions.

Basic Syntax

A dictionary is defined using **curly braces {}**, where each key is mapped to a value using a colon :.

```
# Creating a dictionary
student = {
    "name": "Alice",
    "age": 20,
```

```
    "course": "Computer Science"
}
print(student)
# Output: {'name': 'Alice', 'age': 20, 'course': 'Computer Science'}
```

Key-Value Pairs

- **Keys**: Must be unique and immutable (strings, numbers, or tuples).
- **Values**: Can be of any data type (strings, numbers, lists, other dictionaries, etc.).

Example:-

```
person = {
    "name": "John",
    "age": 25,
    "hobbies": ["reading", "coding"],
    "address": {"city": "New York", "zip": "10001"}
}

print(person["name"])  # Output: John
print(person["hobbies"])  # Output: ['reading', 'coding']
print(person["address"]["city"])  # Output: New York
```

Accessing, adding, updating, and deleting dictionary elements.

1. Accessing Dictionary Elements

You can access dictionary values using **keys**.

Using Square Brackets [] (Direct Access)

```
student = {"name": "Alice", "age": 20, "course": "Computer Science"}
print(student["name"])  # Output: Alice
```

Using .get() Method (Safe Access)

```
print(student.get("age"))  # Output: 20
print(student.get("email", "Not found"))  # Output: Not found
```

The .get() method **prevents errors** and allows a default value if the key is missing.

2. Adding Elements to a Dictionary

You can add new key-value pairs dynamically.

```
student["email"] = "alice@example.com"
print(student)
# Output: {'name': 'Alice', 'age': 20, 'course': 'Computer Science', 'email': 'alice@example.com'}
```

3. Updating Dictionary Elements

Modify values by assigning a new value to an existing key.

```
student["age"] = 21
print(student)
# Output: {'name': 'Alice', 'age': 21, 'course': 'Computer Science', 'email': 'alice@example.com'}
```

Using .update() Method

The .update() method allows updating multiple key-value pairs at once.

```
student.update({"age": 22, "course": "Data Science"})
print(student)
# Output: {'name': 'Alice', 'age': 22, 'course': 'Data Science', 'email': 'alice@example.com'}
```

4. Deleting Dictionary Elements

Using del Statement

```
del student["course"]
print(student)
# Output: {'name': 'Alice', 'age': 22, 'email': 'alice@example.com'}
```

Using .pop() Method

Removes a key and returns its value.

```
email = student.pop("email")
print(email)  # Output: alice@example.com
print(student)  # Output: {'name': 'Alice', 'age': 22}
```

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

Dictionary Methods: keys(), values(), and items() in Python

Dictionaries in Python provide built-in methods to access their keys, values, and key-value pairs efficiently.

1. keys() Method

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

Example:

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

keys = student.keys()

print(keys)

# Output: dict_keys(['name', 'age', 'course'])
```

- The result is a dict_keys object, which acts like a set and reflects changes in the dictionary dynamically.

Iterating Over Keys

```
for key in student.keys():
    print(key)
```

2. values() Method

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

Example:

```
values = student.values()

print(values)

# Output: dict_values(['Alice', 20, 'Computer Science'])
```

Iterating Over Values

```
for value in student.values():
    print(value)
```

3. items() Method

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

Example:

```
items = student.items()
print(items)
# Output: dict_items([('name', 'Alice'), ('age', 20), ('course', 'Computer Science')])
```

Iterating Over Key-Value Pairs

```
for key, value in student.items():
    print(f"{key}: {value}")
```

7. Working with Dictionaries:-

Iterating over a dictionary using loops:-

Dictionaries store data in **key-value pairs**, and Python provides multiple ways to iterate over them efficiently.

1. Iterating Over Keys

By default, a for loop iterates over the **keys** of a dictionary.

Example:

```
student = {"name": "Alice", "age": 20, "course": "Computer Science"}
for key in student:
    print(key)
```

2. Iterating Over Values

To iterate over values, use the .values() method.

Example:

