Unit 7 – Advanced Python Features

Python provides **powerful, flexible, and elegant features** that go beyond basics. These include **decorators, closures, context managers, itertools, type hints, enums, namedtuples, and concurrency considerations (GIL & threading)**.

7.1: Decorators (Function and Class Decorators)

1. Definition / Concept

- A **decorator** is a function that **modifies or extends another function or class** without changing its actual code.
- They use the @decorator_name syntax.
- Commonly used for **logging, authentication, caching, measuring execution time, access control**.

📌 Two main types:

- **Function decorators** → Modify functions.
- **Class decorators** → Modify classes.

2. Analogy / Real-Life Connection

Think of **decorators** like **gift wrapping**:

- The **gift (function)** remains the same.
- The **wrapper (decorator)** adds something extra (beauty, protection) before handing it over.

3. Syntax

```
def decorator(func):
    def wrapper():
        # extra code
        return func()
    return wrapper

@decorator
def my_function():
    ...
```

4. Step-by-Step Explanation

1. A decorator is a function that **takes another function as input**.
2. Inside, it defines a wrapper function to add extra behavior.
3. Returns the wrapper instead of the original function.
4. Using @decorator is just **syntactic sugar** for func = decorator(func).

5. Example Code

(a) Basic Function Decorator

```
def greet_decorator(func):
    def wrapper():
        print("Hello!")
        func()
        print("Goodbye!")
    return wrapper

@greet_decorator
def say_name():
    print("I am Alice")

say_name()
```

Output:

```
Hello!
I am Alice
Goodbye!
```

(b) Decorator with Arguments

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

@log_decorator
def add(a, b):
    return a + b
```

```
print(add(3, 5))
```

Output:

```
Calling add with (3, 5), {}
8
```

(c) Class Decorator

```
def decorator(cls):
    class Wrapped(cls):
        def __init__(self, *args, **kwargs):
            print("Creating instance of", cls.__name__)
            super().__init__(*args, **kwargs)
    return Wrapped

@decorator
class Person:
    def __init__(self, name):
        self.name = name

p = Person("Bob")
```

Output:

```
Creating instance of Person
```

(d) Built-in Decorators

- @staticmethod
- @classmethod
- @property

```
class Demo:
    @staticmethod
    def static_method():
        print("I am static")

    @classmethod
    def class_method(cls):
        print("I am class method")
```

```
    @property
    def prop(self):
        return "I am a property"

obj = Demo()
Demo.static_method()
Demo.class_method()
print(obj.prop)
```

6. Diagram / Flow

📌 **How Decorators Work**

Original Function → Wrapped by Decorator → Enhanced Function

📌 Example:

say_name()
  ↓
greet_decorator(say_name) → wrapper() adds extra behavior

7. Output

- Decorators **don't change the original function logic**.
- They **wrap extra features** around it.

8. Common Errors & Debugging

❌ **Error 1: Forgetting to return wrapper**

```
def decorator(func):
    def wrapper():
        print("Before")
        func()
        print("After")
    # missing return ❌
```

✅ Fix: return wrapper

❌ **Error 2: Losing function metadata**

print(add.__name__) # shows "wrapper" instead of "add"

✅ Fix: Use functools.wraps(func)

❌ **Error 3: Misusing arguments**

- Always use *args, **kwargs to support flexible function signatures.

9. Interview / Industry Insight

- Interview Qs:
  - What is a decorator in Python?
  - Difference between function decorator and class decorator?
  - What's the use of functools.wraps?
- Industry:
  - Heavily used in **Flask/Django** for routes, permissions, logging.
  - Used in **APIs** for validation and authentication.
  - Used in **ML frameworks** for caching and performance monitoring.

7.2: Closures

1. Definition / Concept

- A **closure** is a function that **remembers variables from its enclosing scope** even after that scope has finished execution.
- Inner functions can access variables from the outer function → this "closes over" those variables.

📌 Requirements for a closure:

1. A nested (inner) function.
2. Inner function uses variables from outer function.
3. Outer function returns the inner function.

2. Analogy / Real-Life Connection

Think of a **child remembering a lullaby from their parent**:

- The parent (outer function) may not be around anymore.
- But the child (inner function) remembers the song (variable).

3. Syntax

