Decorators and Generators in Python

- **Submitted by**: Srivathsa Tirumala
- **Course**: Data Science
- **Submitted to**: Plasmid Innovation

Table of Contents

---

1. Introduction

Python is a powerful, high-level programming language known for its simplicity and versatility. Among its many advanced features, **decorators** and **generators** stand out as tools that enhance the functionality and efficiency of Python programs.

- **Decorators** provide a way to extend or modify the behavior of functions or methods without changing their structure.
- **Generators** allow for efficient iteration through large data sets by producing values on the fly using the `yield` keyword.

This report explores the concept, syntax, examples, and practical use cases of decorators and generators in Python programming.

---

2. Understanding Decorators

What is a Decorator?

A decorator in Python is a function that modifies or extends the behavior of another function or method. It "wraps" another function and can execute code before or after the target function runs.

Decorators follow the principle that **functions are first-class objects** in Python, meaning they can be passed as arguments, returned from other functions, or assigned to variables.

Basic Syntax and Structure

A decorator is implemented using the `@my_decorator` syntax. Here is an example:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

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

say_hello()
```

Output:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

Examples of Decorators

1. **Simple Decorator**:

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

@simple_decorator
def test():
    print("Inside the function")

test()
```

2. **Decorator with Parameters**:

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

Output:

```
Hello, Alice!
Hello, Alice!
Hello, Alice!
```

Built-in Decorators in Python

- **`@staticmethod`**: Defines a static method inside a class.
- **`@classmethod`**: Defines a class-level method that receives `cls` as the first parameter.
- **`@property`**: Converts a method into a property.

Example:

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

p = Person("Alice")
print(p.name)  # Output: Alice
```

Practical Use Cases

- **Logging**: Decorators can log information before or after a function executes.
- **Access Control**: Restrict access to certain parts of code.
- **Performance Measurement**: Measure execution time of a function.

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Execution Time: {end - start} seconds")
        return result
    return wrapper

@timer
def slow_function():
    time.sleep(2)

slow_function()
```

---

3. Understanding Generators

What is a Generator?

A generator in Python is a function that returns an **iterator** using the `yield` keyword. Generators allow you to iterate over values **one at a time**, making them memory-efficient.

Syntax and Use of `yield`

A generator function contains the `yield` statement, which pauses the function's execution and returns a value to the caller. Execution resumes from where it left off.

Example:

```
def simple_generator():
    yield 1
    yield 2
    yield 3

for value in simple_generator():
    print(value)
```

Output:

```
1
2
3
```

Difference Between Iterators and Generators

Feature	Iterator	Generator
Memory Usage	Stores all values in memory	Produces values on the fly
Syntax	Uses `__iter__` and `__next__`	Uses `yield` statement
Complexity	Explicitly implemented	Simplified syntax

Examples and Use Cases

1. **Simple Generator Function**:

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

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

2. **Infinite Sequence Generator**:

```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1

gen = infinite_sequence()
print(next(gen))  # 0
print(next(gen))  # 1
print(next(gen))  # 2
```

3. **Reading Large Files**: Generators are useful for streaming large data.

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

for line in read_large_file("large_file.txt"):
    print(line)
```

---

4. Combining Decorators and Generators

Generators and decorators can be combined for enhanced functionality.

Example:

```
def generator_decorator(func):
    def wrapper(*args, **kwargs):
        print("Generator is being called")
        yield from func(*args, **kwargs)
    return wrapper

@generator_decorator
def square_numbers(n):
    for i in range(1, n + 1):
        yield i * i

for square in square_numbers(5):
    print(square)
```

Output:

```
Generator is being called
1
```

```
4
9
16
25
```

---

5. Conclusion

Decorators and generators are powerful features in Python that allow for code reusability, memory efficiency, and cleaner syntax. Decorators simplify modifying function behavior, while generators make it easier to handle large data streams. Understanding and combining these concepts can significantly improve the quality and performance of Python programs.

---

6. References

1. Python Documentation: https://docs.python.org/3/
2. "Fluent Python" by Luciano Ramalho
3. Online Tutorials and Resources
