Python

Full stack Skills Bootcamp

Introducing Python Decorators

- What are Decorators?
  - Decorators are higher-order functions that modify or extend the behaviour of functions or methods. They accept a function as an argument and return a new function that adds functionality.
- Purpose:
  - Enhance code readability and reusability.
  - Commonly used for logging, enforcing access control, instrumentation, and caching.

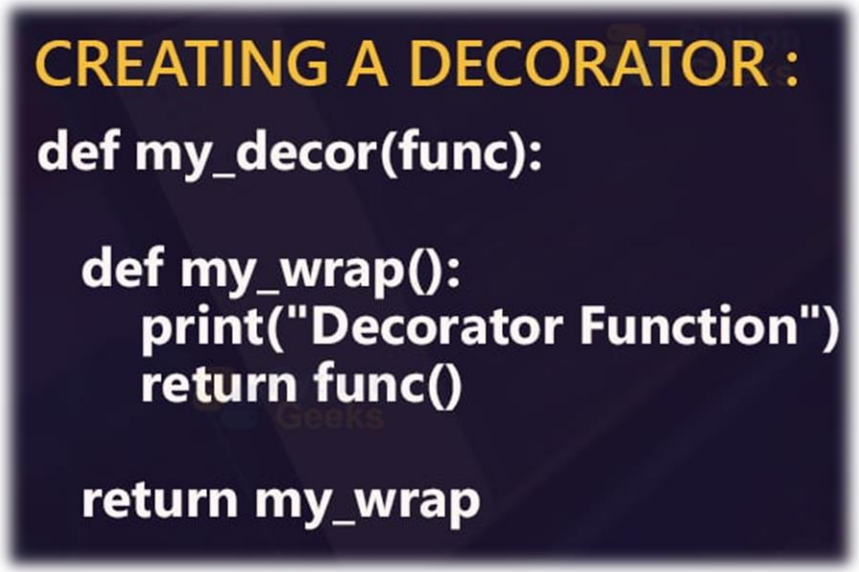
Defining a Decorator

- Creating a Decorator:

```python
def simple_decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

@simple_decorator
def greet():
    print("Hello!")

greet()
```



- simple_decorator() wraps the greet function, adding behaviour before and after its execution.
- Output: When we call greet(), we get output as

  Before function call
  Hello!
  After function call

Understanding the Wrapper Function

What is a Wrapper ?

A function that "wraps" another function, allowing additional functionality to be added before or after the wrapped function executes.

Key Benefits:

- Keeps the original function intact.
- Allows for reusability and separation of concerns.

Decorator with Parameters Example

```python
from functools import wraps
import time

def timing_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Execution time: {end_time - start_time} seconds")
        return result
    return wrapper

@timing_decorator
def long_running_task():
    time.sleep(2)  # Simulates a long-running task
    print("Task complete")

long_running_task()
```

- The timing_decorator measures how long function takes to execute, printing the execution time.
- Output: Execution time: 2.002345 seconds
  Task complete

Use Cases for Decorators

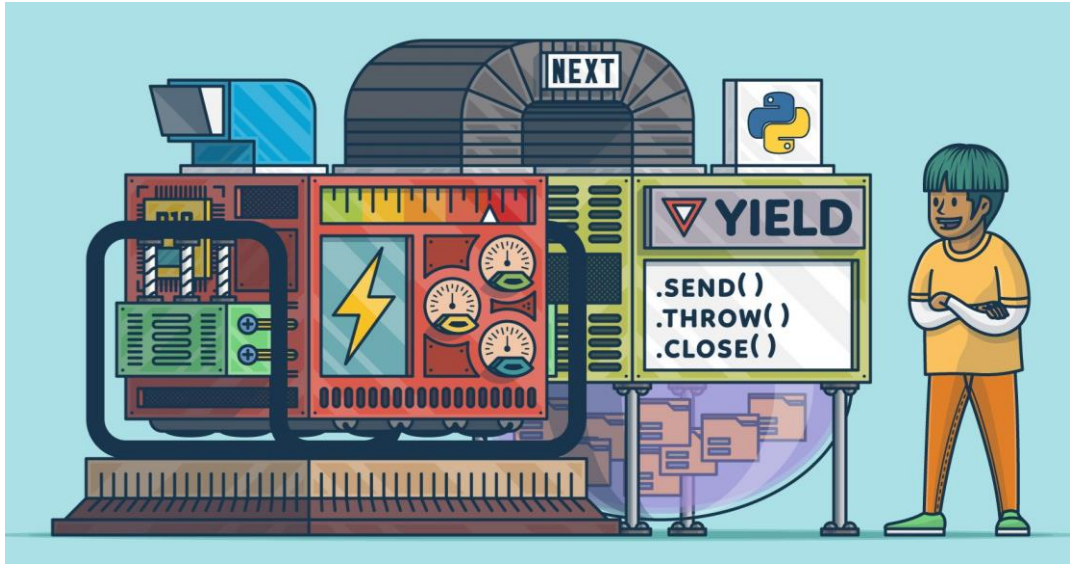
- Logging: Track function usage and debug issues.
- Timing: Measure execution duration to optimize performance
- Access Control: Restrict access to certain functions based on conditions.
- Caching: Store results of expensive function calls and return cached results when the same inputs occur.

