# **Advanced Python Cookbook for Production Development**

## **Table of Contents**

1. [Basic Concepts](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#basic-concepts)
   * [Python Environment Setup](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#python-environment-setup)
   * [Virtual Environments](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#virtual-environments)
   * [Package Management](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#package-management)
   * [Code Organization](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#code-organization)
   * [Type Hints](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#type-hints)
2. [Intermediate Concepts](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#intermediate-concepts)
   * [Object-Oriented Programming](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#object-oriented-programming)
   * [Functional Programming](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#functional-programming)
   * [Error Handling and Exceptions](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#error-handling-and-exceptions)
   * [Context Managers](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#context-managers)
   * [Generators and Iterators](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#generators-and-iterators)
   * [Decorators](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#decorators)
   * [Concurrency Basics](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#concurrency-basics)
3. [Advanced Concepts](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#advanced-concepts)
   * [Metaprogramming](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#metaprogramming)
   * [Advanced Concurrency](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#advanced-concurrency)
   * [Performance Optimization](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#performance-optimization)
   * [Memory Management](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#memory-management)
   * [Design Patterns](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#design-patterns)
   * [Testing Strategies](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#testing-strategies)
   * [Packaging and Distribution](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#packaging-and-distribution)
4. [Enterprise-Level Practices](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#enterprise-level-practices)
   * [Application Architecture](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#application-architecture)
   * [API Development](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#api-development)
   * [Database Interaction](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#database-interaction)
   * [Logging and Monitoring](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#logging-and-monitoring)
   * [Security Best Practices](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#security-best-practices)
   * [Configuration Management](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#configuration-management)
   * [Continuous Integration/Continuous Deployment](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#continuous-integrationcontinuous-deployment)
   * [Containerization and Orchestration](https://claude.ai/chat/22485c74-9b96-45bb-9f38-f240a9172a3a#containerization-and-orchestration)

## **Basic Concepts**

### **Python Environment Setup**

#### **Python Versions and Installation**

While you're already familiar with Python, ensuring you have the right setup is crucial for production development:

# Check Python version

import sys

print(f"Python version: {sys.version}")

print(f"Python version info: {sys.version\_info}")

For production applications, Python 3.10+ is recommended due to performance improvements and important language features.

#### **Managing Multiple Python Versions with pyenv**

# Install pyenv (Linux/Mac)

curl https://pyenv.run | bash

# Install specific Python version

pyenv install 3.11.2

# Set global Python version

pyenv global 3.11.2

# Set local Python version for a project

cd your\_project

pyenv local 3.10.9

### **Virtual Environments**

Virtual environments are essential for isolating project dependencies and ensuring reproducible environments.

#### **Using venv (built-in)**

# Create a virtual environment

python -m venv myenv

# Activate the virtual environment

# On Windows

myenv\Scripts\activate

# On Unix or MacOS

source myenv/bin/activate

# Deactivate the virtual environment

deactivate

#### **Using Poetry (modern dependency management)**

# Install Poetry

curl -sSL https://install.python-poetry.org | python3 -

# Create a new project

poetry new my-project

# Initialize Poetry in an existing project

cd existing-project

poetry init

# Add dependencies

poetry add requests

# Add development dependencies

poetry add --dev pytest

# Install dependencies

poetry install

# Activate the virtual environment

poetry shell

# Update dependencies

poetry update

### **Package Management**

#### **Requirements Files**

# Generate requirements.txt

pip freeze > requirements.txt

# Install from requirements.txt

pip install -r requirements.txt

#### **Using Poetry for Dependency Management**

Poetry provides more sophisticated dependency resolution than pip:

# pyproject.toml

[tool.poetry]

name = "my-project"

version = "0.1.0"

description = "My Python project"

authors = ["Your Name <your.email@example.com>"]

[tool.poetry.dependencies]

python = "^3.10"

requests = "^2.28.1"

[tool.poetry.group.dev.dependencies]

pytest = "^7.2.0"

black = "^23.1.0"

[build-system]

requires = ["poetry-core>=1.0.0"]

build-backend = "poetry.core.masonry.api"

#### **Version Pinning Strategies**

# Fixed versions (exact match)

requests = "2.28.1"

# Compatible releases (SemVer)

requests = "^2.28.1" # >= 2.28.1, < 3.0.0

# Greater than or equal to

requests = ">=2.28.1"

# Range specification

requests = ">=2.28.1,<2.30.0"

### **Code Organization**

#### **Project Structure**

A well-organized Python project structure:

my\_project/

│

├── pyproject.toml # Project metadata and dependencies

├── README.md # Project documentation

├── LICENSE # License information

├── .gitignore # Git ignore file

│

├── src/ # Source code

│ └── my\_project/ # Main package

│ ├── \_\_init\_\_.py # Package initialization

│ ├── module1.py # Module 1

│ ├── module2.py # Module 2

│ └── subpackage/ # Subpackage

│ ├── \_\_init\_\_.py

│ └── module3.py

│

├── tests/ # Test directory

│ ├── \_\_init\_\_.py

│ ├── test\_module1.py

│ └── test\_module2.py

│

├── docs/ # Documentation

│

└── scripts/ # Utility scripts

#### **Modules and Packages**

# Creating a package

# \_\_init\_\_.py

"""My awesome package."""

from .module1 import function1

from .module2 import Class1

\_\_all\_\_ = ['function1', 'Class1']

#### **Import Best Practices**

# Absolute imports (preferred)

from my\_project.subpackage import module3

from my\_project.module1 import function1

# Relative imports (use with caution)

from ..module1 import function1 # Go up one directory level

from . import module3 # From the same directory

### **Type Hints**

Type hints improve code readability, enable static analysis, and help catch bugs early:

# Basic type hints

def calculate\_total(price: float, quantity: int) -> float:

return price \* quantity

# Complex type hints

from typing import List, Dict, Optional, Union, Callable, TypeVar, Generic

# For lists

def process\_items(items: List[str]) -> List[int]:

return [len(item) for item in items]

