Intermediate Python Programming: The Complete Guide

Mastering Python Beyond the Basics

Table of Contents

Chapter 1: Advanced Data Structures and Algorithms {#advanced-data-structures}

Collections Module Deep Dive

python

```python
from collections import (
    defaultdict, Counter, OrderedDict, deque,
    namedtuple, ChainMap, UserDict, UserList
)

# defaultdict - Dictionary with default values
word_count = defaultdict(int)
text = "the quick brown fox jumps over the lazy dog the fox"
for word in text.split():
    word_count[word] += 1
print(dict(word_count))  # {'the': 2, 'quick': 1, ...}

# Nested defaultdict
nested_dict = defaultdict(lambda: defaultdict(list))
nested_dict['users']['admin'].append('Alice')
nested_dict['users']['admin'].append('Bob')
nested_dict['users']['guest'].append('Charlie')

# Counter - Multiset/Bag implementation
counter = Counter(text.split())
print(counter.most_common(3))  # [('the', 2), ('fox', 2), ('quick', 1)]

# Counter arithmetic
counter1 = Counter(['a', 'b', 'c', 'a', 'b'])
counter2 = Counter(['a', 'b', 'b', 'd'])
print(counter1 + counter2)  # Counter({'b': 4, 'a': 3, 'c': 1, 'd': 1})
print(counter1 - counter2)  # Counter({'c': 1, 'a': 1})
print(counter1 & counter2)  # Counter({'b': 2, 'a': 1})

# deque - Double-ended queue
dq = deque([1, 2, 3], maxlen=5)
dq.append(4)      # Add to right
dq.appendleft(0)  # Add to left
dq.rotate(2)      # Rotate right by 2
print(dq)         # deque([3, 4, 0, 1, 2], maxlen=5)

# ChainMap - Chain multiple dictionaries
defaults = {'theme': 'light', 'language': 'en'}
user_settings = {'theme': 'dark'}
config = ChainMap(user_settings, defaults)
print(config['theme'])     # 'dark' (from user_settings)
print(config['language'])  # 'en' (from defaults)
```

Custom Data Structures

python

```python
class Node:
    """Node for linked list implementation"""
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    """Singly linked list implementation"""
    def __init__(self):
        self.head = None
        self.size = 0

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
        self.size += 1

    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def __iter__(self):
        current = self.head
        while current:
            yield current.data
            current = current.next

    def __len__(self):
        return self.size

    def __repr__(self):
        return ' -> '.join(str(item) for item in self)

# Binary Search Tree
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
```

```
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def inorder_traversal(self):
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        if node:
            self._inorder_recursive(node.left, result)
            result.append(node.value)
            self._inorder_recursive(node.right, result)
```

Advanced Sorting and Searching

python

```python
import heapq
from typing import List, Tuple

# Heap operations
class PriorityQueue:
    """Min-heap based priority queue"""
    def __init__(self):
        self.heap = []
        self.counter = 0

    def push(self, item, priority):
        # Use counter to ensure stable sorting
        heapq.heappush(self.heap, (priority, self.counter, item))
        self.counter += 1

    def pop(self):
        if self.heap:
            return heapq.heappop(self.heap)[2]
        raise IndexError("pop from empty queue")

    def __len__(self):
        return len(self.heap)

# Binary search variations
def binary_search_leftmost(arr: List[int], target: int) -> int:
    """Find leftmost occurrence of target"""
    left, right = 0, len(arr) - 1
    result = -1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            result = mid
            right = mid - 1  # Continue searching left
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return result

def binary_search_range(arr: List[int], target: int) -> Tuple[int, int]:
    """Find range [start, end] of target occurrences"""
    def find_bound(is_left: bool) -> int:
        left, right = 0, len(arr) - 1
        result = -1
```

```
        while left <= right:
            mid = (left + right) // 2
            if arr[mid] == target:
                result = mid
                if is_left:
                    right = mid - 1
                else:
                    left = mid + 1
            elif arr[mid] < target:
                left = mid + 1
            else:
                right = mid - 1

        return result

    return (find_bound(True), find_bound(False))

# Custom sorting with key functions
class Student:
    def __init__(self, name, grade, age):
        self.name = name
        self.grade = grade
        self.age = age

    def __repr__(self):
        return f"Student({self.name}, {self.grade}, {self.age})"

students = [
    Student("Alice", 85, 20),
    Student("Bob", 85, 19),
    Student("Charlie", 90, 21)
]

# Sort by multiple criteria
students.sort(key=lambda s: (-s.grade, s.age, s.name))
print(students)
```

Graph Algorithms

python

```python
from collections import defaultdict, deque
from typing import Dict, List, Set, Tuple

class Graph:
    """Graph implementation with common algorithms"""
    def __init__(self, directed=False):
        self.graph = defaultdict(list)
        self.directed = directed

    def add_edge(self, u, v, weight=1):
        self.graph[u].append((v, weight))
        if not self.directed:
            self.graph[v].append((u, weight))

    def bfs(self, start):
        """Breadth-first search"""
        visited = set()
        queue = deque([start])
        result = []

        while queue:
            vertex = queue.popleft()
            if vertex not in visited:
                visited.add(vertex)
                result.append(vertex)

                for neighbor, _ in self.graph[vertex]:
                    if neighbor not in visited:
                        queue.append(neighbor)

        return result

    def dfs(self, start):
        """Depth-first search"""
        visited = set()
        result = []

        def dfs_recursive(vertex):
            visited.add(vertex)
            result.append(vertex)

            for neighbor, _ in self.graph[vertex]:
                if neighbor not in visited:
                    dfs_recursive(neighbor)

        dfs_recursive(start)
```

```
        return result

    def dijkstra(self, start):
        """Dijkstra's shortest path algorithm"""
        import heapq

        distances = {vertex: float('infinity') for vertex in self.graph}
        distances[start] = 0
        pq = [(0, start)]
        visited = set()

        while pq:
            current_distance, current_vertex = heapq.heappop(pq)

            if current_vertex in visited:
                continue

            visited.add(current_vertex)

            for neighbor, weight in self.graph[current_vertex]:
                distance = current_distance + weight

                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    heapq.heappush(pq, (distance, neighbor))

        return distances

    def has_cycle(self):
        """Detect cycle in graph (for directed graphs)"""
        WHITE, GRAY, BLACK = 0, 1, 2
        color = defaultdict(int)

        def visit(vertex):
            if color[vertex] == GRAY:
                return True  # Back edge found

            if color[vertex] == BLACK:
                return False  # Already processed

            color[vertex] = GRAY

            for neighbor, _ in self.graph[vertex]:
                if visit(neighbor):
                    return True

            color[vertex] = BLACK
```

```
        return False

    for vertex in self.graph:
        if color[vertex] == WHITE:
            if visit(vertex):
                return True

    return False
```

Chapter 2: Decorators and Functional Programming {#decorators-functional}

Advanced Decorator Patterns

python

```python
import functools
import time
from typing import Any, Callable, TypeVar

# Decorator with arguments
def retry(max_attempts: int = 3, delay: float = 1.0):
    """Retry decorator with configurable attempts and delay"""
    def decorator(func: Callable) -> Callable:
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            last_exception = None
            for attempt in range(max_attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    last_exception = e
                    if attempt < max_attempts - 1:
                        time.sleep(delay)
                    else:
                        raise last_exception
            return None
        return wrapper
    return decorator

@retry(max_attempts=3, delay=0.5)
def unreliable_network_call():
    import random
    if random.random() < 0.7:
        raise ConnectionError("Network error")
    return "Success!"

# Class decorator
def singleton(cls):
    """Make a class a singleton"""
    instances = {}

    @functools.wraps(cls)
    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return get_instance

@singleton
class DatabaseConnection:
```

```
    def __init__(self):
        self.connection = "Connected to database"

# Property decorator with validation
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature below absolute zero")
        self._celsius = value

    @property
    def fahrenheit(self):
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) * 5/9

# Caching decorator
def memoize(func):
    """Cache function results"""
    cache = {}

    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        # Create a key from args and kwargs
        key = str(args) + str(kwargs)

        if key not in cache:
            cache[key] = func(*args, **kwargs)

        return cache[key]

    wrapper.cache_clear = lambda: cache.clear()
    return wrapper

@memoize
def fibonacci(n):
    if n < 2:
```

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

# Decorator factory with state
class CountCalls:
    """Decorator that counts function calls"""
    def __init__(self, func):
        self.func = func
        self.count = 0
        functools.update_wrapper(self, func)

    def __call__(self, *args, **kwargs):
        self.count += 1
        print(f"{self.func.__name__} called {self.count} times")
        return self.func(*args, **kwargs)

    def reset_count(self):
        self.count = 0

@CountCalls
def greet(name):
    return f"Hello, {name}!"
```

Functional Programming Concepts

python

```python
from functools import partial, reduce
from operator import add, mul
from typing import Callable, List, TypeVar, Iterable

T = TypeVar('T')
U = TypeVar('U')

# Higher-order functions
def compose(*functions):
    """Compose multiple functions"""
    def inner(arg):
        result = arg
        for func in reversed(functions):
            result = func(result)
        return result
    return inner

# Usage
add_one = lambda x: x + 1
multiply_two = lambda x: x * 2
square = lambda x: x ** 2

combined = compose(square, multiply_two, add_one)
print(combined(3))  # ((3 + 1) * 2) ** 2 = 64

# Currying
def curry(func):
    """Convert a function to curried form"""
    @functools.wraps(func)
    def curried(*args, **kwargs):
        if len(args) + len(kwargs) >= func.__code__.co_argcount:
            return func(*args, **kwargs)
        return partial(func, *args, **kwargs)
    return curried

@curry
def add_three_numbers(a, b, c):
    return a + b + c

# Can be called in multiple ways
print(add_three_numbers(1)(2)(3))  # 6
print(add_three_numbers(1, 2)(3))  # 6
print(add_three_numbers(1)(2, 3))  # 6

# Monadic-style operations
class Maybe:
```

```
    """Simple Maybe monad implementation"""
    def __init__(self, value):
        self.value = value

    def bind(self, func):
        if self.value is None:
            return Maybe(None)
        return Maybe(func(self.value))

    def is_nothing(self):
        return self.value is None

    def get_or_else(self, default):
        return default if self.value is None else self.value

# Pipeline operations
def pipeline(*functions):
    """Create a pipeline of functions"""
    return compose(*reversed(functions))

# Custom map, filter, reduce implementations
def my_map(func: Callable[[T], U], iterable: Iterable[T]) -> List[U]:
    """Custom map implementation"""
    return [func(item) for item in iterable]

def my_filter(predicate: Callable[[T], bool], iterable: Iterable[T]) -> List[T]:
    """Custom filter implementation"""
    return [item for item in iterable if predicate(item)]

def my_reduce(func: Callable[[T, T], T], iterable: Iterable[T], initial=None) -> T:
    """Custom reduce implementation"""
    it = iter(iterable)
    if initial is None:
        value = next(it)
    else:
        value = initial

    for item in it:
        value = func(value, item)

    return value

# Lazy evaluation
class LazyList:
    """Lazy list implementation"""
    def __init__(self, generator_func):
        self.generator_func = generator_func
```

```
        self._cache = []
        self._generator = None

    def __getitem__(self, index):
        if self._generator is None:
            self._generator = self.generator_func()

        while len(self._cache) <= index:
            try:
                self._cache.append(next(self._generator))
            except StopIteration:
                raise IndexError("Index out of range")

        return self._cache[index]

# Usage
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib_list = LazyList(fibonacci_generator)
print(fib_list[10])  # 55
print(fib_list[20])  # 6765
```

Chapter 3: Advanced Object-Oriented Programming {#advanced-oop}

Abstract Base Classes and Interfaces

python

```python
from abc import ABC, abstractmethod, ABCMeta
from typing import List, Protocol

# Abstract Base Class
class Shape(ABC):
    """Abstract base class for shapes"""

    @abstractmethod
    def area(self) -> float:
        """Calculate area of the shape"""
        pass

    @abstractmethod
    def perimeter(self) -> float:
        """Calculate perimeter of the shape"""
        pass

    def describe(self) -> str:
        return f"Shape with area {self.area():.2f} and perimeter {self.perimeter():.2f}"

class Circle(Shape):
    def __init__(self, radius: float):
        self.radius = radius

    def area(self) -> float:
        return 3.14159 * self.radius ** 2

    def perimeter(self) -> float:
        return 2 * 3.14159 * self.radius

# Protocol (Structural subtyping)
class Drawable(Protocol):
    """Protocol for drawable objects"""
    def draw(self) -> None: ...
    def get_position(self) -> tuple[float, float]: ...

class Button:
    def __init__(self, x: float, y: float, text: str):
        self.x = x
        self.y = y
        self.text = text

    def draw(self) -> None:
        print(f"Drawing button '{self.text}' at ({self.x}, {self.y})")

    def get_position(self) -> tuple[float, float]:
```

```
        return (self.x, self.y)

# Button implements Drawable protocol without inheritance
def render_drawable(drawable: Drawable) -> None:
    drawable.draw()
    pos = drawable.get_position()
    print(f"Position: {pos}")

# Custom metaclass for validation
class ValidatedMeta(type):
    """Metaclass that validates class attributes"""
    def __new__(mcs, name, bases, namespace, **kwargs):
        # Validate that all methods have docstrings
        for attr_name, attr_value in namespace.items():
            if callable(attr_value) and not attr_name.startswith('_'):
                if not attr_value.__doc__:
                    raise ValueError(f"Method {attr_name} lacks docstring")

        return super().__new__(mcs, name, bases, namespace)

class DocumentedClass(metaclass=ValidatedMeta):
    def method_with_doc(self):
        """This method has documentation"""
        pass
```

Multiple Inheritance and MRO

python

```python
# Diamond problem and Method Resolution Order (MRO)
class A:
    def method(self):
        print("A.method()")

class B(A):
    def method(self):
        print("B.method()")
        super().method()

class C(A):
    def method(self):
        print("C.method()")
        super().method()

class D(B, C):
    def method(self):
        print("D.method()")
        super().method()

# Understanding MRO
d = D()
d.method()  # D -> B -> C -> A
print(D.__mro__)  # (<class 'D'>, <class 'B'>, <class 'C'>, <class 'A'>, <class 'object'>)

# Mixins
class TimestampMixin:
    """Mixin to add timestamp functionality"""
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.created_at = time.time()
        self.updated_at = self.created_at

    def touch(self):
        """Update the timestamp"""
        self.updated_at = time.time()

class SerializableMixin:
    """Mixin to add JSON serialization"""
    def to_dict(self):
        return {
            key: value
            for key, value in self.__dict__.items()
            if not key.startswith('_')
        }
```

```
    def to_json(self):
        import json
        return json.dumps(self.to_dict(), indent=2)

class User(TimestampMixin, SerializableMixin):
    def __init__(self, name, email):
        super().__init__()
        self.name = name
        self.email = email

# Composition over inheritance
class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower

    def start(self):
        print(f"Engine with {self.horsepower}hp started")

class Transmission:
    def __init__(self, gears):
        self.gears = gears
        self.current_gear = 0

    def shift_up(self):
        if self.current_gear < self.gears:
            self.current_gear += 1

class Car:
    def __init__(self, engine, transmission):
        self.engine = engine
        self.transmission = transmission

    def start(self):
        self.engine.start()
        print("Car is ready to drive")
```

Advanced Magic Methods

python

