FUNCTIONS

1. What is the difference between a function and a method in Python?

📌 **Answer:**

✅ **Difference between a Function and a Method in Python**

Feature	Function	Method
Definition	A block of code that performs a task and can be called independently.	A function that is associated with an object (like a list, string, dict, etc.).
Called on	Called directly by its name.	Called on an object (using dot notation).
Belongs to	Not bound to any object.	Bound to a data type (like list, dict, string).
Syntax	function_name()	object.method_name()

🔷 **Example of a Function:**

```
def greet(name):
    return "Hello " + name

print(greet("Yash"))
```

📌 Output:

```
Hello Yash
```

🔶 **Example of a Method:**

```
name = "yash"
print(name.upper())  # 'upper()' is a string method
```

📌 Output:

```
YASH
```

💡 **In Short:**

- **Function** = Independent block.
- **Method** = Function tied to an object.

FUNCTIONS

2. Explain the concept of function arguments and parameters in Python.

📌 **Answer:**

In Python, **parameters** and **arguments** are related to how we pass data into functions — but they are **not the same thing**.

🔷 **Parameter:**

A **parameter** is a variable listed inside the parentheses when a function is defined.
It acts like a **placeholder**.
👉 **Think of it as a label** that the function uses internally.

```
def greet(name):  # 'name' is a parameter
    print("Hello", name)
```

🔶 **Argument:**

An **argument** is the **actual value** that is passed to the function when it is called.

```
greet("Yash")  # "Yash" is an argument
```

✅ **Types of Arguments in Python:**

1. **Positional Arguments** – Based on order
2. **Keyword Arguments** – Using key=value
3. **Default Arguments** – Have default values
4. **Variable-length Arguments** – *args, **kwargs

🧠 **Example using multiple types:**

```
def info(name, age=18):
    print("Name:", name)
    print("Age:", age)

info("Yash")          # Positional + default
info(name="Yash", age=23) # Keyword
```

📌 Output:

```
Name: Yash
Age: 18

Name: Yash
Age: 23
```

📜 **Summary:**

- **Parameters** → Defined in the function header.
- **Arguments** → Passed during function call.

FUNCTIONS

3. What are the different ways to define and call a function in Python?

📌 **Answer:**

In Python, functions can be defined and called in several ways depending on how you want to **organize, reuse, or pass** information.

✅ **1. Defining a Function (using def keyword):**

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

This defines a simple function named greet with **no parameters**.

✅ **2. Calling a Function:**

```
greet()  # This will print "Hello, World!"
```

📌 **Different Ways to Define Functions:**

🔷 **A. Function with Parameters:**

```
def greet(name):
    print("Hello,", name)
greet("Yash")  # Output: Hello, Yash
```

🔷 **B. Function with Default Parameters:**

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

greet()       # Output: Hello, User
greet("Yash")   # Output: Hello, Yash
```

🔷 **C. Function with Return Value:**

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

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

🔷 **D. Lambda (Anonymous) Function:**

A quick, single-expression function using lambda.

```
square = lambda x: x ** 2
print(square(4))  # Output: 16
```

🔷 **E. Recursive Function (Calls Itself):**

```
def factorial(n):
    if n == 0:
```

```
    return 1
  return n * factorial(n - 1)

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

🧠 **Summary:**

FUNCTION TYPE	USE CASE
REGULAR FUNCTION	Most common
WITH PARAMETERS	When input is needed
WITH RETURN	When output is needed
DEFAULT PARAMETERS	For optional values
LAMBDA FUNCTION	One-liner expressions
RECURSIVE FUNCTION	When solving by subproblems

FUNCTIONS

4. What is the purpose of the `return` statement in a Python function?

📌 **Answer:**

The return statement in a Python function is used to **send a value back** to the caller (the place where the function was called). It **ends the function execution** and provides the result.

✅ **Purpose of return:**

1. **Output data** from a function.
2. **Pass results** for further use.
3. **Exit the function** early if needed.

🔷 **Syntax:**

```
def function_name():
    return value
```

🔷 **Example 1: Returning a value**

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

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

Here, return a + b sends the sum to the variable result.

🔷 **Example 2: Returning multiple values**

```
def get_name_age():
    return "Yash", 23

name, age = get_name_age()
print(name)  # Output: Yash
print(age)   # Output: 23
```

🔷 **Example 3: Early exit**

```
def check_even(n):
    if n % 2 != 0:
        return "Not Even"
    return "Even"
```

🧠 **Summary:**

Use	Behavior
Return value	Sends data back to caller
No return	Returns None by default
Return multiple values	Tuple is returned
Return ends function	Function stops executing immediately

FUNCTIONS

5. What are iterators in Python and how do they differ from iterables?

📌 **Answer:**

In Python, **iterables** and **iterators** are two key components used in looping (like in for loops), but they are **not the same**.