```
def outer_function(msg):
    def inner_function():
        print(msg)   # uses outer variable
    return inner_function

closure = outer_function("Hello")
closure()   # prints "Hello"
```

4. Step-by-Step Explanation

1. Outer function defines a variable.
2. Inner function uses that variable.
3. Outer function returns the inner function.
4. Even after outer function finishes, inner function remembers variable.

5. Example Code

(a) Simple Closure

```
def outer(name):
    def inner():
        print("Hello,", name)
    return inner

greet = outer("Alice")
greet()
```

Output:

Hello, Alice

(b) Closure with Counter

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter
```

```
c1 = make_counter()
print(c1())  # 1
print(c1())  # 2
print(c1())  # 3
```

(c) Closure as Function Factory

```
def power_factory(exp):
    def power(base):
        return base ** exp
    return power

square = power_factory(2)
cube = power_factory(3)

print(square(5))  # 25
print(cube(5))    # 125
```

(d) Closure Remembering State

```
def multiplier(n):
    def multiply(x):
        return x * n
    return multiply

double = multiplier(2)
triple = multiplier(3)

print(double(10))  # 20
print(triple(10))  # 30
```

6. Diagram / Flow

📌 **Closure Flow**

```
outer("Alice")
  → returns inner()
  → inner() still remembers name="Alice"
```

7. Output

- Inner function remembers outer variables.
- Useful for stateful functions without using classes.

8. Common Errors & Debugging

❌ **Error 1: Forgetting nonlocal**

```
def counter():
    count = 0
    def inner():
        count += 1   # ❌ UnboundLocalError
        return count
```

✅ Fix: Use nonlocal count.

❌ **Error 2: Misunderstanding scope**

- Inner functions only "close over" variables used inside them.

❌ **Error 3: Forgetting to return inner function**

- Otherwise closure won’t form.

9. Interview / Industry Insight

- Interview Qs:
    - What is a closure?
    - Difference between closure and global variable?
    - When would you use a closure?
- Industry:
    - Closures used in **decorators**, **callbacks**, and **event-driven programming**.
    - Common in **Flask/Django** → route handlers capture request context.
    - Useful for creating **function factories** and **stateful behavior** without OOP overhead.

7.3: Context Managers (custom using __enter__ and __exit__)

1. Definition / Concept

- A **context manager** is an object that **manages resources** using the with statement.
- Custom context managers are created by defining:
    - __enter__(self) → Code to run when entering the context (setup).
    - __exit__(self, exc_type, exc_value, traceback) → Code to run when leaving the context (cleanup).

📌 Purpose: Ensure **resources are released properly** (files, DB connections, network).

2. Analogy / Real-Life Connection

Think of **borrowing a library book**:

- __enter__ → You borrow the book (setup).
- Use the book (your code runs).
- __exit__ → You return the book (cleanup), even if you tore a page (error occurred).

3. Syntax

```
class MyContext:
    def __enter__(self):
        # setup
        return self
    def __exit__(self, exc_type, exc_value, traceback):
        # cleanup
        return False  # propagate exceptions (True suppresses)
```

4. Step-by-Step Explanation

1. Create a class with __enter__ and __exit__.
2. Use it in a with statement.
3. __enter__ runs before the block.
4. Block executes.
5. __exit__ runs automatically, handling cleanup (even if error occurs).

5. Example Code

(a) Basic Custom Context Manager

```
class MyContext:
    def __enter__(self):
        print("Entering context")
```

```
        return self
    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting context")
        if exc_type:
            print("Error handled:", exc_value)
        return True   # suppress error

with MyContext():
    print("Inside context")
```

Output:

Entering context
Inside context
Exiting context

(b) Handling Errors with Context Manager

```
with MyContext():
    print("Before error")
    x = 1 / 0   # ZeroDivisionError
    print("After error")
```

Output:

Entering context
Before error
Exiting context
Error handled: division by zero

(c) File Handling Example (Manual)

```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
```

```
        self.file.close()
        print("File closed")

with FileManager("test.txt", "w") as f:
    f.write("Hello with custom context manager")
```

Output:

File closed

(d) Suppressing vs Propagating Errors

```
class Demo:
    def __enter__(self):
        print("Start")
        return self
    def __exit__(self, exc_type, exc_value, traceback):
        print("End")
        return False  # propagate error

with Demo():
    x = 1 / 0
```

Output:

Start

End

ZeroDivisionError: division by zero

6. Diagram / Flow

📌 **Context Manager Lifecycle**

```
with MyContext() as obj:
    __enter__() → setup → block executes → __exit__() → cleanup
```

7. Output

- Guarantees cleanup.
- Can choose to suppress or propagate exceptions.

