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

Ans:-

Creating a List
A list is a collection of items (elements) that are ordered and changeable. Lists are written using square brackets [].

```
# Example of a list
my_list = [10, 20, 30, 40, 50]
```

Accessing Elements in a List
You access list elements by indexing — using square brackets with an index number.

```
Indexing (starts at 0)
print(my_list[0])  # First element: 10
print(my_list[2])  # Third element: 30

Negative Indexing (starts from -1, the last element)
print(my_list[-1])  # Last element: 50
print(my_list[-2])  # Second-to-last: 40

Examples
colors = ['red', 'green', 'blue', 'yellow']
print(colors[1])     # Output: 'green'
print(colors[-3])    # Output: 'green'
```

Common Errors
IndexError: Happens when you try to access an index that doesn't exist.
print(my_list[10]) # Error: Index out of range

Quick Tips

- Lists can contain mixed data types: my_list = [1, "hello", 3.14, True]
- You can modify elements: my_list[1] = 99
- Use len(my_list) to get the number of elements.

2. Indexing in lists (positive and negative indexing).
Ans:-

Indexing in Lists (Positive & Negative Indexing)
Indexing allows you to access individual elements in a list by referring to their position.

Positive Indexing

- Index starts from 0 (left to right)
- First item is at index 0, second at 1, and so on.

```
fruits = ['apple', 'banana', 'cherry', 'date']
print(fruits[0])  # 'apple'
print(fruits[2])  # 'cherry'
```

Think of the index positions:

Index: 0 1 2 3
List: ['apple', 'banana', 'cherry', 'date']

Negative Indexing

- Index starts from -1 (right to left)
- Last item is at index -1, second-last at -2, and so on.

```
print(fruits[-1])  # 'date'
print(fruits[-3])  # 'banana'
```

Negative index positions:
Index: -4 -3 -2 -1
List: ['apple', 'banana', 'cherry', 'date']

Common Mistake
Trying to access an index that doesn't exist will raise an IndexError.

```
print(fruits[10])   # IndexError: list index out of range
```

Bonus: Loop with Indexing

```
for i in range(len(fruits)):
    print(f"Index {i} has {fruits[i]}")
```

3. Slicing a list: accessing a range of elements.

Ans:-

Slicing a List: Accessing a Range of Elements
Slicing allows you to access a sublist or a range of elements from a list using the syntax:

list[start:stop]

- start is the index to begin the slice (inclusive).
- stop is the index to end the slice (exclusive).
- So, it includes the element at start, but not at stop.

Basic Slicing Example

```
colors = ['red', 'green', 'blue', 'yellow', 'purple']
print(colors[1:4])  # ['green', 'blue', 'yellow']
```

Indexes used: 1, 2, and 3. Index 4 ('purple') is excluded.

Omitting Start or Stop

- If you omit start, it defaults to the beginning of the list.
- If you omit stop, it goes to the end of the list.

```
print(colors[:3])   # ['red', 'green', 'blue']
print(colors[2:])   # ['blue', 'yellow', 'purple']
```

Using Negative Indexes in Slicing

You can use negative numbers to slice from the end.

```
print(colors[-3:])     # ['blue', 'yellow', 'purple']
print(colors[:-2])     # ['red', 'green', 'blue']
```

Full Copy of a List

You can copy a list using slicing:

```
copied_list = colors[:]  # Makes a full shallow copy
```

Slicing with Step
You can also specify a step value:
print(colors[::2]) # ['red', 'blue', 'purple'] (every second element)
print(colors[::-1]) # ['purple', 'yellow', 'blue', 'green', 'red'] (reversed list)

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

Ans:-

1. Concatenation (+)
Combines two or more lists into one.

```
a = [1, 2, 3]
b = [4, 5]
result = a + b
print(result)  # [1, 2, 3, 4, 5]
```

The original lists are not changed unless you assign the result to a new variable or back to the original.

2. Repetition (*)
Repeats the elements in a list multiple times.

```
a = ['hi']
result = a * 3
print(result)  # ['hi', 'hi', 'hi']
nums = [0, 1]
print(nums * 2)  # [0, 1, 0, 1]
```

3. Membership (in, not in)
Checks whether an item exists in the list.

```
fruits = ['apple', 'banana', 'cherry']
print('banana' in fruits)     # True
print('orange' not in fruits) # True
```

This is commonly used in if statements:

```
if 'apple' in fruits:
    print("Yes, it's in the list.")
```

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

Ans:-

1. append() — Add to the end of the list
Adds a single element to the end.