🔷 **What is an Iterable?**

An **iterable** is any Python object capable of **returning its elements one at a time**. It can be looped over using a for loop.

🧩 **Examples of iterables**:

- list
- tuple
- string
- set
- dictionary

```
my_list = [1, 2, 3]
for num in my_list:
   print(num)
```

🔷 **What is an Iterator?**

An **iterator** is an object that **remembers its state** and provides the **next item** from an iterable using the __next__() method.

You can create an iterator using the built-in iter() function.

🔄 **Difference Between Iterable and Iterator:**

Feature	Iterable	Iterator
Definition	Object that can be looped over	Object that produces values from iterable
Method support	Has __iter__()	Has __iter__() and __next__()
Usage	Used in for loops	Used with next() function
Examples	list, string, tuple, set	Object returned by iter()

🔷 **Example:**

```
# Iterable
numbers = [1, 2, 3]  # This is an iterable

# Convert to Iterator
it = iter(numbers)   # Now it's an iterator

print(next(it))  # Output: 1
print(next(it))  # Output: 2
print(next(it))  # Output: 3
```

If you call next(it) again, it will raise StopIteration.

FUNCTIONS

🔍 **Summary:**

- **Iterable**: Can be looped over (for, in).
- **Iterator**: Fetches elements **one by one** using next().

FUNCTIONS

6. Explain the concept of generators in Python and how they are defined.

📌 **Answer:**

In Python, **generators** are a special type of **iterator** that allow you to **generate values on the fly** using the yield keyword instead of storing them all in memory.

🔷 **What is a Generator?**

A **generator** is a function that **yields values one at a time**, **pausing its state** between each call so it can resume from where it left off.

🔷 **Why Use Generators?**

- **Memory-efficient**: Does not store the entire sequence in memory.
- **Lazy Evaluation**: Values are generated only when needed.

🔷 **How to Define a Generator:**

You define a generator just like a normal function, but use yield instead of return.

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1
```

🔷 **How to Use a Generator:**

```
gen = count_up_to(3)

print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

After all values are exhausted, next() raises StopIteration.

🔷 **Generator vs Normal Function:**

Feature	Generator Function	Normal Function
Uses	yield	return
Returns	Generator object	Final result
Memory usage	Low (lazy evaluation)	High (entire data stored)
Resumes from point	Yes	No

🔷 **Example Use Case:**

Reading large files line by line:

```
def read_lines(file_path):
    with open(file_path) as file:
        for line in file:
```

```
    yield line.strip()
```

🔍 **Summary:**

- Generators **produce items one at a time**.
- They are **more memory-efficient** than storing large lists.
- Use the yield keyword to **pause and resume** execution.

---

7. What are the advantages of using generators over regular functions?

📌 **Answer:**

Generators offer several powerful advantages over regular functions, especially when dealing with **large datasets**, **infinite sequences**, or when **memory efficiency** is critical.

🔷 **Advantages of Generators Over Regular Functions:**

1. Memory Efficiency

- **Generators do not store** the entire sequence in memory.
- Instead, they **yield one value at a time**, which is ideal for large datasets.

🧪 **Example:**

```
def big_range():  # Generator
    for i in range(10**6):
        yield i
```

🔁 Using a generator saves memory compared to creating a list with range(10**6).

---

2. Lazy Evaluation

- Values are computed **only when needed**, reducing unnecessary calculations.

🧪 **Example:**

```
gen = (x * x for x in range(5))  # generator expression
print(next(gen))  # Output: 0
```

---

3. Infinite Sequences Support

- Generators can handle **infinite streams of data**, unlike lists or regular functions.

🧪 **Example:**

```
def infinite_counter():
    i = 0
    while True:
        yield i
        i += 1
```

---

4. Improved Performance

- Since they avoid the overhead of storing full sequences, generators often **run faster** for iteration tasks.

---

5. Clean and Readable Code

- Generators help simplify code for large pipelines and data processing (e.g., map(), filter(), zip() with generators).