8. Common Errors & Debugging

❌ **Error 1: Forgetting __exit__**

- Cleanup won't happen → resources leak.

❌ **Error 2: Returning wrong value in __exit__**

- Returning True hides exceptions → may cause silent bugs.

❌ **Error 3: Misunderstanding scope**

- Resource exists **only inside with block**.

9. Interview / Industry Insight

- Interview Qs:
  - What is a context manager?
  - How does __enter__ and __exit__ work?
  - What's the difference between __exit__ returning True vs False?
- Industry:
  - Used in **file I/O, DB connections, network sockets, threading locks**.
  - Example: with open(...) → ensures file closes even if error occurs.
  - Django/Flask use context managers for **request handling and DB sessions**.

7.4: functools Module (lru_cache, partial, wraps)

1. Definition / Concept

The **functools module** provides **higher-order functions** and utilities that act on or return other functions.

Three widely used tools are:

- **lru_cache** → Caches function results (memoization).
- **partial** → Fixes some function arguments (creates a new function).
- **wraps** → Preserves metadata of decorated functions.

2. Analogy / Real-Life Connection

- **lru_cache** → Like a **restaurant remembering your last orders** (so next time they serve faster).
- **partial** → Like **pre-filling a form** with default values, so you only fill the remaining fields.

- **wraps** → Like giving credit to the **original author** of a book even if you reprinted it with your own cover.

3. Syntax

```
from functools import lru_cache, partial, wraps
```

4. Step-by-Step Explanation

1. **lru_cache** → Decorator to store results of expensive function calls.
2. **partial** → Pre-fills some arguments of a function.
3. **wraps** → Used inside custom decorators to keep original function name/docstring.

5. Example Code

(a) lru_cache Example

```
from functools import lru_cache
import time

@lru_cache(maxsize=3)  # cache last 3 results
def slow_square(n):
   time.sleep(2)
   return n * n

print(slow_square(4))  # slow first time
print(slow_square(4))  # fast second time (cached)
```

Output:

```
16   # (after 2 seconds)
16   # (instant)
```

(b) partial Example

```
from functools import partial

def power(base, exp):
   return base ** exp

square = partial(power, exp=2)
```

```
cube = partial(power, exp=3)

print(square(5))  # 25
print(cube(5))    # 125
```

(c) wraps Example in Decorators

```
from functools import wraps

def log_decorator(func):
    @wraps(func)   # preserves metadata
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@log_decorator
def greet(name):
    """This function greets someone"""
    return f"Hello, {name}"

print(greet("Alice"))
print(greet.__name__)  # preserved
print(greet.__doc__)   # preserved
```

Output:

```
Calling greet
Hello, Alice
greet
This function greets someone
```

6. Diagram / Flow

📌 **functools utilities**

```
lru_cache → stores results → faster repeated calls
partial   → creates new function with pre-filled args
wraps     → keeps original metadata in decorators
```

7. Output

- lru_cache → reduces computation time.
- partial → simplifies function usage.
- wraps → keeps function name, docstring intact.

8. Common Errors & Debugging

❌ **Error 1: Forgetting maxsize in lru_cache**

- If maxsize=None, cache grows indefinitely.

❌ **Error 2: Misusing partial**

square = partial(power, 2) # ❌ assigns base=2, not exp

✅ Fix: Use keyword → partial(power, exp=2).

❌ **Error 3: Not using wraps in decorators**

- Function metadata lost → debugging & introspection harder.