Concluding Decorators

- So,
  - Decorators are a powerful feature in Python that enhance the functionality of functions and methods in a clean and maintainable way.
  - Experiment with decorators in your own projects to see their benefits firsthand.

Introducing Python Generators

- What are Generators?
  - Generators are a type of iterable that allows on-the-fly value generation, without storing the entire dataset in memory.
  - Useful for large datasets where loading everything into memory is inefficient or impossible.
- Purpose:
  - Generators vs Lists: Lists store all elements in memory, while generators "yield" elements one at a time as needed.



Defining a Generator

- Creating a Generator:

```python
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1
```

- Yield: Pauses the function, returns a value, and resumes from the last state when called again.
- Looping through the generator:

```
for number in count_up_to(5):
    print(number)
```

Output:

```css
Counting up to 5:
1
2
3
4
5
```

Manual Retrieval from a Generator

- Manual Generator Control with next():

```python
generator = count_up_to(3)
print(next(generator))  # Output: 1
print(next(generator))  # Output: 2
print(next(generator))  # Output: 3
```

- Generators can be controlled manually using the next() function.
- If the generator is exhausted (no more yields), calling next() raises a StopIteration exeception.

Comparison with Lists

Generators vs Lists

Using a list to store numbers up to 5. All elements are stored in memory, which can be inefficient for large datasets

```python
numbers_list = list(range(1, 6))
print(numbers_list)  # Output: [1, 2, 3, 4, 5]
```

Using a generator to yield numbers up to 5:

```python
for number in count_up_to(5):
    print(number)
```

Generators are more efficient for large or infinite data streams where storing the entire dataset is impractical.

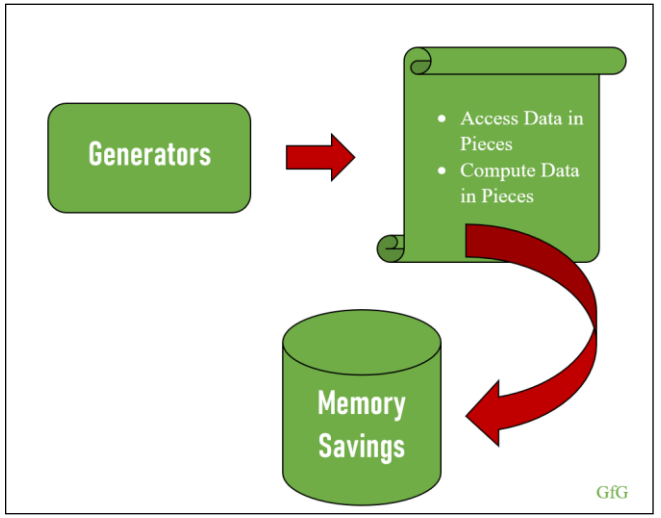
Practical Use Case of Generators

- Memory Efficiency in Real Scenarios

When to Use Generators: When dealing with large datasets (e.g., logs, real-time sensor data) where loading everything into memory would be inefficient.

For example:

Processing large files line by line without loading the entire file. Streaming data from an API, fetching it in chunks, rather than all at once.



Generators provide a way to handle such data efficiently, yielding one item at a time as needed.

Generator with Filtering Logic

```python
def filter_even_numbers(data):
    for num in data:
        if num % 2 == 0:
            yield num
```

```python
for even_num in filter_even_numbers(range(1, 101)):
    print(even_num)
```

- Generators can be combined with conditions to yield only certain values (e.g., even numbers).
- This generator will yield only even numbers from 1 to 100

Benefits of Generators

- Memory Efficiency: Ideal for large datasets because generators only yield data as needed.
- Performance: They reduce overhead by avoiding the need to load and process entire datasets at once.
- Lazy Evaluation: Generators only compute values when they are required, making them useful for performance-critical applications.

Advanced Generator Usage – Chaining Generators

Chaining Generators for Data Pipelines

Generators can be chained to build powerful data processing pipelines.

```python
def number_gen():
    yield from range(1, 10)

def square_gen(numbers):
    for num in numbers:
        yield num ** 2

for squared in square_gen(number_gen()):
    print(squared)
```

This shows how multiple generators can work together in a pipeline

```
1, 4, 9, 16, ..., 81
```

Advanced Generator Usage – Data Streaming

- Simulating Data Streaming

Generators can simulate streaming data from a file or API, yielding chunks of data as they are read.

```python
def data_stream(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip()

for line in data_stream('large_file.txt'):
    print(line)
```

Real-World Use: Ideal for scenarios like processing log files, real-time monitoring, or streaming large datasets.

Concluding Generators

- So,
  - Generators are a key tool for efficient, real-time data processing and memory management.
  - Use generators whenever you need lazy evaluation and memory-efficient processing of large sequences.