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

2. insert() — Add at a specific index
Inserts an element at a specific position, shifting the rest to the right.

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

insert(index, element)

3. remove() — Remove by value
Removes the first occurrence of a specific value.

```
fruits = ['apple', 'banana', 'apple']
fruits.remove('apple')
print(fruits)  # ['banana', 'apple']
```

If the value isn't found, it raises a ValueError.

4. pop() — Remove by index (or last item by default)

- Removes and returns the item at the given index.
- If no index is given, it removes the last item.

```
fruits = ['apple', 'banana', 'cherry']
last = fruits.pop()
print(last)    # 'cherry'
print(fruits)  # ['apple', 'banana']
first = fruits.pop(0)
print(first)   # 'apple'
```

6. Iterating over a list using loops.

Ans:-

1. Using a for loop (most common)

Basic Example:

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

Output:

```
apple
```

banana
Cherry

This loop:

- Goes through each item in the list.
- Stores the current item in the variable fruit.
- Runs the loop body (prints the fruit).

2. Using for loop with range() and indexing
This is useful if you need the index number as well.

```
fruits = ['apple', 'banana', 'cherry']
for i in range(len(fruits)):
    print(f"Index {i}: {fruits[i]}")
```

Output:
Index 0: apple
Index 1: banana
Index 2: cherry

3. Using enumerate() (best of both worlds)
Gives both index and value:

```
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")
```

Output:
0: apple
1: banana

2: cherry

Bonus: while Loop (less common)
You can also use a while loop with an index:

```
fruits = ['apple', 'banana', 'cherry']
i = 0
while i < len(fruits):
    print(fruits[i])
    i += 1
```

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

Ans:-

1. sort() – Sorts the list in place
   - Changes the original list.
   - Default is ascending order.
   - For strings, it's alphabetical.

```
numbers = [3, 1, 4, 2]
numbers.sort()
print(numbers)  # [1, 2, 3, 4]
words = ['banana', 'apple', 'cherry']
words.sort()

print(words)  # ['apple', 'banana', 'cherry']
```

Optional parameter: reverse=True for descending order:
numbers.sort(reverse=True)
print(numbers) # [4, 3, 2, 1]

2. sorted() – Returns a new sorted list

- Original list stays the same.
- Can be used with any iterable (like tuples, strings, etc.)

```
nums = [5, 2, 9, 1]
sorted_nums = sorted(nums)
print(sorted_nums)  # [1, 2, 5, 9]
print(nums)         # [5, 2, 9, 1] — unchanged
```

Also supports reverse=True:

```
descending = sorted(nums, reverse=True)
print(descending)  # [9, 5, 2, 1]
```

3. reverse() – Just reverses the order, doesn't sort

```
nums = [1, 2, 3]
nums.reverse()
print(nums)  # [3, 2, 1]
```

Note: reverse() does not sort — it just flips the order of the current list.

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

Ans:-

1. Addition (Adding Elements)

append() – Adds one item to the end

```
nums = [1, 2, 3]
nums.append(4)
print(nums)  # [1, 2, 3, 4]
```

insert(index, value) – Adds at a specific position

```
nums.insert(1, 10)
print(nums)  # [1, 10, 2, 3, 4]
```

extend() – Adds multiple items

```
nums.extend([5, 6])
print(nums)  # [1, 10, 2, 3, 4, 5, 6]
```

2. Deletion (Removing Elements)

remove(value) – Removes the first occurrence of the value

```
nums.remove(10)
print(nums)  # [1, 2, 3, 4, 5, 6]
```

pop(index) – Removes and returns item at index (last if no index)

```
last_item = nums.pop()
print(last_item)  # 6
print(nums)       # [1, 2, 3, 4, 5]
```

del – Deletes item at index or a slice

```
del nums[1]
print(nums)  # [1, 3, 4, 5]
```
python
Copy code
```
del nums[1:3]
print(nums)  # [1, 5]
```

3. Updating Elements
You can update items by assigning new values using indexing:
```
nums = [1, 2, 3]
nums[0] = 10
print(nums)  # [10, 2, 3]
```

You can also update slices:
```
nums[1:3] = [20, 30]
print(nums)  # [10, 20, 30]
```

4. Slicing (Accessing a Range of Elements)
ist[start:stop] – Returns a new list from index start to stop-1
```
nums = [10, 20, 30, 40, 50]
print(nums[1:4])   # [20, 30, 40]
print(nums[:3])    # [10, 20, 30]
print(nums[2:])    # [30, 40, 50]
```
Slicing with step
```
print(nums[::2])   # [10, 30, 50]
```

```
print(nums[::-1])  # [50, 40, 30, 20, 10]  # reversed list
```

9. Introduction to tuples, immutability.

Ans:-

What Is a Tuple?