# For dictionaries

def process\_data(data: Dict[str, int]) -> Dict[str, float]:

return {key: value / 2 for key, value in data.items()}

# Optional parameters

def fetch\_user(user\_id: int, include\_details: Optional[bool] = None) -> Dict[str, Union[str, int]]:

result = {"id": user\_id, "name": "John Doe"}

if include\_details:

result["details"] = "Some details"

return result

# Type aliases

UserId = int

UserData = Dict[str, Union[str, int, bool]]

def get\_user\_info(user\_id: UserId) -> UserData:

return {"id": user\_id, "name": "Jane Doe", "active": True}

# Generic types

T = TypeVar('T')

def first\_element(collection: List[T]) -> Optional[T]:

return collection[0] if collection else None

#### **Type Checking with mypy**

# Install mypy

pip install mypy

# Run mypy on your code

mypy my\_project/

# mypy.ini configuration

[mypy]

python\_version = 3.10

warn\_return\_any = True

warn\_unused\_configs = True

disallow\_untyped\_defs = True

disallow\_incomplete\_defs = True

[mypy.plugins.numpy.dynamo]

enable = True

## **Intermediate Concepts**

### **Object-Oriented Programming**

#### **Class Design**

# Basic class structure

class Customer:

"""A customer representation."""

def \_\_init\_\_(self, customer\_id: int, name: str) -> None:

self.customer\_id = customer\_id

self.name = name

self.\_purchases: List[Purchase] = []

def add\_purchase(self, purchase: 'Purchase') -> None:

self.\_purchases.append(purchase)

@property

def total\_spent(self) -> float:

return sum(purchase.amount for purchase in self.\_purchases)

def \_\_str\_\_(self) -> str:

return f"Customer {self.name} (ID: {self.customer\_id})"

def \_\_repr\_\_(self) -> str:

return f"Customer(customer\_id={self.customer\_id}, name='{self.name}')"

#### **Inheritance and Composition**

# Inheritance

class Person:

def \_\_init\_\_(self, name: str, age: int) -> None:

self.name = name

self.age = age

def display\_info(self) -> str:

return f"{self.name}, {self.age} years old"

class Employee(Person):

def \_\_init\_\_(self, name: str, age: int, employee\_id: str) -> None:

super().\_\_init\_\_(name, age)

self.employee\_id = employee\_id

def display\_info(self) -> str:

return f"{super().display\_info()}, ID: {self.employee\_id}"

# Composition (often preferred over inheritance)

class Address:

def \_\_init\_\_(self, street: str, city: str, state: str, zip\_code: str) -> None:

self.street = street

self.city = city

self.state = state

self.zip\_code = zip\_code

class Customer:

def \_\_init\_\_(self, name: str, address: Address) -> None:

self.name = name

self.address = address # Composition

#### **Abstract Base Classes and Interfaces**

from abc import ABC, abstractmethod

class PaymentProcessor(ABC):

@abstractmethod

def process\_payment(self, amount: float) -> bool:

"""Process a payment."""

pass

@abstractmethod

def refund(self, amount: float) -> bool:

"""Process a refund."""

pass

class CreditCardProcessor(PaymentProcessor):

def process\_payment(self, amount: float) -> bool:

print(f"Processing credit card payment of ${amount}")

return True

def refund(self, amount: float) -> bool:

print(f"Processing credit card refund of ${amount}")

return True

#### **Dataclasses**

from dataclasses import dataclass, field

from datetime import datetime

from typing import List, Optional

@dataclass

class Product:

id: int

name: str

price: float

description: Optional[str] = None

created\_at: datetime = field(default\_factory=datetime.now)

tags: List[str] = field(default\_factory=list)

def discount\_price(self, percentage: float) -> float:

return self.price \* (1 - percentage / 100)

### **Functional Programming**

#### **Lambda Functions**

# Simple lambda

multiply = lambda x, y: x \* y

result = multiply(5, 3) # 15

# Lambda with sorting

people = [{'name': 'John', 'age': 30}, {'name': 'Jane', 'age': 25}]

sorted\_people = sorted(people, key=lambda person: person['age'])

#### **Map, Filter, and Reduce**

from functools import reduce

# Map: Apply function to each item

numbers = [1, 2, 3, 4, 5]

squared = list(map(lambda x: x\*\*2, numbers)) # [1, 4, 9, 16, 25]

# Filter: Select items based on condition

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers)) # [2, 4]

# Reduce: Aggregate items using a function

sum\_all = reduce(lambda acc, val: acc + val, numbers) # 15

# List comprehensions (often preferred over map/filter)

squared = [x\*\*2 for x in numbers] # [1, 4, 9, 16, 25]

even\_numbers = [x for x in numbers if x % 2 == 0] # [2, 4]

#### **Higher-Order Functions**

def create\_multiplier(factor: int) -> Callable[[int], int]:

def multiply(x: int) -> int:

return x \* factor

return multiply

double = create\_multiplier(2)

triple = create\_multiplier(3)

print(double(5)) # 10

print(triple(5)) # 15

#### **Immutability and Pure Functions**

# Pure function

def add\_pure(x: int, y: int) -> int:

return x + y

# Impure function

total = 0

def add\_impure(x: int) -> int:

global total

total += x

return total

# Working with immutable data

from typing import Dict, Tuple, Any

def update\_dict\_immutable(data: Dict[str, Any], key: str, value: Any) -> Dict[str, Any]:

return {\*\*data, key: value} # Create a new dictionary

original = {"name": "John", "age": 30}

updated = update\_dict\_immutable(original, "age", 31)

# original = {"name": "John", "age": 30}

# updated = {"name": "John", "age": 31}

### **Error Handling and Exceptions**

#### **Try-Except Blocks**

def divide(a: float, b: float) -> float:

try:

return a / b

except ZeroDivisionError:

print("Error: Division by zero!")

return float('inf')

except TypeError as e:

print(f"Error: {e}")

raise

finally:

print("Division operation attempted")

#### **Custom Exceptions**

class ApplicationError(Exception):

"""Base class for application-specific exceptions."""

pass

class ResourceNotFoundError(ApplicationError):

"""Raised when a requested resource is not found."""

def \_\_init\_\_(self, resource\_type: str, resource\_id: str) -> None:

self.resource\_type = resource\_type

self.resource\_id = resource\_id

message = f"{resource\_type} with ID '{resource\_id}' not found"

super().\_\_init\_\_(message)

# Using custom exceptions

def fetch\_user(user\_id: str) -> Dict[str, Any]:

if not user\_exists(user\_id):

raise ResourceNotFoundError("User", user\_id)