```
for value in student.values():
    print(value)
```

3. Iterating Over Key-Value Pairs

To iterate over both keys and values, use the .items() method.

Example:

```
for key, value in student.items():
```

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

4. Iterating and Modifying a Dictionary

Since dictionaries are mutable, you can update values while iterating.

Example (Doubling Numeric Values):

```
student_scores = {"Alice": 85, "Bob": 90, "Charlie": 78}
for key in student_scores:
    student_scores[key] += 5  # Increase score by 5
print(student_scores)
```

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

1. Using a for Loop

You can iterate over both lists simultaneously using range().

Example:

```
keys = ["name", "age", "city"]
values = ["Alice", 25, "New York"]
# Creating dictionary using loop
merged_dict = {}
for i in range(len(keys)):
    merged_dict[keys[i]] = values[i]
print(merged_dict)
```

2. Using zip()

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

Example:

```
merged_dict = dict(zip(keys, values))
print(merged_dict)
```

Counting occurrences of characters in a string using dictionaries.:-

1. Using a for Loop

We iterate over each character in the string and update its count in the dictionary.

Example:

```
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)
```

8. Functions:-

Defining functions in Python.

A **function** in Python is a reusable block of code that performs a specific task. Functions help in organizing code, improving readability, and reducing redundancy.

1. Defining a Function

A function is defined using the def keyword, followed by the function name and parentheses ().

Syntax:

```
def function_name(parameters):
    """Docstring (optional): Describes the function."""
    # Function body
    return value  # (Optional) Returns a result
```

2. Basic Function Example

```
def greet():
    print("Hello, welcome to Python!")
```

Calling the Function:

```
greet()
```

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

1. Function Without Parameters and Without Return Value

These functions perform an action but do not accept any input and do not return any result.

Example:

```
def greet():
    print("Hello, welcome to Python!")
```

Calling the Function:

```
greet()
```

2. Function With Parameters and Without Return Value

These functions accept inputs but do not return any result.

Example:

```
def greet_user(name):
    print(f"Hello, {name}!")
```

Calling the Function:

```
greet_user("Alice")
```

Different Types of Functions in Python

Functions in Python can be classified based on whether they take **parameters** and whether they **return a value**.

1. Function Without Parameters and Without Return Value

These functions perform an action but do not accept any input and do not return any result.

Example:

```
def greet():
    print("Hello, welcome to Python!")
```

Calling the Function:

```
greet()
```

Output:

Hello, welcome to Python!

2. Function With Parameters and Without Return Value

These functions accept inputs but do not return any result.

Example:

```
def greet_user(name):
    print(f"Hello, {name}!")
```

greet_user("Alice")

Output:

Hello, Alice!

3. Function Without Parameters and With Return Value

These functions do not accept parameters but return a result.

Example:

```
def get_message():
    return "Hello, welcome to Python!"
```

Calling the Function:

```
message = get_message()
print(message)
```

4. Function With Parameters and With Return Value

These functions accept inputs and return a result.

Example:

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

Calling the Function:

```
result = add_numbers(3, 5)
print(result)
```

Anonymous functions (lambda functions).

1. Syntax of a Lambda Function

lambda arguments: expression

- **lambda**: The keyword to define an anonymous function.
- **arguments**: Input parameters (can be multiple).
- **expression**: The computation, which is automatically returned.

2. Basic Example

```
square = lambda x: x * x
```

```
print(square(5))
```

9. Modules

Introduction to Python modules and importing modules.

A **module** in Python is a file containing Python code, which can include functions, classes, and variables. It allows you to organize your code into reusable components, making it easier to maintain and scale.

Types of Modules in Python

1. **Built-in Modules** – Pre-installed modules in Python (e.g., math, os, sys).
2. **User-defined Modules** – Custom modules created by users.
3. **Third-party Modules** – External libraries installed via pip (e.g., requests, numpy).

Importing Modules in Python

Python provides several ways to import modules:

1. Import the Entire Module

```
import math

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

- You need to prefix functions with the module name (math.sqrt()).