9. Interview / Industry Insight

- Interview Qs:
    - What is memoization? How does lru_cache help?
    - Difference between functools.partial and default arguments?
    - Why use wraps in decorators?
- Industry:
    - **lru_cache** → widely used in **dynamic programming, caching API responses**.
    - **partial** → used in **event-driven frameworks** to bind functions with predefined arguments.
    - **wraps** → critical in decorators for **maintaining clean code and metadata**.

7.5: itertools Module

1. Definition / Concept

- **itertools** is a Python standard library module that provides **fast, memory-efficient tools** for working with iterators.

- Common utilities:
  - combinations() → all possible pairs/groups.
  - permutations() → all possible orderings.
  - cycle() → infinite loop through iterable.
  - chain() → join multiple iterables.

2. Analogy / Real-Life Connection

- **Combinations** → Choosing toppings for a pizza (order doesn't matter).
- **Permutations** → Arranging people in seats (order matters).
- **Cycle** → A merry-go-round ride (keeps looping).
- **Chain** → Linking multiple chains together (combining lists).

3. Syntax

```
import itertools

itertools.combinations(iterable, r)
itertools.permutations(iterable, r)
itertools.cycle(iterable)
itertools.chain(iter1, iter2, ...)
```

4. Step-by-Step Explanation

1. Import itertools.
2. Use functions to generate iterators.
3. Convert iterators to list (if needed) or loop over them.
4. Save memory because results are generated lazily.

5. Example Code

(a) Combinations

```
import itertools

items = ['A', 'B', 'C']
print(list(itertools.combinations(items, 2)))
```

Output:

```
[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

(b) Permutations

```
print(list(itertools.permutations(items, 2)))
```

Output:

[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]

(c) Cycle

```
count = 0
for x in itertools.cycle("AB"):
    print(x, end=" ")
    count += 1
    if count == 6:
        break
```

Output:

A B A B A B

(d) Chain

```
nums1 = [1, 2]
nums2 = [3, 4]
print(list(itertools.chain(nums1, nums2)))
```

Output:

[1, 2, 3, 4]

(e) Real-Life Example – Password Generator (Permutations)

```
chars = ['1', '2', 'A']
for p in itertools.permutations(chars, 2):
    print("".join(p))
```

Output:

12
1A
21
2A
A1
A2

6. Diagram / Flow

📌 **itertools Functions**

combinations(['A','B','C'],2) → ('A','B'), ('A','C'), ('B','C')
permutations(['A','B','C'],2) → ('A','B'), ('B','A'), ...
cycle("AB") → A, B, A, B, ...
chain([1,2],[3,4]) → 1, 2, 3, 4

7. Output

- Iterators generated lazily.
- Efficient for large datasets.

8. Common Errors & Debugging

❌ **Error 1: Forgetting to convert to list**

print(itertools.combinations("ABC", 2)) # iterator object

✅ Fix: list(itertools.combinations("ABC", 2)).

❌ **Error 2: Confusing combinations with permutations**

- **Combinations** → order doesn't matter.
- **Permutations** → order matters.

❌ **Error 3: Infinite loop with cycle**

- Always use a counter to break out.

9. Interview / Industry Insight

- Interview Qs:
    - Difference between combinations and permutations?
    - How does itertools.chain work?
    - Why are itertools functions memory efficient?
- Industry:
    - **Combinations/permutations** → used in **testing, probability, password generation**.
    - **Cycle** → useful for **round-robin scheduling**.
    - **Chain** → used in **data pipelines to merge datasets**.

7.6: Type Hinting (typing module`)

1. Definition / Concept

- **Type Hinting** → A way to specify the **expected data types** of variables, function parameters, and return values.
- Introduced in Python 3.5 with the **typing module**.
- Python is **dynamically typed**, so type hints **don't enforce types at runtime** — they serve as **documentation** and help with **static analysis tools** (like mypy, IDE autocompletion).

2. Analogy / Real-Life Connection

Think of **labels on food packages**:

- The actual contents may vary (dynamic typing).
- But the label (type hint) tells you what you're **supposed** to expect (milk, not soda).

3. Syntax