```
class Vector:
    """Advanced vector class with full operator support"""
    def __init__(self, *components):
        self._components = list(components)

    def __repr__(self):
        return f"Vector{tuple(self._components)}"

    def __str__(self):
        return f"<{', '.join(map(str, self._components))}>"

    def __len__(self):
        return len(self._components)

    def __getitem__(self, index):
        return self._components[index]

    def __setitem__(self, index, value):
        self._components[index] = value

    def __iter__(self):
        return iter(self._components)

    def __eq__(self, other):
        if not isinstance(other, Vector):
            return NotImplemented
        return self._components == other._components

    def __add__(self, other):
        if isinstance(other, Vector):
            if len(self) != len(other):
                raise ValueError("Vectors must have same dimension")
            return Vector(*(a + b for a, b in zip(self, other)))
        elif isinstance(other, (int, float)):
            return Vector(*(a + other for a in self))
        return NotImplemented

    def __radd__(self, other):
        return self.__add__(other)

    def __mul__(self, scalar):
        if isinstance(scalar, (int, float)):
            return Vector(*(a * scalar for a in self))
        return NotImplemented

    def __rmul__(self, scalar):
```

```
        return self.__mul__(scalar)

    def __matmul__(self, other):
        """Dot product using @ operator"""
        if not isinstance(other, Vector) or len(self) != len(other):
            return NotImplemented
        return sum(a * b for a, b in zip(self, other))

    def __abs__(self):
        """Magnitude of vector"""
        return sum(a ** 2 for a in self) ** 0.5

    def __bool__(self):
        """Vector is truthy if any component is non-zero"""
        return any(self._components)

    def __hash__(self):
        """Make vector hashable if immutable"""
        return hash(tuple(self._components))

# Context manager protocol
class ManagedResource:
    """Resource that implements context manager protocol"""
    def __init__(self, name):
        self.name = name
        self.is_open = False

    def __enter__(self):
        print(f"Acquiring resource: {self.name}")
        self.is_open = True
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print(f"Releasing resource: {self.name}")
        self.is_open = False
        if exc_type is not None:
            print(f"Exception occurred: {exc_type.__name__}: {exc_value}")
        # Return False to propagate exception
        return False

# Descriptor protocol
class ValidatedAttribute:
    """Descriptor for validated attributes"""
    def __init__(self, validator):
        self.validator = validator
        self.data = {}
```

```
    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        return self.data.get(id(obj), None)

    def __set__(self, obj, value):
        self.validator(value)
        self.data[id(obj)] = value

    def __delete__(self, obj):
        del self.data[id(obj)]

class Person:
    name = ValidatedAttribute(lambda x: isinstance(x, str) or (_ for _ in ()).throw(TypeError('
    age = ValidatedAttribute(lambda x: x >= 0 or (_ for _ in ()).throw(ValueError("Age must be

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Chapter 4: Metaclasses and Descriptors {#metaclasses-descriptors}

Understanding Metaclasses

python

```python
# Metaclass basics
class Meta(type):
    """Custom metaclass"""
    def __new__(mcs, name, bases, namespace, **kwargs):
        print(f"Creating class {name}")
        # Modify namespace before class creation
        namespace['class_id'] = f"{name}_{id(namespace)}"
        return super().__new__(mcs, name, bases, namespace)

    def __init__(cls, name, bases, namespace, **kwargs):
        print(f"Initializing class {name}")
        super().__init__(name, bases, namespace)
        cls.instances_count = 0

    def __call__(cls, *args, **kwargs):
        print(f"Creating instance of {cls.__name__}")
        instance = super().__call__(*args, **kwargs)
        cls.instances_count += 1
        return instance

class MyClass(metaclass=Meta):
    def __init__(self, value):
        self.value = value

# Singleton metaclass
class SingletonMeta(type):
    """Metaclass that creates singleton classes"""
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class Database(metaclass=SingletonMeta):
    def __init__(self):
        self.connection = "Database connection"

# Registry metaclass
class RegistryMeta(type):
    """Metaclass that maintains a registry of classes"""
    registry = {}

    def __new__(mcs, name, bases, namespace, **kwargs):
        cls = super().__new__(mcs, name, bases, namespace)
        if 'abstract' not in namespace or not namespace['abstract']:
```

```
            RegistryMeta.registry[name] = cls
        return cls

    @classmethod
    def get_class(mcs, name):
        return mcs.registry.get(name)

class Plugin(metaclass=RegistryMeta):
    abstract = True

    def execute(self):
        raise NotImplementedError

class AudioPlugin(Plugin):
    def execute(self):
        return "Processing audio"

class VideoPlugin(Plugin):
    def execute(self):
        return "Processing video"

# Metaclass for automatic property creation
class AutoPropertyMeta(type):
    """Automatically create properties for attributes starting with _"""
    def __new__(mcs, name, bases, namespace):
        for key, value in list(namespace.items()):
            if key.startswith('_') and not key.startswith('__'):
                # Create property for private attribute
                def make_property(attr_name):
                    def getter(self):
                        return getattr(self, attr_name)
                    def setter(self, value):
                        setattr(self, attr_name, value)
                    return property(getter, setter)

                public_name = key[1:]  # Remove underscore
                namespace[public_name] = make_property(key)

        return super().__new__(mcs, name, bases, namespace)
```

Advanced Descriptors

python

```python
# Type checking descriptor
class TypedDescriptor:
    """Descriptor that enforces type checking"""
    def __init__(self, name, expected_type, default=None):
        self.name = name
        self.expected_type = expected_type
        self.default = default

    def __get__(self, obj, owner):
        if obj is None:
            return self
        return obj.__dict__.get(self.name, self.default)

    def __set__(self, obj, value):
        if not isinstance(value, self.expected_type):
            raise TypeError(f"{self.name} must be {self.expected_type.__name__}")
        obj.__dict__[self.name] = value

    def __delete__(self, obj):
        del obj.__dict__[self.name]

# Lazy property descriptor
class LazyProperty:
    """Descriptor that computes value only once"""
    def __init__(self, func):
        self.func = func
        self.name = func.__name__

    def __get__(self, obj, owner):
        if obj is None:
            return self
        value = self.func(obj)
        # Replace descriptor with computed value
        setattr(obj, self.name, value)
        return value

# Cached property with expiration
import time
from functools import wraps

class CachedProperty:
    """Property that caches value with expiration"""
    def __init__(self, ttl=60):
        self.ttl = ttl
        self.cache = {}
        self.timestamps = {}
```

```
    def __call__(self, func):
        self.func = func
        return self

    def __get__(self, obj, owner):
        if obj is None:
            return self

        obj_id = id(obj)
        now = time.time()

        # Check if cached value exists and is valid
        if obj_id in self.cache:
            if now - self.timestamps[obj_id] < self.ttl:
                return self.cache[obj_id]

        # Compute and cache value
        value = self.func(obj)
        self.cache[obj_id] = value
        self.timestamps[obj_id] = now
        return value

    def invalidate(self, obj):
        """Invalidate cache for specific object"""
        obj_id = id(obj)
        if obj_id in self.cache:
            del self.cache[obj_id]
            del self.timestamps[obj_id]

# Method descriptor
class BoundMethod:
    """Custom method descriptor"""
    def __init__(self, func):
        self.func = func

    def __get__(self, obj, owner):
        if obj is None:
            return self.func
        return MethodWrapper(self.func, obj)

class MethodWrapper:
    def __init__(self, func, obj):
        self.func = func
        self.obj = obj

    def __call__(self, *args, **kwargs):
```

```
        return self.func(self.obj, *args, **kwargs)

# Validated descriptor with custom validators
class ValidatedDescriptor:
    """Descriptor with pluggable validators"""
    def __init__(self, *validators, default=None):
        self.validators = validators
        self.default = default
        self.data = {}

    def __get__(self, obj, owner):
        if obj is None:
            return self
        return self.data.get(id(obj), self.default)

    def __set__(self, obj, value):
        for validator in self.validators:
            if not validator(value):
                raise ValueError(f"Validation failed for {value}")
        self.data[id(obj)] = value

# Example usage
class Product:
    name = TypedDescriptor('name', str)
    price = TypedDescriptor('price', (int, float), 0)
    quantity = ValidatedDescriptor(
        lambda x: x >= 0,
        lambda x: isinstance(x, int),
        default=0
    )

    @LazyProperty
    def inventory_value(self):
        print("Computing inventory value...")
        return self.price * self.quantity

    @CachedProperty(ttl=5)
    def discounted_price(self):
        print("Computing discounted price...")
        import random
        discount = random.uniform(0.1, 0.3)
        return self.price * (1 - discount)
```

Chapter 5: Context Managers and Protocols {#context-managers-protocols}

Advanced Context Managers

python

```
import contextlib
import sys
from typing import Any, Optional
import threading
import sqlite3

# Nested context managers
@contextlib.contextmanager
def managed_transaction(connection):
    """Context manager for database transactions"""
    try:
        yield connection
        connection.commit()
        print("Transaction committed")
    except Exception as e:
        connection.rollback()
        print(f"Transaction rolled back: {e}")
        raise
    finally:
        print("Transaction ended")

# Reentrant context manager
class ReentrantLock:
    """Reentrant lock context manager"""
    def __init__(self):
        self._lock = threading.RLock()
        self._count = 0

    def __enter__(self):
        self._lock.acquire()
        self._count += 1
        print(f"Lock acquired (count: {self._count})")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self._count -= 1
        print(f"Lock released (count: {self._count})")
        self._lock.release()
        return False

# Context manager for temporary changes
@contextlib.contextmanager
def temporary_attr(obj, attr, new_value):
    """Temporarily change an attribute"""
    old_value = getattr(obj, attr)
    setattr(obj, attr, new_value)
```

```
    try:
        yield
    finally:
        setattr(obj, attr, old_value)

# Suppress specific exceptions
@contextlib.contextmanager
def suppress_and_log(*exceptions):
    """Suppress exceptions and log them"""
    try:
        yield
    except exceptions as e:
        print(f"Suppressed exception: {type(e).__name__}: {e}")

# Context manager class with cleanup
class ResourcePool:
    """Pool of resources with context management"""
    def __init__(self, factory, size=5):
        self.factory = factory
        self.size = size
        self.resources = []
        self.available = []
        self._lock = threading.Lock()

    def __enter__(self):
        with self._lock:
            if not self.available and len(self.resources) < self.size:
                resource = self.factory()
                self.resources.append(resource)
                return resource
            elif self.available:
                return self.available.pop()
            else:
                raise RuntimeError("No resources available")

    def __exit__(self, exc_type, exc_val, exc_tb):
        # Return resource to pool
        with self._lock:
            # Get the resource that was checked out
            # In real implementation, track which resource was given
            pass

# ExitStack for dynamic context management
def process_files(filenames):
    """Process multiple files with dynamic context management"""
    with contextlib.ExitStack() as stack:
        files = [
```

```
            stack.enter_context(open(fname, 'r'))
            for fname in filenames
        ]

        # Process all files
        for f in files:
            print(f"Processing {f.name}")
            # Files will be automatically closed

# Async context managers
import asyncio

class AsyncResource:
    """Async context manager example"""
    async def __aenter__(self):
        print("Acquiring async resource")
        await asyncio.sleep(0.1)  # Simulate async operation
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        print("Releasing async resource")
        await asyncio.sleep(0.1)  # Simulate async cleanup
        return False

# Context manager decorator with parameters
def timing_context(label="Operation"):
    """Context manager that times operations"""
    @contextlib.contextmanager
    def timer():
        import time
        start = time.time()
        print(f"{label} started")
        try:
            yield
        finally:
            end = time.time()
            print(f"{label} completed in {end - start:.4f} seconds")
    return timer()
```

Python Protocols Deep Dive

python

```python
from typing import Protocol, runtime_checkable
import collections.abc

# Iterator protocol
class CountdownIterator:
    """Iterator that counts down from n to 1"""
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.n <= 0:
            raise StopIteration
        self.n -= 1
        return self.n + 1

# Container protocol
class RangeContainer:
    """Container that checks if value is in range"""
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __contains__(self, value):
        return self.start <= value < self.end

    def __len__(self):
        return max(0, self.end - self.start)

# Callable protocol
class Multiplier:
    """Callable object that multiplies by a factor"""
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, value):
        return value * self.factor

    def __repr__(self):
        return f"Multiplier({self.factor})"

# Sequence protocol
class FibonacciSequence:
    """Sequence of Fibonacci numbers"""
```

```
    def __init__(self, max_n=None):
        self.max_n = max_n
        self._cache = [0, 1]

    def __getitem__(self, index):
        if isinstance(index, slice):
            # Handle slicing
            start, stop, step = index.indices(len(self))
            return [self[i] for i in range(start, stop, step)]

        if index < 0:
            raise IndexError("Negative indices not supported")

        # Extend cache if needed
        while len(self._cache) <= index:
            self._cache.append(self._cache[-1] + self._cache[-2])

        return self._cache[index]

    def __len__(self):
        if self.max_n is None:
            raise TypeError("Infinite sequence has no length")
        return self.max_n

# Runtime checkable protocol
@runtime_checkable
class Comparable(Protocol):
    """Protocol for comparable objects"""
    def __lt__(self, other: Any) -> bool: ...
    def __le__(self, other: Any) -> bool: ...
    def __gt__(self, other: Any) -> bool: ...
    def __ge__(self, other: Any) -> bool: ...

# Buffer protocol
class BitArray:
    """Simple bit array with buffer protocol"""
    def __init__(self, size):
        self.size = size
        self.bytes = bytearray((size + 7) // 8)

    def __getitem__(self, index):
        if not 0 <= index < self.size:
            raise IndexError("Index out of range")
        byte_index = index // 8
        bit_index = index % 8
        return bool(self.bytes[byte_index] & (1 << bit_index))
```

```
    def __setitem__(self, index, value):
        if not 0 <= index < self.size:
            raise IndexError("Index out of range")
        byte_index = index // 8
        bit_index = index % 8
        if value:
            self.bytes[byte_index] |= (1 << bit_index)
        else:
            self.bytes[byte_index] &= ~(1 << bit_index)

    def __len__(self):
        return self.size

# Numeric protocols
class ModularInt:
    """Integer with modular arithmetic"""
    def __init__(self, value, modulus):
        self.value = value % modulus
        self.modulus = modulus

    def __add__(self, other):
        if isinstance(other, ModularInt):
            if self.modulus != other.modulus:
                raise ValueError("Moduli must match")
            return ModularInt(self.value + other.value, self.modulus)
        return ModularInt(self.value + other, self.modulus)

    def __mul__(self, other):
        if isinstance(other, ModularInt):
            if self.modulus != other.modulus:
                raise ValueError("Moduli must match")
            return ModularInt(self.value * other.value, self.modulus)
        return ModularInt(self.value * other, self.modulus)

    def __pow__(self, exponent):
        # Fast modular exponentiation
        result = 1
        base = self.value
        exp = exponent

        while exp > 0:
            if exp % 2 == 1:
                result = (result * base) % self.modulus
            exp = exp >> 1
            base = (base * base) % self.modulus

        return ModularInt(result, self.modulus)
```

```
    def __repr__(self):
        return f"ModularInt({self.value}, mod={self.modulus})"
```

