DECORATORS AND GENERATORS IN PYTHON

A Minor Project Report
submitted in partial fulfilment of the requirements for
the award of the degree of

Internship

in

Data Science

By

Name:P. Sri Harsha

Email:srihrshapadala@gmail.com

Under the esteemed guidance of

CHAPTER-1

INTRODUCTION

1.1 Overview of python

Python is a high-level, versatile programming language that has gained immense popularity due to its simplicity, readability, and wide range of applications. Developed by Guido van Rossum and first released in 1991, Python was designed to emphasize code readability and allow programmers to express concepts in fewer lines of code compared to languages such as C++ or Java..

Whether you are developing a web application, analyzing data, or automating tasks, Python provides the tools and flexibility needed to accomplish your goals efficiently. As a result, it continues to be a popular choice among developers, researchers, and educators worldwide.

1.2 PURPOSE OF THE REPORT

The purpose of this report is to provide an in-depth exploration of two advanced features in Python: decorators and generators. These features are instrumental in enhancing the functionality and efficiency of Python programming and are widely used by developers to write cleaner, more efficient, and more maintainable code.

Importance of Decorators and Generators

Understanding decorators and generators is crucial for any Python programmer aiming to write efficient, scalable, and maintainable code. These features are not only used to enhance code functionality but also to improve performance and resource management.

Decorators provide a means to extend and modify behavior without altering existing code, promoting code reuse and separation of concerns.

Generators, on the other hand, offer a memory-efficient way to handle large datasets and perform iterative operations.

By mastering these advanced features, programmers can leverage Python's full potential, creating more robust and efficient applications. This report aims to equip readers with a comprehensive understanding of decorators and generators, empowering them to use these tools effectively in their own Python projects

CHAPTER-2

DECORATER IN PYTHON

DECORATER

In Python, a decorator is a design pattern that allows you to modify the functionality of a function by wrapping it in another function.

The outer function is called the decorator, which takes the original function as an argument and returns a modified version of it.

In fact, any object which implements the special __call__()method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable

Syntax example for Decorators

```
def make_pretty(func):
    # define the inner function
    def inner():
        # add some additional beha
        print("I got decorated")

        # call original function
        func()
    # return the inner function
    return inner

# define ordinary function
def ordinary():
    print("I am ordinary")

# decorate the ordinary function
decorated_func = make_pretty(ordin

# call the decorated function
decorated_func()
```

```
I got decorated
I am ordinary
```

- Here make_pretty () is a decorator and we are passing the ordinary () function as the argument to the make_pretty ()
- The male_pretty () function returns the inner function and it is now assigned to the decorated_func variable

Types of Decorators

1.Function decorator: Function decorators are the most common type of decorators and are used to modify or extend the behavior of functions. They take a function as input and return a new function with added or altered functionality

```
def make_pretty(func):
    # define the inner function
    def inner():
        # add some additional beha
        print("I got decorated")

        # call original function
        func()
    # return the inner function
    return inner

# define ordinary function
def ordinary():
    print("I am ordinary")

# decorate the ordinary function
decorated_func = make_pretty(ordin

# call the decorated function
decorated_func()
```

```
I got decorated
I am ordinary
```

- Here make_pretty () is a decorator and we are passing the ordinary () function as the argument to the make_pretty ()
- The male_pretty () function returns the inner function and it is now assigned to the decorated_func variable

2.MethodDecoder: Method decorators are similar to function decorators but are used to modify methods within classes. They take an instance method as input and return a new method with added or altered functionality

```
def method_decorator(func):
    def wrapper(self, *args, **kwargs):
        print(f"Calling method {func.__name__}")
        return func(self, *args, **kwargs)
    return wrapper

class MyClass:
    @method_decorator
    def my_method(self):
        print("Method called")

obj = MyClass()
obj.my_method()
```

```
Calling method my_method
Method called
```

- The inner function wrapper will replace the original method and prints the message indicating that the method is being called including the name of the method
- When my_method is called on an instance of MyClass, the decorator prints a message before and after the method's execution

3. Class Decorators: Class decorators are used to modify or extend the behavior of entire classes. They take a class as input and return a new class with added or altered functionality.

```
def class_decorator(cls):
    cls.decorated = True
    return cls

@class_decorator
class MyClass:
    pass

print(MyClass.decorated)
```

```
True
```

- The decorator is applied to '**MyClass'** modifying it by adding the '**decorated'** attribute
- When accessing '**MyClass.decorated'**, the value **True** is printed because the class decorator set this attribute
-

4.Built-in Decorators: Python provides several built-in decorators for common use cases, such as:

- Static-method: Used to define a static method that does not require access to the class or instance.
- Class-method: Used to define a class method that receives the class as the first argument.
- property: Used to define a method as a property, which can be accessed like an attribute.

```python
class MyClass:
    def __init__(self, value):
        self._value = value

    @staticmethod
    def static_method():
        print("This is a static method.")

    @classmethod
    def class_method(cls):
        print("This is a class method.")

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, new_value):
        self._value = new_value

obj = MyClass(10)
MyClass.static_method()
MyClass.class_method()
print(obj.value)
obj.value = 20
print(obj.value)
```

```
This is a static method.
This is a class method.
10
```

- Class Definition: MyClass is defined with an initializer to set an instance variable _value.
- Static Method: static_method is a static method that prints a message.
- Class Method: class_method is a class method that prints a message.

