**Python Generators and Decorators — Complete Notes**

**1. Generators with yield and next()**

A **generator** is a special function that returns values **one at a time** using yield.  
Unlike normal functions, generators don’t return all data at once — they **pause** at yield and resume when called again.

def get\_chai\_gen():

    yield "cup 1"

    yield "cup 2"

    yield "cup 3"

chai = get\_chai\_gen()

print(next(chai))  # cup 1

print(next(chai))  # cup 2

print(next(chai))  # cup 3

# next(chai)  # StopIteration error

✅ **Key Points**:

* yield → returns value and pauses function.
* next() → resumes generator from last pause.
* Once finished → raises StopIteration.

👉 **Use Case**: Generators are useful when working with **large data (files, streams, logs)** where loading everything at once is costly.

**2. Infinite Generators**

A generator can be made **infinite** by using while True.

def infinite\_chai():

    count = 1

    while True:

        yield f"Refill #{count}"

        count += 1

refill = infinite\_chai()

for \_ in range(3):

    print(next(refill))

Output:

Refill #1

Refill #2

Refill #3

✅ **Key Points**:

* No memory issues (values are generated on-demand).
* Must be **controlled** (like using break or limited loop).

👉 **Use Case**: Endless counters, data streams, random number streams, server logs.

**3. Sending Values to Generators (send)**

With send(), you can **send values into a generator**.

def chai\_customer():

    print("! What chai would you like?")

    order = yield

    while True:

        print(f"Preparing: {order}")

        order = yield

stall = chai\_customer()

next(stall)  # Start generator

stall.send("Masala chai")

stall.send("Green chai")

# Output:

# ! What chai would you like?

# Preparing: Masala chai

# Preparing: Green chai

✅ **Key Points**:

* First next() is required to start.
* send(value) sends data into generator.
* Useful for **coroutines** (two-way communication).

👉 **Use Case**: Async pipelines, task scheduling, message passing.

**4. yield from & Closing Generators**

**🔹 yield from**

Instead of writing multiple loops, yield from allows one generator to delegate to another.

def local\_chai():

    yield "lemon chai"

    yield "masala chai"

def imported\_chai():

    yield "Matcha"

    yield "Oolong"

def full\_menu():

    yield from local\_chai()

    yield from imported\_chai()

for chai in full\_menu():

    print(chai)

Output:

lemon chai

masala chai

Matcha

Oolong

👉 **Use Case**: Combining multiple generators easily.

**🔹 Closing a Generator (close)**

def chai\_stall():

    try:

        while True:

            order = yield "Waiting for chai Order"

    except:

        print("Stall closed. No more chai.")

stall = chai\_stall()

print(next(stall))   # Waiting for chai Order

stall.close()        # Gracefully stops generator

✅ **Key Points**:

* close() → stops generator safely.
* Good for resource cleanup (like closing DB connections).

**5. Basic Decorators**

A **decorator** is a function that takes another function and adds **extra behavior** without changing original code.

from functools import wraps

def my\_decorators(func):

    @wraps(func)

    def wrapper():

        print("Before function run")

        func()

        print("After function run")

    return wrapper

@my\_decorators

def greet():

    print("Hello from decorator class!")

greet()

print(greet.\_\_name\_\_)  # greet (not wrapper)

Output:

Before function run

Hello from decorator class!

After function run

greet

✅ **Key Points**:

* @decorator syntax is shortcut for func = decorator(func).
* wraps preserves original function name & docs.

👉 **Use Case**: Logging, authentication, caching, timing.

**6. Logger with Decorators**

from functools import wraps

def log\_activity(func):

    @wraps(func)

    def wrapper(\*args, \*\*kwargs):

        print(f"Calling: {func.\_\_name\_\_}")

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

        print(f"✅ Finished: {func.\_\_name\_\_}")

        return result

    return wrapper

@log\_activity

def brew\_chai(type, milk="no"):

    print(f"Brewing {type} Chai, milk: {milk}")

brew\_chai("Masala Chai")

Output:

Calling: brew\_chai

Brewing Masala Chai, milk: no

✅ Finished: brew\_chai

👉 **Use Case**: Debugging, tracking function calls.

**7. Authorization Decorator**

from functools import wraps

def require\_admin(func):

    @wraps(func)

    def wrapper(user\_role):

        if user\_role != "admin":

            print("Access Denied! Admin only")

            return None

        else:

            return func(user\_role)

    return wrapper

@require\_admin

def access\_tea\_inventory(role):

    print("Access granted to tea inventory")

access\_tea\_inventory("admin")  # ✅ granted

access\_tea\_inventory("user")   # ❌ denied

Output:

Access granted to tea inventory

Access Denied! Admin only

👉 **Use Case**: Role-based access control (RBAC) in web apps.

**📌 Summary (Cheat Sheet)**

**Generators**

* yield → pauses & returns value.
* next(gen) → resumes generator.
* Infinite generators → use while True.
* send(value) → send data to generator.
* yield from → delegate to another generator.
* close() → stop generator safely.

**Decorators**

* Add extra behavior to functions without modifying them.
* @wraps → keeps function metadata.
* Types:
  + **Basic**: run code before/after function.
  + **Logger**: track function calls.
  + **Authorization**: role-based restrictions.
  + **Others**: caching, timing, validation.

👉 So basically:

* **Generators** = **produce data on-demand**.
* **Decorators** = **modify functions easily**.