

**Ideation Axis**

**Backend Intermediate Development**

Edwin Asare

# Decorators

Decorators are a powerful feature in Python that allow you to modify the behavior of functions or classes without directly altering their source code. They provide a way to add additional functionality to existing code in a modular and reusable manner.

## Use Cases:

* Logging: You can create a decorator to log function calls and execution times.
* Authentication: For web applications, decorators can check if a user is authorized to access certain endpoints.

## Real-world Use Case: Authentication

Decorators are used in authentication systems to restrict access to certain parts of a web application.

Below is a demonstration of a real-world example of decorators in authentication:

# The @wraps function, imported from functools, is used in Python decorators to

# ensure that the decorated function retains its original metadata, such as its

# name, docstring, and other attributes. Without @wraps, the decorated function

# will lose these attributes and take on the metadata of the wrapper function

# inside the decorator.

from functools import wraps

# Database for this software

users\_db = {

    "edwin": "password123",

    "joseph": "qwerty",

    "ideation axis": "letmein"

}

# Global variable to track logged-in user

current\_user = None

# Authentication decorator

def login\_required(func):

    """

    This decorator checks if the current\_user is logged in

    before executing the decorated function. If the user is not logged in,

    it prints an error message and returns None. Otherwise, it executes the decorated

    """

    @wraps(func)

    def check\_status(\*args, \*\*kwargs):

        if not current\_user:

            print("You need to log in first!")

            return None

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

    return check\_status

# Logout Function

def logout():

    global current\_user

    if current\_user:

        print(f"Goodbye, {current\_user}!")

        current\_user = None

    else:

        print("No user is logged in.")

# Protected function that requires login

# The @login\_required decorator is used on top of the view\_sensitive\_data()

# function to enforce access control. This means that only authenticated

# users (those who are logged in) are allowed to access this function

@login\_required

def view\_sensitive\_data():

    print(f"Ideation Axis Bank \n Ye'llo valued customer {current\_user}: Your bank balance is GHC1500.")

# Usage Example, that is typing it manually

if \_\_name\_\_ == "\_\_main\_\_":

    # Trying to access sensitive data without login

    view\_sensitive\_data()

    # Logging in

    login("edwin", "password123")

    # Trying again after login

    view\_sensitive\_data()

    # Logging out

    logout()

    # Trying to access sensitive data after logout

    view\_sensitive\_data()

## The output

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/User/Desktop/Ideation Axis/decorators.py"

You need to log in first!

Login successful! Welcome, edwin.

Ideation Axix Bank

Ye'llo valued customer edwin: Your bank balance is GHC1500.

Goodbye, edwin!

You need to log in first!

PS C:\Users\User>

# Generators

Generators are functions that return an iterator, allowing you to iterate over a sequence of values without creating the entire sequence in memory at once. This is particularly useful for working with large datasets or infinite sequences.

## Use Cases:

* Large Data Processing: Generators can handle large datasets efficiently by yielding items one at a time, reducing memory usage.
* Streaming Data: Useful in scenarios like reading files line by line or processing data streams.

## Real-World Example: Generating Infinite Data Streams

Imagine you're dealing with **real-time data** from a sensor or an API that generates continuous data (e.g., temperature data). You don’t want to load all of this data into memory, especially since it's infinite. A generator can produce this data on-demand. Below is a demonstration:

import random

import time

# Generate random temperature readings from a sensor

def generate\_sensor\_data():

    while True:

        # Generate random temperature between 10 and 25 on at a time using yield function

        yield random.uniform(10.0, 25.0)

        # delay 1s between readings

        time.sleep(1)

# Process the infinite stream of sensor data

for data in generate\_sensor\_data():

    print(f"Sensor reading: {data}")

    if data > 24.5:

        print("Warning: High temperature!")

## Output

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/User/Desktop/Ideation Axis/generators.py"

Sensor reading: 11.079964829136571

Sensor reading: 11.59397585361

Sensor reading: 11.369006914217607

Sensor reading: 12.642230908814556

Sensor reading: 20.57985302151274

Warning: High temperature!

Sensor reading: 23.161044497054675

Warning: High temperature!

Sensor reading: 29.730460066053197

Warning: High temperature!

# Lambda Expressions

Lambda expressions, also known as anonymous functions, are concise functions defined without a name. They are often used as arguments to functions that expect functions as input.

## Use Cases:

* Sorting and Filtering: Using lambda functions as a key for sorting or within functions like `filter()` and `map()`.
* Data Manipulation: Quick transformations in data processing tasks.

## Real-World Example: Sorting

A **real-world practical example** of using lambda functions in Python is when you need **short, anonymous functions** for simple operations, especially when working with functions like **sort(), filter(),** and **map()** that expect other functions as arguments. Lambdas allow you to create these functions quickly and inline, without the need to define a separate function with def.

Imagine you have a list of dictionaries where each dictionary represents a person, and you want to sort the list based on the ID. This is a common task when dealing with data, such as sorting records from a database or a JSON response.

Below is a demonstration:

# List of people at Ideation Axis

ideation\_db = [

    {"name": "Edwin", "position": "Intern @ Ideation Axis", "id": 22},

    {"name": "Joseph Nyarko", "position": "Facilitator @ Ideation Axis", "id": 25},

    {"name": "Esegbe Able Katapu", "position": "Facilitator @ Ideation Axis", "id": 25},

    {"name": "Philip Appiah Gyima", "position": "CEO @ Ideation Axis", "id": 35}

]

# Sort the list by id using a lambda function

sorted\_interns = sorted(ideation\_db, key=lambda intern: intern['id'])

# Print the sorted list

for intern in sorted\_interns:

    if(intern["id"] == 22):

        print('Interns: \n')

        print(f"{intern['name']}: {intern['position']} | ID : {intern['id']}  \n")

## Output

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/User/Desktop/Ideation Axis/lambda.py"

Interns:

Edwin: Intern @ Ideation Axis | ID : 22

PS C:\Users\User>

# List Comprehensions

List comprehensions provide a concise way to create lists based on existing lists or other iterable objects. They are often used to perform transformations or filtering on data.

## Use Cases:

* Transformations: Quickly transforming data or filtering lists.
* Code Clarity: It enhances readability in cases of simple list operations.

## Practical Example: Filtering and Transforming Data from an API

Imagine you're dealing with an API response from an e-commerce platform where you need to extract the names of all products that are "in stock" and convert their names to lowercase.

products = [

    {"name": "Laptop", "stock": 5, "price": 1500},

    {"name": "Smartphone", "stock": 0, "price": 700},

    {"name": "Tablet", "stock": 10, "price": 400},

    {"name": "Television", "stock": 5, "price": 1900},

    {"name": "Fridge", "stock": 6, "price": 1700},

    {"name": "Speaker", "stock": 15, "price": 200},

]

# List comprehension to get in-stock product names in lowercase

in\_stock\_products = [product['name'].lower() for product in products if product['stock'] > 0]

print('Products available in stock: ')

print('==============================')

print(in\_stock\_products)

## Output

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/User/Desktop/Ideation Axis/list-comprehension.py"

Products available in stock:

==============================

['laptop', 'tablet', 'television', 'fridge', 'speaker']

PS C:\Users\User>

In conclusion, decorators, generators, lambda expressions, and list comprehension are powerful Python features that enhances code efficiency, readability, and expressiveness making it easier to write well-structured and maintainable code in real world application.