```
from typing import List, Dict, Tuple, Optional

def func(a: int, b: str) -> bool:
    ...
```

4. Step-by-Step Explanation

1. Type hints use **colon :** for parameters.
2. Use **->** to indicate return type.
3. Use typing module for advanced types → List, Dict, Tuple, Optional, Union.
4. They help **readability, maintainability, and IDE assistance**.

5. Example Code

(a) Basic Function with Hints

```
def add(a: int, b: int) -> int:
    return a + b

print(add(2, 3))
```

(b) Using List and Dict

```
from typing import List, Dict
```

```
def process_scores(scores: List[int]) -> Dict[str, float]:
    return {"avg": sum(scores)/len(scores)}
```

(c) Tuple and Optional

```
from typing import Tuple, Optional

def divide(a: int, b: int) -> Optional[Tuple[int, int]]:
    if b == 0:
        return None
    return divmod(a, b)
```

(d) Union (Multiple Possible Types)

```
from typing import Union

def get_value(flag: bool) -> Union[int, str]:
    return 42 if flag else "forty-two"
```

(e) Modern Syntax (Python 3.9+)

```
def square_all(nums: list[int]) -> list[int]:
    return [x*x for x in nums]
```

6. Diagram / Flow

📌 **How Type Hints Work**

Code → With Type Hints → IDE/static tool checks → Runtime still dynamic

7. Output

- Type hints don't affect runtime output.
- They improve **developer understanding** and **tooling support**.

8. Common Errors & Debugging

❌ **Error 1: Assuming type hints enforce rules**

```
def add(a: int, b: int) -> int:
```

return str(a) + str(b) # ❌ still works at runtime

✅ Fix: Type hints are for static checking only.

❌ **Error 2: Forgetting to import from typing**

def func(data: List[int]): # ❌ NameError if List not imported

✅ Fix: from typing import List.

❌ **Error 3: Overusing type hints**

- Too many hints can reduce readability.
- Use **where clarity is needed most**.

9. Interview / Industry Insight

- Interview Qs:
  - Do type hints enforce data types in Python?
  - Difference between Optional and Union.
  - Why use type hints if Python is dynamically typed?
- Industry:
  - Heavily used in **large-scale projects** to make codebases maintainable.
  - Crucial in **API design, data science pipelines, backend frameworks**.
  - Tools like mypy, pylance, pyright rely on type hints.

7.7: Enumerations (enum module`)

1. Definition / Concept

- **Enumeration (Enum)** → a set of **named constant values**.
- Provided by the enum module (Python 3.4+).
- Enums make code **more readable, safe, and self-documenting** instead of using raw numbers/strings.

📌 Each member of an Enum has:

- **Name** → identifier.
- **Value** → constant value.

2. Analogy / Real-Life Connection

Think of **traffic lights**:

- RED = stop, GREEN = go, YELLOW = wait.
- Instead of using **0, 1, 2**, we use named constants → more meaningful.

3. Syntax

```
from enum import Enum

class Color(Enum):
   RED = 1
   GREEN = 2
   BLUE = 3
```

4. Step-by-Step Explanation

1. Import Enum from enum module.
2. Define class inheriting from Enum.
3. Define members as constants.
4. Access with ClassName.MEMBER.
5. Each member is unique and iterable.

5. Example Code

(a) Basic Enum

```
from enum import Enum

class Color(Enum):
   RED = 1
   GREEN = 2
   BLUE = 3

print(Color.RED)
print(Color.RED.name)
print(Color.RED.value)
```

Output:

```
Color.RED
```



```
RED
1
```

(b) Iterating over Enum

```python
for color in Color:
    print(color)
```

Output:

```
Color.RED
Color.GREEN
Color.BLUE
```

(c) Comparison

```python
print(Color.RED == Color.GREEN)  # False
print(Color.RED == Color.RED)    # True
```

(d) Enum with String Values

```python
class Status(Enum):
    SUCCESS = "success"
    FAILURE = "failure"
    PENDING = "pending"

print(Status.SUCCESS.value)  # "success"
```

(e) Accessing by Value or Name

```python
print(Color(1))       # Color.RED
print(Color["GREEN"]) # Color.GREEN
```

(f) Auto Values with auto()

