Decorators, Generators, and Iterators in Python

1. Decorators in Python

Decorators are functions that modify the behavior of another function or method. They are used for a variety of purposes like logging, access control, and memoization.

How They Work:

A decorator is applied to a function using the @decorator_name syntax. It wraps the original function and can modify its behavior or return value.

Example:

```python
# A basic decorator
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.
```

Using Arguments with Decorators:

```python
def greet_decorator(func):
    def wrapper(name):
        print("Welcome!")
        func(name)
        print("Goodbye!")
    return wrapper

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

greet("Rishabh")
```

Output:

```
Welcome!
Hello, Rishabh!
Goodbye!
```

Built-in Decorators:

- @staticmethod
- @classmethod
- @property

2. Generators in Python

Generators are a type of iterable, like lists or tuples, but instead of returning all their values at once, they yield values one at a time, pausing and resuming their state between calls. They are memory-efficient and used for large datasets.

How to Create Generators:

A generator is a function that uses the yield statement instead of return.

Example:

```python
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

Use Cases of Generators:

- Reading large files line by line.
- Infinite sequences like Fibonacci numbers.

Fibonacci Generator Example:

```python
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

for num in fibonacci(5):
    print(num)
```

Output:

```
0
1
1
2
3
```

3. Iterators in Python

An iterator is an object that contains a sequence of values and provides a way to iterate over them using the __iter__() and __next__() methods.

Creating an Iterator:

```python
class MyIterator:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self
    def __next__(self):
        if self.current >= self.end:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

it = MyIterator(1, 5)
for num in it:
    print(num)
```

Output:

```
1
2
3
4
```

Using Built-in Iterators:

You can make any iterable (like lists, strings, or tuples) into an iterator using the iter() function.

```python
my_list = [10, 20, 30]
it = iter(my_list)
```

```
print(next(it))  # Output: 10
print(next(it))  # Output: 20
print(next(it))  # Output: 30
```

Difference Between Iterators and Generators:

Feature	**Iterator**	**Generator**
Definition	An object with `__iter__()` and `__next__()`.	A function that uses `yield`.
State	Maintains its own state explicitly.	Maintains state automatically.
Syntax	Requires custom implementation.	Simple and concise with `yield`.
Memory Usage	May use more memory.	Memory-efficient as values are produced on demand.

Combining Them:

Decorators, generators, and iterators can work together for powerful, flexible functionality. For example, a decorator can wrap a generator to add logging.

```python
def log_decorator(gen_func):
    def wrapper(*args, **kwargs):
        print(f"Calling generator: {gen_func.__name__}")
        return gen_func(*args, **kwargs)
    return wrapper


@log_decorator
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1


for num in count_up_to(3):
    print(num)
```

Output:

```
Calling generator: count_up_to
1
2
3
```