# ...

#### **Exception Chaining**

try:

# Some operation

process\_data()

except ValueError as e:

# Chain exceptions to preserve context

raise ApplicationError("Invalid data format") from e

#### **Handling Multiple Exceptions**

try:

result = complex\_operation()

except (ValueError, TypeError) as e:

# Handle multiple exception types

print(f"Operation error: {e}")

except Exception as e:

# Catch-all for other exceptions

print(f"Unexpected error: {e}")

# Re-raise to preserve the error

raise

### **Context Managers**

#### **Using with Statement**

# File operations with context manager

with open('file.txt', 'r') as file:

content = file.read()

# File is automatically closed when the block exits

# Multiple context managers

with open('input.txt', 'r') as input\_file, open('output.txt', 'w') as output\_file:

content = input\_file.read()

output\_file.write(content.upper())

#### **Creating Custom Context Managers**

# Using a class

class DatabaseConnection:

def \_\_init\_\_(self, connection\_string: str) -> None:

self.connection\_string = connection\_string

self.connection = None

def \_\_enter\_\_(self):

print(f"Connecting to database: {self.connection\_string}")

self.connection = connect\_to\_db(self.connection\_string)

return self.connection

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

print("Closing database connection")

if self.connection:

self.connection.close()

# Return True to suppress exceptions, False to propagate them

return False

# Using contextlib

from contextlib import contextmanager

@contextmanager

def database\_connection(connection\_string: str):

connection = None

try:

print(f"Connecting to database: {connection\_string}")

connection = connect\_to\_db(connection\_string)

yield connection

finally:

print("Closing database connection")

if connection:

connection.close()

# Usage

with database\_connection("postgres://user:pass@localhost:5432/db") as conn:

results = conn.execute("SELECT \* FROM users")

### **Generators and Iterators**

#### **Creating Generators**

def count\_up\_to(limit: int):

count = 1

while count <= limit:

yield count

count += 1

# Using a generator

for number in count\_up\_to(5):

print(number) # Prints 1, 2, 3, 4, 5

# Generator expressions

squares = (x \* x for x in range(10))

for square in squares:

print(square)

#### **Implementing Iterators**

class Fibonacci:

def \_\_init\_\_(self, limit: int) -> None:

self.limit = limit

self.previous = 0

self.current = 1

self.count = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.count >= self.limit:

raise StopIteration

self.count += 1

if self.count == 1:

return 0

result = self.current

self.current, self.previous = self.previous + self.current, self.current

return result

# Using the iterator

for num in Fibonacci(10):

print(num) # Prints the first 10 Fibonacci numbers

#### **Generator Use Cases**

def read\_large\_file(file\_path: str, chunk\_size: int = 1024):

"""Memory-efficient file reading."""

with open(file\_path, 'r') as file:

while True:

chunk = file.read(chunk\_size)

if not chunk:

break

yield chunk

# Processing a large file without loading it entirely into memory

for chunk in read\_large\_file('large\_file.txt'):

process\_data(chunk)

### **Decorators**

#### **Basic Decorators**

# Simple decorator

def log\_function\_call(func):

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

print(f"Calling {func.\_\_name\_\_} with args: {args}, kwargs: {kwargs}")

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

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

return result

return wrapper

@log\_function\_call

def add(a, b):

return a + b

# Equivalent to:

# add = log\_function\_call(add)

result = add(3, 5) # Logs the call and returns 8

#### **Decorators with Parameters**

def repeat(times: int):

def decorator(func):

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

results = []

for \_ in range(times):

results.append(func(\*args, \*\*kwargs))

return results

return wrapper

return decorator

@repeat(times=3)

def greet(name: str) -> str:

return f"Hello, {name}!"

print(greet("World")) # Returns ["Hello, World!", "Hello, World!", "Hello, World!"]

#### **Class Decorators**

def singleton(cls):

instances = {}

def get\_instance(\*args, \*\*kwargs):

if cls not in instances:

instances[cls] = cls(\*args, \*\*kwargs)

return instances[cls]

return get\_instance

@singleton

class Database:

def \_\_init\_\_(self, connection\_string: str) -> None:

print(f"Initializing database with {connection\_string}")

self.connection\_string = connection\_string

# Only one instance will be created

db1 = Database("connection1")

db2 = Database("connection2") # No initialization happens here

print(db1 is db2) # True

#### **Preserving Function Metadata**

from functools import wraps

def log\_function\_call(func):

@wraps(func) # Preserves the original function's metadata

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

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

return func(\*args, \*\*kwargs)

return wrapper

@log\_function\_call

def add(a, b):

"""Add two numbers."""

return a + b

print(add.\_\_name\_\_) # 'add' (not 'wrapper')

print(add.\_\_doc\_\_) # 'Add two numbers.' (preserved)

### **Concurrency Basics**

#### **Threading**

import threading

import time

from typing import List

def worker(name: str, delay: float) -> None:

print(f"Worker {name} starting")

time.sleep(delay)

print(f"Worker {name} finished")

# Create threads

threads: List[threading.Thread] = []

for i in range(5):

t = threading.Thread(target=worker, args=(f"Thread-{i}", i \* 0.5))

threads.append(t)

t.start()

# Wait for all threads to complete

for t in threads:

t.join()

print("All threads completed")

#### **Thread Safety**

import threading

# Thread-safe counter using a lock

class Counter:

def \_\_init\_\_(self) -> None:

self.value = 0

self.lock = threading.Lock()

def increment(self) -> None:

with self.lock:

self.value += 1

def get\_value(self) -> int:

with self.lock:

return self.value

# Thread-local storage

thread\_local = threading.local()

thread\_local.value = 0

def increment\_local():

thread\_local.value += 1

print(f"Thread {threading.current\_thread().name}: {thread\_local.value}")

#### **Multiprocessing**

import multiprocessing

import time

def worker(name: str, delay: float) -> None:

print(f"Process {name} starting")

time.sleep(delay)

print(f"Process {name} finished")

if \_\_name\_\_ == "\_\_main\_\_":

# Create processes

processes = []

for i in range(5):

p = multiprocessing.Process(target=worker, args=(f"Process-{i}", i \* 0.5))

processes.append(p)

p.start()

