8️⃣

Chapter 8

🧠 Python Chapter 8 – Functions

(Functions help you write reusable, modular code — aka less typing, more doing!)

🔷 What is a Function?

> A function is a reusable block of code that only runs when it is called. It helps break programs into smaller chunks to make them more organized and readable.

📌 Basic Function Syntax:

```
def function_name():
    # Code block (indented)
    ...
```

Example:

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

To run (or “call”) this function:

```
greet()
```

✨ Real Example – `avg()` and `greet()`

```
def avg():
    a = int(input("Enter Number 1: "))
    b = int(input("Enter Number 2: "))
    c = int(input("Enter Number 3: "))
    average = (a + b + c) / 3
    print("Average:", int(average))  # int() rounds down
```

Explanation:

- `def avg():` → We define a new function.
- We take 3 numbers as input.
- Calculate the average.
- Print the result.
- **This function does NOT return anything**, it only prints.

👋 Function with Input (`greet()`)

```
def greet():
    name = input("Enter your name: ")
    print("Good day", name)

greet()  # Calling the function
```

Here:

- The function `greet()` asks your name.

- Then it prints a friendly greeting using that input.

🧠 Function Parameters (Inputs You Control)

```
def Goodday(name, ending):
    print("Good Day,", name)
    return "Danke"
```

Here, `name` and `ending` are **parameters** (like input slots).

They get **values** when we call the function:

```
Goodday("Prathamesh", "Thank you")
Goodday("Harry", "Thanks")
```

But we are not using `ending` inside the function — we could update the print to:

```
print(ending)
```

So now the full function becomes useful.

🔁 Default Parameters

```
def Goodday(name, ending="Thanks"):
    print("Good Day,", name)
    print(ending)
```

Calling it like this:

```
Goodday("Prathamesh")
```

Gives:

```
Good Day, Prathamesh
Thanks
```

🧠 If you **don't** provide the second argument, Python uses the **default** (`"Thanks"`).

🔄 Return vs Print

- `print()` → Shows output to the user
- `return` → Sends data back to where the function was called

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

result = add(4, 5)
print(result)  # 9
```

♻️ Recursion – Function Calling Itself

```
def factorial(n):
    if n == 1 or n == 0:
        return 1
    return n * factorial(n - 1)
```

Example:

```
factorial(5)
= 5 * factorial(4)
= 5 ∗ 4 * factorial(3)
= 5 ∗ 4 ∗ 3 * factorial(2)
= 5 ∗ 4 ∗ 3 ∗ 2 * factorial(1)
= 5 ∗ 4 ∗ 3 ∗ 2 ∗ 1 = 120
```

💡 Key Concepts in Recursion:

1. **Base Case** – Where recursion stops

   → `if n == 0 or n == 1: return 1`

2. **Recursive Case** – Where function keeps calling itself

   → `return n * factorial(n - 1)`

⚠️ Risk:

If you forget the base case, the function will **run infinitely** and crash.

✅ Tips

- Use **functions** to avoid repeating code.
- Always try to `return` values when needed, instead of just printing them inside.
- Avoid infinite recursion by setting a **base case**.
- Keep function names **clear and specific**.

🧠 Advanced Concepts – Functions & Recursion (Python Deep Dive)

🔶 1. Function Parameters — Positional, Keyword, Default, Arbitrary

Python functions accept multiple types of arguments. These are **core for flexibility**, especially in larger projects.

✅ Positional Arguments

These are passed in order.

```
def greet(name, msg):
    print(f"{msg}, {name}!")

greet("Prathamesh", "Hello")
```

Order matters.

✅ Keyword Arguments

Allows passing arguments using their **names**, irrespective of order.

```
greet(msg="Welcome", name="Harry")  # Output: Welcome, Harry!
```

✅ Clean, readable, reduces error.

✅ Default Arguments

Give default value, which can be **overwritten**.