Chapter 6: Iterators, Generators, and Coroutines {#iterators-generators}

Advanced Iterator Patterns

python

```python
from itertools import tee, chain, islice
from typing import Iterator, Iterable, TypeVar, Tuple
import collections

T = TypeVar('T')

class PeekableIterator:
    """Iterator with peek functionality"""
    def __init__(self, iterable: Iterable[T]):
        self._iterator = iter(iterable)
        self._buffer = collections.deque()

    def __iter__(self):
        return self

    def __next__(self):
        if self._buffer:
            return self._buffer.popleft()
        return next(self._iterator)

    def peek(self, n=1):
        """Peek at next n items without consuming them"""
        while len(self._buffer) < n:
            try:
                self._buffer.append(next(self._iterator))
            except StopIteration:
                break
        return list(self._buffer)[:n]

    def has_next(self):
        """Check if iterator has more items"""
        try:
            self.peek()
            return True
        except:
            return False

# Infinite iterators
def fibonacci():
    """Infinite Fibonacci sequence generator"""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

def primes():
```

```
    """Infinite prime number generator using Sieve of Eratosthenes"""
    composites = {}
    n = 2
    while True:
        if n not in composites:
            yield n
            composites[n * n] = [n]
        else:
            for p in composites[n]:
                composites.setdefault(n + p, []).append(p)
            del composites[n]
        n += 1

# Generator with send() and close()
def running_average():
    """Coroutine that maintains running average"""
    total = 0.0
    count = 0
    average = 0.0

    while True:
        value = yield average
        if value is not None:
            total += value
            count += 1
            average = total / count

# Two-way generator
def echo_coroutine():
    """Coroutine that echoes values with transformation"""
    print("Coroutine started")
    try:
        while True:
            value = yield
            yield value.upper() if isinstance(value, str) else value * 2
    except GeneratorExit:
        print("Coroutine closing")

# Pipeline generators
def read_lines(filename):
    """Generator that reads lines from file"""
    with open(filename, 'r') as f:
        for line in f:
            yield line.strip()

def filter_lines(lines, pattern):
    """Filter lines containing pattern"""
```

```
    for line in lines:
        if pattern in line:
            yield line

def parse_numbers(lines):
    """Extract numbers from lines"""
    for line in lines:
        numbers = [int(x) for x in line.split() if x.isdigit()]
        yield from numbers

# Advanced generator expressions
class GeneratorChain:
    """Chain multiple generators efficiently"""
    def __init__(self, *generators):
        self.generators = generators

    def __iter__(self):
        for gen in self.generators:
            yield from gen

# Delegating generators
def flatten(nested_list):
    """Recursively flatten nested lists"""
    for item in nested_list:
        if isinstance(item, list):
            yield from flatten(item)
        else:
            yield item

# Generator with cleanup
def managed_generator(resource_factory):
    """Generator with resource management"""
    resource = resource_factory()
    try:
        while True:
            data = resource.read()
            if not data:
                break
            yield data
    finally:
        resource.close()
```

Coroutines and Async Generators

python

```python
import asyncio
from typing import AsyncIterator, AsyncIterable

# Basic async generator
async def async_counter(start=0, stop=10):
    """Async generator that counts with delay"""
    for i in range(start, stop):
        await asyncio.sleep(0.1)  # Simulate async work
        yield i

# Async generator with cleanup
class AsyncFileReader:
    """Async context manager and generator for file reading"""
    def __init__(self, filename):
        self.filename = filename
        self.file = None

    async def __aenter__(self):
        # In real async code, use aiofiles
        self.file = open(self.filename, 'r')
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()

    async def read_lines(self):
        """Async generator for reading lines"""
        for line in self.file:
            await asyncio.sleep(0.01)  # Simulate async I/O
            yield line.strip()

# Coroutine with multiple entry points
async def stateful_coroutine():
    """Coroutine that maintains state between calls"""
    state = {"count": 0, "total": 0}

    while True:
        command = yield state

        if command is None:
            continue
        elif command == "increment":
            state["count"] += 1
        elif command == "reset":
            state["count"] = 0
```

```
            state["total"] = 0
        elif isinstance(command, (int, float)):
            state["total"] += command

# Async generator with error handling
async def resilient_data_stream(sources):
    """Stream data from multiple sources with error handling"""
    for source in sources:
        try:
            async for data in source:
                yield data
        except Exception as e:
            print(f"Error in source {source}: {e}")
            continue

# Bidirectional async communication
class AsyncPipeline:
    """Async pipeline for data processing"""
    def __init__(self):
        self.processors = []

    def add_processor(self, processor):
        self.processors.append(processor)
        return self

    async def process(self, data_stream):
        """Process data through pipeline"""
        current_stream = data_stream

        for processor in self.processors:
            current_stream = processor(current_stream)

        async for result in current_stream:
            yield result

# Example processors
async def async_filter(stream, predicate):
    """Async filter processor"""
    async for item in stream:
        if await predicate(item):
            yield item

async def async_map(stream, func):
    """Async map processor"""
    async for item in stream:
        result = await func(item)
        yield result
```

```python
# Async generator comprehension
async def fetch_data(urls):
    """Fetch data from multiple URLs concurrently"""
    async def fetch_one(session, url):
        # Simulated async fetch
        await asyncio.sleep(0.1)
        return f"Data from {url}"

    # In real code, use aiohttp
    tasks = [fetch_one(None, url) for url in urls]
    results = await asyncio.gather(*tasks)

    for result in results:
        yield result
```

Chapter 7: Concurrent Programming {#concurrent-programming}

Threading Advanced Patterns

python

```python
import threading
import queue
import time
from concurrent.futures import ThreadPoolExecutor, as_completed
from contextlib import contextmanager
import weakref

# Thread-safe singleton with double-checked locking
class ThreadSafeSingleton:
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        if cls._instance is None:
            with cls._lock:
                if cls._instance is None:
                    cls._instance = super().__new__(cls)
        return cls._instance

# Read-Write lock implementation
class ReadWriteLock:
    """Multiple readers, single writer lock"""
    def __init__(self):
        self._read_ready = threading.Semaphore(1)
        self._readers = 0
        self._writers = 0
        self._read_counter_lock = threading.Lock()
        self._write_lock = threading.Lock()

    @contextmanager
    def read_lock(self):
        self._read_ready.acquire()
        try:
            self._read_counter_lock.acquire()
            try:
                self._readers += 1
                if self._readers == 1:
                    self._write_lock.acquire()
            finally:
                self._read_counter_lock.release()
        finally:
            self._read_ready.release()

        try:
            yield
        finally:
```

```
            self._read_counter_lock.acquire()
            try:
                self._readers -= 1
                if self._readers == 0:
                    self._write_lock.release()
            finally:
                self._read_counter_lock.release()

    @contextmanager
    def write_lock(self):
        self._read_ready.acquire()
        self._write_lock.acquire()
        try:
            yield
        finally:
            self._write_lock.release()
            self._read_ready.release()

# Thread pool with custom task queue
class PriorityThreadPool:
    """Thread pool that processes tasks by priority"""
    def __init__(self, num_threads=4):
        self.queue = queue.PriorityQueue()
        self.threads = []
        self.shutdown_event = threading.Event()

        for _ in range(num_threads):
            thread = threading.Thread(target=self._worker)
            thread.daemon = True
            thread.start()
            self.threads.append(thread)

    def _worker(self):
        while not self.shutdown_event.is_set():
            try:
                priority, task, args, kwargs, future = self.queue.get(timeout=1)
                try:
                    result = task(*args, **kwargs)
                    future.set_result(result)
                except Exception as e:
                    future.set_exception(e)
                finally:
                    self.queue.task_done()
            except queue.Empty:
                continue

    def submit(self, task, *args, priority=0, **kwargs):
```

```
        future = threading.Event()
        future.result = None
        future.exception = None

        class Future:
            def set_result(self, result):
                future.result = result
                future.set()

            def set_exception(self, exc):
                future.exception = exc
                future.set()

            def result(self, timeout=None):
                if future.wait(timeout):
                    if future.exception:
                        raise future.exception
                    return future.result
                raise TimeoutError()

        future_obj = Future()
        self.queue.put((priority, task, args, kwargs, future_obj))
        return future_obj

    def shutdown(self):
        self.shutdown_event.set()
        for thread in self.threads:
            thread.join()

# Producer-Consumer with backpressure
class BoundedBuffer:
    """Thread-safe bounded buffer with backpressure"""
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = collections.deque()
        self.lock = threading.Lock()
        self.not_full = threading.Condition(self.lock)
        self.not_empty = threading.Condition(self.lock)

    def put(self, item, timeout=None):
        with self.not_full:
            while len(self.buffer) >= self.capacity:
                if not self.not_full.wait(timeout):
                    raise TimeoutError("Buffer is full")

            self.buffer.append(item)
            self.not_empty.notify()
```

```
    def get(self, timeout=None):
        with self.not_empty:
            while not self.buffer:
                if not self.not_empty.wait(timeout):
                    raise TimeoutError("Buffer is empty")

            item = self.buffer.popleft()
            self.not_full.notify()
            return item

# Thread-local storage pattern
class ThreadLocalStack:
    """Thread-local stack implementation"""
    def __init__(self):
        self._local = threading.local()

    def _get_stack(self):
        if not hasattr(self._local, 'stack'):
            self._local.stack = []
        return self._local.stack

    def push(self, item):
        self._get_stack().append(item)

    def pop(self):
        stack = self._get_stack()
        if not stack:
            raise IndexError("pop from empty stack")
        return stack.pop()

    def top(self):
        stack = self._get_stack()
        if not stack:
            return None
        return stack[-1]
```

Multiprocessing Patterns

python

```python
import multiprocessing as mp
from multiprocessing import Process, Queue, Pool, Manager, Value, Array
import ctypes
import pickle

# Shared memory with complex data structures
class SharedCounter:
    """Process-safe counter using shared memory"""
    def __init__(self, init_value=0):
        self.val = Value(ctypes.c_int, init_value)
        self.lock = mp.Lock()

    def increment(self):
        with self.lock:
            self.val.value += 1

    def decrement(self):
        with self.lock:
            self.val.value -= 1

    @property
    def value(self):
        with self.lock:
            return self.val.value

# Process pool with initialization
def init_worker(shared_data):
    """Initialize worker process with shared data"""
    global worker_data
    worker_data = shared_data

def process_item(item):
    """Process item using shared worker data"""
    # Access global worker_data
    return item * worker_data.value

# Custom process pool executor
class ProcessPoolExecutor:
    """Enhanced process pool with better control"""
    def __init__(self, max_workers=None, initializer=None, initargs=()):
        self.max_workers = max_workers or mp.cpu_count()
        self.pool = Pool(
            processes=self.max_workers,
            initializer=initializer,
            initargs=initargs
        )
```

```
        self.futures = {}

    def submit(self, fn, *args, **kwargs):
        """Submit task to pool"""
        future = self.pool.apply_async(fn, args, kwargs)
        self.futures[future] = True
        return future

    def map(self, fn, iterable, chunksize=None):
        """Map function over iterable"""
        return self.pool.map(fn, iterable, chunksize=chunksize)

    def shutdown(self, wait=True):
        """Shutdown the pool"""
        if wait:
            self.pool.close()
            self.pool.join()
        else:
            self.pool.terminate()

# Inter-process communication patterns
class MessageBroker:
    """Simple message broker for IPC"""
    def __init__(self):
        self.manager = Manager()
        self.topics = self.manager.dict()
        self.subscribers = self.manager.dict()
        self.lock = mp.Lock()

    def subscribe(self, topic, queue):
        """Subscribe queue to topic"""
        with self.lock:
            if topic not in self.subscribers:
                self.subscribers[topic] = self.manager.list()
            self.subscribers[topic].append(queue)

    def publish(self, topic, message):
        """Publish message to topic"""
        with self.lock:
            if topic in self.subscribers:
                for queue in self.subscribers[topic]:
                    try:
                        queue.put(message, block=False)
                    except:
                        pass  # Queue might be full

# Parallel map-reduce implementation
```

```python
class ParallelMapReduce:
    """Parallel map-reduce framework"""
    def __init__(self, num_workers=None):
        self.num_workers = num_workers or mp.cpu_count()

    def partition(self, data, num_partitions):
        """Partition data for workers"""
        partitions = [[] for _ in range(num_partitions)]
        for i, item in enumerate(data):
            partitions[i % num_partitions].append(item)
        return partitions

    def run(self, data, map_func, reduce_func):
        """Run map-reduce on data"""
        # Partition data
        partitions = self.partition(data, self.num_workers)

        # Map phase
        with Pool(self.num_workers) as pool:
            mapped_results = pool.map(map_func, partitions)

        # Reduce phase
        result = mapped_results[0]
        for partial_result in mapped_results[1:]:
            result = reduce_func(result, partial_result)

        return result

# Distributed task queue
class DistributedTaskQueue:
    """Simple distributed task queue"""
    def __init__(self, num_workers=4):
        self.task_queue = mp.Queue()
        self.result_queue = mp.Queue()
        self.workers = []

        for i in range(num_workers):
            worker = mp.Process(
                target=self._worker,
                args=(self.task_queue, self.result_queue)
            )
            worker.start()
            self.workers.append(worker)

    @staticmethod
    def _worker(task_queue, result_queue):
        """Worker process"""
```

```
        while True:
            task = task_queue.get()
            if task is None:  # Poison pill
                break

            func, args, kwargs, task_id = task
            try:
                result = func(*args, **kwargs)
                result_queue.put((task_id, result, None))
            except Exception as e:
                result_queue.put((task_id, None, e))

    def submit(self, func, *args, **kwargs):
        """Submit task to queue"""
        task_id = id((func, args, kwargs))
        self.task_queue.put((func, args, kwargs, task_id))
        return task_id

    def get_result(self, timeout=None):
        """Get result from queue"""
        return self.result_queue.get(timeout=timeout)

    def shutdown(self):
        """Shutdown all workers"""
        for _ in self.workers:
            self.task_queue.put(None)
        for worker in self.workers:
            worker.join()
```