# Wait for all processes to complete

for p in processes:

p.join()

print("All processes completed")

#### **Concurrent.futures**

import concurrent.futures

import time

def worker(name: str) -> str:

time.sleep(1)

return f"Worker {name} completed"

# Using ThreadPoolExecutor

with concurrent.futures.ThreadPoolExecutor(max\_workers=5) as executor:

futures = {executor.submit(worker, f"Thread-{i}"): i for i in range(5)}

for future in concurrent.futures.as\_completed(futures):

worker\_id = futures[future]

try:

result = future.result()

print(f"Worker {worker\_id} result: {result}")

except Exception as e:

print(f"Worker {worker\_id} error: {e}")

# Using ProcessPoolExecutor

with concurrent.futures.ProcessPoolExecutor(max\_workers=3) as executor:

results = list(executor.map(worker, [f"Process-{i}" for i in range(5)]))

for result in results:

print(result)

## **Advanced Concepts**

### **Metaprogramming**

#### **Metaclasses**

# Metaclass example

class MetaLogger(type):

def \_\_new\_\_(mcs, name, bases, attrs):

# Add logging to all methods

for attr\_name, attr\_value in attrs.items():

if callable(attr\_value) and not attr\_name.startswith("\_\_"):

attrs[attr\_name] = MetaLogger.log\_method(attr\_value)

return super().\_\_new\_\_(mcs, name, bases, attrs)

@staticmethod

def log\_method(method):

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

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

return method(\*args, \*\*kwargs)

return wrapper

# Using the metaclass

class Service(metaclass=MetaLogger):

def process(self, data):

return f"Processed: {data}"

def analyze(self, data):

return f"Analysis result: {data}"

service = Service()

result = service.process("sample data") # Logs "Calling method process"

#### **Class Decorators**

def add\_methods(cls):

def new\_method(self, x):

return x \* 2

cls.new\_method = new\_method

return cls

@add\_methods

class MyClass:

def \_\_init\_\_(self, value):

self.value = value

# Now MyClass has a new\_method

obj = MyClass(5)

print(obj.new\_method(10)) # 20

#### **Descriptors**

class Validator:

def \_\_init\_\_(self, min\_value=None, max\_value=None):

self.min\_value = min\_value

self.max\_value = max\_value

self.name = None

def \_\_set\_name\_\_(self, owner, name):

self.name = name

def \_\_get\_\_(self, instance, owner):

if instance is None:

return self

return instance.\_\_dict\_\_[self.name]

def \_\_set\_\_(self, instance, value):

if self.min\_value is not None and value < self.min\_value:

raise ValueError(f"{self.name} must be at least {self.min\_value}")

if self.max\_value is not None and value > self.max\_value:

raise ValueError(f"{self.name} cannot exceed {self.max\_value}")

instance.\_\_dict\_\_[self.name] = value

class Person:

age = Validator(min\_value=0, max\_value=120)

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# This works

person = Person("John", 30)

# This raises ValueError

try:

person.age = 150

except ValueError as e:

print(e) # "age cannot exceed 120"

#### **Dynamic Attribute Access**

class DynamicAttributes:

def \_\_init\_\_(self):

self.\_data = {}

def \_\_getattr\_\_(self, name):

if name in self.\_data:

return self.\_data[name]

raise AttributeError(f"{self.\_\_class\_\_.\_\_name\_\_} has no attribute '{name}'")

def \_\_setattr\_\_(self, name, value):

if name == "\_data":

super().\_\_setattr\_\_(name, value)

else:

self.\_data[name] = value

def \_\_delattr\_\_(self, name):

if name in self.\_data:

del self.\_data[name]

else:

raise AttributeError(f"{self.\_\_class\_\_.\_\_name\_\_} has no attribute '{name}'")

# Using dynamic attributes

obj = DynamicAttributes()

obj.name = "John"

obj.age = 30

print(obj.name) # John

### **Advanced Concurrency**

#### **Asyncio Basics**

import asyncio

async def async\_task(name, delay):

print(f"Task {name} started")

await asyncio.sleep(delay) # Non-blocking sleep

print(f"Task {name} completed after {delay} seconds")

return f"Result from {name}"

async def main():

# Run tasks concurrently

results = await asyncio.gather(

async\_task("Task 1", 2),

async\_task("Task 2", 1),

async\_task("Task 3", 3)

)

print("All tasks completed")

print(f"Results: {results}")

# Run the event loop

asyncio.run(main())

#### **Async Context Managers**

import asyncio

class AsyncDatabase:

async def \_\_aenter\_\_(self):

print("Connecting to database")

await asyncio.sleep(1) # Simulating connection time

return self

async def \_\_aexit\_\_(self, exc\_type, exc\_val, exc\_tb):

print("Closing database connection")

await asyncio.sleep(0.5) # Simulating disconnection time

async def query(self, sql):

print(f"Executing query: {sql}")

await asyncio.sleep(0.5) # Simulating query execution

return ["result1", "result2"]

async def main():

async with AsyncDatabase() as db:

results = await db.query("SELECT \* FROM users")

print(f"Query results: {results}")

asyncio.run(main())

#### **Async Generators**

import asyncio

async def async\_range(start, stop):

for i in range(start, stop):

await asyncio.sleep(0.1) # Non-blocking sleep

yield i

async def main():

async for i in async\_range(1, 5):

print(f"Received: {i}")

asyncio.run(main())

#### **Combining Asyncio with Threads and Processes**

import asyncio

import concurrent.futures

import time

def cpu\_bound\_task(n):

"""A CPU-bound task to demonstrate ProcessPoolExecutor."""

result = 0

for i in range(n \* 10000000):

result += i

return result

async def main():

# For CPU-bound tasks, use ProcessPoolExecutor

with concurrent.futures.ProcessPoolExecutor() as process\_pool:

loop = asyncio.get\_running\_loop()

# Schedule the CPU-bound task in the process pool

result = await loop.run\_in\_executor(

process\_pool, cpu\_bound\_task, 1

)

print(f"CPU-bound task result: {result}")

# For I/O-bound blocking operations, use ThreadPoolExecutor

with concurrent.futures.ThreadPoolExecutor() as thread\_pool:

