---

1. What is a Decorator?

A **decorator** is a function that **takes another function as input and returns a new function** with enhanced behavior.

In simple terms: **You "wrap" a function to add extra functionality to it.**

---

2. Basic Decorator Example

```
def my_decorator(func):
    def wrapper():
        print("Before the function call")
        func()
        print("After the function call")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Before the function call
Hello!
After the function call
```

---

3. What does `@my_decorator` mean?

This:

```
@my_decorator
def say_hello():
    ...
```

Is the same as:

```
say_hello = my_decorator(say_hello)
```

---

4. Decorator with Arguments

You can pass arguments to the function being decorated:

```
def decorator(func):
    def wrapper(name):
        print("Before function")
        func(name)
        print("After function")
    return wrapper

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

greet("Alice")
```

---

5. Using `*args` and `**kwargs` in Decorators

To make your decorator work with **any number of arguments**:

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

@debug
def add(a, b):
    return a + b

print(add(3, 4))  # Output: Calling add with (3, 4) {}
                  #         7
```

---

6. Built-in Decorators

Python includes several built-in decorators:

Decorator	Use Case
`@staticmethod`	Declares a static method in a class
`@classmethod`	Declares a class method
`@property`	Turns a method into a read-only property

Example: `@staticmethod`

```
class Math:
    @staticmethod
    def square(x):
        return x * x

print(Math.square(5))  # 25
```

---

7. Chaining Multiple Decorators

You can stack multiple decorators:

```
def bold(func):
    def wrapper():
        return "<b>" + func() + "</b>"
    return wrapper

def italic(func):
    def wrapper():
        return "<i>" + func() + "</i>"
    return wrapper

@bold
@italic
def text():
    return "Hello"

print(text())  # <b><i>Hello</i></b>
```

---

8. `functools.wraps` (Preserve Metadata)

Without it, the decorated function loses its name and docstring.

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper
```

Summary

Concept	Meaning
`@decorator`	Shorthand for `func = decorator(func)`
`*args, **kwargs`	Let decorator handle any kind of function
`functools.wraps`	Preserves original function metadata
Built-in	`@staticmethod`, `@classmethod`, `@property`

---

1. What is a Generator?

A **generator** is a special type of **iterator** that **yields items one by one** instead of returning them all at once. They are defined like functions but use `yield` instead of `return`.

---

2. Creating a Simple Generator

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

gen = count_up_to(5)
for num in gen:
    print(num)
```

Output:

```
1
2
3
4
5
```

---

3. `yield` vs `return`

`return`	`yield`
Ends the function	Pauses the function
Returns one value	Returns a generator object

`return`	`yield`
Does not remember state	Remembers where it left off

4. Generator Object

You can get values from a generator manually using `next()`:

```
gen = count_up_to(3)
print(next(gen))  # 1
print(next(gen))  # 2
```

---

5. Generator Expression (like list comprehension)

```
gen = (x * x for x in range(5))
print(next(gen))  # 0
print(next(gen))  # 1
```

Same as:

```
def gen_squares():
    for x in range(5):
        yield x * x
```

---

6. Why Use Generators?

 Memory efficient – Doesn't store all values in memory. **Lazy evaluation** – Generates values on the fly. **Faster for large data** – Especially in loops and pipelines.

---

7. Use Case: Reading Large Files

```
def read_lines(file):
    with open(file, 'r') as f:
        for line in f:
            yield line.strip()
```

Summary

Concept	Explanation
`yield`	Returns a value and pauses the

Concept	Explanation
	function
Generator	Produces items one at a time (lazy)
`next()`	Moves to the next value
Generator Expr	`(x for x in range(5))`

What is a Context Manager?

A **context manager** in Python is an object that defines runtime **setup and cleanup** actions using:

- `__enter__()` → Code to run *before* the block
- `__exit__()` → Code to run *after* the block (even if there's an error)

The most common example:

```
with open("file.txt", "r") as file:
    data = file.read()
```

No need to explicitly call `file.close()` — it's handled automatically.

---

1. Behind the Scenes

This:

```
with open("file.txt") as f:
    data = f.read()
```

Is similar to:

```
f = open("file.txt")
try:
    data = f.read()
finally:
    f.close()
```

---

2. Creating a Custom Context Manager (Using Class)

```
class MyManager:
    def __enter__(self):
        print("Entering context...")
        return "Resource Ready"
```

```
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Exiting context...")

with MyManager() as val:
    print(val)
```

Output:

```
Entering context...
Resource Ready
Exiting context...
```

---

3. Using `contextlib` Module (Simpler Way)

```
from contextlib import contextmanager

@contextmanager
def my_context():
    print("Start")
    yield "Hello"
    print("End")

with my_context() as msg:
    print(msg)
```

Why Use Context Managers?

Use Case	Benefit
File handling	Automatically closes files
Database connection	Auto commit/rollback
Thread locks	Ensures proper locking
Resource management	Clean & safe code

4. Another Real Example – File Writer