A tuple is an ordered, immutable collection of elements in Python.

- Defined with parentheses: ()
- Elements can be of any type (numbers, strings, other tuples, etc.)
- Similar to lists, but immutable

```
my_tuple = (10, 20, 30)
print(my_tuple)  # Output: (10, 20, 30)
```

What Does Immutability Mean?

Once a tuple is created, it cannot be changed.

You cannot:

- Modify an element
- Add or remove elements
- Sort or reverse it in place

```
my_tuple[0] = 99     # Error: 'tuple' object does not support item assignment
```

This makes tuples safe for storing constant data that shouldn’t be changed by mistake.

When to Use Tuples

- When the data is fixed and should not be changed.
- As keys in dictionaries (lists can't be used as keys, but tuples can).
- To return multiple values from a function.
- For faster performance compared to lists (slightly more efficient).

Creating Tuples
Multiple elements:

```
colors = ('red', 'green', 'blue')
```

Single-element tuple:
You must use a trailing comma or it won’t be a tuple:

```
single = (42,)      # This is a tuple
not_tuple = (42)    # Just an integer
```

Without parentheses (tuple packing):

```
data = 1, 2, 3
```

Accessing Tuple Elements
Tuples are indexed, just like lists:

```
colors = ('red', 'green', 'blue')
print(colors[0])    # Output: red
print(colors[-1])   # Output: blue
```

Tuple Methods
Tuples only support a few built-in methods:

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

```
print(my_tuple.count(2))  # Output: 2
print(my_tuple.index(3))  # Output: 3
```

10. Creating and accessing elements in a tuple.

Ans:-

1. Creating a Tuple

You create a tuple using parentheses () (or by just separating values with commas).

Basic Tuple

```
my_tuple = (10, 20, 30)
```

Tuple without parentheses (tuple packing)

```
my_tuple = 10, 20, 30  # Still a tuple
```

Single-element Tuple (⚠ Trailing comma required!)

```
one_item = (5,)
not_a_tuple = (5)  # This is just an int
```

2. Accessing Elements in a Tuple

Tuples are indexed, just like lists.

Positive Indexing (starts at 0)

```
colors = ('red', 'green', 'blue')
print(colors[0])  # 'red'
print(colors[2])  # 'blue'
```

Negative Indexing (starts at -1 from the end)
print(colors[-1]) # 'blue'
print(colors[-2]) # 'green'

Example
person = ('Alice', 30, 'Engineer')

# Accessing elements
name = person[0]
age = person[1]

print(f"{name} is {age} years old.") # Alice is 30 years old.
Tuples Are Immutable
You can access items, but cannot change them:

person[1] = 31 # Error: 'tuple' object does not support item assignment

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

Ans:-

1. Concatenation (+)
You can combine two tuples using the + operator.
t1 = (1, 2, 3)
t2 = (4, 5)

```
result = t1 + t2
print(result)  # (1, 2, 3, 4, 5)
```
This creates a new tuple — original tuples are not modified.

2. Repetition (*)
You can repeat a tuple multiple times using the * operator.
```
t = (1, 2)
result = t * 3
print(result)  # (1, 2, 1, 2, 1, 2)
```
Like concatenation, this creates a new tuple.

3. Membership (in, not in)
You can check if a value exists in a tuple using in or not in.
```
colors = ('red', 'green', 'blue')
print('green' in colors)     # True
print('yellow' not in colors)  # True
```
This is commonly used in if statements:

```
if 'blue' in colors:
    print("Blue is in the tuple.")
```

12. Accessing tuple elements using positive and negative indexing.

Ans:-

1. Positive Indexing
   - Starts at 0 (from left to right).
   - First element is at index 0, second at 1, etc.

```
my_tuple = ('a', 'b', 'c', 'd')
print(my_tuple[0])  # 'a'
print(my_tuple[2])  # 'c'
```

Index positions:

```
  Index:     0     1     2     3
  Tuple:   ('a', 'b', 'c', 'd')
```

2. Negative Indexing
   - Starts at -1 (from right to left).
   - -1 is the last element, -2 is second-last, and so on.

```
print(my_tuple[-1])  # 'd'
print(my_tuple[-3])  # 'b'
```