Chapter 8: Asynchronous Programming {#async-programming}

Advanced Async Patterns

python

```python
import asyncio
import aiohttp
from typing import List, Dict, Any, Coroutine
import weakref
from contextlib import asynccontextmanager
import time

# Async context manager for rate limiting
class AsyncRateLimiter:
    """Async rate limiter using token bucket algorithm"""
    def __init__(self, rate: int, capacity: int):
        self.rate = rate
        self.capacity = capacity
        self.tokens = capacity
        self.last_update = time.time()
        self._lock = asyncio.Lock()

    async def acquire(self, tokens: int = 1):
        async with self._lock:
            while self.tokens < tokens:
                now = time.time()
                elapsed = now - self.last_update
                self.tokens = min(
                    self.capacity,
                    self.tokens + elapsed * self.rate
                )
                self.last_update = now

                if self.tokens < tokens:
                    sleep_time = (tokens - self.tokens) / self.rate
                    await asyncio.sleep(sleep_time)

    @asynccontextmanager
    async def limit(self):
        await self.acquire()
        try:
            yield
        finally:
            pass

# Async connection pool
class AsyncConnectionPool:
    """Reusable async connection pool"""
    def __init__(self, factory, max_size=10):
        self.factory = factory
        self.max_size = max_size
```

```
        self.pool = asyncio.Queue(maxsize=max_size)
        self.size = 0
        self._lock = asyncio.Lock()

    async def acquire(self):
        try:
            # Try to get from pool
            conn = self.pool.get_nowait()
        except asyncio.QueueEmpty:
            # Create new connection if under limit
            async with self._lock:
                if self.size < self.max_size:
                    conn = await self.factory()
                    self.size += 1
                else:
                    # Wait for available connection
                    conn = await self.pool.get()

        return conn

    async def release(self, conn):
        await self.pool.put(conn)

    @asynccontextmanager
    async def connection(self):
        conn = await self.acquire()
        try:
            yield conn
        finally:
            await self.release(conn)

# Async task scheduler
class AsyncScheduler:
    """Schedule async tasks with delays and intervals"""
    def __init__(self):
        self.tasks = []
        self.running = False

    def schedule_once(self, delay: float, coro: Coroutine):
        """Schedule task to run once after delay"""
        async def wrapper():
            await asyncio.sleep(delay)
            await coro

        task = asyncio.create_task(wrapper())
        self.tasks.append(task)
        return task
```

```python
    def schedule_interval(self, interval: float, coro_func):
        """Schedule task to run at intervals"""
        async def wrapper():
            while self.running:
                await coro_func()
                await asyncio.sleep(interval)

        task = asyncio.create_task(wrapper())
        self.tasks.append(task)
        return task

    async def start(self):
        self.running = True

    async def stop(self):
        self.running = False
        await asyncio.gather(*self.tasks, return_exceptions=True)

# Async event emitter
class AsyncEventEmitter:
    """Async version of event emitter pattern"""
    def __init__(self):
        self._handlers = {}
        self._lock = asyncio.Lock()

    async def on(self, event: str, handler: Coroutine):
        async with self._lock:
            if event not in self._handlers:
                self._handlers[event] = []
            self._handlers[event].append(handler)

    async def emit(self, event: str, *args, **kwargs):
        handlers = self._handlers.get(event, [])
        if handlers:
            await asyncio.gather(
                *(handler(*args, **kwargs) for handler in handlers),
                return_exceptions=True
            )

    async def remove_handler(self, event: str, handler: Coroutine):
        async with self._lock:
            if event in self._handlers:
                self._handlers[event].remove(handler)

# Async retry with exponential backoff
async def async_retry(
```

```
    coro_func,
    max_attempts=3,
    base_delay=1,
    max_delay=60,
    exponential_base=2
):
    """Retry async function with exponential backoff"""
    attempt = 0
    delay = base_delay

    while attempt < max_attempts:
        try:
            return await coro_func()
        except Exception as e:
            attempt += 1
            if attempt >= max_attempts:
                raise

            await asyncio.sleep(min(delay, max_delay))
            delay *= exponential_base

# Async pipeline processing
class AsyncPipeline:
    """Build async processing pipelines"""
    def __init__(self):
        self.stages = []

    def add_stage(self, processor):
        self.stages.append(processor)
        return self

    async def process(self, data):
        result = data
        for stage in self.stages:
            if asyncio.iscoroutinefunction(stage):
                result = await stage(result)
            else:
                result = stage(result)
        return result

    async def process_stream(self, stream):
        async for item in stream:
            yield await self.process(item)

# Async cache with TTL
class AsyncCache:
    """Async cache with time-to-live"""
```

```
    def __init__(self, ttl=300):
        self.ttl = ttl
        self.cache = {}
        self.timestamps = {}
        self._lock = asyncio.Lock()

    async def get(self, key, factory=None):
        async with self._lock:
            if key in self.cache:
                if time.time() - self.timestamps[key] < self.ttl:
                    return self.cache[key]
                else:
                    del self.cache[key]
                    del self.timestamps[key]

        if factory:
            value = await factory()
            await self.set(key, value)
            return value

        return None

    async def set(self, key, value):
        async with self._lock:
            self.cache[key] = value
            self.timestamps[key] = time.time()

    async def invalidate(self, key):
        async with self._lock:
            self.cache.pop(key, None)
            self.timestamps.pop(key, None)

# Async worker pool
class AsyncWorkerPool:
    """Pool of async workers processing from queue"""
    def __init__(self, num_workers=5):
        self.num_workers = num_workers
        self.queue = asyncio.Queue()
        self.workers = []
        self.running = False

    async def _worker(self):
        while self.running:
            try:
                task = await asyncio.wait_for(
                    self.queue.get(),
                    timeout=1.0
```

```
                )
                await task
            except asyncio.TimeoutError:
                continue
            except Exception as e:
                print(f"Worker error: {e}")

    async def start(self):
        self.running = True
        self.workers = [
            asyncio.create_task(self._worker())
            for _ in range(self.num_workers)
        ]

    async def submit(self, coro):
        await self.queue.put(coro)

    async def stop(self):
        self.running = False
        await asyncio.gather(*self.workers, return_exceptions=True)
```

Async Streaming and Protocols

python

```python
# Async stream processing
class AsyncStreamProcessor:
    """Process async streams with transformations"""

    @staticmethod
    async def map(stream, func):
        """Async map over stream"""
        async for item in stream:
            if asyncio.iscoroutinefunction(func):
                yield await func(item)
            else:
                yield func(item)

    @staticmethod
    async def filter(stream, predicate):
        """Async filter stream"""
        async for item in stream:
            if asyncio.iscoroutinefunction(predicate):
                if await predicate(item):
                    yield item
            else:
                if predicate(item):
                    yield item

    @staticmethod
    async def batch(stream, size):
        """Batch stream items"""
        batch = []
        async for item in stream:
            batch.append(item)
            if len(batch) >= size:
                yield batch
                batch = []
        if batch:
            yield batch

    @staticmethod
    async def throttle(stream, rate):
        """Throttle stream to specific rate"""
        interval = 1.0 / rate
        last_time = 0

        async for item in stream:
            current_time = time.time()
            elapsed = current_time - last_time
```

```
            if elapsed < interval:
                await asyncio.sleep(interval - elapsed)

            yield item
            last_time = time.time()

# Async protocol implementation
class AsyncProtocol:
    """Base class for async protocols"""
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer
        self.buffer = bytearray()

    async def read_until(self, delimiter):
        """Read until delimiter is found"""
        while delimiter not in self.buffer:
            chunk = await self.reader.read(1024)
            if not chunk:
                raise ConnectionError("Connection closed")
            self.buffer.extend(chunk)

        index = self.buffer.index(delimiter)
        data = bytes(self.buffer[:index])
        self.buffer = self.buffer[index + len(delimiter):]
        return data

    async def write_message(self, message):
        """Write message with proper encoding"""
        self.writer.write(message.encode() + b'\n')
        await self.writer.drain()

    async def close(self):
        """Close the connection"""
        self.writer.close()
        await self.writer.wait_closed()

# Async server with protocol handling
class AsyncServer:
    """Async TCP server with custom protocol"""
    def __init__(self, host='127.0.0.1', port=8888):
        self.host = host
        self.port = port
        self.clients = weakref.WeakSet()
        self.running = False

    async def handle_client(self, reader, writer):
```

```
    """Handle individual client connection"""
    client = AsyncProtocol(reader, writer)
    self.clients.add(client)

    try:
        while self.running:
            try:
                message = await asyncio.wait_for(
                    client.read_until(b'\n'),
                    timeout=30.0
                )
                response = await self.process_message(message)
                await client.write_message(response)
            except asyncio.TimeoutError:
                await client.write_message("PING")
            except Exception as e:
                print(f"Client error: {e}")
                break
    finally:
        await client.close()
        self.clients.discard(client)

async def process_message(self, message):
    """Process incoming message"""
    return f"ECHO: {message.decode()}"

async def start(self):
    """Start the server"""
    self.running = True
    server = await asyncio.start_server(
        self.handle_client,
        self.host,
        self.port
    )

    async with server:
        await server.serve_forever()

async def stop(self):
    """Stop the server"""
    self.running = False
    # Close all client connections
    for client in list(self.clients):
        await client.close()
```

Chapter 9: Design Patterns in Python {#design-patterns}

Creational Patterns

python

```python
from abc import ABC, abstractmethod
from typing import Dict, Type, Any
import copy

# Advanced Singleton with metaclass
class SingletonMeta(type):
    """Thread-safe Singleton metaclass"""
    _instances = {}
    _lock = threading.Lock()

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            with cls._lock:
                if cls not in cls._instances:
                    instance = super().__call__(*args, **kwargs)
                    cls._instances[cls] = instance
        return cls._instances[cls]

# Factory Pattern with registration
class ShapeFactory:
    """Factory with automatic registration"""
    _shapes: Dict[str, Type['Shape']] = {}

    @classmethod
    def register(cls, name: str):
        """Decorator to register shapes"""
        def wrapper(shape_class):
            cls._shapes[name] = shape_class
            return shape_class
        return wrapper

    @classmethod
    def create(cls, name: str, **kwargs) -> 'Shape':
        shape_class = cls._shapes.get(name)
        if not shape_class:
            raise ValueError(f"Unknown shape: {name}")
        return shape_class(**kwargs)

@ShapeFactory.register("circle")
class Circle:
    def __init__(self, radius):
        self.radius = radius

# Abstract Factory
class GUIFactory(ABC):
    """Abstract factory for GUI elements"""
```

```
    @abstractmethod
    def create_button(self) -> 'Button':
        pass

    @abstractmethod
    def create_window(self) -> 'Window':
        pass

class WindowsFactory(GUIFactory):
    def create_button(self):
        return WindowsButton()

    def create_window(self):
        return WindowsWindow()

class MacFactory(GUIFactory):
    def create_button(self):
        return MacButton()

    def create_window(self):
        return MacWindow()

# Builder Pattern with fluent interface
class QueryBuilder:
    """SQL query builder with fluent interface"""
    def __init__(self):
        self._reset()

    def _reset(self):
        self._select = []
        self._from = None
        self._where = []
        self._group_by = []
        self._order_by = []
        self._limit = None

    def select(self, *fields):
        self._select.extend(fields)
        return self

    def from_table(self, table):
        self._from = table
        return self

    def where(self, condition):
        self._where.append(condition)
        return self
```

```
    def group_by(self, *fields):
        self._group_by.extend(fields)
        return self

    def order_by(self, field, direction='ASC'):
        self._order_by.append(f"{field} {direction}")
        return self

    def limit(self, count):
        self._limit = count
        return self

    def build(self):
        query_parts = []

        # SELECT
        if self._select:
            query_parts.append(f"SELECT {', '.join(self._select)}")
        else:
            query_parts.append("SELECT *")

        # FROM
        if self._from:
            query_parts.append(f"FROM {self._from}")

        # WHERE
        if self._where:
            query_parts.append(f"WHERE {' AND '.join(self._where)}")

        # GROUP BY
        if self._group_by:
            query_parts.append(f"GROUP BY {', '.join(self._group_by)}")

        # ORDER BY
        if self._order_by:
            query_parts.append(f"ORDER BY {', '.join(self._order_by)}")

        # LIMIT
        if self._limit:
            query_parts.append(f"LIMIT {self._limit}")

        return ' '.join(query_parts)

# Prototype Pattern with deep copy
class Prototype:
    """Base prototype class"""
```

```
    def clone(self):
        return copy.deepcopy(self)

class DocumentPrototype(Prototype):
    def __init__(self):
        self.pages = []
        self.metadata = {}
        self.formatting = {
            'font': 'Arial',
            'size': 12,
            'margins': [1, 1, 1, 1]
        }

    def add_page(self, content):
        self.pages.append(content)
        return self

# Object Pool Pattern
class ObjectPool:
    """Generic object pool"""
    def __init__(self, factory, max_size=10):
        self._factory = factory
        self._max_size = max_size
        self._pool = []
        self._in_use = weakref.WeakSet()
        self._lock = threading.Lock()

    def acquire(self):
        with self._lock:
            if self._pool:
                obj = self._pool.pop()
            else:
                obj = self._factory()

            self._in_use.add(obj)
            return obj

    def release(self, obj):
        with self._lock:
            if obj in self._in_use:
                self._in_use.remove(obj)
                if len(self._pool) < self._max_size:
                    self._reset_object(obj)
                    self._pool.append(obj)

    def _reset_object(self, obj):
        """Reset object to initial state"""
```

```
        if hasattr(obj, 'reset'):
            obj.reset()
```

Structural Patterns

python

```python
# Advanced Decorator Pattern
class Component(ABC):
    """Base component interface"""
    @abstractmethod
    def operation(self) -> str:
        pass

class ConcreteComponent(Component):
    def operation(self) -> str:
        return "ConcreteComponent"

class Decorator(Component):
    """Base decorator"""
    def __init__(self, component: Component):
        self._component = component

    def operation(self) -> str:
        return self._component.operation()

class TimingDecorator(Decorator):
    """Add timing to operations"""
    def operation(self) -> str:
        import time
        start = time.time()
        result = self._component.operation()
        end = time.time()
        return f"{result} (took {end-start:.4f}s)"

class CachingDecorator(Decorator):
    """Add caching to operations"""
    def __init__(self, component: Component):
        super().__init__(component)
        self._cache = {}

    def operation(self) -> str:
        if 'result' not in self._cache:
            self._cache['result'] = self._component.operation()
        return f"{self._cache['result']} (cached)"

# Adapter Pattern with multiple adaptees
class ModernPaymentInterface(ABC):
    @abstractmethod
    def make_payment(self, amount: float) -> bool:
        pass

class OldPaymentSystem:
```

```python
    def pay(self, sum: int) -> str:
        return "success" if sum > 0 else "failure"

class NewPaymentGateway:
    def process_payment(self, value: float) -> dict:
        return {"status": "approved", "amount": value}

class PaymentAdapter(ModernPaymentInterface):
    """Adapter for multiple payment systems"""
    def __init__(self, payment_system):
        self.payment_system = payment_system

    def make_payment(self, amount: float) -> bool:
        if isinstance(self.payment_system, OldPaymentSystem):
            result = self.payment_system.pay(int(amount * 100))
            return result == "success"
        elif isinstance(self.payment_system, NewPaymentGateway):
            result = self.payment_system.process_payment(amount)
            return result["status"] == "approved"
        else:
            raise ValueError("Unknown payment system")

# Composite Pattern with visitors
class FileSystemComponent(ABC):
    @abstractmethod
    def get_size(self) -> int:
        pass

    @abstractmethod
    def accept(self, visitor: 'Visitor'):
        pass

class File(FileSystemComponent):
    def __init__(self, name: str, size: int):
        self.name = name
        self.size = size

    def get_size(self) -> int:
        return self.size

    def accept(self, visitor):
        visitor.visit_file(self)

class Directory(FileSystemComponent):
    def __init__(self, name: str):
        self.name = name
        self.children: List[FileSystemComponent] = []
```

```python
    def add(self, component: FileSystemComponent):
        self.children.append(component)

    def get_size(self) -> int:
        return sum(child.get_size() for child in self.children)

    def accept(self, visitor):
        visitor.visit_directory(self)
        for child in self.children:
            child.accept(visitor)

# Proxy Pattern with lazy loading
class DataProxy:
    """Proxy for expensive data operations"""
    def __init__(self, data_source):
        self._data_source = data_source
        self._data = None
        self._loaded = False

    def _ensure_loaded(self):
        if not self._loaded:
            print("Loading data...")
            self._data = self._data_source.load()
            self._loaded = True

    def get_data(self):
        self._ensure_loaded()
        return self._data

    def process(self):
        self._ensure_loaded()
        return self._data_source.process(self._data)

# Flyweight Pattern
class CharacterFlyweight:
    """Flyweight for character rendering"""
    _characters = {}

    def __new__(cls, char: str, font: str):
        key = (char, font)
        if key not in cls._characters:
            instance = super().__new__(cls)
            instance.char = char
            instance.font = font
```

```
            cls._characters[key] = instance
        return cls._characters[key]
```