```
@contextmanager
def write_to_file(filename):
    f = open(filename, 'w')
    try:
        yield f
    finally:
        f.close()
```

```
with write_to_file("demo.txt") as f:
    f.write("Hello, world!")
```

Summary

Component	Purpose
`with`	Opens and closes resources automatically
`__enter__()`	Sets up resource
`__exit__()`	Cleans up resource
`contextlib`	Create context managers easily

What is a Regular Expression?

A **regular expression** (regex) is a **sequence of characters** that defines a **search pattern**. It's useful for validating emails, extracting numbers, finding patterns, etc.

---

1. Importing the `re` Module

```
import re
```

---

2. Basic Functions in `re`

Function	Description
`re.match()`	Matches pattern at the **beginning**
`re.search()`	Searches pattern **anywhere** in string
`re.findall()`	Returns **all matches** as a list
`re.sub()`	Replaces matches with another string
`re.split()`	Splits string by a regex pattern

3. Basic Regex Patterns

Pattern	Meaning	Example Match
`.`	Any character (except newline)	`a.c` matches `abc`

Pattern	Meaning	Example Match		
`^`	Start of string	`^Hello` matches `Hello world`		
`$`	End of string	`world$` matches `Hello world`		
`*`	0 or more	`a*` matches `aaa` or `""`		
`+`	1 or more	`a+` matches `a`, `aa`		
`?`	0 or 1	`a?` matches `a` or `""`		
`[]`	Set of characters	`[a-z]` matches any lowercase letter		
`\d`	Digit (0–9)	`\d` matches `5`		
`\w`	Word character (a-z, A-Z, 0-9, _)	`\w+` matches `hello_123`		
`\s`	Whitespace	`\s` matches space, tab, etc.		
`|`	`|`	OR	`cat`	`dogmatchescatordog|`
`()`	Group	`(\d+)-(\d+)` matches `123-456`		

4. Examples

`re.match()` – Start of string

```
import re

result = re.match(r"Hello", "Hello world")
print(result.group())  # Hello
```

`re.search()` – Anywhere in string

```
result = re.search(r"world", "Hello world")
print(result.group())  # world
```

`re.findall()` – Find all matches

```
result = re.findall(r"\d+", "My numbers are 123 and 456")
print(result)  # ['123', '456']
```

`re.sub()` – Replace matches

```
result = re.sub(r"\d+", "X", "ID 123, Code 456")
print(result)  # ID X, Code X
```

---

5. Grouping with `()` and `groups()`

```
match = re.search(r"(\d+)-(\d+)", "Phone: 123-456")
print(match.group(0))  # 123-456
print(match.group(1))  # 123
print(match.group(2))  # 456
```

Summary

Function	Use
`re.match`	Match from start
`re.search`	Match anywhere
`re.findall`	List of all matches
`re.sub`	Substitute
`re.split`	Split by regex

Tip

Always use **raw strings** for regex:

```
r"\d+"   # Correct
"\\d+"   # Less readable
```

We'll briefly cover:

1. **NumPy** – Numerical computing
2. **Pandas** – Data analysis
3. **Matplotlib** – Plotting and visualization
4. **Other Mentionable Libraries** (Scikit-learn, Requests, etc.)

---

1. **NumPy (Numerical Python)**

Used for: Working with arrays, matrices, and numerical operations.

```
import numpy as np

arr = np.array([1, 2, 3])
print(arr + 5)          # [6 7 8]
print(np.mean(arr))     # 2.0
```

Key Features:

- Fast operations on large arrays
- Broadcasting
- Linear algebra, FFT, and more

---

2. **Pandas**

Used for: Data manipulation, analysis, and tabular data handling.

```
import pandas as pd

data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)

print(df.head())      # View top rows
print(df['Age'].mean())  # 27.5
```

Key Objects:

- `Series`: 1D labeled array
- `DataFrame`: 2D labeled data structure (like Excel)

---

3. **Matplotlib**

Used for: Creating plots and charts.

```
import matplotlib.pyplot as plt

x = [1, 2, 3]
y = [4, 5, 6]

plt.plot(x, y)
plt.title("Simple Line Plot")
plt.show()
```

---

4. **Seaborn** (built on Matplotlib)

Used for: High-level visualizations, like box plots, heatmaps, etc.

```
import seaborn as sns
import pandas as pd

tips = sns.load_dataset("tips")
```

```
sns.boxplot(x="day", y="total_bill", data=tips)
plt.show()
```

---

5. **Scikit-learn**

Used for: Machine learning (classification, regression, clustering).

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

---

6. **Other Useful Libraries**

Library	Purpose
`requests`	HTTP requests (API calls)
`beautifulsoup4`	Web scraping
`openpyxl`	Reading/writing Excel files
`os` / `shutil`	File system operations
`math`	Math functions like `sqrt`, `log`
`random`	Random number generation

Summary Table

Library	Purpose
NumPy	Fast array and math operations
Pandas	DataFrames and data analysis
Matplotlib	Basic plotting
Seaborn	Statistical visualization
Scikit-learn	Machine learning models
Requests	API calls, HTTP requests