# Schedule blocking I/O in the thread pool

result = await loop.run\_in\_executor(

thread\_pool, time.sleep, 1

)

print("I/O-bound task completed")

asyncio.run(main())

### **Performance Optimization**

#### **Profiling**

import cProfile

import pstats

from pstats import SortKey

# Profile a function

def profile\_function(func, \*args, \*\*kwargs):

profile = cProfile.Profile()

profile.enable()

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

profile.disable()

ps = pstats.Stats(profile).sort\_stats(SortKey.CUMULATIVE)

ps.print\_stats(10)

return result

# Usage example

def fibonacci(n):

if n <= 1:

return n

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

profile\_function(fibonacci, 30)

# Line profiler for detailed line-by-line analysis

# pip install line\_profiler

# Usage with IPython: %load\_ext line\_profiler

# %lprun -f function\_name function\_name(arguments)

#### **CPU and Memory Optimization**

# Use built-in functions and libraries when possible

# Slow

result = 0

for i in range(1000000):

result += i

# Fast

result = sum(range(1000000))

# Use appropriate data structures

from collections import defaultdict, Counter

# Instead of:

word\_counts = {}

for word in words:

if word not in word\_counts:

word\_counts[word] = 0

word\_counts[word] += 1

# Use:

word\_counts = defaultdict(int)

for word in words:

word\_counts[word] += 1

# Or even better:

word\_counts = Counter(words)

# Use generators for memory efficiency

# Memory-intensive

all\_data = [process(item) for item in large\_dataset]

result = sum(all\_data)

# Memory-efficient

result = sum(process(item) for item in large\_dataset)

#### **Numba for Performance**

# pip install numba

import numpy as np

from numba import jit

# Regular Python function

def python\_sum\_array(arr):

total = 0

for i in range(len(arr)):

total += arr[i]

return total

# Numba-optimized function

@jit(nopython=True)

def numba\_sum\_array(arr):

total = 0

for i in range(len(arr)):

total += arr[i]

return total

# Comparison

data = np.random.random(10000000)

%time result1 = python\_sum\_array(data)

%time result2 = numba\_sum\_array(data)

# Numba typically provides significant performance improvements

#### **Cython for Performance**

# demo.pyx file

def fibonacci\_cy(int n):

cdef int a = 0, b = 1, i, temp

if n <= 0:

return a

for i in range(2, n + 1):

temp = a + b

a = b

b = temp

return b

# setup.py

from setuptools import setup

from Cython.Build import cythonize

setup(

ext\_modules=cythonize("demo.pyx")

)

# Build with: python setup.py build\_ext --inplace

# Then import and use:

from demo import fibonacci\_cy

result = fibonacci\_cy(30)

### **Memory Management**

#### **Monitoring Memory Usage**

import tracemalloc

import linecache

import os

def display\_top(snapshot, key\_type='lineno', limit=10):

snapshot = snapshot.filter\_traces((

tracemalloc.Filter(False, "<frozen importlib.\_bootstrap>"),

tracemalloc.Filter(False, "<unknown>"),

))

top\_stats = snapshot.statistics(key\_type)

print(f"Top {limit} lines")

for index, stat in enumerate(top\_stats[:limit], 1):

frame = stat.traceback[0]

filename = os.path.basename(frame.filename)

print(f"#{index}: {filename}:{frame.lineno}: {stat.size / 1024:.1f} KiB")

line = linecache.getline(frame.filename, frame.lineno).strip()

if line:

print(f" {line}")

other = top\_stats[limit:]

if other:

size = sum(stat.size for stat in other)

print(f"{len(other)} other: {size / 1024:.1f} KiB")

total = sum(stat.size for stat in top\_stats)

print(f"Total allocated size: {total / 1024:.1f} KiB")

# Track memory allocations

tracemalloc.start()

# Run your code here

data = [object() for \_ in range(1000)]

# Take snapshot and display statistics

snapshot = tracemalloc.take\_snapshot()

display\_top(snapshot)

#### **Garbage Collection**

import gc

# Force garbage collection

gc.collect()

# Get reference counts

import sys

x = []

print(sys.getrefcount(x)) # Always at least 2 (variable + getrefcount param)

# Inspect garbage collector

print(gc.get\_count()) # Get collection counters

print(gc.get\_threshold()) # Get collection thresholds

# Disable automatic garbage collection (for performance-critical sections)

gc.disable()

# ... critical code ...

gc.enable()

#### **Memory Leaks and Circular References**

# Circular reference example

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.children = []

def add\_child(self, node):

self.children.append(node)

# Create a circular reference

node1 = Node(1)

node2 = Node(2)

node1.add\_child(node2)

node2.add\_child(node1) # Circular reference

# Using weak references to avoid memory leaks

import weakref

class BetterNode:

def \_\_init\_\_(self, value):

self.value = value

self.children = []

self.parent = None

def add\_child(self, node):

self.children.append(node)

node.parent = weakref.ref(self) # Weak reference

# Create references without circular memory issues

node1 = BetterNode(1)

node2 = BetterNode(2)

node1.add\_child(node2) # parent is a weak reference

### **Design Patterns**

#### **Creational Patterns**

# Singleton

class Singleton:

\_instance = None

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if cls.\_instance is None:

cls.\_instance = super().\_\_new\_\_(cls)

return cls.\_instance

# Factory

class AnimalFactory:

@staticmethod

def create\_animal(animal\_type):

if animal\_type == "dog":

return Dog()

elif animal\_type == "cat":

return Cat()

else:

raise ValueError(f"Unknown animal type: {animal\_type}")

# Builder

class Computer:

def \_\_init\_\_(self):

self.cpu = None

self.memory = None

self.storage = None

self.gpu = None

class ComputerBuilder:

def \_\_init\_\_(self):

self.computer = Computer()

def with\_cpu(self, cpu):

self.computer.cpu = cpu

return self

def with\_memory(self, memory):

self.computer.memory = memory

return self

def with\_storage(self, storage):

self.computer.storage = storage

return self

def with\_gpu(self, gpu):

self.computer.gpu = gpu

return self

def build(self):

return self.computer

# Usage

computer = (ComputerBuilder()

.with\_cpu("Intel i7")

.with\_memory("16GB")

.with\_storage("512GB SSD")

.with\_gpu("NVIDIA RTX 3080")

.build())