Behavioral Patterns

python

```python
# Advanced Observer Pattern
class EventManager:
    """Event manager with priority and filtering"""
    def __init__(self):
        self._observers = {}
        self._event_queue = []
        self._processing = False

    def subscribe(self, event_type: str, callback, priority: int = 0):
        if event_type not in self._observers:
            self._observers[event_type] = []

        self._observers[event_type].append((priority, callback))
        self._observers[event_type].sort(key=lambda x: x[0], reverse=True)

    def unsubscribe(self, event_type: str, callback):
        if event_type in self._observers:
            self._observers[event_type] = [
                (p, c) for p, c in self._observers[event_type]
                if c != callback
            ]

    def emit(self, event_type: str, data: Any = None):
        self._event_queue.append((event_type, data))

        if not self._processing:
            self._process_events()

    def _process_events(self):
        self._processing = True

        while self._event_queue:
            event_type, data = self._event_queue.pop(0)

            if event_type in self._observers:
                for priority, callback in self._observers[event_type]:
                    try:
                        callback(data)
                    except Exception as e:
                        print(f"Observer error: {e}")

        self._processing = False

# Strategy Pattern with context
class SortStrategy(ABC):
    @abstractmethod
```

```python
    def sort(self, data: List[Any]) -> List[Any]:
        pass

class QuickSortStrategy(SortStrategy):
    def sort(self, data: List[Any]) -> List[Any]:
        if len(data) <= 1:
            return data
        pivot = data[len(data) // 2]
        left = [x for x in data if x < pivot]
        middle = [x for x in data if x == pivot]
        right = [x for x in data if x > pivot]
        return self.sort(left) + middle + self.sort(right)

class MergeSortStrategy(SortStrategy):
    def sort(self, data: List[Any]) -> List[Any]:
        if len(data) <= 1:
            return data

        mid = len(data) // 2
        left = self.sort(data[:mid])
        right = self.sort(data[mid:])

        return self._merge(left, right)

    def _merge(self, left: List[Any], right: List[Any]) -> List[Any]:
        result = []
        i = j = 0

        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1

        result.extend(left[i:])
        result.extend(right[j:])
        return result

class Sorter:
    def __init__(self, strategy: SortStrategy = None):
        self._strategy = strategy or QuickSortStrategy()

    def set_strategy(self, strategy: SortStrategy):
        self._strategy = strategy
```

```python
    def sort(self, data: List[Any]) -> List[Any]:
        return self._strategy.sort(data.copy())

# Chain of Responsibility
class Handler(ABC):
    def __init__(self):
        self._next_handler = None

    def set_next(self, handler: 'Handler') -> 'Handler':
        self._next_handler = handler
        return handler

    @abstractmethod
    def handle(self, request: Any) -> Any:
        if self._next_handler:
            return self._next_handler.handle(request)
        return None

class AuthenticationHandler(Handler):
    def handle(self, request: Dict[str, Any]) -> Any:
        if 'token' not in request:
            return {"error": "Authentication required"}

        # Validate token
        if request['token'] != "valid_token":
            return {"error": "Invalid token"}

        return super().handle(request)

class AuthorizationHandler(Handler):
    def handle(self, request: Dict[str, Any]) -> Any:
        if request.get('role') != 'admin':
            return {"error": "Insufficient permissions"}

        return super().handle(request)

class ValidationHandler(Handler):
    def handle(self, request: Dict[str, Any]) -> Any:
        required_fields = ['user_id', 'action']
        for field in required_fields:
            if field not in request:
                return {"error": f"Missing field: {field}"}

        return super().handle(request)

# Command Pattern with undo/redo
class Command(ABC):
```

```
    @abstractmethod
    def execute(self):
        pass

    @abstractmethod
    def undo(self):
        pass

class TextEditor:
    def __init__(self):
        self.content = ""
        self._history = []
        self._redo_stack = []

    def write(self, text: str):
        command = WriteCommand(self, text)
        command.execute()
        self._history.append(command)
        self._redo_stack.clear()

    def delete(self, count: int):
        command = DeleteCommand(self, count)
        command.execute()
        self._history.append(command)
        self._redo_stack.clear()

    def undo(self):
        if self._history:
            command = self._history.pop()
            command.undo()
            self._redo_stack.append(command)

    def redo(self):
        if self._redo_stack:
            command = self._redo_stack.pop()
            command.execute()
            self._history.append(command)

class WriteCommand(Command):
    def __init__(self, editor: TextEditor, text: str):
        self.editor = editor
        self.text = text
        self.position = len(editor.content)

    def execute(self):
        self.editor.content += self.text
```

```
    def undo(self):
        self.editor.content = self.editor.content[:self.position]

class DeleteCommand(Command):
    def __init__(self, editor: TextEditor, count: int):
        self.editor = editor
        self.count = count
        self.deleted_text = ""

    def execute(self):
        self.deleted_text = self.editor.content[-self.count:]
        self.editor.content = self.editor.content[:-self.count]

    def undo(self):
        self.editor.content += self.deleted_text

# State Pattern
class State(ABC):
    @abstractmethod
    def handle(self, context: 'StateMachine'):
        pass

class StateMachine:
    def __init__(self, initial_state: State):
        self._state = initial_state
        self._state_history = [initial_state]

    def transition_to(self, state: State):
        print(f"Transitioning from {self._state.__class__.__name__} to {state.__class__.__name_
        self._state = state
        self._state_history.append(state)

    def handle(self):
        self._state.handle(self)
```

Chapter 10: Testing and Debugging {#testing-debugging}

Advanced Testing Techniques

python

```python
import unittest
import pytest
from unittest.mock import Mock, patch, MagicMock
from hypothesis import given, strategies as st
import asyncio

# Parameterized testing with pytest
class MathOperations:
    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def multiply(a, b):
        return a * b

    @staticmethod
    def divide(a, b):
        if b == 0:
            raise ValueError("Division by zero")
        return a / b

@pytest.mark.parametrize("a,b,expected", [
    (2, 3, 5),
    (-1, 1, 0),
    (0, 0, 0),
    (1.5, 2.5, 4.0),
])
def test_add(a, b, expected):
    assert MathOperations.add(a, b) == expected

# Property-based testing with Hypothesis
@given(
    a=st.integers(),
    b=st.integers()
)
def test_add_commutative(a, b):
    """Test that addition is commutative"""
    assert MathOperations.add(a, b) == MathOperations.add(b, a)

@given(
    a=st.floats(allow_nan=False, allow_infinity=False),
    b=st.floats(allow_nan=False, allow_infinity=False).filter(lambda x: x != 0)
)
def test_divide_multiply_inverse(a, b):
    """Test that division and multiplication are inverse operations"""
```

```
    result = MathOperations.divide(a, b)
    assert abs(MathOperations.multiply(result, b) - a) < 1e-10

# Advanced mocking techniques
class DatabaseService:
    def __init__(self, connection):
        self.connection = connection

    def get_user(self, user_id):
        cursor = self.connection.cursor()
        cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))
        return cursor.fetchone()

    def save_user(self, user_data):
        cursor = self.connection.cursor()
        cursor.execute(
            "INSERT INTO users (name, email) VALUES (?, ?)",
            (user_data['name'], user_data['email'])
        )
        self.connection.commit()
        return cursor.lastrowid

class TestDatabaseService(unittest.TestCase):
    def setUp(self):
        self.mock_connection = Mock()
        self.mock_cursor = Mock()
        self.mock_connection.cursor.return_value = self.mock_cursor
        self.service = DatabaseService(self.mock_connection)

    def test_get_user(self):
        # Setup mock
        expected_user = ('John', 'john@example.com')
        self.mock_cursor.fetchone.return_value = expected_user

        # Test
        result = self.service.get_user(1)

        # Assertions
        self.mock_cursor.execute.assert_called_once_with(
            "SELECT * FROM users WHERE id = ?", (1,)
        )
        self.assertEqual(result, expected_user)

    @patch('database_service.time')
    def test_save_user_with_timestamp(self, mock_time):
        # Mock time
        mock_time.time.return_value = 1234567890
```

```
        self.mock_cursor.lastrowid = 42

        # Test
        user_data = {'name': 'Jane', 'email': 'jane@example.com'}
        result = self.service.save_user(user_data)

        # Assertions
        self.assertEqual(result, 42)
        self.mock_connection.commit.assert_called_once()

# Testing async code
class AsyncService:
    async def fetch_data(self, url):
        async with aiohttp.ClientSession() as session:
            async with session.get(url) as response:
                return await response.json()

    async def process_urls(self, urls):
        tasks = [self.fetch_data(url) for url in urls]
        return await asyncio.gather(*tasks)

@pytest.mark.asyncio
async def test_fetch_data():
    service = AsyncService()

    # Mock aiohttp
    with patch('aiohttp.ClientSession') as mock_session:
        mock_response = AsyncMock()
        mock_response.json = AsyncMock(return_value={'data': 'test'})

        mock_session.return_value.__aenter__.return_value.get.return_value.__aenter__.return_va

        result = await service.fetch_data('http://example.com')
        assert result == {'data': 'test'}

# Custom test fixtures
@pytest.fixture
def database():
    """Database fixture with setup and teardown"""
    import sqlite3
    conn = sqlite3.connect(':memory:')
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE users (
            id INTEGER PRIMARY KEY,
            name TEXT,
            email TEXT
```

```
        )
    ''')
    conn.commit()

    yield conn

    conn.close()

@pytest.fixture
def populated_database(database):
    """Database with test data"""
    cursor = database.cursor()
    test_users = [
        ('Alice', 'alice@example.com'),
        ('Bob', 'bob@example.com'),
        ('Charlie', 'charlie@example.com')
    ]
    cursor.executemany(
        "INSERT INTO users (name, email) VALUES (?, ?)",
        test_users
    )
    database.commit()
    return database

# Test doubles and spies
class EmailSpy:
    """Spy to track email sending"""
    def __init__(self):
        self.sent_emails = []

    def send(self, to, subject, body):
        self.sent_emails.append({
            'to': to,
            'subject': subject,
            'body': body,
            'timestamp': time.time()
        })
        return True

    def get_sent_count(self):
        return len(self.sent_emails)

    def was_email_sent_to(self, email):
        return any(e['to'] == email for e in self.sent_emails)

# Testing with context managers
class TestContextManager(unittest.TestCase):
```

```
    def test_file_handling(self):
        with patch('builtins.open', create=True) as mock_open:
            mock_file = MagicMock()
            mock_open.return_value.__enter__.return_value = mock_file

            with open('test.txt', 'r') as f:
                f.read()

            mock_file.read.assert_called_once()
            mock_open.assert_called_with('test.txt', 'r')
```

Debugging and Profiling

python

```
import cProfile
import pstats
import tracemalloc
import logging
from functools import wraps
import sys
import linecache

# Custom debugging decorator
def debug(func):
    """Decorator to debug function calls"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Print function call info
        args_repr = [repr(a) for a in args]
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]
        signature = ", ".join(args_repr + kwargs_repr)
        print(f"Calling {func.__name__}({signature})")

        try:
            result = func(*args, **kwargs)
            print(f"{func.__name__} returned {result!r}")
            return result
        except Exception as e:
            print(f"{func.__name__} raised {e!r}")
            raise

    return wrapper

# Advanced logging configuration
class ColoredFormatter(logging.Formatter):
    """Colored log formatter"""

    COLORS = {
        'DEBUG': '\033[36m',    # Cyan
        'INFO': '\033[32m',     # Green
        'WARNING': '\033[33m',  # Yellow
        'ERROR': '\033[31m',    # Red
        'CRITICAL': '\033[35m', # Magenta
    }
    RESET = '\033[0m'

    def format(self, record):
        log_color = self.COLORS.get(record.levelname, self.RESET)
        record.levelname = f"{log_color}{record.levelname}{self.RESET}"
        return super().format(record)
```

```
def setup_logging():
    """Setup advanced logging configuration"""
    logger = logging.getLogger()
    logger.setLevel(logging.DEBUG)

    # Console handler with colors
    console_handler = logging.StreamHandler()
    console_handler.setFormatter(
        ColoredFormatter(
            '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
        )
    )

    # File handler with rotation
    from logging.handlers import RotatingFileHandler
    file_handler = RotatingFileHandler(
        'app.log',
        maxBytes=10*1024*1024,  # 10MB
        backupCount=5
    )
    file_handler.setFormatter(
        logging.Formatter(
            '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
        )
    )

    logger.addHandler(console_handler)
    logger.addHandler(file_handler)

# Memory profiling
class MemoryProfiler:
    """Context manager for memory profiling"""
    def __init__(self, top_n=10):
        self.top_n = top_n
        self.snapshot1 = None
        self.snapshot2 = None

    def __enter__(self):
        tracemalloc.start()
        self.snapshot1 = tracemalloc.take_snapshot()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.snapshot2 = tracemalloc.take_snapshot()
        top_stats = self.snapshot2.compare_to(self.snapshot1, 'lineno')
```

```
        print(f"[ Top {self.top_n} memory allocations ]")
        for stat in top_stats[:self.top_n]:
            print(stat)

        tracemalloc.stop()

# Performance profiling decorator
def profile(sort_by='cumulative', lines_to_print=10):
    """Decorator to profile function performance"""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            profiler = cProfile.Profile()
            profiler.enable()

            try:
                result = func(*args, **kwargs)
            finally:
                profiler.disable()

                stats = pstats.Stats(profiler)
                stats.sort_stats(sort_by)
                stats.print_stats(lines_to_print)

            return result
        return wrapper
    return decorator

# Custom exception hook
def custom_excepthook(exc_type, exc_value, exc_traceback):
    """Enhanced exception display"""
    if issubclass(exc_type, KeyboardInterrupt):
        sys.__excepthook__(exc_type, exc_value, exc_traceback)
        return

    print("=" * 80)
    print("EXCEPTION OCCURRED:")
    print("=" * 80)

    # Print exception type and value
    print(f"Type: {exc_type.__name__}")
    print(f"Value: {exc_value}")
    print()

    # Print traceback with context
    print("Traceback (most recent call last):")
    tb = exc_traceback
```

```
    while tb is not None:
        frame = tb.tb_frame
        filename = frame.f_code.co_filename
        line_number = tb.tb_lineno
        function_name = frame.f_code.co_name

        print(f'  File "{filename}", line {line_number}, in {function_name}')

        # Print source code line
        line = linecache.getline(filename, line_number, frame.f_globals)
        if line:
            print(f"    {line.strip()}")

        # Print local variables
        print("    Local variables:")
        for key, value in frame.f_locals.items():
            try:
                value_repr = repr(value)
                if len(value_repr) > 100:
                    value_repr = value_repr[:97] + "..."
                print(f"      {key} = {value_repr}")
            except:
                print(f"      {key} = <unable to get value>")

        tb = tb.tb_next

    print("=" * 80)

sys.excepthook = custom_excepthook

# Debugging utilities
class Debugger:
    """Interactive debugger utilities"""

    @staticmethod
    def trace_calls(func):
        """Trace all function calls"""
        def tracer(frame, event, arg):
            if event == 'call':
                code = frame.f_code
                print(f"Calling: {code.co_filename}:{code.co_name}")
            return tracer

        @wraps(func)
        def wrapper(*args, **kwargs):
            sys.settrace(tracer)
            try:
```

```
                result = func(*args, **kwargs)
            finally:
                sys.settrace(None)
            return result

        return wrapper

    @staticmethod
    def breakpoint():
        """Custom breakpoint implementation"""
        import pdb
        import inspect

        frame = inspect.currentframe().f_back
        namespace = frame.f_locals.copy()
        namespace.update(frame.f_globals)

        print("Entering debugger. Type 'help' for commands.")
        pdb.set_trace()
```

