MINOR PROJECT ON

Decorators and Generators in Python

INTRODUCTION

Python is a popular, high-level programming language known for its clear syntax and readability. Two of its advanced features, decorators and generators, provide powerful tools for extending and optimizing code functionality. This report explores these concepts, explaining how they work and their practical applications.

Decorators in Python

Decorators are a flexible feature that allow you to modify or extend the behaviour of functions or methods without changing their actual code. They are often used for tasks like logging, enforcing access control, and caching.

What Are Decorators?

A decorator is a function that takes another function as its argument and returns a new function. This returned function usually enhances or modifies the behavior of the original function. Decorators help adhere to the"don't repeat yourself" (DRY) principle by allowing you to reuse code across multiple functions.

Basic Decorator Example

python

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()

In this example, `my\_decorator` wraps the `say\_hello` function, adding behavior before and after the original function runs. The `@my\_decorator` syntax is a shorthand for applying the decorator to the function.

How Do Decorators Work?

Decorators work by using closures, which allow a function to access variables from its enclosing scope. When you apply a decorator, it returns a new function that has access to both the original function and any additional arguments or state maintained by the decorator.

Types of Decorators

1. Function Decorators: The most common type, used to modify the behavior of functions.

2. Class Decorators: Used to modify or enhance the behavior of classes.

Common Uses of Decorators

1. Logging: To track function calls and data changes.

python

def log(func):

def wrapper(\*args, \*\*kwargs):

print(f"Calling {func.\_\_name\_\_} with {args} and {kwargs}")

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

return wrapper

2. Authorization: To check if a user has the right to access a particular function.

python

def requires\_permission(func):

def wrapper(user, \*args, \*\*kwargs):

if not user.has\_permission:

raise PermissionError("Unauthorized access")

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

return wrapper

3. Memoization: To cache results of expensive function calls.

python

def memoize(func):

cache = {}

def wrapper(\*args):

if args in cache:

return cache[args]

result = func(\*args)

cache[args] = result

return result

return wrapper

4. Validation: To check inputs before passing them to the function.

python

def validate\_input(func):

def wrapper(x, y):

if x < 0 or y < 0:

raise ValueError("Inputs must be non-negative")

return func(x, y)

return wrapper

Generators in Python

Generators are a type of iterable, like lists or tuples, but they don't store all their values in memory. Instead, they generate values on the fly, which makes them memory efficient, especially for handling large datasets.

What Are Generators?

Generators are functions that use the `yield` keyword to return values one at a time, preserving their state between each yield. Unlike normal functions that terminate with a `return` statement,

generators can pause their execution and resume from the same point.

Basic Generator Example

python

def simple\_generator():

yield 1

yield 2

yield 3

for value in simple\_generator():

print(value)

This will output:

1

2

3

How Do Generators Work?

When a generator function is called, it returns a generator object without starting execution immediately. Each call to `next()` on the generator object resumes the function from where it left off, until it encounters another `yield` statement or raises a `StopIteration` exception.

Generator Expressions

Generator expressions provide a concise way to create generators. They are similar to list comprehensions but use parentheses instead of square brackets.

Example:

python

gen = (x\*x for x in range(5))

for val in gen:

print(val)

This will output:

0

1

4

9

16

Common Uses of Generators

1. Lazy Evaluation: Generators compute values on demand, which is useful for large datasets or streams of data.

2. Pipelining: Generators can be chained together, passing data from one generator to the next.

python

def read\_lines(file):

with open(file) as f:

for line in f:

yield line.strip()

def filter\_lines(lines, keyword):

for line in lines:

if keyword in line:

yield line

3. Memory Efficiency: Because they only produce one item at a time, generators are very memory efficient.

python

def infinite\_sequence():

num = 0

while True:

yield num

num += 1

Differences Between Decorators and Generators

Decorators and generators serve different purposes in Python:

1. Decorators:

- Used to modify or enhance the behavior of functions or methods.

- Commonly applied to cross-cutting concerns like logging, authorization, and memoization.

- Implemented using higher-order functions and closures.

2. Generators:

- Used to produce values one at a time using `yield`.

- Ideal for working with large datasets or implementing lazy evaluation.

- Efficient in memory usage as they do not store all values at once.

Conclusion

Decorators and generators are essential features in Python, offering ways to extend functionality and optimize performance. Decorators provide a clean and reusable way to modify functions or methods, while generators offer an efficient method for handling large datasets and implementing lazy evaluation. Mastering these features can lead to more efficient and maintainable Python code,making them invaluable tools in any Python developer's toolkit.