# REPORT ON DECORATORS AND GENERATORS

## INTRODUCTION: -

Python is known for its simplicity and versatility, giving developers the tools they need to produce clean, efficient, and reusable code. There are many such features in Python, and decorators and generators are one of them, which provide advanced concepts to the Python developers. This document discusses both of those capabilities in detail: usage, syntax, benefits, and best practices.

## DECORATORS

### What Are Decorators?

In Python, decorators are a design pattern that lets you add new functionality to an existing object without modifying its structure. They can be thought of as wrappers that give extended functionality to an existing function, which makes them very helpful for following the DRY (Don't Repeat Yourself) principle. How Decorators Work Simply put, a decorator is a function that adds functionality to an existing function, object, or classes. Applying a decorator takes the original function and replaces it with the wrapper function, which wraps the additional functionality.

### Syntax of Decorators

Decorator Process with Improved Usage Decorators are used in a much simpler way with @decorator_name.

Here is a basic example:

```
# Defining a decorator
def my_decorator(func):
def wrapper():
print("Something happens before the function is called.")
func()
print("Something happens after the function is called.")
return wrapper

# Using the decorator
@my_decorator
def say_hello():
```

```
print("Hello, world!")

say_hello()
```

In this example, my decorator function wraps additional behaviour around the say hello function.

## TYPES OF DECORATORS

These decorators are special functions or callable objects that modify the behaviour of other functions, methods, or classes without permanently changing their source code.

Here's an in-depth explanation of the main types of decorators:

1. **Function Decorators:** Alter the behaviour of functions.
2. **Method decorators extend:** extend or alter class methods.
3. **Class Decorators:** Modify the behaviour of whole classes.

## 1. Function Decorators

It is applied to standalone functions that alter or enhance their behaviour.

**Common Use Cases:**

- Logging function calls
- Measuring execution time
- Adding pre/post-processing steps
- Validating inputs and outputs

**Example: -**

```
def log_decorator(func):
def wrapper(*args, **kwargs):
print(f"Calling function: {func.__name__}")
result = func(*args, **kwargs)
print(f"Function {func.__name__} completed.")
return result
return wrapper

@log_decorator
```

```
def greet(name):
print(f"Hello, {name}!")

greet("Alice")
```

## 2. Method Decorators

decorators are mainly applied to class methods in order to alter or enrich their behaviour.

**Most Common Applications:**

- Adding supplementary functionality to methods of instances or methods of a class
- Controlling access, for example, checking user roles
- Logging or profiling methods

**Example: -**

```
def access_control(role_required):
def decorator(func):
def wrapper(self, *args, **kwargs):
if self.role != role_required:
print("Access Denied")
return
return func(self, *args, **kwargs)
return wrapper
return decorator

class User:
def __init__(self, name, role):
self.name = name
self.role = role

@access_control("admin")
def view_admin_panel(self):
print("Welcome to the admin panel.")
user = User("Bob", "user")
admin = User("Alice", "admin")
user.view_admin_panel() # Access Denied
admin.view_admin_panel() # Welcome to the admin panel.
```

## 3. Class Decorators

These decorators operate on entire classes and are often used to modify or enhance class behaviour as a whole.

**Common Use Cases:**

- Adding attributes or methods dynamically
- Validating data during class initialization
- Implementing singleton patterns or similar design patterns

**Example: -**

```
def add_methods(cls):
cls.new_method = lambda self: print("This is a new method.")
return cls

@add_methods
class MyClass:
def __init__(self):
print("MyClass initialized.")

obj = MyClass()
obj.new_method() # This is a new method.
```

## BUILT-IN DECORATORS IN PYTHON

Python has several built-in decorators that make common patterns easier to express:

**@staticmethod**: A static method within a class that does not depend on instance variables.

**@classmethod**: A method bound to the class; it allows access to class variables.

**@property**: Used to create managed attributes; it enables getter, setter, and delete methods.

**Use Cases of Decorators:-**

1. **Logging:** Automatically log function calls and their outputs.
2. **Access Control:** Restrict or grant access based on user roles.
3. **Memorization:** cache the results of expensive computations to improve performance.

4. **Input validation:** Validate arguments passed to functions.
5. **Performance Measurement:** Measure execution time of functions for profiling.
6. **Authorization:** Implement role-based access control.

## ADVANCED TOPICS IN DECORATORS

### 1. Decorators with Arguments:

A decorator can accept arguments by nesting functions.

**For example:**

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

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

### 2. Chaining Decorators:

More than one decorator can be applied to a single function

**For example:**

```
def bold(func):
def wrapper():
return f"<b>{func()}</b>"
return wrapper

def italic(func):
def wrapper():
return f"<i>{func()}</i>"
return wrapper

@bold
@italic
```

```
def text():
return "Decorated Text"
print(text())
```

### 3. Class decorators:

It is used to modify the behavior of classes

**For example:**

```
def add_method(cls):
cls.new_method = lambda self: "New method added!"
return cls
@add_method
class SampleClass:
pass
obj = SampleClass()
print(obj.new_method())
```

## GENERATORS

### What Are Generators?

Generators are one of the coolest iterable types from Python, specialized for producing some sequence of values lazily instead of storing their elements in the memory, whereas lists store all in memory. Thereby, each item is outputted one at a time; they are memory-safe.

### How Work Generators?

Generators actually use the `yield` keyword to return one value and freeze their execution during the process. From where it actually left off each time it resumed, the memory state is well preserved.

### Syntax of Generators

Generators can be implemented using generator functions or generator expressions.

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

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

Generator expressions provide a concise way to create generators:

```
squares = (x**2 for x in range(5))
for square in squares:
print(square)
```

## ADVANTAGES OF GENERATORS

**1. Memory Efficiency**: Produce values on demand instead of loading the whole sequence into memory.
**2. Lazy Evaluation:** Helpful with huge datasets or infinite streams.
**3. Composable:** May be chained together to create complex pipelines by lots of generators.
**4. Parallel Processing:** breaking down tasks into smaller bits and executing them one after another.
Use Cases of Generators
**5.Stream Processing:** Process data streams efficiently, perhaps logs or network data.
**6.File Reading:** read large files line by line without loading a whole file in memory.
**7.Infinite Sequences:** generate Fibonacci numbers or prime numbers.
**8.Data Transformation Pipelines:** apply a series of transformations on data streams.

### Example: Fibonacci Generator

```
def fibonacci():
a, b = 0, 1
while True:
yield a
a, b = b, a + b

fib = fibonacci()
for _ in range(10):
print(next(fib))
```

## Comparison: Decorators vs. Generators

| USE CASE | PURPOSE | EXAMPLE |
|---|---|---|
| Logging | Track function calls and their outputs | Debugging and analytics |
| Access Control | Restrict access based on conditions or user roles | Role-based functionality |
| Memorization | Cache results of functions to optimize performance | Recursive computations like Fibonacci sequence |
| Input Validation | Ensure input arguments meet certain criteria | Avoid runtime errors from invalid input |
| Performance Measurement | Profile the execution time of a function | Identify slow parts of code |
| Authorization | Enforce security by limiting access to certain users | Protect sensitive operations |

## CONCLUSION: -

Decorators and generators are two of the most important tools in Python to write succinct, efficient, and beautiful code. Decorators facilitate function modification by wrapping additional behaviour in it, and generators give a memory-friendly approach for handling great datasets or streams. The more one learns about using these tools, the more a developer's ability to write strong and scalable Python applications increases.

Including decorators and generators in daily programming would bring out advanced techniques to produce reusable, maintainable, and performance-optimized software solutions.