Chapter 11: Performance Optimization {#performance-optimization}

Profiling and Benchmarking

python

```python
import timeit
import memory_profiler
from line_profiler import LineProfiler
import numpy as np
from functools import lru_cache, cache
import numba

# Benchmarking utilities
class Benchmark:
    """Comprehensive benchmarking tool"""
    def __init__(self, name="Benchmark"):
        self.name = name
        self.results = []

    def run(self, func, *args, number=1000, **kwargs):
        """Run benchmark on function"""
        # Warm up
        func(*args, **kwargs)

        # Time execution
        timer = timeit.Timer(
            lambda: func(*args, **kwargs)
        )

        times = timer.repeat(repeat=5, number=number)

        result = {
            'function': func.__name__,
            'min': min(times),
            'max': max(times),
            'avg': sum(times) / len(times),
            'times': times
        }

        self.results.append(result)
        return result

    def compare(self):
        """Compare all benchmark results"""
        if not self.results:
            print("No results to compare")
            return

        print(f"\n{self.name} Results:")
        print("-" * 60)
```

```
        # Sort by average time
        sorted_results = sorted(self.results, key=lambda x: x['avg'])
        baseline = sorted_results[0]['avg']

        for result in sorted_results:
            ratio = result['avg'] / baseline
            print(f"{result['function']:20} | "
                  f"Avg: {result['avg']:.6f}s | "
                  f"Ratio: {ratio:.2f}x")

# Optimization techniques
class OptimizationExamples:
    """Examples of various optimization techniques"""

    # 1. Algorithm optimization
    @staticmethod
    def find_duplicates_naive(lst):
        """O(n²) approach"""
        duplicates = []
        for i in range(len(lst)):
            for j in range(i + 1, len(lst)):
                if lst[i] == lst[j] and lst[i] not in duplicates:
                    duplicates.append(lst[i])
        return duplicates

    @staticmethod
    def find_duplicates_optimized(lst):
        """O(n) approach using set"""
        seen = set()
        duplicates = set()
        for item in lst:
            if item in seen:
                duplicates.add(item)
            seen.add(item)
        return list(duplicates)

    # 2. Caching optimization
    @staticmethod
    def fibonacci_naive(n):
        """Naive recursive approach"""
        if n <= 1:
            return n
        return OptimizationExamples.fibonacci_naive(n-1) + OptimizationExamples.fibonacci_naive

    @staticmethod
    @lru_cache(maxsize=128)
    def fibonacci_cached(n):
```

```
        """Cached recursive approach"""
        if n <= 1:
            return n
        return OptimizationExamples.fibonacci_cached(n-1) + OptimizationExamples.fibonacci_cach

    # 3. Vectorization with NumPy
    @staticmethod
    def calculate_distances_loops(points):
        """Calculate pairwise distances using loops"""
        n = len(points)
        distances = np.zeros((n, n))
        for i in range(n):
            for j in range(n):
                distances[i, j] = np.sqrt(
                    sum((points[i][k] - points[j][k])**2
                        for k in range(len(points[0])))
                )
        return distances

    @staticmethod
    def calculate_distances_vectorized(points):
        """Calculate pairwise distances using vectorization"""
        points = np.array(points)
        # Broadcasting magic
        diff = points[:, np.newaxis, :] - points[np.newaxis, :, :]
        distances = np.sqrt(np.sum(diff**2, axis=2))
        return distances

# Memory optimization
class MemoryOptimizer:
    """Memory optimization techniques"""

    # Using __slots__ to reduce memory
    class RegularClass:
        def __init__(self, x, y, z):
            self.x = x
            self.y = y
            self.z = z

    class SlottedClass:
        __slots__ = ['x', 'y', 'z']

        def __init__(self, x, y, z):
            self.x = x
            self.y = y
            self.z = z
```

```
    @staticmethod
    def compare_memory_usage():
        """Compare memory usage of different approaches"""
        import sys

        regular = MemoryOptimizer.RegularClass(1, 2, 3)
        slotted = MemoryOptimizer.SlottedClass(1, 2, 3)

        print(f"Regular class size: {sys.getsizeof(regular.__dict__)} bytes")
        print(f"Slotted class size: {sys.getsizeof(slotted)} bytes")

# JIT compilation with Numba
@numba.jit(nopython=True)
def monte_carlo_pi_numba(n):
    """Calculate PI using Monte Carlo method with Numba"""
    count = 0
    for i in range(n):
        x = np.random.random()
        y = np.random.random()
        if x*x + y*y <= 1:
            count += 1
    return 4.0 * count / n

def monte_carlo_pi_pure(n):
    """Calculate PI using Monte Carlo method in pure Python"""
    import random
    count = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if x*x + y*y <= 1:
            count += 1
    return 4.0 * count / n

# String optimization
class StringOptimizer:
    """String handling optimizations"""

    @staticmethod
    def concatenate_naive(strings):
        """Naive string concatenation"""
        result = ""
        for s in strings:
            result += s
        return result

    @staticmethod
```

```
    def concatenate_join(strings):
        """Optimized using join"""
        return "".join(strings)

    @staticmethod
    def concatenate_list(strings):
        """Using list and join"""
        result = []
        for s in strings:
            result.append(s)
        return "".join(result)

# Data structure optimizations
class DataStructureOptimizer:
    """Optimized data structure usage"""

    @staticmethod
    def membership_test_list(items, queries):
        """Membership testing with list"""
        results = []
        for query in queries:
            results.append(query in items)
        return results

    @staticmethod
    def membership_test_set(items, queries):
        """Membership testing with set"""
        item_set = set(items)
        results = []
        for query in queries:
            results.append(query in item_set)
        return results

    @staticmethod
    @profile  # for line_profiler
    def process_large_data(data):
        """Example of processing large dataset efficiently"""
        # Use generator expressions instead of list comprehensions
        filtered = (x for x in data if x > 0)

        # Process in chunks
        chunk_size = 1000
        results = []

        chunk = []
        for item in filtered:
            chunk.append(item * 2)
```

```
        if len(chunk) >= chunk_size:
            # Process chunk
            results.extend(chunk)
            chunk = []

    # Don't forget the last chunk
    if chunk:
        results.extend(chunk)

    return results
```

Concurrent Performance

python

```python
import concurrent.futures
import multiprocessing as mp
import asyncio
import aiohttp
from typing import List, Callable, Any
import time

class ConcurrentOptimizer:
    """Concurrent execution optimizations"""

    @staticmethod
    def cpu_bound_task(n):
        """Simulate CPU-bound task"""
        result = 0
        for i in range(n):
            result += i ** 2
        return result

    @staticmethod
    def sequential_execution(tasks):
        """Sequential execution baseline"""
        results = []
        for task in tasks:
            results.append(ConcurrentOptimizer.cpu_bound_task(task))
        return results

    @staticmethod
    def thread_pool_execution(tasks, max_workers=4):
        """Thread pool for I/O-bound tasks"""
        with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
            futures = [executor.submit(ConcurrentOptimizer.cpu_bound_task, task)
                       for task in tasks]
            return [future.result() for future in concurrent.futures.as_completed(futures)]

    @staticmethod
    def process_pool_execution(tasks, max_workers=None):
        """Process pool for CPU-bound tasks"""
        with concurrent.futures.ProcessPoolExecutor(max_workers=max_workers) as executor:
            return list(executor.map(ConcurrentOptimizer.cpu_bound_task, tasks))

    @staticmethod
    async def async_io_task(session, url):
        """Simulate async I/O task"""
        async with session.get(url) as response:
            return await response.text()
```

```python
    @staticmethod
    async def async_execution(urls):
        """Async execution for I/O-bound tasks"""
        async with aiohttp.ClientSession() as session:
            tasks = [ConcurrentOptimizer.async_io_task(session, url) for url in urls]
            return await asyncio.gather(*tasks)

# Batch processing optimization
class BatchProcessor:
    """Optimized batch processing"""

    def __init__(self, batch_size=1000):
        self.batch_size = batch_size
        self.buffer = []
        self.results = []

    def process_batch(self, batch):
        """Process a single batch"""
        # Simulate batch processing
        return [item * 2 for item in batch]

    def add(self, item):
        """Add item to buffer"""
        self.buffer.append(item)

        if len(self.buffer) >= self.batch_size:
            self.flush()

    def flush(self):
        """Process remaining items in buffer"""
        if self.buffer:
            result = self.process_batch(self.buffer)
            self.results.extend(result)
            self.buffer = []

    def get_results(self):
        """Get all processed results"""
        self.flush()  # Process any remaining items
        return self.results

# Lazy evaluation
class LazyEvaluator:
    """Lazy evaluation patterns"""

    class LazyProperty:
        """Lazy property decorator"""
        def __init__(self, func):
```

```
            self.func = func
            self.name = func.__name__

        def __get__(self, obj, type=None):
            if obj is None:
                return self

            value = self.func(obj)
            # Replace the descriptor with the computed value
            setattr(obj, self.name, value)
            return value

    class LazyList:
        """Lazy list that computes values on demand"""
        def __init__(self, generator_func, length=None):
            self.generator_func = generator_func
            self.length = length
            self._cache = {}

        def __getitem__(self, index):
            if index not in self._cache:
                # Generate all values up to index
                gen = self.generator_func()
                for i in range(index + 1):
                    if i not in self._cache:
                        self._cache[i] = next(gen)

            return self._cache[index]

        def __len__(self):
            if self.length is None:
                raise TypeError("Length not known")
            return self.length

# C extension optimization example
"""
# Example Cython code (save as fast_math.pyx)
# cython: language_level=3

import numpy as np
cimport numpy as np
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def fast_matrix_multiply(np.ndarray[double, ndim=2] a,
                         np.ndarray[double, ndim=2] b):
```

```
    cdef int i, j, k
    cdef int n = a.shape[0]
    cdef int m = a.shape[1]
    cdef int p = b.shape[1]

    cdef np.ndarray[double, ndim=2] result = np.zeros((n, p))

    for i in range(n):
        for j in range(p):
            for k in range(m):
                result[i, j] += a[i, k] * b[k, j]

    return result
"""

# Performance monitoring
class PerformanceMonitor:
    """Real-time performance monitoring"""

    def __init__(self):
        self.metrics = {
            'cpu_times': [],
            'memory_usage': [],
            'function_calls': {}
        }

    def measure_function(self, func):
        """Decorator to measure function performance"""
        @wraps(func)
        def wrapper(*args, **kwargs):
            start_time = time.perf_counter()
            start_memory = self._get_memory_usage()

            try:
                result = func(*args, **kwargs)
            finally:
                end_time = time.perf_counter()
                end_memory = self._get_memory_usage()

                # Record metrics
                func_name = func.__name__
                if func_name not in self.metrics['function_calls']:
                    self.metrics['function_calls'][func_name] = {
                        'count': 0,
                        'total_time': 0,
                        'avg_time': 0,
                        'max_time': 0,
```

```
                    'min_time': float('inf')
                }

            elapsed = end_time - start_time
            metrics = self.metrics['function_calls'][func_name]
            metrics['count'] += 1
            metrics['total_time'] += elapsed
            metrics['avg_time'] = metrics['total_time'] / metrics['count']
            metrics['max_time'] = max(metrics['max_time'], elapsed)
            metrics['min_time'] = min(metrics['min_time'], elapsed)

        return result

    return wrapper

def _get_memory_usage(self):
    """Get current memory usage"""
    import psutil
    process = psutil.Process()
    return process.memory_info().rss / 1024 / 1024  # MB

def report(self):
    """Generate performance report"""
    print("\nPerformance Report")
    print("=" * 60)

    for func_name, metrics in self.metrics['function_calls'].items():
        print(f"\n{func_name}:")
        print(f"  Calls: {metrics['count']}")
        print(f"  Total time: {metrics['total_time']:.4f}s")
        print(f"  Avg time: {metrics['avg_time']:.4f}s")
        print(f"  Min time: {metrics['min_time']:.4f}s")
        print(f"  Max time: {metrics['max_time']:.4f}s")
```

Chapter 12: Package Development and Distribution {#package-development}

Creating Python Packages

python

```
# Project structure for a professional package
"""
my_package/
├── setup.py
├── setup.cfg
├── pyproject.toml
├── README.md
├── LICENSE
├── MANIFEST.in
├── requirements.txt
├── requirements-dev.txt
├── .gitignore
├── .github/
│   └── workflows/
│       └── python-package.yml
├── docs/
│   ├── conf.py
│   ├── index.rst
│   └── ...
├── tests/
│   ├── __init__.py
│   ├── test_core.py
│   └── ...
├── src/
│   └── my_package/
│       ├── __init__.py
│       ├── __version__.py
│       ├── core.py
│       ├── utils.py
│       └── submodule/
│           ├── __init__.py
│           └── ...
└── examples/
    ├── basic_usage.py
    └── ...
"""

# setup.py - Modern setup with setuptools
from setuptools import setup, find_packages
import pathlib

here = pathlib.Path(__file__).parent.resolve()
long_description = (here / "README.md").read_text(encoding="utf-8")

# Read version from __version__.py
version = {}
```

```
with open("src/my_package/__version__.py") as fp:
    exec(fp.read(), version)

setup(
    name="my-package",
    version=version['__version__'],
    author="Your Name",
    author_email="your.email@example.com",
    description="A brief description of the package",
    long_description=long_description,
    long_description_content_type="text/markdown",
    url="https://github.com/yourusername/my-package",
    project_urls={
        "Bug Reports": "https://github.com/yourusername/my-package/issues",
        "Source": "https://github.com/yourusername/my-package",
    },
    classifiers=[
        "Development Status :: 4 - Beta",
        "Intended Audience :: Developers",
        "Topic :: Software Development :: Libraries",
        "License :: OSI Approved :: MIT License",
        "Programming Language :: Python :: 3",
        "Programming Language :: Python :: 3.8",
        "Programming Language :: Python :: 3.9",
        "Programming Language :: Python :: 3.10",
        "Programming Language :: Python :: 3 :: Only",
    ],
    keywords="sample, setuptools, development",
    package_dir={"": "src"},
    packages=find_packages(where="src"),
    python_requires=">=3.8, <4",
    install_requires=[
        "requests>=2.25.0",
        "numpy>=1.19.0",
    ],
    extras_require={
        "dev": [
            "pytest>=6.0",
            "pytest-cov>=2.0",
            "black>=21.0",
            "flake8>=3.9",
            "mypy>=0.900",
            "sphinx>=4.0",
        ],
        "test": [
            "pytest>=6.0",
            "pytest-cov>=2.0",
```

```
        ],
    },
    entry_points={
        "console_scripts": [
            "my-command=my_package.cli:main",
        ],
    },
    include_package_data=True,
)

# pyproject.toml - PEP 518 configuration
"""
[build-system]
requires = ["setuptools>=45", "wheel", "setuptools_scm[toml]>=6.2"]
build-backend = "setuptools.build_meta"

[project]
name = "my-package"
dynamic = ["version"]
description = "A comprehensive Python package"
readme = "README.md"
requires-python = ">=3.8"
license = {text = "MIT"}
authors = [
    {name = "Your Name", email = "your.email@example.com"},
]
keywords = ["example", "package", "development"]
classifiers = [
    "Development Status :: 4 - Beta",
    "Intended Audience :: Developers",
    "License :: OSI Approved :: MIT License",
    "Programming Language :: Python :: 3",
]
dependencies = [
    "requests>=2.25.0",
    "numpy>=1.19.0",
]

[project.optional-dependencies]
dev = [
    "pytest>=6.0",
    "black>=21.0",
    "flake8>=3.9",
    "mypy>=0.900",
]

[project.urls]
```

```
Homepage = "https://github.com/yourusername/my-package"
Documentation = "https://my-package.readthedocs.io"
Repository = "https://github.com/yourusername/my-package.git"
Changelog = "https://github.com/yourusername/my-package/blob/main/CHANGELOG.md"

[tool.setuptools]
package-dir = {"" = "src"}

[tool.setuptools.packages.find]
where = ["src"]

[tool.setuptools_scm]
write_to = "src/my_package/_version.py"

[tool.black]
line-length = 88
target-version = ['py38', 'py39', 'py310']
include = '\.pyi?

[tool.pytest.ini_options]
testpaths = ["tests"]
python_files = ["test_*.py", "*_test.py"]
addopts = [
    "--cov=my_package",
    "--cov-report=html",
    "--cov-report=term-missing",
    "-v",
]

[tool.mypy]
python_version = "3.8"
warn_return_any = true
warn_unused_configs = true
disallow_untyped_defs = true
"""

# Package initialization with proper exports
# src/my_package/__init__.py
"""
My Package - A comprehensive Python package

This package provides...
"""

from .__version__ import __version__
from .core import MainClass, main_function
from .utils import utility_function
```

```python
# Define what is exported with 'from my_package import *'
__all__ = [
    "__version__",
    "MainClass",
    "main_function",
    "utility_function",
]

# Package metadata
__author__ = "Your Name"
__email__ = "your.email@example.com"
__license__ = "MIT"

# Lazy imports for heavy dependencies
def _lazy_import():
    """Lazy import heavy dependencies"""
    global heavy_module
    import heavy_module as _heavy_module
    heavy_module = _heavy_module

# Version handling
# src/my_package/__version__.py
__version__ = "0.1.0"

# CLI entry point
# src/my_package/cli.py
import argparse
import sys
from . import __version__

def create_parser():
    """Create command-line argument parser"""
    parser = argparse.ArgumentParser(
        description="My Package CLI",
        formatter_class=argparse.RawDescriptionHelpFormatter,
    )
    parser.add_argument(
        "--version",
        action="version",
        version=f"%(prog)s {__version__}",
    )

    subparsers = parser.add_subparsers(dest="command", help="Available commands")

    # Add subcommands
    run_parser = subparsers.add_parser("run", help="Run the main function")
```

```
    run_parser.add_argument("input", help="Input file")
    run_parser.add_argument("-o", "--output", help="Output file")

    return parser

def main(argv=None):
    """Main CLI entry point"""
    parser = create_parser()
    args = parser.parse_args(argv)

    if args.command == "run":
        from .core import process_file
        result = process_file(args.input, args.output)
        print(f"Processed: {result}")
    else:
        parser.print_help()
        return 1

    return 0

if __name__ == "__main__":
    sys.exit(main())
```