```
def greet(name, msg="Hi"):
    print(f"{msg}, {name}!")

greet("Prathamesh")          # Uses default msg
greet("Prathamesh", "Hello")  # Overrides default
```

⚠️ Rule: Default arguments **must come after** non-default.

✅ Arbitrary Arguments – `args` and `*kwargs`

👉 `args` → Multiple positional arguments as a tuple

```
def adder(*nums):
    total = 0
    for n in nums:
        total += n
    return total

print(adder(1, 2, 3, 4, 5))  # Output: 15
```

👉 `*kwargs` → Multiple keyword arguments as a dictionary

```
def show_info(**info):
    for key, value in info.items():
        print(f"{key} = {value}")

show_info(name="Prathamesh", age=17, lang="Python")
```

✅ Real-world use: APIs, class initializations, configs.

🔶 2. Function Scope – Local, Global & `global` Keyword

✅ Local Scope

Variables created inside a function are **local**.

```
def foo():
    x = 10  # Local to foo()
    print(x)

foo()
# print(x)  # Error: x is not defined
```

✅ Global Scope

Accessible throughout the file unless shadowed.

```
x = 5

def bar():
    print(x)  # Works

bar()
```

⚠️ Shadowing and the `global` Keyword

If you assign to a variable inside a function, Python assumes it's **local** unless you use `global`.

```
x = 5

def change():
    global x
    x = 10

change()
print(x)  # Output: 10
```

🔶 3. Return Statements — Single & Multiple Values

```
def compute():
    return 1, 2, 3  # Tuple of values

a, b, c = compute()
```

✅ Return multiple results easily.

✅ Often used in: coordinates, status + result combos, etc.

🔶 4. Docstrings

Use triple quotes to document functions:

```
def square(n):
    """Returns square of a number"""
    return n * n

print(square.__doc__)
```

✅ Helps auto documentation. Essential for team projects.

🔶 5. Lambda Functions (Anonymous Functions)

Single-expression functions — super useful with **sorting**, **filtering**, etc.

```
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

👉 Real use:

```
pairs = [(1, 2), (3, 1), (5, 0)]
pairs.sort(key=lambda x: x[1])
print(pairs)  # Sorted by second item
```

🔶 6. Higher Order Functions

Functions that take other functions as **arguments** or return functions.

```
def apply(func, x):
    return func(x)

print(apply(lambda x: x**2, 5))  # Output: 25
```

✅ Core to functional programming.

🔶 7. Recursion — The Real Game

Recursion = Function calling itself.

> Every recursive function has:
>
> - ✅ **Base Case** — Stop recursion
> - 🔁 **Recursive Case** — Keep going

⚠️ Common Mistakes:

1. **No base case** → Infinite loop
2. **Wrong base case** → Logic error
3. **Not reducing input** → Stack overflow

🔶 8. Recursive Patterns and Examples

🔁 Factorial

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

🔁 Fibonacci (Bad Version)

```
def fib(n):
    if n == 0: return 0
    if n == 1: return 1
    return fib(n-1) + fib(n-2)
```

⚠️ Very inefficient — exponential time!

✅ Fibonacci (With Memoization)

```
memo = {}

def fib(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```

🚀 Now it's **linear time**.

🧠 Use Recursion For:

- Tree traversal
- Backtracking problems
- Divide and conquer (merge sort, quicksort)

🔶 9. Function Composition

Calling functions **inside other functions**.

```
def square(n):
    return n * n

def cube(n):
    return square(n) * n

print(cube(3))  # Output: 27
```

✅ Keeps code **modular** and **readable**.

🔶 10. Best Practices in Functions

Tip	Why it matters
Small functions	Easy to read, test
Use meaningful names	`get_area()` > `f1()`
Use docstrings	Self-documenting code
Avoid global variables	Maintain encapsulation
Prefer return over print	More reusable

✅ Chapter 8 – Practice Problems (Functions)

🔷 Problem 1: Find Greatest of 3 Numbers

```
n1 = int(input("Enter Number 1: "))
n2 = int(input("Enter Number 2: "))
n3 = int(input("Enter Number 3: "))

def greatest(n1, n2, n3):
    l1 = [n1, n2, n3]     # Store numbers in a list
    l1.sort()            # Sort the list in ascending order
    print(f"The greatest number is {l1[2]}")  # Last element is greatest

greatest(n1, n2, n3)
```

✅ Notes:

- List sorting gives a clean one-liner way to get the max.
- Alternate method: Use nested `if-else`, but it's verbose and less readable.
- Efficient, readable, and Pythonic.

🔷 Problem 2: Celsius to Fahrenheit Converter

