RAMAIAH
Institute of Technology

Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science & Engineering

Data Visualization with Python Lab(CSL48)

USN: **Week #: 02**

Semester: **Section:** **Date:**

Functional Programming in Python

Functional programming is a programming paradigm that treats computation as the evaluation of **pure functions** and avoids changing state or mutable data.

Key Concepts in Functional Programming

1. **Functions** – Reusable blocks of code that perform a specific task.
2. **Lambda Functions** – Anonymous functions defined using the `lambda` keyword.
3. **Higher-Order Functions** – Functions that take other functions as arguments or return functions.
4. **Map, Filter, Reduce** – Built-in functions that apply operations on iterables.
5. **Decorators** – Special functions that modify the behavior of other functions.

---

1. Functions

Definition:

A function in Python is defined using the `def` keyword. It allows code reuse and improves readability.

Programs

1a. Function to return the sum of two numbers

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

# Example
print(add_numbers(5, 10))  # Output: 15
```

1b. Function to return the maximum number in a list

```
def find_max(lst):
    return max(lst)

# Example
numbers = [3, 7, 2, 9, 5]
print(find_max(numbers))  # Output: 9
```

---

2. Lambda Functions

Definition:

Lambda functions are **anonymous functions** that can have multiple arguments but only one expression.

Programs

2a. Lambda function to find the product of two numbers

```
multiply = lambda x, y: x * y

# Example
print(multiply(4, 5))  # Output: 20
```

2b. Lambda function to sort a list of tuples based on the second element

```
tuples_list = [(1, 4), (3, 1), (5, 2)]
sorted_list = sorted(tuples_list, key=lambda x: x[1])

print(sorted_list)  # Output: [(3, 1), (5, 2), (1, 4)]
```

---

3. Map Function

Definition:

`map(function, iterable)` applies a function to every element in an iterable.

Programs

3a. Convert a list of strings to uppercase using `map()`

```
words = ["hello", "world", "python"]
uppercased = list(map(str.upper, words))

print(uppercased)  # Output: ['HELLO', 'WORLD', 'PYTHON']
```

3b. Compute the square of each number in a list using `map()`

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))

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

---

4. Filter Function

Definition:

`filter(function, iterable)` filters elements based on a condition.

Programs

4a. Extract even numbers from a list using `filter()`

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

print(evens)  # Output: [2, 4, 6]
```

4b. Remove empty strings from a list using `filter()`

```
words = ["Python", "", "Functional", "", "Programming"]
non_empty_words = list(filter(lambda x: x != "", words))

print(non_empty_words)  # Output: ['Python', 'Functional',
'Programming']
```

---

5. Reduce Function

Definition:

`reduce(function, iterable)` applies a function cumulatively to reduce an iterable to a single value.
Requires importing from `functools`.

Programs

5a. Find the maximum number in a list using `reduce()`

```
from functools import reduce

numbers = [3, 7, 2, 9, 5]
max_num = reduce(lambda x, y: x if x > y else y, numbers)

print(max_num)  # Output: 9
```

5b. Compute the product of all numbers in a list using `reduce()`

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)

print(product)  # Output: 120
```

6. Decorators

Definition:

A **decorator** is a function that takes another function as input and modifies its behavior.

Programs

6a. Decorator to print messages before and after calling a function

```
def before_after_decorator(func):
    def wrapper():
        print("Before calling function")
        func()
        print("After calling function")
    return wrapper

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

say_hello()
```

Output:

```
Before calling function
Hello!
After calling function
```

6b. Decorator to convert function output to uppercase

```
def uppercase_decorator(func):
    def wrapper():
        return func().upper()
    return wrapper

@uppercase_decorator
def greet():
    return "hello, world"

print(greet())  # Output: HELLO, WORLD
```

7. Higher-Order Functions

Definition:

A **higher-order function** is a function that either takes another function as an argument or returns a function.

Program

7. Function that returns a lambda function for multiplication

```
def multiplier(factor):
    return lambda x: x * factor

double = multiplier(2)
print(double(5))  # Output: 10
```

---

8. Function Taking Two Functions as Arguments

Definition:

A function that takes two functions as arguments and applies both to a value.

Program

8. Function that applies two functions in sequence

```
def apply_functions(func1, func2, value):
    return func2(func1(value))

# Example functions
def add_five(x):
    return x + 5

def square(x):
    return x ** 2

# Applying both functions in sequence
result = apply_functions(add_five, square, 3)
print(result)  # Output: (3+5)^2 = 64
```

Summary

Topic	Concept	Function Used
Functions	Reusable code blocks	`def`
Lambda Functions	Anonymous functions	`lambda`
Map Function	Apply function to list elements	`map()`
Filter Function	Filter elements based on condition	`filter()`
Reduce Function	Reduce iterable to a single value	`reduce()`
Decorators	Modify function behavior	`@decorator`
Higher-Order Functions	Functions that return/take functions	Function as argument