Testing and CI/CD

python

```python
# tests/test_core.py
import pytest
from unittest.mock import Mock, patch
import tempfile
import pathlib

from my_package import MainClass, main_function

class TestMainClass:
    """Test suite for MainClass"""

    @pytest.fixture
    def instance(self):
        """Create instance for testing"""
        return MainClass()

    def test_initialization(self, instance):
        """Test proper initialization"""
        assert instance is not None
        assert hasattr(instance, 'data')

    @pytest.mark.parametrize("input_data,expected", [
        ([1, 2, 3], 6),
        ([], 0),
        ([-1, 1], 0),
    ])
    def test_process_data(self, instance, input_data, expected):
        """Test data processing with various inputs"""
        result = instance.process_data(input_data)
        assert result == expected

    def test_error_handling(self, instance):
        """Test error handling"""
        with pytest.raises(ValueError):
            instance.process_data(None)

# GitHub Actions workflow
# .github/workflows/python-package.yml
"""
name: Python Package

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

```yaml
jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        python-version: ['3.8', '3.9', '3.10']

    steps:
    - uses: actions/checkout@v2

    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v2
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -e .[dev]

    - name: Lint with flake8
      run: |
        flake8 src tests

    - name: Type check with mypy
      run: |
        mypy src

    - name: Test with pytest
      run: |
        pytest --cov=my_package --cov-report=xml

    - name: Upload coverage
      uses: codecov/codecov-action@v1
      with:
        file: ./coverage.xml

  build:
    needs: test
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Build package
```

```
      run: |
        python -m pip install --upgrade pip
        pip install build
        python -m build

    - name: Upload artifacts
      uses: actions/upload-artifact@v2
      with:
        name: dist
        path: dist/
"""

# Documentation generation
# docs/conf.py
"""
import os
import sys
sys.path.insert(0, os.path.abspath('../src'))

project = 'My Package'
copyright = '2024, Your Name'
author = 'Your Name'

extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.napoleon',
    'sphinx.ext.viewcode',
    'sphinx.ext.intersphinx',
    'sphinx_rtd_theme',
]

templates_path = ['_templates']
exclude_patterns = ['_build', 'Thumbs.db', '.DS_Store']

html_theme = 'sphinx_rtd_theme'
html_static_path = ['_static']

intersphinx_mapping = {
    'python': ('https://docs.python.org/3', None),
    'numpy': ('https://numpy.org/doc/stable/', None),
}
"""

# Publishing utilities
class PackagePublisher:
    """Utilities for publishing packages"""
```

```python
    @staticmethod
    def build_package():
        """Build package distributions"""
        import subprocess

        # Clean previous builds
        subprocess.run(["rm", "-rf", "dist/", "build/", "*.egg-info"])

        # Build source distribution and wheel
        subprocess.run([sys.executable, "-m", "build"])

    @staticmethod
    def check_package():
        """Check package with twine"""
        import subprocess

        result = subprocess.run(
            ["twine", "check", "dist/*"],
            capture_output=True,
            text=True
        )

        if result.returncode != 0:
            print(f"Package check failed: {result.stderr}")
            return False

        print("Package check passed!")
        return True

    @staticmethod
    def upload_to_test_pypi():
        """Upload to TestPyPI for testing"""
        import subprocess

        subprocess.run([
            "twine", "upload",
            "--repository-url", "https://test.pypi.org/legacy/",
            "dist/*"
        ])

    @staticmethod
    def upload_to_pypi():
        """Upload to PyPI"""
        import subprocess

        # Confirm before uploading
        response = input("Upload to PyPI? (y/N): ")
```

```
    if response.lower() != 'y':
        print("Upload cancelled")
        return

    subprocess.run(["twine", "upload", "dist/*"])
```

Chapter 13: Network Programming {#network-programming}

Advanced Socket Programming

python

```python
import socket
import select
import struct
import threading
import ssl
import asyncio
from typing import Optional, Tuple, List
import json

# Non-blocking socket server
class NonBlockingServer:
    """High-performance non-blocking socket server"""

    def __init__(self, host='127.0.0.1', port=9999):
        self.host = host
        self.port = port
        self.server_socket = None
        self.client_sockets = []
        self.socket_to_data = {}
        self.running = False

    def start(self):
        """Start the non-blocking server"""
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.server_socket.setblocking(False)
        self.server_socket.bind((self.host, self.port))
        self.server_socket.listen(5)

        self.running = True
        print(f"Server listening on {self.host}:{self.port}")

        while self.running:
            # Use select for non-blocking I/O
            readable, writable, exceptional = select.select(
                [self.server_socket] + self.client_sockets,
                [],
                self.client_sockets,
                1.0  # 1 second timeout
            )

            for sock in readable:
                if sock is self.server_socket:
                    self._accept_connection()
                else:
                    self._handle_client_data(sock)
```

```
        for sock in exceptional:
            self._remove_client(sock)

def _accept_connection(self):
    """Accept new client connection"""
    try:
        client_socket, address = self.server_socket.accept()
        client_socket.setblocking(False)
        self.client_sockets.append(client_socket)
        self.socket_to_data[client_socket] = b''
        print(f"New connection from {address}")
    except socket.error:
        pass

def _handle_client_data(self, client_socket):
    """Handle data from client"""
    try:
        data = client_socket.recv(4096)
        if data:
            self.socket_to_data[client_socket] += data
            # Process complete messages
            self._process_messages(client_socket)
        else:
            self._remove_client(client_socket)
    except socket.error:
        self._remove_client(client_socket)

def _process_messages(self, client_socket):
    """Process complete messages from client"""
    data = self.socket_to_data[client_socket]

    # Simple protocol: messages end with newline
    while b'\n' in data:
        message, data = data.split(b'\n', 1)
        self.socket_to_data[client_socket] = data

        # Echo back to client
        response = b"Echo: " + message + b'\n'
        try:
            client_socket.send(response)
        except socket.error:
            self._remove_client(client_socket)
            break

def _remove_client(self, client_socket):
    """Remove client connection"""
```

```
        if client_socket in self.client_sockets:
            self.client_sockets.remove(client_socket)
            del self.socket_to_data[client_socket]
            client_socket.close()
            print("Client disconnected")

    def stop(self):
        """Stop the server"""
        self.running = False
        if self.server_socket:
            self.server_socket.close()
        for sock in self.client_sockets:
            sock.close()

# Protocol implementation
class Protocol:
    """Custom network protocol implementation"""

    HEADER_FORMAT = '!IH'  # Network byte order: uint32 + uint16
    HEADER_SIZE = struct.calcsize(HEADER_FORMAT)

    class MessageType:
        HANDSHAKE = 1
        DATA = 2
        HEARTBEAT = 3
        CLOSE = 4

    @staticmethod
    def pack_message(msg_type: int, data: bytes) -> bytes:
        """Pack message with header"""
        length = len(data)
        header = struct.pack(Protocol.HEADER_FORMAT, length, msg_type)
        return header + data

    @staticmethod
    def unpack_header(data: bytes) -> Optional[Tuple[int, int]]:
        """Unpack message header"""
        if len(data) < Protocol.HEADER_SIZE:
            return None

        length, msg_type = struct.unpack(
            Protocol.HEADER_FORMAT,
            data[:Protocol.HEADER_SIZE]
        )
        return length, msg_type

# SSL/TLS secure server
```

```python
class SecureServer:
    """SSL/TLS enabled server"""

    def __init__(self, host='127.0.0.1', port=9443, certfile='server.crt', keyfile='server.key'
        self.host = host
        self.port = port
        self.certfile = certfile
        self.keyfile = keyfile
        self.context = self._create_ssl_context()

    def _create_ssl_context(self):
        """Create SSL context"""
        context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
        context.load_cert_chain(self.certfile, self.keyfile)

        # Set secure options
        context.options |= ssl.OP_NO_TLSv1 | ssl.OP_NO_TLSv1_1
        context.set_ciphers('ECDHE+AESGCM:ECDHE+CHACHA20:DHE+AESGCM:DHE+CHACHA20:!aNULL:!MD5:!D

        return context

    def start(self):
        """Start secure server"""
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
            sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            sock.bind((self.host, self.port))
            sock.listen(5)

            with self.context.wrap_socket(sock, server_side=True) as ssock:
                print(f"Secure server listening on {self.host}:{self.port}")

                while True:
                    client_socket, address = ssock.accept()
                    print(f"Secure connection from {address}")

                    # Handle client in separate thread
                    thread = threading.Thread(
                        target=self._handle_client,
                        args=(client_socket,)
                    )
                    thread.daemon = True
                    thread.start()

    def _handle_client(self, client_socket):
        """Handle secure client connection"""
        try:
            # Get peer certificate info
```

```
            cert = client_socket.getpeercert()
            if cert:
                print(f"Client certificate: {cert}")

            while True:
                data = client_socket.recv(4096)
                if not data:
                    break

                # Process and respond
                response = self._process_request(data)
                client_socket.send(response)

        except Exception as e:
            print(f"Error handling client: {e}")
        finally:
            client_socket.close()

    def _process_request(self, data: bytes) -> bytes:
        """Process client request"""
        try:
            request = json.loads(data.decode('utf-8'))
            response = {
                'status': 'success',
                'data': f"Processed: {request}"
            }
        except Exception as e:
            response = {
                'status': 'error',
                'message': str(e)
            }

        return json.dumps(response).encode('utf-8')

# Async network programming
class AsyncTCPServer:
    """Asynchronous TCP server using asyncio"""

    def __init__(self, host='127.0.0.1', port=8888):
        self.host = host
        self.port = port
        self.clients = {}
        self.client_id_counter = 0

    async def handle_client(self, reader, writer):
        """Handle individual client connection"""
        client_id = self.client_id_counter
```

```
        self.client_id_counter += 1

        client_address = writer.get_extra_info('peername')
        self.clients[client_id] = (reader, writer)

        print(f"Client {client_id} connected from {client_address}")

        try:
            while True:
                # Read data with timeout
                try:
                    data = await asyncio.wait_for(
                        reader.read(1024),
                        timeout=30.0
                    )
                except asyncio.TimeoutError:
                    await self._send_to_client(writer, b"PING\n")
                    continue

                if not data:
                    break

                message = data.decode('utf-8').strip()
                print(f"Client {client_id}: {message}")

                # Broadcast to all clients
                await self._broadcast(f"Client {client_id}: {message}\n", exclude=client_id)

                # Echo back to sender
                await self._send_to_client(writer, f"You said: {message}\n".encode())

        except Exception as e:
            print(f"Error with client {client_id}: {e}")

        finally:
            del self.clients[client_id]
            writer.close()
            await writer.wait_closed()
            print(f"Client {client_id} disconnected")

    async def _send_to_client(self, writer, data: bytes):
        """Send data to specific client"""
        writer.write(data)
        await writer.drain()

    async def _broadcast(self, message: str, exclude=None):
        """Broadcast message to all clients"""
```

```
        tasks = []
        for client_id, (_, writer) in self.clients.items():
            if client_id != exclude:
                tasks.append(self._send_to_client(writer, message.encode()))

        if tasks:
            await asyncio.gather(*tasks, return_exceptions=True)

    async def start(self):
        """Start the async server"""
        server = await asyncio.start_server(
            self.handle_client,
            self.host,
            self.port
        )

        addr = server.sockets[0].getsockname()
        print(f"Async server running on {addr}")

        async with server:
            await server.serve_forever()

# UDP server with reliability
class ReliableUDPServer:
    """UDP server with reliability features"""

    def __init__(self, host='127.0.0.1', port=9999):
        self.host = host
        self.port = port
        self.socket = None
        self.clients = {}  # address -> client_state
        self.sequence_numbers = {}

    def start(self):
        """Start UDP server"""
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.socket.bind((self.host, self.port))

        print(f"Reliable UDP server listening on {self.host}:{self.port}")

        while True:
            data, address = self.socket.recvfrom(4096)
            self._handle_packet(data, address)

    def _handle_packet(self, data: bytes, address: Tuple[str, int]):
        """Handle incoming UDP packet"""
        try:
```

```
            # Parse packet header
            if len(data) < 8:
                return

            seq_num, ack_num, flags = struct.unpack('!IIB', data[:9])
            payload = data[9:]

            # Initialize client state if new
            if address not in self.clients:
                self.clients[address] = {
                    'last_seq': 0,
                    'expected_seq': 0
                }

            client = self.clients[address]

            # Check for duplicate or out-of-order packets
            if seq_num <= client['last_seq']:
                # Send ACK for duplicate
                self._send_ack(address, seq_num)
                return

            # Process payload
            response = self._process_payload(payload)

            # Update state
            client['last_seq'] = seq_num

            # Send response with ACK
            self._send_response(address, seq_num, response)

        except Exception as e:
            print(f"Error handling packet: {e}")

    def _send_ack(self, address: Tuple[str, int], seq_num: int):
        """Send acknowledgment packet"""
        ack_packet = struct.pack('!IIB', 0, seq_num, 0x01)  # ACK flag
        self.socket.sendto(ack_packet, address)

    def _send_response(self, address: Tuple[str, int], ack_num: int, data: bytes):
        """Send response with sequence number"""
        if address not in self.sequence_numbers:
            self.sequence_numbers[address] = 0

        seq_num = self.sequence_numbers[address]
        self.sequence_numbers[address] += 1
```