🔍 **Summary Table:**

Feature	Generator	Regular Function
Memory Usage	Low	High (stores entire data)
Value Evaluation	Lazy (on-demand)	Eager (all at once)

FUNCTIONS

Infinite Sequence	Supported	Not practical
State Persistence	Maintains state between calls	Starts fresh every time

FUNCTIONS

8. **What is a lambda function in Python and when is it typically used? 8. What is a lambda function in Python and when is it typically used?**

📌 **Answer:**

A **lambda function** in Python is a **small anonymous function** defined using the lambda keyword. It can take any number of arguments but can **only have one expression**.

It's typically used when a **simple function** is required for a **short period of time**, usually as an **argument to higher-order functions** like map(), filter(), or sorted().

🔷 **Syntax of a Lambda Function:**

```
lambda arguments: expression
```

🧪 **Example 1: Simple lambda function to square a number**

```
square = lambda x: x * x
print(square(5))  # Output: 25
```

🔷 **Typical Use Cases:**

✅ **1. With map()**

Applies a function to all elements in a list.

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x * x, nums))
print(squares)  # Output: [1, 4, 9, 16]
```

✅ **2. With filter()**

Filters elements based on a condition.

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

✅ **3. With sorted()**

Sorts based on a custom key.

```
data = [(1, 'apple'), (3, 'banana'), (2, 'cherry')]
sorted_data = sorted(data, key=lambda x: x[0])
print(sorted_data)  # Output: [(1, 'apple'), (2, 'cherry'), (3, 'banana')]
```

🔍 **Key Points:**

Feature	Lambda Function
Name	Anonymous (unnamed)
Use Case	Short, simple, throwaway functions
Return Keyword Needed?	❌ No (lambda returns the value itself)
Scope	Can be used wherever a function is valid

🧠 **Tip:** Use lambdas for **concise logic**. Avoid for complex tasks — use def instead.

FUNCTIONS

9. Explain the purpose and usage of the `map()` function in Python.

📌 **Answer:**

The map() function in Python is used to **apply a function to every item in an iterable (like a list, tuple, etc.)** and returns a new iterable (specifically, a map object) containing the results.

🔷 **Syntax:**

map(function, iterable)

- function: A function that you want to apply.
- iterable: A sequence (like a list, tuple, etc.) whose items will be passed to the function.

🔷 **Example 1: Square all numbers in a list**

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # Output: [1, 4, 9, 16, 25]
```

🔷 **Example 2: Convert list of strings to uppercase**

```
words = ['hello', 'world']
upper_words = list(map(str.upper, words))
print(upper_words)  # Output: ['HELLO', 'WORLD']
```

🔷 **Using map() with a user-defined function:**

```
def double(n):
    return n * 2

data = [10, 20, 30]
result = list(map(double, data))
print(result)  # Output: [20, 40, 60]
```

🔍 **Key Benefits:**

Feature	Advantage
Fast & Efficient	No need for manual for loops
Cleaner Code	More readable and expressive
Works with lambda	Ideal for quick one-line operations

⚠️ **Note:**

- The result of map() must be converted to a list or tuple if you want to see or manipulate the results immediately.
- map() is lazy; it computes values only when needed.

💡 **Use Case Tip:** Use map() when you need to **transform each item** in a sequence using a function — especially for operations like type conversion, math, formatting, etc.

FUNCTIONS

10. What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

📌 **Answer:**

map(), reduce(), and filter() are **higher-order functions** in Python that allow functional-style programming by processing iterables (like lists or tuples). Each serves a different purpose:

🔷 **1. map() → Transforms elements**

- **Purpose:** Applies a function to **every item** in an iterable and returns a new iterable with the transformed items.

✅ **Example:**

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # Output: [1, 4, 9, 16]
```

🔷 **2. filter() → Filters elements**

- **Purpose:** Applies a function that returns True or False to each item, and **only keeps the items** that return True.

✅ **Example:**

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

🔷 **3. reduce() → Reduces to a single value**

- **Purpose:** Applies a function cumulatively to the items of an iterable, **reducing it to a single result**.
- 🔶 Comes from the functools module.

✅ **Example:**

```
from functools import reduce
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 24
```

🔁 **Summary Table:**

FUNCTION	PURPOSE	OUTPUT TYPE	EXAMPLE USE CASE
MAP()	Transform each item	New iterable	Squaring each number
FILTER()	Select items that meet condition	Filtered iterable	Keep only even numbers
REDUCE()	Combine all items into one value	Single result	Sum, product, etc.

📌 **Real-life analogy**:

Imagine a **fruit factory**:

- map() = Peeling each fruit (modify every fruit)
- filter() = Selecting only ripe fruits (choose some)
- reduce() = Making juice by blending all (combine into one)