# Module Python -Fundamentals of Python Language

Vrinda Mavadhiya 10th June Python

**How functional programming works in Python.**

** Using map(), reduce(), and filter() functions for processing data.**

** Introduction to closures and decorators.**

Functional programming is a style where **functions are treated as first-class citizens** (they can be passed around just like data).  
It focuses on **pure functions**, **immutability**, and **avoiding side effects**.

**🔹 1. Using map(), filter(), and reduce() Functions**

These are **higher-order functions**—they take another function as an argument.

**✅ map() – Apply a function to each item in a sequence**

**🔸 Syntax:**

map(function, iterable)

**Example:**

numbers = [1, 2, 3, 4]

squared = list(map(lambda x: x \*\* 2, numbers))

print(squared) # Output: [1, 4, 9, 16]

**✅ filter() – Keep items that match a condition**

**🔸 Syntax:**

filter(function, iterable)

**Example:**

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

evens = list(filter(lambda x: x % 2 == 0, numbers))

print(evens) # Output: [2, 4]

**✅ reduce() – Reduce all items to a single value**

* It comes from the functools module.

**🔸 Syntax:**

from functools import reduce

reduce(function, iterable)

**Example:**

from functools import reduce

numbers = [1, 2, 3, 4]

total = reduce(lambda x, y: x + y, numbers)

print(total) # Output: 10

**🔹 2. Closures in Python**

A **closure** is a function defined inside another function that **remembers the variables** from its enclosing scope even after the outer function has finished executing.

**✅ Example:**

def outer(x):

def inner(y):

return x + y # x is remembered from outer()

return inner

add\_five = outer(5)

print(add\_five(3)) # Output: 8

✅ **Explanation:**

* inner() remembers x even after outer() has finished.
* This is **useful in decorators and functional programming**.

**🔹 3. Decorators in Python**

A **decorator** is a function that **modifies another function’s behavior** without changing its code.

**✅ Basic Decorator Example:**

def my\_decorator(func):

def wrapper():

print("Before the function runs")

func()

print("After the function runs")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

**Output:**

Before the function runs

Hello!

After the function runs

✅ **Explanation:**

* @my\_decorator is **syntactic sugar** for:

say\_hello = my\_decorator(say\_hello)

* It wraps the say\_hello() function with additional behavior.