1️⃣ Python Lambda Functions

1. What is a lambda?

A **lambda function** is a small, **anonymous** function (no name) defined using the lambda keyword.

Normal function:

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

Same with lambda:

```
add = lambda a, b: a + b

print(add(3, 5))  # 8
```

👉 Structure: lambda arguments: expression

- No return keyword (expression is automatically returned)
- Typically used for **short, simple functions**

---

2. Basic examples

Example 1: Single argument

```
square = lambda x: x * x

print(square(4))  # 16
```

Example 2: Multiple arguments

```
multiply = lambda a, b: a * b

print(multiply(3, 7))  # 21
```

Example 3: Used with sorted()

```
students = [("Deepesh", 90), ("Amit", 75), ("Riya", 85)]


# sort by marks (index 1)

sorted_students = sorted(students, key=lambda s: s[1])
```

```
print(sorted_students)

# [('Amit', 75), ('Riya', 85), ('Deepesh', 90)]
```

---

3. Lambdas with map, filter, sorted

map(function, iterable) – apply a function to each item

```
nums = [1, 2, 3, 4, 5]

squares = list(map(lambda x: x * x, nums))

print(squares)   # [1, 4, 9, 16, 25]
```

filter(function, iterable) – keep items where function returns True

```
nums = [1, 2, 3, 4, 5, 6]

even_nums = list(filter(lambda x: x % 2 == 0, nums))

print(even_nums)  # [2, 4, 6]
```

✅ **Practice for Lambda**

Try these:

1. Write a lambda that:
    - Takes a number and returns "Even" or "Odd".
2. Use filter + lambda to get all names longer than 4 characters from a list.
3. Use sorted + lambda to sort a list of dictionaries by a key, e.g.:
4. employees = [
5. {"name": "Deepesh", "age": 30},
6. {"name": "Amit", "age": 25},
7. {"name": "Riya", "age": 28},
8.]

2️⃣ **Python Generators**

1. What is a generator?

A **generator** is a special type of function that **remembers its state** and **yields values one by one** instead of returning them all at once.

- Uses yield instead of return
- Produces values **lazily** (on demand)
- Saves memory (doesn't create the whole list in advance)

---

2. Normal function vs Generator

Normal function returns a list:

```
def get_squares(n):
    result = []
    for i in range(1, n+1):
        result.append(i * i)
    return result


print(get_squares(5))  # [1, 4, 9, 16, 25]
```

Generator yields one square at a time:

```
def generate_squares(n):
    for i in range(1, n+1):
        yield i * i


gen = generate_squares(5)
print(gen)         # <generator object ...>
print(next(gen))    # 1
print(next(gen))    # 4
print(next(gen))    # 9
```

---

3. How yield works (step-by-step)

When Python sees yield:

1. It **returns** the value after yield (like return, but not final).
2. **Pauses** the function and **remembers** where it left off.
3. Next time you call next() on the generator, it **continues from the paused point**.

Example:

```
def simple_gen():
    print("Step 1")
    yield 10

    print("Step 2")
    yield 20

    print("Step 3")
    yield 30

g = simple_gen()

print(next(g))  # prints "Step 1" then 10
print(next(g))  # prints "Step 2" then 20
print(next(g))  # prints "Step 3" then 30
# next(g) now will raise StopIteration
```

---

4. Using generators in loops

You rarely use next() manually in real code.
Usually:

```
for value in generate_squares(5):
    print(value)
```

This will print: 1 4 9 16 25

---

5. Generator Expressions (short syntax)

Similar to list comprehension, but with () instead of [].

```
# List comprehension – builds whole list
squares_list = [x * x for x in range(1, 6)]


# Generator expression – generates one by one
squares_gen = (x * x for x in range(1, 6))


print(squares_gen)      # <generator object ...>
print(next(squares_gen))  # 1
print(list(squares_gen))  # remaining values: [4, 9, 16, 25]
```

✅ **Practice for Generators**

Try:

1. Write a generator countdown(n) that yields n, n-1, ..., 1.
2. Write a generator even_numbers(n) that yields all even numbers from 1 to n.
3. Convert a list comprehension into a generator expression:
4. # from:
5. cubes = [x**3 for x in range(10)]
6. # to:

7. cubes_gen = ...

3️⃣ Python Decorators

Decorators are usually the most confusing at first.
We'll go **very slowly** and build understanding step-by-step.

---

1. First concept: Functions are objects

In Python, **functions are first-class objects**:

- You can assign them to variables
- You can pass them as arguments
- You can return them from other functions

Example:

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


say_hello = greet   # assign function to variable
print(say_hello())  # Hello!
```

You can also pass a function to another function:

```
def call_function(func):
    print("Calling function...")
    print(func())


call_function(greet)
# Output:
# Calling function...
# Hello!
```

---

2. Inner functions (functions inside functions)

```
def outer():
    def inner():
        print("I am inner function")
    inner()


outer()
```

Inner functions can be **returned** too:

```
def outer():
    def inner():
        print("I am inner")
    return inner   # return function, not call


func = outer()
func()  # "I am inner"
```

This concept is critical for understanding decorators.

---

3. What is a decorator?

A **decorator** is a function that:

- Takes another function as input
- Adds some extra behavior
- Returns a new function

Syntax using @decorator_name is just a shortcut.

---

4. First decorator example (without @ syntax)

Let’s build a decorator that prints messages **before and after** a function call.

```
def my_decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper
```

Now use it:

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


decorated_function = my_decorator(say_hello)
decorated_function()
```

Output:

```
Before function call
Hello!
After function call
```

What happened?

- my_decorator(say_hello) returns wrapper
- decorated_function now **is** wrapper
- When you call decorated_function(), it runs the extra behavior + original function

---

5. Using @ decorator syntax

The same example, written more cleanly:

```
def my_decorator(func):
    def wrapper():
```

```
        print("Before function call")
        func()
        print("After function call")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

This:

```
@my_decorator
def say_hello():
    ...
```

is equivalent to:

```
def say_hello():
    ...
say_hello = my_decorator(say_hello)
```

---

6. Decorators with arguments

We need *args and **kwargs so the decorator can support **any** function signature.

```
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with args={args}, kwargs={kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned {result}")
```

```
        return result
    return wrapper

@logger
def add(a, b):
    return a + b

@logger
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

add(3, 5)
greet("Deepesh", greeting="Hi")
```

---

7. Practical decorator examples

Example 1: Timing decorator

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f} seconds")
        return result
    return wrapper
```

```
@timer
def compute():
    total = 0
    for i in range(1_000_000):
        total += i
    return total

compute()
```

Example 2: Access control (very simple)

```
USER_ROLE = "admin"

def require_admin(func):
    def wrapper(*args, **kwargs):
        if USER_ROLE != "admin":
            print("Access denied! Admins only.")
            return None
        return func(*args, **kwargs)
    return wrapper

@require_admin
def delete_user(user_id):
    print(f"User {user_id} deleted.")

delete_user(101)
```

✅ **Practice for Decorators**

Try these:

1. Create a decorator uppercase_output that:
    - Calls the original function
    - Converts its string return value to **uppercase**
2. @uppercase_output
3. def get_message():
4. return "hello world"
5. # Expected: "HELLO WORLD"
6. Create a decorator repeat(n) that calls the function n times.
   Hint: you'll need a decorator that **takes an argument**, so:
7. def repeat(n):
8. def decorator(func):
9. def wrapper(*args, **kwargs):
10. # call func n times
11. return wrapper
12. return decorator
13. Add a decorator to log function calls in your own small project code (e.g., in a script you already have).