#### **Structural Patterns**

# Adapter

class OldSystem:

def old\_operation(self, data):

return f"Old operation with {data}"

class NewSystem:

def new\_operation(self, info):

return f"New operation with {info}"

class SystemAdapter:

def \_\_init\_\_(self, new\_system):

self.new\_system = new\_system

def old\_operation(self, data):

return self.new\_system.new\_operation(data)

# Decorator (OOP implementation)

class Component:

def operation(self):

pass

class ConcreteComponent(Component):

def operation(self):

return "Base operation"

class Decorator(Component):

def \_\_init\_\_(self, component):

self.component = component

def operation(self):

return self.component.operation()

class LoggingDecorator(Decorator):

def operation(self):

print("Logging: Operation started")

result = self.component.operation()

print("Logging: Operation completed")

return result

# Usage

component = ConcreteComponent()

decorated\_component = LoggingDecorator(component)

result = decorated\_component.operation()

#### **Behavioral Patterns**

# Observer

class Subject:

def \_\_init\_\_(self):

self.\_observers = []

def attach(self, observer):

self.\_observers.append(observer)

def detach(self, observer):

self.\_observers.remove(observer)

def notify(self, message):

for observer in self.\_observers:

observer.update(message)

class Observer:

def update(self, message):

pass

class ConcreteObserver(Observer):

def \_\_init\_\_(self, name):

self.name = name

def update(self, message):

print(f"Observer {self.name} received: {message}")

# Strategy

class Strategy:

def execute(self, data):

pass

class ConcreteStrategyA(Strategy):

def execute(self, data):

return sorted(data)

class ConcreteStrategyB(Strategy):

def execute(self, data):

return sorted(data, reverse=True)

class Context:

def \_\_init\_\_(self, strategy):

self.strategy = strategy

def set\_strategy(self, strategy):

self.strategy = strategy

def execute\_strategy(self, data):

return self.strategy.execute(data)

# Chain of Responsibility

class Handler:

def \_\_init\_\_(self):

self.successor = None

def set\_successor(self, successor):

self.successor = successor

def handle(self, request):

pass

class ConcreteHandler1(Handler):

def handle(self, request):

if request < 10:

return f"Handler 1 processed request: {request}"

elif self.successor:

return self.successor.handle(request)

return None

class ConcreteHandler2(Handler):

def handle(self, request):

if 10 <= request < 20:

return f"Handler 2 processed request: {request}"

elif self.successor:

return self.successor.handle(request)

return None

### **Testing Strategies**

#### **Unit Testing**

# test\_calculator.py

import unittest

from calculator import Calculator

class TestCalculator(unittest.TestCase):

def setUp(self):

self.calc = Calculator()

def test\_add(self):

self.assertEqual(self.calc.add(1, 2), 3)

self.assertEqual(self.calc.add(-1, 1), 0)

self.assertEqual(self.calc.add(-1, -1), -2)

def test\_divide(self):

self.assertEqual(self.calc.divide(10, 2), 5)

with self.assertRaises(ValueError):

self.calc.divide(10, 0)

def tearDown(self):

pass

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

#### **Pytest**

# test\_calculator\_pytest.py

import pytest

from calculator import Calculator

@pytest.fixture

def calculator():

return Calculator()

def test\_add(calculator):

assert calculator.add(1, 2) == 3

assert calculator.add(-1, 1) == 0

assert calculator.add(-1, -1) == -2

def test\_divide(calculator):

assert calculator.divide(10, 2) == 5

with pytest.raises(ValueError):

calculator.divide(10, 0)

# Parameterized tests

@pytest.mark.parametrize("a,b,expected", [

(1, 2, 3),

(-1, 1, 0),

(-1, -1, -2)

])

def test\_add\_params(calculator, a, b, expected):

assert calculator.add(a, b) == expected

#### **Mocking**

# Using unittest.mock

from unittest.mock import Mock, patch, MagicMock

import unittest

from user\_service import UserService

class TestUserService(unittest.TestCase):

def test\_get\_user\_details(self):

# Create a mock database

mock\_db = Mock()

mock\_db.query.return\_value = {"id": 1, "name": "John Doe"}

# Inject the mock into the service

service = UserService(mock\_db)

user = service.get\_user\_details(1)

# Assertions

self.assertEqual(user["name"], "John Doe")

mock\_db.query.assert\_called\_once\_with("SELECT \* FROM users WHERE id = 1")

@patch('user\_service.requests')

def test\_fetch\_external\_data(self, mock\_requests):

# Setup the mock response

mock\_response = Mock()

mock\_response.status\_code = 200

mock\_response.json.return\_value = {"data": "example"}

mock\_requests.get.return\_value = mock\_response

# Create service and call method

service = UserService(Mock())

result = service.fetch\_external\_data("example\_api")

# Assertions

self.assertEqual(result, {"data": "example"})

mock\_requests.get.assert\_called\_once\_with("https://api.example.com/example\_api")

#### **Integration Testing**

# Integration test with actual database

import pytest

import os

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

from models import Base, User

from user\_service import UserService

@pytest.fixture(scope="module")

def db\_session():

# Create test database

engine = create\_engine('sqlite:///test.db')

Base.metadata.create\_all(engine)

Session = sessionmaker(bind=engine)

session = Session()

# Setup test data

user = User(name="Test User", email="test@example.com")

session.add(user)

session.commit()

yield session

# Teardown

session.close()

os.remove("test.db")

def test\_user\_service\_integration(db\_session):

service = UserService(db\_session)

user = service.get\_user\_by\_email("test@example.com")

assert user is not None

assert user.name == "Test User"

### **Packaging and Distribution**

#### **Package Structure**

my\_package/

├── pyproject.toml # Project configuration

├── setup.py # Build script

├── setup.cfg # Configuration for setup.py

├── README.md # Project readme

├── LICENSE # License file

├── CHANGELOG.md # Change history

├── requirements.txt # Dependencies

├── requirements-dev.txt # Development dependencies

├── src/ # Source code

│ └── my\_package/ # Main package directory

│ ├── \_\_init\_\_.py # Package initialization

│ ├── module1.py # Module 1

│ └── subpackage/ # Subpackage

│ ├── \_\_init\_\_.py

│ └── module2.py

├── tests/ # Test directory

│ ├── \_\_init\_\_.py

│ ├── test\_module1.py

│ └── test\_subpackage/

│ └── test\_module2.py

└── docs/ # Documentation

├── conf.py

├── index.rst

└── api.rst

#### **Modern Packaging with pyproject.toml**

# pyproject.toml

[build-system]

requires = ["setuptools>=61.0", "wheel"]

build-backend = "setuptools.build\_meta"