Negative index positions:

```
  Index:    -4    -3    -2    -1
  Tuple:   ('a', 'b', 'c', 'd')
```

IndexError
Trying to access an index that doesn't exist will raise an error:

```
print(my_tuple[10])  #IndexError: tuple index out of range
```

13. Slicing a tuple to access ranges of elements.

Ans:-

Tuple Slicing Syntax

tuple[start:stop:step]

- start – index to begin the slice (inclusive)
- stop – index to end the slice (exclusive)
- step – (optional) how many steps to move at a time

Examples

```
t = ('a', 'b', 'c', 'd', 'e', 'f')

Basic slicing
print(t[1:4])    # ('b', 'c', 'd')
print(t[:3])     # ('a', 'b', 'c')      # from start to index 2
print(t[3:])     # ('d', 'e', 'f')      # from index 3 to end

With step value
print(t[::2])    # ('a', 'c', 'e')      # every second element
print(t[1:5:2])  # ('b', 'd')           # from index 1 to 4, step 2

Using negative indexes
print(t[-4:-1])  # ('c', 'd', 'e')      # same as t[2:5]

Reversing a tuple with slicing
print(t[::-1])   # ('f', 'e', 'd', 'c', 'b', 'a')
```

Slicing Creates a New Tuple

```
slice = t[1:4]
print(slice)     # ('b', 'c', 'd')
```
Tuples are immutable, but slicing creates a new tuple — the original is unchanged.

Invalid slices don’t raise errors
They just return an empty tuple:
```
print(t[10:20])  # ()
```

14. Introduction to dictionaries: key-value pairs.

Ans:-

What Is a Dictionary?

- A dictionary is written using curly braces {}
- Each item is a 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 and can repeat

Example Dictionary
```
person = {
    "name": "Alice",
    "age": 30,
    "job": "Engineer"
}
```
This dictionary contains:

- Key "name" with value "Alice"
- Key "age" with value 30
- Key "job" with value "Engineer"

Accessing Values by Key
Use square brackets [] with the key:
print(person["name"]) # Output: Alice
print(person["job"])   # Output: Engineer
Using a key that doesn't exist will raise a KeyError.

Creating Dictionaries
Empty dictionary:
empty_dict = {}

With dict() function:
person = dict(name="Bob", age=25)

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

Ans:-

1. Accessing Elements
Use the key inside square brackets [] to get the value.
person = {"name": "Alice", "age": 30}
print(person["name"]) # Alice
print(person["age"])   # 30

Safer Access with .get()

```
print(person.get("name"))      # Alice
print(person.get("email"))     # None (no error)
```

You can also provide a default value:

```
print(person.get("email", "Not provided"))  # Not provided
```

2. Adding Elements

Just assign a new key:

```
person["job"] = "Engineer"
print(person)
# {'name': 'Alice', 'age': 30, 'job': 'Engineer'}
```

3. Updating Elements

Reassign a new value to an existing key:

```
person["age"] = 31
print(person)
# {'name': 'Alice', 'age': 31, 'job': 'Engineer'}
```

You can also use the .update() method:

```
person.update({"age": 32, "city": "New York"})
print(person)
# {'name': 'Alice', 'age': 32, 'job': 'Engineer', 'city': 'New
York'}
```

4. Deleting Elements

Using del:

```
del person["job"]
print(person)
```

{'name': 'Alice', 'age': 32, 'city': 'New York'}
Raises a KeyError if the key doesn’t exist.

Using .pop() (removes and returns the value):
age = person.pop("age")
print(age)   # 32
print(person)   # {'name': 'Alice', 'city': 'New York'}

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

Ans:-

Suppose You Have This Dictionary:

```
person = {
    "name": "Alice",
    "age": 30,
    "job": "Engineer"
}
```

1. keys() – Get All Keys
Returns a view object containing all the keys.
print(person.keys()) # dict_keys(['name', 'age', 'job'])
You can convert it to a list if needed:
print(list(person.keys())) # ['name', 'age', 'job']

2. values() – Get All Values
Returns a view object of all the values in the dictionary.

```
print(person.values())  # dict_values(['Alice', 30,
'Engineer'])
```
Convert to list:
```
print(list(person.values()))  # ['Alice', 30, 'Engineer']
```

