1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1

</ Table of contents

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

What is Function?

- A function is like a little machine that:
  - Takes some input.
  - Does some work.
  - Gives back some output.
- Functions are first-class object, you can pass them around, store them, etc.

1. Basic function.
2. Function as variable.
3. Passing function to another function.
4. Return a function from another function.

- **Why this important in OOP?**
  - You can pass behavior (functions) around just like data
  - You can decide which function to call while the program is running
  - It's how many advanced Python features work (like decorators)

- functions are just like any other value this what makes Python very flexible.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

What Is a Decorator

- A decorator is like a wrapper that adds extra functionality to an existing function without changing the original function itself.

- **How decorators work?**

  The @my_decorator syntax is actually just a shortcut for this:

```
def say_hello():
    print("Hello!")
say_hello = my_decorator(say_hello)  # This is what @my_decorator does
say_hello()
```

- **Why Use Decorators?**
  - Adding logging (keeping records of when functions run)
  - Timing how long functions take to run
  - Checking if a user is logged in before running a function
  - Adding retry logic if a function fails
  - Many other "around the function" behaviors

- **Task:** Write a decorator that logs "Start" and "End" before and after a function runs.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Decorators with Arguments

- **Why Do We Need Decorators with Arguments?**
  - If the decorator doesn’t handle arguments properly, it will break when used on different functions.

- ***args** and ****kwargs**: makes the decorator flexible and work with any function.
  - ***args**: Captures all positional arguments (like name, a, b).
  - ****kwargs**: Captures all keyword arguments (like age=25, city="NY").

- Example: A Decorator That Logs Arguments

- **Task:** build a decorator that measures how long a function takes using time.time().

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Class-Based Decorators

- Decorators don’t have to be just functions—they can also be classes! This makes them more powerful and keeps things organized, especially in Object-Oriented Programming (OOP).

- **Why Use Class-Based Decorators?**
  - Reusable logic.
  - Cleaner structure.
  - More control. __init__, __call__
- **How Class-Based Decorators Work?**
  - Takes a function (func) in __init__.
  - Implements __call__ to run extra code before/after the function.
- **When to use class-based decorator?**
  - Need to track state.
  - Complex decorators.
  - OOP-heavy projects
- **Task:** create a decorator that counts how many times a function is called.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Built-in Decorators

- Python comes with ready-made decorators that help you write cleaner and more efficient object-oriented code.
- **@staticmethod** - Standalone Functions in a Class
  - A static method is a function inside a class that doesn't need self or cls. It behaves like a normal function but belongs to the class.
  - For utility functions that don't need class/instance data.
  - **e.x:** Formatting, validation, calculations
- **@classmethod** - Methods That Work with the Class Itself
  - A class method takes cls as the first argument. It can modify class-level data.
  - When you need to access/modify class-level data.
  - For alternative constructors.
  - **e.x:** tracking book count.
- **@property** - Treat Methods Like Attributes.
  - When you want to compute a value dynamically but access it like an attribute.
  - Turns a method into a **read-only** attribute.
- **@<property>.setter** - Modify "Read-Only" Properties
  - To validate before setting a value.
  - To control modifications.
- **@<property>.deleter** - Clean Up Resources
  - To clean up when a property is deleted.
  - Example: Closing files, resetting values.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Real-World Use Cases

1. **Logging – Track Function Calls:**
   You want to log when a function runs (for debugging).

2. **Authentication – Check User Permissions:**
   Only allow certain users to access a function.

3. **Input Validation – Check Data Before Processing:**
   Ensure function inputs are correct (e.g., age ≥ 18).

4. **Caching – Store Results for Faster Performance:**
   Avoid recalculating expensive operations.

5. **Rate Limiting – Control How Often a Function Runs:**
   Prevent users from calling an API too many times.

6. **Timing – Measure Function Execution Time**
   Find out how long a function takes to run.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Lab

1. validate_input Decorator: Ensure a number is non-negative.
2. classmethod to Count Instances: Track how many objects are created.
3. property for Rectangle Area: Auto-calculate area/perimeter

Bonus:

Combined Example
Features:

- validate_input: Blocks negative radius.
- classmethod: Counts circles.
- property: Auto-calculates area/circumference.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1