[project]

name = "my-package"

version = "0.1.0"

description = "A sample Python package"

readme = "README.md"

requires-python = ">=3.8"

license = {text = "MIT"}

authors = [

{name = "Your Name", email = "your.email@example.com"},

]

classifiers = [

"Development Status :: 4 - Beta",

"Intended Audience :: Developers",

"License :: OSI Approved :: MIT License",

"Programming Language :: Python :: 3",

"Programming Language :: Python :: 3.8",

"Programming Language :: Python :: 3.9",

"Programming Language :: Python :: 3.10",

]

dependencies = [

"requests>=2.28.0",

"numpy>=1.23.0",

]

[project.optional-dependencies]

dev = [

"pytest>=7.0.0",

"pytest-cov>=3.0.0",

"black>=22.3.0",

"isort>=5.10.1",

"mypy>=0.950",

]

docs = [

"sphinx>=5.0.0",

"sphinx-rtd-theme>=1.0.0",

]

[project.urls]

"Homepage" = "https://github.com/username/my-package"

"Bug Tracker" = "https://github.com/username/my-package/issues"

[project.scripts]

my-command = "my\_package.cli:main"

[tool.black]

line-length = 88

target-version = ["py38", "py39", "py310"]

[tool.isort]

profile = "black"

[tool.mypy]

python\_version = "3.8"

warn\_return\_any = true

warn\_unused\_configs = true

disallow\_untyped\_defs = true

disallow\_incomplete\_defs = true

#### **Building and Publishing**

# Build the package

python -m build

# Upload to PyPI (requires twine)

python -m pip install twine

python -m twine upload dist/\*

# Upload to Test PyPI

python -m twine upload --repository testpypi dist/\*

## **Enterprise-Level Practices**

### **Application Architecture**

#### **Layered Architecture**

my\_application/

├── presentation/ # UI/API layer

│ ├── \_\_init\_\_.py

│ ├── api.py

│ └── web.py

├── application/ # Application/Service layer

│ ├── \_\_init\_\_.py

│ ├── services/

│ │ ├── \_\_init\_\_.py

│ │ ├── user\_service.py

│ │ └── product\_service.py

│ └── dto/ # Data Transfer Objects

│ ├── \_\_init\_\_.py

│ └── user\_dto.py

├── domain/ # Domain/Business layer

│ ├── \_\_init\_\_.py

│ ├── models/

│ │ ├── \_\_init\_\_.py

│ │ ├── user.py

│ │ └── product.py

│ └── repositories/ # Repository interfaces

│ ├── \_\_init\_\_.py

│ ├── user\_repository.py

│ └── product\_repository.py

└── infrastructure/ # Infrastructure layer

├── \_\_init\_\_.py

├── config.py

├── logging.py

├── database/

│ ├── \_\_init\_\_.py

│ ├── models.py

│ └── repositories/ # Repository implementations

│ ├── \_\_init\_\_.py

│ ├── user\_repository.py

│ └── product\_repository.py

└── external/

├── \_\_init\_\_.py

└── payment\_gateway.py

#### **Dependency Injection**

# Simple dependency injection without a framework

class UserRepository:

def get\_user(self, user\_id):

pass

class DatabaseUserRepository(UserRepository):

def \_\_init\_\_(self, db\_connection):

self.db\_connection = db\_connection

def get\_user(self, user\_id):

return self.db\_connection.query(f"SELECT \* FROM users WHERE id = {user\_id}")

class UserService:

def \_\_init\_\_(self, user\_repository):

self.user\_repository = user\_repository

def get\_user\_details(self, user\_id):

return self.user\_repository.get\_user(user\_id)

# Dependency injection with a container

from dependency\_injector import containers, providers

class Container(containers.DeclarativeContainer):

config = providers.Configuration()

db\_connection = providers.Singleton(

Database,

connection\_string=config.db.connection\_string,

)

user\_repository = providers.Factory(

DatabaseUserRepository,

db\_connection=db\_connection,

)

user\_service = providers.Factory(

UserService,

user\_repository=user\_repository,

)

# Usage

container = Container()

container.config.db.connection\_string.from\_env("DATABASE\_URL")

user\_service = container.user\_service()

user = user\_service.get\_user\_details(1)

#### **Domain-Driven Design**

# Value Object

@dataclass(frozen=True)

class Money:

amount: Decimal

currency: str

def \_\_add\_\_(self, other):

if self.currency != other.currency:

raise ValueError("Cannot add different currencies")

return Money(self.amount + other.amount, self.currency)

def \_\_sub\_\_(self, other):

if self.currency != other.currency:

raise ValueError("Cannot subtract different currencies")

return Money(self.amount - other.amount, self.currency)

# Entity

class Order:

def \_\_init\_\_(self, order\_id: str, customer\_id: str):

self.order\_id = order\_id

self.customer\_id = customer\_id

self.items = []

self.status = "new"

def add\_item(self, product\_id: str, quantity: int, unit\_price: Money):

item = OrderItem(product\_id, quantity, unit\_price)

self.items.append(item)

def total\_price(self) -> Money:

if not self.items:

return Money(Decimal('0'), 'USD')

currency = self.items[0].unit\_price.currency

total = Decimal('0')

for item in self.items:

if item.unit\_price.currency != currency:

raise ValueError("Mixed currencies in order")

total += item.unit\_price.amount \* item.quantity

return Money(total, currency)

def place(self):

if not self.items:

raise ValueError("Cannot place an empty order")

self.status = "placed"