3. items() – Get All Key-Value Pairs
Returns a view of tuples (key, value) pairs.
```
print(person.items())
# dict_items([('name', 'Alice'), ('age', 30), ('job',
'Engineer')])
```
Useful when looping through both keys and values:
```
for key, value in person.items():
    print(f"{key}: {value}")
```
Output:
```
name: Alice
age: 30
job: Engineer
```

17. Iterating over a dictionary using loops.

Ans:-

Iterating Over a Dictionary Using Loops in Python
You can loop through a dictionary in several ways depending on what you need: keys, values, or key-value pairs.
Let’s use this example:
```
person = {
```

```
    "name": "Alice",
    "age": 30,
    "job": "Engineer"
}
```

1. Loop Through Keys (Default)

```
for key in person:
    print(key)
```

Output:

```
name
age
Job
```

This is the same as:

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

2. Loop Through Values

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

Output:

```
Alice
30
Engineer
```

3. Loop Through Key-Value Pairs

Use .items() to get both key and value in one go:

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

```
    print(f"{key}: {value}")
```
Output:
```
name: Alice
age: 30
job: Engineer
```

Bonus: Use enumerate() with keys or values
```
for i, key in enumerate(person):
    print(i, key, person[key])
```

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

Ans:-

Merging Two Lists Into a Dictionary in Python

You can combine two lists — one for keys, one for values — into a dictionary using either:

1. A loop
2. The built-in zip() function

Example Lists:
```
keys = ['name', 'age', 'job']
values = ['Alice', 30, 'Engineer']
```

1. Using zip() and dict()

The easiest and cleanest method:

```
person = dict(zip(keys, values))
print(person)
```
Output:
```
{'name': 'Alice', 'age': 30, 'job': 'Engineer'}
```
zip() pairs elements together: ('name', 'Alice'), etc.

2. Using a for Loop
You can build the dictionary manually:
```
person = {}
for i in range(len(keys)):
    person[keys[i]] = values[i]
print(person)
```
Output:
```
{'name': 'Alice', 'age': 30, 'job': 'Engineer'}
```
This assumes both lists are the same length.

What If the Lists Are Different Lengths?
zip() automatically stops at the shorter list:
```
keys = ['name', 'age']
values = ['Alice', 30, 'Engineer']
print(dict(zip(keys, values)))  # {'name': 'Alice', 'age': 30}
```
You can handle mismatches manually if needed (e.g., pad with None).

19. Counting occurrences of characters in a string using dictionaries.

Ans:-

Counting Character Occurrences in a String Using a Dictionary

You can use a dictionary in Python to count how many times each character appears in a string.

Example Input

```
text = "hello world"
```

Method 1: Using a for loop

```
char_count = {}
for char in text:
    if char in char_count:
        char_count[char] += 1
    else:
        char_count[char] = 1
print(char_count)
```

Output:

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

Method 2: Using dict.get() for cleaner code

```
char_count = {}
for char in text:
    char_count[char] = char_count.get(char, 0) + 1
print(char_count)
```

get(char, 0) returns the current count or 0 if the character is not yet in the dictionary.

Bonus: Count Only Letters (Ignore Spaces and Punctuation)

```
text = "hello world"
char_count = {}
for char in text:
    if char.isalpha():  # Only count letters
        char_count[char] = char_count.get(char, 0) + 1
print(char_count)
```

20. Defining functions in Python.

Ans:-

Defining Functions in Python

A function is a reusable block of code that performs a specific task. You define a function once and can use it many times.

1. Basic Function Syntax

```
def function_name():
    # code block
    print("Hello!")
```

Example:

```
def greet():
    print("Hello, world!")
```

Call the function:

```
greet()  # Output: Hello, world!
```

2. Function with Parameters
Parameters allow you to pass information into the function.

```
def greet(name):
    print("Hello,", name)
```

Call it with an argument:
python
Copy code

```
greet("Alice")  # Output: Hello, Alice
```

3. Function with Return Value
Use return to send a result back to the caller.

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

Use the result:

```
sum_result = add(3, 5)
print(sum_result)  # Output: 8
```

4. Default Parameter Values
You can provide default values for parameters:

```
def greet(name="friend"):
    print("Hello,", name)
greet("Bob")      # Hello, Bob
greet()           # Hello, friend
```

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

Ans:-

Different Types of Functions in Python

Functions in Python can vary based on whether they:

- Take parameters
- Return values

Let's break it down into the 4 main types:

1. No Parameters, No Return Value

A simple function that just does something when called.

```
def greet():
    print("Hello!")
```

Usage:

```
greet()  # Output: Hello!
```

2. With Parameters, No Return Value

Takes input (parameters), but does not return a value — it performs an action.

