**Generators:**

Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value. Each time the yield statement is executed the function generates a new value. So a generator is also an iterator. When a generator function is called, it returns a generator object without even beginning execution of the function. When next method is called for the first time, the function starts executing until it reaches yield statement. The yielded value is returned by the next call.

**Generator Expressions:**

Generator Expressions are generator version of list comprehensions. They look like list comprehensions, but returns a generator back instead of a list.

**Example:**

def simpleGeneratorFun():

yield 1

yield 2

yield 3

# x is a generator object

x = simpleGeneratorFun()

# Iterating over the generator object using next

print(next(x))

print(next(x))

print(next(x))

**Exanple:2**

def square\_generator(n):

for i in range(n):

yield i \*\* 2

# Using the generator

n\_values = 5

squares = square\_generator(n\_values)

# Iterating over the generator

for square in squares:

print(square)

**Note:**In Python, yield is used as a keyword in the context of generators, and it is not a function that you call like yield().

When you use yield in a function, it turns that function into a generator. The yield statement is used to produce a value from the generator and temporarily suspends the function's state. The next time the generator is called, it resumes from where it was suspended.

**Decorators :**

Decorators allow you to make simple modifications to callable objects like functions, methods, or classes. a decorator takes in a function, adds some functionality and returns it.

A decorator is the name used for a software design pattern. Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated.

We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. Doecorator acts as a wrapper for the original function

**Use of Decorators**

1. Avoid code duplication

2. Cluttering main logic of function with additional functional it

**Example:**

def make\_pretty(func):

def inner():

print("I got decorated")

func()

return inner

@make\_pretty

def ordinary():

print("I am ordinary")

**Example:2**

def ensure\_positive\_integer(func):

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

result = func(\*args, \*\*kwargs)

if isinstance(result, int) and result > 0:

return result

else:

print("Error: Result is not a positive integer. Setting result to 1.")

return 1

return wrapper

@ensure\_positive\_integer

def multiply(a, b):

return a \* b

@ensure\_positive\_integer

def subtract(x, y):

return x - y

# Calling decorated functions

result\_multiply = multiply(5, 3)

result\_subtract = subtract(10, 7)

print("Result of multiply:", result\_multiply)

print("Result of subtract:", result\_subtract)

**Note:::::**The line def wrapper(\*args, \*\*kwargs): defines a function named wrapper that takes any number of positional and keyword arguments. This kind of function signature using \*args and \*\*kwargs is often used in decorators to create a generic wrapper that can be applied to functions with different sets of parameters.

Here's a breakdown of the syntax:

* \*args: Collects any number of positional arguments into a tuple named args. This allows the decorator to accept functions with different numbers of positional arguments.
* \*\*kwargs: Collects any number of keyword arguments into a dictionary named kwargs. This allows the decorator to accept functions with different sets of keyword arguments.

So, when you see def wrapper(\*args, \*\*kwargs): in the context of a decorator, it means that wrapper is a generic wrapper that can be applied to various functions, regardless of their argument signatures.

**Decorators in Python (Function having arguments & return statement)**

A decorator in Python is a design pattern that allows the modification or extension of the behavior of functions or methods without changing their actual code. Decorators are applied to functions using the **@decorator\_name** syntax, and they wrap the original function with additional functionality.

**Basic Example:**

def my\_decorator(func):

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

# Additional code to be executed before the original function

print("Something is happening before the function is called.")

# Call the original function

result = func(\*args, \*\*kwargs)

# Additional code to be executed after the original function

print("Something is happening after the function is called.")

return result

return wrapper

# Applying the decorator to a function

@my\_decorator

def my\_function():

print("I am the original function.")

# Calling the decorated function

my\_function()

**Explanation**

**Decorator Definition:**

* + The my\_decorator function is a decorator that takes another function (func) as its argument.
  + It defines a nested function called wrapper that wraps the original function with additional behavior.
  + The wrapper function accepts any number of positional and keyword arguments (\*args and \*\*kwargs).
  + It prints a message before calling the original function, calls the original function (func), and then prints a message after the original function is called.
  + Finally, it returns the result of the original function.

1. **Applying the Decorator:**
   * The @my\_decorator syntax is used to apply the my\_decorator to the my\_function.
   * This is equivalent to my\_function = my\_decorator(my\_function). The decorator is essentially wrapping the original function.
2. **Calling the Decorated Function:**
   * When my\_function() is called, it is the decorated version (wrapper) that gets executed.
   * The decorated function prints messages before and after the original function is called.

**Recursion**

Recursion is a programming and mathematical concept where a function calls itself directly or indirectly in order to solve a particular problem. In other words, a recursive function is a function that performs a task in part and delegates the remaining task to itself.

**Example:1**

def factorial(n):

# Base case: factorial of 0 or 1 is 1

if n == 0 or n == 1:

return 1

# Recursive case: n! = n \* (n-1)!

else:

return n \* factorial(n-1)

# Example usage:

result = factorial(5)

print("Factorial of 5 is:", result)

**Example:2**

def fibonacci(n):

# Base case: fib(0) = 0, fib(1) = 1

if n == 0:

return 0

elif n == 1:

return 1

# Recursive case: fib(n) = fib(n-1) + fib(n-2)

else:

return fibonacci(n-1) + fibonacci(n-2)

# Example usage:

result = fibonacci(6)

print("Fibonacci number at position 6 is:", result)