```python
from enum import auto

class Day(Enum):
    MONDAY = auto()
    TUESDAY = auto()

print(Day.MONDAY.value)  # 1
```

print(Day.TUESDAY.value) # 2

6. Diagram / Flow

📌 **Enum Mapping**

Enum Class → Members → Each has Name + Value

Example: Color.RED
  name = "RED"
  value = 1

7. Output

- Enums give human-readable constants.
- Prevents accidental use of wrong values.

8. Common Errors & Debugging

❌ **Error 1: Duplicate values not unique by default**

```
class Example(Enum):
    A = 1
    B = 1
```

✅ Fix: Use @unique decorator.

❌ **Error 2: Confusing member name with value**

print(Color.RED == 1) # ❌ False

✅ Fix: Compare Color.RED.value == 1.

❌ **Error 3: Forgetting Enum immutability**

- Enum members **cannot be reassigned** after definition.

9. Interview / Industry Insight

- Interview Qs:
  - What is an Enum in Python?

- Difference between Enum and constants?
- How do you enforce unique values in Enum?
- Industry:
  - Used for **status codes, error handling, modes, categories**.
  - Example:
    - HTTP Status Codes (OK, NOT_FOUND, INTERNAL_ERROR).
    - Order status in e-commerce (PLACED, SHIPPED, DELIVERED).
  - Improves **code readability and maintainability**.

7.8: Named Tuples (collections.namedtuple)

1. Definition / Concept

- A **namedtuple** is like a **regular tuple**, but with **named fields** for better readability.
- Provided by the collections module.
- Elements can be accessed **both by index and by name**.

📌 Think of it as a **lightweight, immutable class** without methods.

2. Analogy / Real-Life Connection

- A **normal tuple** is like a list of ingredients without labels: ("sugar", 2, "cups").
- A **namedtuple** is like a labeled recipe card:
  - ingredient="sugar", quantity=2, unit="cups"
- Easier to read and understand.

3. Syntax

```
from collections import namedtuple

Point = namedtuple("Point", ["x", "y"])
p = Point(10, 20)
```

4. Step-by-Step Explanation

1. Import namedtuple from collections.
2. Create a new namedtuple type → namedtuple("TypeName", [fields]).
3. Create instances like a class.
4. Access fields via **dot notation** or index.

5. Immutable → values cannot be reassigned.

5. Example Code

(a) Basic NamedTuple

```
from collections import namedtuple

Point = namedtuple("Point", ["x", "y"])
p1 = Point(10, 20)

print(p1)        # Point(x=10, y=20)
print(p1.x, p1.y)
print(p1[0], p1[1])  # still works like tuple
```

Output:

```
Point(x=10, y=20)
10 20
10 20
```

(b) NamedTuple as Record

```
Student = namedtuple("Student", ["name", "age", "marks"])
s = Student("Alice", 21, 88)

print(s.name, s.age, s.marks)
```

Output:

```
Alice 21 88
```

(c) Using _fields and _asdict()

```
print(s._fields)     # ('name', 'age', 'marks')
print(s._asdict())   # OrderedDict([('name', 'Alice'), ('age', 21), ('marks', 88)])
```

(d) Replacing Values (_replace)

```
s2 = s._replace(marks=95)
print(s2)
```

Output:

