**Assignment10**

**Advanced Python (map(), reduce(), filter(), Closures and Decorators**)

**How functional programming works in Python**

Functional programming in python focuses on writing code using pure functions , avoiding shared states ,and relying heavily on functions as first class citizens.

This means functions can be passed around as arguments and returned from other functions , and assigned to variables.

**Key concepts of functional programming in Python:**

**1 Pure functions:**

A pure function’s output depends only on its input and has no side effects**.**

**Example:**

def add(x,y):

return x+y

**2 Immutability:**

Data is not modified directly. Instead new data structures are created.

**Example:**

Original\_list=[1,2,3]

New\_list=list(map(lambda x: x\*2,original\_list))

**3 Higher-Order functions:**

Functions that take other functions as arguments or return them.

Functional programming in Python focuses on writing code using pure functions, avoiding shared states, and relying heavily on functions as first-class citizens. This means functions can be passed around as arguments, returned from other functions, and assigned to variables.

def apply(func, value):

return func(value)

def square(x):

return x \* x

result = apply(square, 5) # 25

1. **Anonymous Functions (Lambda Functions):**

Functions created without a name, usually used for small operations.

Example:

square = lambda x: x \*\* 2

print(square(4)) # 16

1. **Recursion:**

Functions that call themselves to solve problems.

Example:

def factorial(n):

if n == 1:

return 1

return n \* factorial(n - 1)

print(factorial(5)) # 120

1. **List Comprehensions**:

A functional way to generate new lists by applying a function to each element.

Example:

squares = [x \*\* 2 for x in range(5)] # [0, 1, 4, 9, 16]

**Benefits of Functional Programming:**

* Cleaner and more concise code
* Easier debugging and testing (due to pure functions)
* Better support for parallel processing
* Reduced side effects and bugs

**Limitations:**

* Not as performant as imperative programming for certain tasks.
* Can be less intuitive for complex, state-heavy operations.

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

### 1. ****map()**** – Transform Data

map() applies a function to every item in an iterable (like a list) and returns a map object (which can be converted to a list).  
**Use Case:** Apply transformations to all elements.

**Example:**

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

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

print(squared) # Output: [1, 4, 9, 16, 25]

### 2. ****filter()**** – Filter Data

filter() applies a function to each element and returns only those that evaluate to True.  
**Use Case:** Select items based on a condition.

**Example:**

numbers = [1, 2, 3, 4, 5, 6]

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even\_numbers) # Output: [2, 4, 6]

### 3. ****reduce()**** – Aggregate Data

reduce() repeatedly applies a function to pairs of elements, reducing the iterable to a single cumulative result. It’s part of the functools module.  
**Use Case:** Aggregate values to produce a single output (like sum or product).

**Example:**

from functools import reduce

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

product = reduce(lambda x, y: x \* y, numbers)

print(product) # Output: 120 (1\*2\*3\*4\*5)

### ****Combining map(), filter(), and reduce():****

### ****Example:****

numbers = [1, 2, 3, 4, 5, 6]

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

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

result = reduce(lambda x, y: x + y, squared)

print(result) # Output: 56 (2^2 + 4^2 + 6^2)

**Introduction to closures and decorators.**

### ****1. Closures****

A **closure** is a function that remembers the environment (variables) in which it was created, even if it is called outside of that environment.

**How it Works:**

* A nested function can access variables from its outer function.
* Even after the outer function has finished executing, the inner function retains access to those variables.

**Example:**

def outer\_function(x):

def inner\_function(y):

return x + y

return inner\_function

closure\_example = outer\_function(10)

print(closure\_example(5)) # Output: 15

**Explanation:**

* inner\_function remembers x from outer\_function even after outer\_function has executed.

**Use Case:**

* Data hiding and maintaining state between function calls.

### ****2. Decorators****

A **decorator** is a design pattern in Python that allows you to modify the behavior of a function or method. It "wraps" another function to add functionality before or after the wrapped function runs, without modifying the function itself.

**How it Works:**

* A decorator is a function that takes another function as an argument, performs some operations, and returns a new function.

**Example:**

def decorator\_function(original\_function):

def wrapper\_function():

print("Wrapper executed before", original\_function.\_\_name\_\_)

return original\_function()

return wrapper\_function

@decorator\_function

def say\_hello():

print("Hello!")

say\_hello()

**Output:**

Wrapper executed before say\_hello

Hello!

**Explanation:**

* @decorator\_function is syntactic sugar for:
* say\_hello = decorator\_function(say\_hello)

### ****Decorators with Arguments:****

def repeat(n):

def decorator(func):

def wrapper(\*args, \*\*kwargs):

for \_ in range(n):

func(\*args, \*\*kwargs)

return wrapper

return decorator

@repeat(3)

def greet():

print("Hello!")

greet()

**Output:**

Hello!

Hello!

Hello!

### ****Key Points:****

* **Closures** capture and remember variables from their enclosing scope.
* **Decorators** modify or enhance functions, making code reusable and clean.
* **Real-World Use:** Logging, authentication, timing functions, and access control.