# Aggregate

class Cart:

def \_\_init\_\_(self, cart\_id: str, customer\_id: str):

self.cart\_id = cart\_id

self.customer\_id = customer\_id

self.items = {} # product\_id -> CartItem

def add\_item(self, product\_id: str, quantity: int, unit\_price: Money):

if product\_id in self.items:

self.items[product\_id].increase\_quantity(quantity)

else:

self.items[product\_id] = CartItem(product\_id, quantity, unit\_price)

def remove\_item(self, product\_id: str):

if product\_id in self.items:

del self.items[product\_id]

def checkout(self) -> Order:

if not self.items:

raise ValueError("Cannot checkout empty cart")

order = Order(str(uuid.uuid4()), self.customer\_id)

for item in self.items.values():

order.add\_item(item.product\_id, item.quantity, item.unit\_price)

order.place()

self.items.clear()

return order

# Repository

class OrderRepository:

def save(self, order: Order):

pass

def find\_by\_id(self, order\_id: str) -> Optional[Order]:

pass

def find\_by\_customer\_id(self, customer\_id: str) -> List[Order]:

pass

### **API Development**

#### **RESTful API with Flask**

from flask import Flask, request, jsonify

from flask\_restful import Resource, Api

app = Flask(\_\_name\_\_)

api = Api(app)

class UserResource(Resource):

def get(self, user\_id):

# Get user by ID

user = get\_user\_from\_db(user\_id)

if user:

return jsonify(user)

return {"error": "User not found"}, 404

def put(self, user\_id):

# Update user

data = request.get\_json()

success = update\_user(user\_id, data)

if success:

return {"message": "User updated successfully"}

return {"error": "Failed to update user"}, 400

def delete(self, user\_id):

# Delete user

success = delete\_user(user\_id)

if success:

return {"message": "User deleted successfully"}

return {"error": "Failed to delete user"}, 400

class UsersResource(Resource):

def get(self):

# List all users

users = get\_all\_users()

return jsonify(users)

def post(self):

# Create new user

data = request.get\_json()

user\_id = create\_user(data)

return {"message": "User created successfully", "user\_id": user\_id}, 201

# Register resources

api.add\_resource(UsersResource, '/users')

api.add\_resource(UserResource, '/users/<int:user\_id>')

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

#### **FastAPI Example**

from fastapi import FastAPI, HTTPException, Depends, status

from pydantic import BaseModel

from typing import List, Optional

from sqlalchemy.orm import Session

from . import models, schemas

from .database import get\_db

app = FastAPI()

class User(BaseModel):

id: Optional[int] = None

name: str

email: str

active: bool = True

@app.post("/users/", response\_model=schemas.User, status\_code=status.HTTP\_201\_CREATED)

def create\_user(user: schemas.UserCreate, db: Session = Depends(get\_db)):

db\_user = models.User(name=user.name, email=user.email)

db.add(db\_user)

db.commit()

db.refresh(db\_user)

return db\_user

@app.get("/users/", response\_model=List[schemas.User])

def read\_users(skip: int = 0, limit: int = 100, db: Session = Depends(get\_db)):

users = db.query(models.User).offset(skip).limit(limit).all()

return users

@app.get("/users/{user\_id}", response\_model=schemas.User)

def read\_user(user\_id: int, db: Session = Depends(get\_db)):

db\_user = db.query(models.User).filter(models.User.id == user\_id).first()

if db\_user is None:

raise HTTPException(status\_code=404, detail="User not found")

return db\_user

@app.put("/users/{user\_id}", response\_model=schemas.User)

def update\_user(user\_id: int, user: schemas.UserUpdate, db: Session = Depends(get\_db)):

db\_user = db.query(models.User).filter(models.User.id == user\_id).first()

if db\_user is None:

raise HTTPException(status\_code=404, detail="User not found")

for key, value in user.dict(exclude\_unset=True).items():

setattr(db\_user, key, value)

db.commit()

db.refresh(db\_user)

return db\_user

@app.delete("/users/{user\_id}", status\_code=status.HTTP\_204\_NO\_CONTENT)

def delete\_user(user\_id: int, db: Session = Depends(get\_db)):

db\_user = db.query(models.User).filter(models.User.id == user\_id).first()

if db\_user is None:

raise HTTPException(status\_code=404, detail="User not found")

db.delete(db\_user)

db.commit()

return None

#### **API Authentication**

from flask import Flask, request, jsonify

from flask\_jwt\_extended import JWTManager, create\_access\_token, jwt\_required, get\_jwt\_identity

app = Flask(\_\_name\_\_)

app.config['JWT\_SECRET\_KEY'] = 'your-secret-key' # Change this in production!

jwt = JWTManager(app)

@app.route('/login', methods=['POST'])

def login():

username = request.json.get('username', None)

password = request.json.get('password', None)

# Check username and password (implement proper authentication!)

if username != 'test' or password != 'test':

return jsonify({"message": "Invalid credentials"}), 401

# Create access token

access\_token = create\_access\_token(identity=username)

return jsonify(access\_token=access\_token)

@app.route('/protected', methods=['GET'])

@jwt\_required()

def protected():

# Access the identity of the current user with get\_jwt\_identity

current\_user = get\_jwt\_identity()

return jsonify(logged\_in\_as=current\_user), 200

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

### **Database Interaction**

#### **SQLAlchemy ORM**

from sqlalchemy import create\_engine, Column, Integer, String, ForeignKey, Text, DateTime, Boolean

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import relationship, sessionmaker

from datetime import datetime

# Create engine and base

engine = create\_engine('sqlite:///app.db', echo=True)

Base = declarative\_base()

# Define models

class User(Base):

\_\_tablename\_\_ = 'users'

id = Column(Integer, primary\_key=True)

username = Column(String(50), unique=True, nullable=False)

email = Column(String(100), unique=True, nullable=False)

password\_hash = Column(String(128), nullable=False)

is\_active = Column(Boolean, default=True)

created\_at = Column(DateTime, default=datetime.utcnow)

posts = relationship("Post", back\_populates="author")

def \_\_repr\_\_(self):

return f"<User {self.username}>"

class Post(Base):

\_\_tablename\_\_ = 'posts'

id = Column(Integer, primary\_key=True)

title = Column(String(100), nullable=False)

content = Column(Text, nullable=False)

created\_at = Column(DateTime, default=datetime.utcnow)

user\_id = Column(Integer, ForeignKey('users.id'), nullable=False)

author = relationship("User", back\_populates="posts")

def \_\_repr\_\_(self):

return f"<Post {self.title}>"

# Create tables

Base.metadata.create\_all(engine)

# Create session

Session = sessionmaker(bind=engine)