```
Student(name='Alice', age=21, marks=95)
```

(e) Iterating NamedTuple

```
for field in s:
    print(field)
```

6. Diagram / Flow

📌 **Normal Tuple vs NamedTuple**

Tuple: ("Alice", 21, 88)

NamedTuple: Student(name="Alice", age=21, marks=88)

7. Output

- NamedTuples give **clarity & readability**.
- Behave like tuples (indexing, immutability).
- Behave like objects (dot notation).

8. Common Errors & Debugging

❌ **Error 1: Trying to modify value**

s.age = 22 # ❌ AttributeError (immutable)

✅ Fix: Use _replace().

❌ **Error 2: Forgetting field names**

Point = namedtuple("Point", ["x y"]) # ❌ wrong

✅ Fix: Use list → ["x", "y"].

❌ **Error 3: Overusing namedtuple**

- For complex behavior, use **dataclasses or classes**.

9. Interview / Industry Insight

- Interview Qs:
    - What is the difference between tuple and namedtuple?
    - How do you update a value in a namedtuple?
    - Is namedtuple mutable?

- Industry:
    - NamedTuples are used in **data parsing, database rows, structured logging**.
    - Example: Representing **coordinates, database records, CSV rows**.
    - Preferred when you want **lightweight objects without class boilerplate**.

7.9: GIL (Global Interpreter Lock) and Threading Considerations

1. Definition / Concept

- **GIL (Global Interpreter Lock)** → A mutex (lock) in CPython that ensures only **one thread executes Python bytecode at a time**.
- Purpose: Simplifies **memory management** in CPython.
- Effect: Multi-threading in Python **does not provide true parallelism** for CPU-bound tasks.

📌 Key points:

- Threads are good for **I/O-bound tasks** (waiting on files, network).
- For CPU-heavy tasks, use **multiprocessing** (separate processes) instead of threads.

2. Analogy / Real-Life Connection

Think of a **restaurant kitchen with only one knife**:

- Even if multiple chefs (threads) are present, only one can chop at a time (GIL).
- If chefs are waiting for boiling water (I/O), they can take turns efficiently.
- But if everyone wants to chop (CPU-bound), it's a bottleneck.

3. Syntax

Threading example:

```
import threading

def worker():
    print("Working...")

threads = []
for _ in range(5):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()
```


4. Step-by-Step Explanation

1. Python's GIL restricts CPU-bound multi-threading.
2. **I/O-bound tasks** (file read, network, API calls) → threads are useful.
3. **CPU-bound tasks** (math, data processing) → use multiprocessing.
4. Alternatives:
    - threading → for concurrency in I/O.
    - multiprocessing → for true parallelism.
    - asyncio → for asynchronous I/O tasks.

5. Example Code

(a) Threading (I/O-bound task works well)

```
import threading, time

def download_file(num):
   print(f"Downloading file {num}...")
   time.sleep(2)   # simulate I/O
   print(f"File {num} downloaded")

threads = []
for i in range(3):
   t = threading.Thread(target=download_file, args=(i,))
   threads.append(t)
   t.start()

for t in threads:
   t.join()
```

Output:

```
Downloading file 0...
Downloading file 1...
Downloading file 2...
File 0 downloaded
File 1 downloaded
File 2 downloaded
```

(b) CPU-Bound Example (Threads blocked by GIL)

```
import threading

def compute():
    count = 0
    for i in range(10**7):
        count += i

threads = []
for _ in range(4):
    t = threading.Thread(target=compute)
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("Done")
```

- Runs sequentially due to GIL → **no speedup**.

(c) Multiprocessing (True Parallelism)

```
import multiprocessing

def compute():
    count = 0
    for i in range(10**7):
        count += i

if __name__ == "__main__":
    processes = []
    for _ in range(4):
        p = multiprocessing.Process(target=compute)
        processes.append(p)
        p.start()

    for p in processes:
        p.join()
```

```
print("Done")
```

- Runs in **parallel** → much faster.

6. Diagram / Flow

📌 **Threads vs Processes in Python**

Threads:

├── Share memory

├── Blocked by GIL (CPU-bound)

└── Good for I/O

Processes:

├── Independent memory

├── True parallelism

└── Good for CPU-bound tasks

7. Output

- Threads → useful for **waiting tasks**.
- Multiprocessing → required for **heavy computations**.

8. Common Errors & Debugging

❌ **Error 1: Expecting threads to speed up CPU-bound tasks**

- They don't, due to GIL.

❌ **Error 2: Using multiprocessing without if __name__ == "__main__":**

- Causes issues on Windows.

❌ **Error 3: Confusing concurrency with parallelism**

- Concurrency = managing multiple tasks.
- Parallelism = running tasks simultaneously.

9. Interview / Industry Insight

- Interview Qs:
    - What is the GIL in Python?
    - Why are threads not effective for CPU-bound tasks in Python?
    - How do you achieve true parallelism in Python?
- Industry:
    - GIL is a **well-known limitation** of CPython.
    - Alternatives: **PyPy, Jython, IronPython** don't have GIL.
    - For performance:
        - **Threading/asyncio** for I/O-heavy apps (web servers, APIs).
        - **Multiprocessing** for CPU-heavy apps (ML training, data crunching).