```
celsius = int(input("Enter the temperature (in °C): "))

def temp(celsius):
    fahrenheit = 1.8 * celsius + 32
    print(f"The temperature in Fahrenheit is: {round(fahrenheit, 3)}")

temp(celsius)
```

✅ Notes:

- `1.8 * C + 32` → Standard formula.
- `round(value, 3)` ensures up to 3 decimal places — good habit for display precision.
- Shows how math & formatting can be combined in a function.

🔷 Problem 3: Preventing Newline in `print()`

```
print("a")
print("a")
print("a", end="")  # 👈 This prevents a new line
print("a", end="")
```

✅ Notes:

- By default, `print()` adds `\n` (newline).
- `end=""` tells Python what to print **instead** of the default newline.
- Very useful for formatting tables, side-by-side text, or animation frames.

🔷 Problem 4: Recursive Sum of First n Natural Numbers

```
def sum(n):
    if n == 1:
        return 1
    return sum(n - 1) + n

print(sum(4))  # Output: 10
```

✅ How it Works:

For `n = 4`:

```
sum(1) = 1
sum(2) = 1+2
sum(3) = 1+2+3
sum(n) = 1+2+3+4+5+........n
sum(n) = sum(n-1) + n
```

- Base Case: `if n == 1: return 1` stops recursion.
- Recursive Case: `sum(n-1) + n` builds the total.

🔷 Problem 5: Recursive Star Pattern

Print pattern:

```
***
**
*
```

```
a = int(input("Enter the number: "))

def pattern(n):
    if n == 0:
        return
    print("*" * n)   # Prints n stars
    pattern(n - 1)   # Recursive call for next line

pattern(a)
```

✅ Notes:

- Recursion replaces loop here.
- `print("*" * n)` uses string multiplication.
- Useful in pattern problems — shows control over **both logic and recursion**.

🔷 Problem 6: Inches to Centimeters

```
a = int(input("Enter the number (in Inches): "))

def conv(n):
    print(f"The number in cm is: {n * 2.54}")

conv(a)
```

✅ Notes:

- `1 inch = 2.54 cm`
- Simple function shows math + output formatting again.
- Can be turned into a bidirectional unit converter with more logic.

🔷 Problem 7: Remove a Number from List

```
l1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 24, 424]

def r1(n):
    l1.remove(n)  # Removes first occurrence of n
    print(f"The list after removal: {l1}")

x = int(input("Enter the number to remove: "))
r1(x)
```

✅ Notes:

- `remove()` deletes **first occurrence** of a value.
- If value not found, raises `ValueError`.
- Useful for learning list mutation inside functions.

🔁 Alternative Way (More Control):

```
def rem(l1, num):
    return [i for i in l1 if i != num]

print(rem(l1, 2))
```

- This returns a **new list** without `num`.
- More "functional" programming style (no mutation).

🔷 Problem 8: Multiplication Table using Function

```
def multiply(n):
    for i in range(1, 11):
        print(f"{n} X {i} = {n * i}")

x = int(input("Enter the number: "))
multiply(x)
```

✅ Notes:

- Core use of loop inside a function.
- Uses string formatting (`f""`) to make output clean.
- Reusable, extendable — can easily become a 1–20 table generator.

📚 Chapter 8 – Functions & Recursion: Complete Summary

✅ What You Learned

Concept	Description
`def` keyword	Define custom functions
Function calls	Invoke your function logic
Arguments	Values passed to function

Concept	Description
Return values	Send back results from function
`*args` , `**kwargs`	Arbitrary arguments (tuple/dict)
Scope	Local vs global variables
Recursion	Function calling itself
Base Case	Stops recursion
Lambda	One-liner anonymous function
Docstring	Add descriptions to functions
Composition	Use functions inside functions