CHAPTER 3
GENERATORS IN PYTHON

Generators

In Python, a generator is a **function** that returns an **iterator** that produces a sequence of values when iterated over.

Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.

Syntax for Generator

gen = simple_generator ()

for value in gen:

print(values)

Example code for Generator

```python
def my_generator(n):

    # initialize counter
    value = 0

    # loop until counter is less than n
    while value < n:

        # produce the current value of the counter
        yield value

        # increment the counter
        value += 1

# iterate over the generator object produced by my_generator
for value in my_generator(3):

    # print each value produced by generator
    print(value)
```

```
0
1
2
```

- The **my_generator ()** generator function takes an integer n as an argument and produce a sequence of numbers from **0 to n-1** using while loop
- The **yield** keyword is used to produce a value from the generator and pause the generator function's execution until the next value is requested.
- The **for** loop iterates over the generator object produced by **my_generator(),** and the print statement prints each value produced by the generator.

Create python Generator

In Python similar to defining a normal function, we can define a generator function using the **def** keyword, but instead of the **return** statement we use the **yield** statement

Syntax:

Def generator_name(arg):

#statement

Yield something

Here, the **yield** keyword is used to produce a value from the generator

When the generator function is called, It does not execute the function body immediately. Instead, it returns a generator object that can be iterated over to produce the values.

Use of python Generator

There are several reasons that makes generators a powerful implementation

1.Easy to Implement:

Example to implement a sequence of power of 2 using an iterator class.

```
class PowTwo:
    def __init__(self, max=0):
        self.n = 0
        self.max = max

    def __iter__(self):
        return self

    def __next__(self):
        if self.n > self.max:
            raise StopIteration

        result = 2 ** self.n
        self.n += 1
        return result
```

Let’s do the same using a generator function

```
def PowTwoGen(max=0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1
```

Generator keeps track on details automatically

2. Memory Efficient

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill, if the number of items in the sequence is very large.

Generator implementation of such sequences is memory friendly and is preferred since it only produces one item at a time.

3.Represent Infinite Stream

Generators are excellent mediums to represent an infinite stream of data. Infinite streams cannot be stored in memory, and since generators produce only one item at a time, they can represent an infinite stream of data.

The following generator function can generate all the even numbers (at least in theory).

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2
```

4. Pipelining Generators

Multiple generators can be used to pipeline a series of operations. This is best illustrated using an example. If we want to find out the sum of squares of numbers in the Fibonacci series, we can do it in the following way by pipelining the output of generator functions together.

```python
def fibonacci_numbers(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x

def square(nums):
    for num in nums:
        yield num**2

print(sum(square(fibonacci_numbers(10))))

# Output: 4895
```

Python Generator Expression

In Python, a generator expression is a concise way to create a generator object It is similar to a list comprehension , but instead of creating a list, it creates a generator object that can be iterated over to produce the value in the generator

Syntax

(expression for item in iterable)

- **Expression** is a value that will be returned for each item in the **iterable**
- The generator expression creates a generator object that produces the values of **expression** for each item in the **iterable,** one at a time when iterated over

Example

```
# create the generator object
squares_generator = (i * i for i in range(5))

# iterate over the generator and print the values
for i in squares_generator:
    print(i)
```

```
0
1
4
9
16
```

- Here we have created the generator object that will produce the squares of the number **0** through **4** when iterated over

- To iterated over the generator and get the values,we have used the **for** loop

Importance of decorator and generator in python

GENERATOR

- Generators are memory-efficient as they generate items on-the-fly and do not store the entire sequence in memory.
- They implement lazy evaluation, which means values are computed only when needed, reducing unnecessary calculations and memory usage.
- Generators simplify code for iterating over complex data structures or infinite sequences by abstracting the iteration logic

DECORATOR

- Decorators enable code reuse by encapsulating common functionality that can be applied to multiple functions or methods without repeating code.
- They allow the separation of core business logic from auxiliary concerns like logging and validation, leading to cleaner and more maintainable code.
- By using decorators, the code becomes more readable as the core functionality is not cluttered with repetitive tasks.

Conclusion:

In summary we can conclude that Decorators and generators are two advanced features in Python that significantly enhance its functionality and efficiency, enabling developers to write cleaner, more efficient, and maintainable code.

Decorators provide a powerful and flexible way to extend and modify the behavior of functions, methods, and classes, promoting code reusability, separation of concerns, and improved readability. They allow developers to encapsulate common functionalities such as logging, access control, and validation, making code more modular and maintainable.

On the other hand, generators offer a memory-efficient approach to iterating over large datasets or creating infinite sequences by generating items on-the-fly and maintaining their state between yields.

This lazy evaluation leads to reduced memory usage and improved performance, especially when dealing with large streams of data.

Together, decorators and generators empower Python developers to write code that is not only efficient and performant but also clean, modular, and easy to maintain, addressing both structural and performance-related concerns effectively.