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

Usage:

```
greet("Alice")  # Output: Hello, Alice!
```

3. No Parameters, With Return Value
Doesn’t take any input, but returns a result.

```
def get_default_greeting():
    return "Hello!"
```

Usage:

```
msg = get_default_greeting()
print(msg)  # Output: Hello!
```

4. With Parameters and Return Value
The most flexible type: takes input and returns output.

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

Usage:

```
result = add(5, 3)
print(result)  # Output: 8
```

22. Anonymous functions (lambda functions).
Ans:-
Anonymous Functions (Lambda Functions) in Python
A lambda function is a small, anonymous (unnamed) function defined using the lambda keyword.

Basic Syntax

```
lambda arguments: expression
```

It returns the result of the expression when called.

Example: Simple Addition

```
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8
```

Same as:

```
def add(x, y):
    return x + y
```

Use Cases

Lambda functions are often used when you need a short function temporarily — especially with:

- map()
- filter()
- sorted()
- GUI or event-driven code

Examples

1. Square of a number

```
square = lambda x: x * x
print(square(4))  # Output: 16
```

2. Filter even numbers from a list

```
nums = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)  # Output: [2, 4, 6]
```

3. Sort list of tuples by second element
pairs = [(1, 3), (2, 1), (4, 2)]
sorted_pairs = sorted(pairs, key=lambda x: x[1])
print(sorted_pairs) # Output: [(2, 1), (4, 2), (1, 3)]
Limitations of Lambda

- Only one expression allowed — no statements, loops, or multiple lines
- For simple operations only
- Not ideal for complex logic (use def instead)

23. Introduction to Python modules and importing modules.

Ans:-

Introduction to Python Modules & Importing Modules
A module in Python is simply a file that contains Python code — it can include functions, variables, and classes. Modules help you organize your code and reuse functionality across projects.

1. What Is a Module?
A module is:

- A .py file
- Can be built-in (standard library)
- Can be custom (your own Python file)
- Can be external (installed with pip)

2. Using Built-in Modules

Python has many built-in modules, like:

- math – for mathematical operations
- random – for generating random numbers
- datetime – for working with dates and times

Importing a Module:

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

3. Import Specific Items

Use from ... import ... to import only what you need:

```
from math import sqrt, pi

print(sqrt(25))  # Output: 5.0
print(pi)        # Output: 3.141592653589793
```

4. Import with Alias

Use as to give the module or function a shorter name:

```
import datetime as dt
print(dt.datetime.now())
```

5. Creating Your Own Module

Create a Python file (e.g., mymodule.py) with functions:

```
# mymodule.py
def greet(name):
```

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

Then import it into another file:

```
import mymodule
print(mymodule.greet("Alice"))  # Output: Hello, Alice!
```

Python must be able to find your module in the same folder or path.

24. Standard library modules: math, random.

Ans:-

Standard Library Modules in Python: math and random

Python's standard library includes powerful built-in modules like math and random — no installation needed!

1. math Module — Mathematical Functions

Import it first:

```
import math
```

2. random Module — Randomness & Simulations

Import it:

```
import random
```

Example: Dice Roll Simulation

```
import random
def roll_dice():
```

```
    return random.randint(1, 6)
print(roll_dice())  # Output: 1–6
```

25. Creating custom modules.

Ans:-

Creating Custom Modules in Python

A custom module is simply a Python file (.py) that contains functions, variables, or classes you define. Once created, you can import and reuse it in other Python scripts.

1. Step-by-Step: How to Create a Module

Step 1: Create a Python File

Create a file named mymodule.py with some code:

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

This is your module.

Step 2: Use the Module in Another Script

Create another Python file in the same folder, e.g., main.py:

```
# main.py
```

```
import mymodule
print(mymodule.greet("Alice"))   # Output: Hello, Alice!
print(mymodule.add(3, 4))        # Output: 7
```
Python treats any .py file as a module that can be imported.

3. Import Only Specific Functions
```
from mymodule import greet
print(greet("Bob"))  # Output: Hello, Bob
```

4. Using Aliases
```
import mymodule as mm
print(mm.add(2, 5))  # Output: 7
```

Bonus: Check if the Module Is Being Run Directly
Use the special __name__ variable:
```
# mymodule.py
def greet(name):
    return f"Hello, {name}!"
if __name__ == "__main__":
    # Only runs when you execute mymodule.py directly
    print(greet("Tester"))
```