2. Import a Specific Function or Variable

```
from math import sqrt

print(sqrt(25))  # Output: 5.0
```

- No need to prefix math. before sqrt().

3. Import a Module with an Alias

```
import numpy as np

print(np.array([1, 2, 3]))  # Output: [1 2 3]
```

- Useful for renaming long module names.

4. Import All Functions and Variables

```
from math import *

print(sin(0), cos(0))
```

Standard library modules: math, random.

1. math Module (Mathematical Functions)

The math module includes a variety of mathematical operations, such as square roots, trigonometry, logarithms, and rounding functions.

Importing and Using math

To use this module, you need to import it first:

```
import math
```

Common Functions in math

- math.sqrt(x): Returns the square root of x.

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

- math.pow(x, y): Returns x raised to the power y (same as x**y).

```
print(math.pow(2, 3))  # Output: 8.0
```

- math.factorial(x): Returns the factorial of x.

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

- math.floor(x): Rounds x down to the nearest integer.

```
print(math.floor(3.7))  # Output: 3
```

- math.ceil(x): Rounds x up to the nearest integer.

```
print(math.ceil(3.2))  # Output: 4
```

- math.pi: Returns the value of π (3.14159...).

```
print(math.pi)  # Output: 3.141592653589793
```

- math.e: Returns Euler's number (2.718...).

```
print(math.e)  # Output: 2.718281828459045
```

2. random Module (Random Number Generation)

The random module allows you to generate random numbers, shuffle sequences, and select random elements.

Importing and Using random

First, import the module:

```
import random
```

Common Functions in random

- random.random(): Generates a random float between 0.0 and 1.0.

```
print(random.random())  # Example Output: 0.685432
```

- random.randint(a, b): Returns a random integer between a and b (inclusive).

```
print(random.randint(1, 10))  # Example Output: 7
```

- random.uniform(a, b): Returns a random float between a and b.

```
print(random.uniform(1, 5))  # Example Output: 3.47
```

- random.choice(sequence): Selects a random element from a list or tuple.

```
colors = ["red", "blue", "green"]
print(random.choice(colors))  # Example Output: "blue"
```

- random.shuffle(sequence): Shuffles a list in place.

```
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(numbers)  # Example Output: [3, 5, 1, 4, 2]
```

- random.sample(sequence, k): Returns k unique random elements from a sequence.

```
numbers = list(range(10))
print(random.sample(numbers, 3))  # Example Output: [2, 7, 5]
```

Creating custom modules.

A **custom module** in Python is a file containing Python code (functions, classes, or variables) that you create yourself. This allows you to organize your code into reusable components.

1. Creating a Custom Module

A module in Python is simply a .py file.

Step 1: Create a Python File (my_module.py)

Create a new Python file and define some functions inside it:

```
# my_module.py
def greet(name):
    return f"Hello, {name}!"
def add(a, b):
    return a + b
PI = 3.14159
```

2. Importing and Using the Custom Module

Step 2: Import the Module in Another Script

Now, in another Python file, import and use the module:

```
# main.py
import my_module
print(my_module.greet("Alice"))  # Output: Hello, Alice!
print(my_module.add(5, 3))       # Output: 8
print(my_module.PI)              # Output: 3.14159
```

3. Importing Specific Functions or Variables

Instead of importing the whole module, you can import specific functions:

```
from my_module import greet, add
print(greet("Bob"))   # Output: Hello, Bob!
print(add(2, 4))      # Output: 6
```

4. Using an Alias for the Module

```
import my_module as mm
print(mm.greet("Charlie"))  # Output: Hello, Charlie!
print(mm.add(10, 5))        # Output: 15
```