```
        packet = struct.pack('!IIB', seq_num, ack_num, 0x00) + data
        self.socket.sendto(packet, address)

    def _process_payload(self, payload: bytes) -> bytes:
        """Process payload and generate response"""
        return b"Processed: " + payload

# WebSocket server implementation
class SimpleWebSocketServer:
    """Basic WebSocket server implementation"""

    def __init__(self, host='127.0.0.1', port=8765):
        self.host = host
        self.port = port

    async def handle_websocket(self, websocket, path):
        """Handle WebSocket connection"""
        print(f"New WebSocket connection: {path}")

        try:
            async for message in websocket:
                print(f"Received: {message}")

                # Echo back
                await websocket.send(f"Echo: {message}")

                # Send additional data
                if message.lower() == "status":
                    status = {
                        "connections": len(websocket.ws_server.websockets),
                        "uptime": time.time()
                    }
                    await websocket.send(json.dumps(status))

        except Exception as e:
            print(f"WebSocket error: {e}")

    async def start(self):
        """Start WebSocket server"""
        import websockets

        async with websockets.serve(
            self.handle_websocket,
            self.host,
            self.port
        ):
```

```
        print(f"WebSocket server running on ws://{self.host}:{self.port}")
        await asyncio.Future()  # Run forever
```

Chapter 14: Database Programming {#database-programming}

Advanced Database Patterns

python

```python
import sqlite3
import psycopg2
from contextlib import contextmanager
from typing import List, Dict, Any, Optional, Type
import threading
from datetime import datetime
import json

# Database connection pool
class ConnectionPool:
    """Thread-safe database connection pool"""

    def __init__(self, database_url: str, min_size=2, max_size=10):
        self.database_url = database_url
        self.min_size = min_size
        self.max_size = max_size
        self._connections = []
        self._used_connections = set()
        self._lock = threading.Lock()

        # Initialize minimum connections
        for _ in range(min_size):
            conn = self._create_connection()
            self._connections.append(conn)

    def _create_connection(self):
        """Create new database connection"""
        # Example for PostgreSQL
        return psycopg2.connect(self.database_url)

    @contextmanager
    def get_connection(self):
        """Get connection from pool"""
        conn = self._acquire_connection()
        try:
            yield conn
        finally:
            self._release_connection(conn)

    def _acquire_connection(self):
        """Acquire connection from pool"""
        with self._lock:
            # Try to get available connection
            if self._connections:
                conn = self._connections.pop()
                self._used_connections.add(conn)
```

```
                return conn

            # Create new connection if under limit
            if len(self._used_connections) < self.max_size:
                conn = self._create_connection()
                self._used_connections.add(conn)
                return conn

            # Wait for available connection
            raise RuntimeError("Connection pool exhausted")

    def _release_connection(self, conn):
        """Release connection back to pool"""
        with self._lock:
            self._used_connections.remove(conn)
            if len(self._connections) < self.max_size:
                self._connections.append(conn)
            else:
                conn.close()

    def close_all(self):
        """Close all connections"""
        with self._lock:
            for conn in self._connections:
                conn.close()
            for conn in self._used_connections:
                conn.close()
            self._connections.clear()
            self._used_connections.clear()

# Repository pattern
class Repository:
    """Base repository class"""

    def __init__(self, connection_pool: ConnectionPool):
        self.pool = connection_pool

    def execute(self, query: str, params: tuple = None):
        """Execute query with parameters"""
        with self.pool.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute(query, params or ())
            conn.commit()
            return cursor

    def fetch_one(self, query: str, params: tuple = None) -> Optional[Dict[str, Any]]:
        """Fetch single row as dictionary"""
```

```
        with self.pool.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute(query, params or ())
            columns = [desc[0] for desc in cursor.description]
            row = cursor.fetchone()

            if row:
                return dict(zip(columns, row))
            return None

    def fetch_all(self, query: str, params: tuple = None) -> List[Dict[str, Any]]:
        """Fetch all rows as list of dictionaries"""
        with self.pool.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute(query, params or ())
            columns = [desc[0] for desc in cursor.description]

            return [dict(zip(columns, row)) for row in cursor.fetchall()]

# Active Record pattern
class ActiveRecord:
    """Base class for Active Record pattern"""

    __tablename__ = None
    __primary_key__ = 'id'

    def __init__(self, **kwargs):
        for key, value in kwargs.items():
            setattr(self, key, value)

    @classmethod
    def create_table(cls, connection):
        """Create table for model"""
        raise NotImplementedError

    @classmethod
    def find(cls, connection, id):
        """Find record by ID"""
        query = f"SELECT * FROM {cls.__tablename__} WHERE {cls.__primary_key__} = ?"
        cursor = connection.execute(query, (id,))
        row = cursor.fetchone()

        if row:
            columns = [desc[0] for desc in cursor.description]
            data = dict(zip(columns, row))
            return cls(**data)
        return None
```

```
    @classmethod
    def all(cls, connection):
        """Get all records"""
        query = f"SELECT * FROM {cls.__tablename__}"
        cursor = connection.execute(query)

        columns = [desc[0] for desc in cursor.description]
        return [cls(**dict(zip(columns, row))) for row in cursor.fetchall()]

    def save(self, connection):
        """Save record to database"""
        if hasattr(self, self.__primary_key__) and getattr(self, self.__primary_key__):
            self._update(connection)
        else:
            self._insert(connection)

    def _insert(self, connection):
        """Insert new record"""
        fields = []
        values = []

        for key, value in self.__dict__.items():
            if key != self.__primary_key__:
                fields.append(key)
                values.append(value)

        placeholders = ', '.join(['?' for _ in values])
        fields_str = ', '.join(fields)

        query = f"INSERT INTO {self.__tablename__} ({fields_str}) VALUES ({placeholders})"
        cursor = connection.execute(query, values)

        setattr(self, self.__primary_key__, cursor.lastrowid)
        connection.commit()

    def _update(self, connection):
        """Update existing record"""
        fields = []
        values = []

        for key, value in self.__dict__.items():
            if key != self.__primary_key__:
                fields.append(f"{key} = ?")
                values.append(value)

        values.append(getattr(self, self.__primary_key__))
```

```
        fields_str = ', '.join(fields)

        query = f"UPDATE {self.__tablename__} SET {fields_str} WHERE {self.__primary_key__} = ?
        connection.execute(query, values)
        connection.commit()

    def delete(self, connection):
        """Delete record"""
        query = f"DELETE FROM {self.__tablename__} WHERE {self.__primary_key__} = ?"
        connection.execute(query, (getattr(self, self.__primary_key__),))
        connection.commit()

# Example model
class User(ActiveRecord):
    __tablename__ = 'users'

    @classmethod
    def create_table(cls, connection):
        connection.execute('''
            CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                username TEXT UNIQUE NOT NULL,
                email TEXT UNIQUE NOT NULL,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            )
        ''')

# Query builder
class QueryBuilder:
    """SQL query builder"""

    def __init__(self, table: str):
        self.table = table
        self._select_fields = ['*']
        self._where_conditions = []
        self._join_clauses = []
        self._order_by = []
        self._group_by = []
        self._having = []
        self._limit = None
        self._offset = None

    def select(self, *fields):
        """Select specific fields"""
        self._select_fields = list(fields)
        return self
```

```
def where(self, condition: str, value: Any = None):
    """Add WHERE condition"""
    self._where_conditions.append((condition, value))
    return self

def join(self, table: str, on: str):
    """Add JOIN clause"""
    self._join_clauses.append(f"JOIN {table} ON {on}")
    return self

def left_join(self, table: str, on: str):
    """Add LEFT JOIN clause"""
    self._join_clauses.append(f"LEFT JOIN {table} ON {on}")
    return self

def order_by(self, field: str, direction: str = 'ASC'):
    """Add ORDER BY clause"""
    self._order_by.append(f"{field} {direction}")
    return self

def group_by(self, *fields):
    """Add GROUP BY clause"""
    self._group_by.extend(fields)
    return self

def having(self, condition: str):
    """Add HAVING clause"""
    self._having.append(condition)
    return self

def limit(self, limit: int):
    """Set LIMIT"""
    self._limit = limit
    return self

def offset(self, offset: int):
    """Set OFFSET"""
    self._offset = offset
    return self

def build(self) -> tuple:
    """Build SQL query and parameters"""
    query_parts = [
        f"SELECT {', '.join(self._select_fields)}",
        f"FROM {self.table}"
    ]
```

```
        # Add JOINs
        query_parts.extend(self._join_clauses)

        # Add WHERE
        params = []
        if self._where_conditions:
            where_parts = []
            for condition, value in self._where_conditions:
                if value is not None:
                    where_parts.append(condition)
                    params.append(value)
                else:
                    where_parts.append(condition)

            query_parts.append(f"WHERE {' AND '.join(where_parts)}")

        # Add GROUP BY
        if self._group_by:
            query_parts.append(f"GROUP BY {', '.join(self._group_by)}")

        # Add HAVING
        if self._having:
            query_parts.append(f"HAVING {' AND '.join(self._having)}")

        # Add ORDER BY
        if self._order_by:
            query_parts.append(f"ORDER BY {', '.join(self._order_by)}")

        # Add LIMIT and OFFSET
        if self._limit:
            query_parts.append(f"LIMIT {self._limit}")
        if self._offset:
            query_parts.append(f"OFFSET {self._offset}")

        return ' '.join(query_parts), tuple(params)

# Migration system
class Migration:
    """Database migration system"""

    def __init__(self, connection):
        self.connection = connection
        self._create_migrations_table()

    def _create_migrations_table(self):
        """Create migrations tracking table"""
```

```
        self.connection.execute('''
            CREATE TABLE IF NOT EXISTS migrations (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT UNIQUE NOT NULL,
                applied_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            )
        ''')
        self.connection.commit()

    def apply(self, migration_name: str, up_func: callable, down_func: callable = None):
        """Apply migration"""
        # Check if already applied
        cursor = self.connection.execute(
            "SELECT * FROM migrations WHERE name = ?",
            (migration_name,)
        )

        if cursor.fetchone():
            print(f"Migration {migration_name} already applied")
            return

        try:
            # Apply migration
            up_func(self.connection)

            # Record migration
            self.connection.execute(
                "INSERT INTO migrations (name) VALUES (?)",
                (migration_name,)
            )
            self.connection.commit()

            print(f"Applied migration: {migration_name}")

        except Exception as e:
            self.connection.rollback()
            print(f"Failed to apply migration {migration_name}: {e}")
            raise

    def rollback(self, migration_name: str, down_func: callable):
        """Rollback migration"""
        cursor = self.connection.execute(
            "SELECT * FROM migrations WHERE name = ?",
            (migration_name,)
        )

        if not cursor.fetchone():
```

```
            print(f"Migration {migration_name} not found")
            return

        try:
            # Rollback migration
            down_func(self.connection)

            # Remove migration record
            self.connection.execute(
                "DELETE FROM migrations WHERE name = ?",
                (migration_name,)
            )
            self.connection.commit()

            print(f"Rolled back migration: {migration_name}")

        except Exception as e:
            self.connection.rollback()
            print(f"Failed to rollback migration {migration_name}: {e}")
            raise

# Advanced ORM features
class AdvancedORM:
    """Advanced ORM with relationships and lazy loading"""

    class Model:
        """Base model with advanced features"""

        def __init__(self, **kwargs):
            self._data = kwargs
            self._dirty = set()
            self._related = {}

        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.startswith('_'):
                super().__setattr__(name, value)
            else:
                self._data[name] = value
                self._dirty.add(name)

        @classmethod
        def has_many(cls, related_model: Type['Model'], foreign_key: str):
```

```
            """Define one-to-many relationship"""
            def get_related(self):
                if related_model.__name__ not in self._related:
                    # Lazy load related objects
                    query = QueryBuilder(related_model.__tablename__)
                    query.where(f"{foreign_key} = ?", self.id)
                    sql, params = query.build()

                    # Execute query and create objects
                    # This is simplified - real implementation would use connection
                    self._related[related_model.__name__] = []

                return self._related[related_model.__name__]

            return property(get_related)

        @classmethod
        def belongs_to(cls, related_model: Type['Model'], foreign_key: str):
            """Define many-to-one relationship"""
            def get_related(self):
                if related_model.__name__ not in self._related:
                    # Lazy load related object
                    foreign_id = getattr(self, foreign_key)
                    if foreign_id:
                        # Simplified - real implementation would query database
                        self._related[related_model.__name__] = None
                    else:
                        self._related[related_model.__name__] = None

                return self._related[related_model.__name__]

            return property(get_related)

# Database transaction manager
class TransactionManager:
    """Manage database transactions"""

    def __init__(self, connection):
        self.connection = connection
        self.savepoints = []

    @contextmanager
    def transaction(self):
        """Context manager for transactions"""
        try:
            self.begin()
            yield self
```

```
            self.commit()
        except Exception:
            self.rollback()
            raise

    def begin(self):
        """Begin transaction"""
        self.connection.execute("BEGIN")

    def commit(self):
        """Commit transaction"""
        self.connection.commit()
        self.savepoints.clear()

    def rollback(self):
        """Rollback transaction"""
        self.connection.rollback()
        self.savepoints.clear()

    def savepoint(self, name: str):
        """Create savepoint"""
        self.connection.execute(f"SAVEPOINT {name}")
        self.savepoints.append(name)

    def rollback_to_savepoint(self, name: str):
        """Rollback to savepoint"""
        if name in self.savepoints:
            self.connection.execute(f"ROLLBACK TO SAVEPOINT {name}")
            # Remove savepoints after this one
            index = self.savepoints.index(name)
            self.savepoints = self.savepoints[:index]
```

Chapter 15: Web Development {#web-development}

Advanced Flask Patterns

python

```python
from flask import Flask, request, jsonify, g
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from flask_jwt_extended import JWTManager, create_access_token, jwt_required, get_jwt_identity
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
from flask_cors import CORS
from werkzeug.security import generate_password_hash, check_password_hash
from functools import wraps
import redis
from datetime import datetime, timedelta
from typing import Dict, Any, Optional
import logging

# Application factory pattern
def create_app(config_name='development'):
    """Create Flask application with factory pattern"""
    app = Flask(__name__)

    # Load configuration
    app.config.from_object(f'config.{config_name.capitalize()}Config')

    # Initialize extensions
    db.init_app(app)
    migrate.init_app(app, db)
    jwt.init_app(app)
    limiter.init_app(app)
    cors.init_app(app)

    # Register blueprints
    from .api import api_bp
    app.register_blueprint(api_bp, url_prefix='/api/v1')

    from .auth import auth_bp
    app.register_blueprint(auth_bp, url_prefix='/auth')

    # Register error handlers
    register_error_handlers(app)

    # Register middleware
    register_middleware(app)

    return app

# Extensions
db = SQLAlchemy()
```

```
migrate = Migrate()
jwt = JWTManager()
limiter = Limiter(key_func=get_remote_address)
cors = CORS()

# Models with advanced features
class BaseModel(db.Model):
    """Base model with common fields"""
    __abstract__ = True

    id = db.Column(db.Integer, primary_key=True)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
    updated_at = db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

    def to_dict(self):
        """Convert model to dictionary"""
        return {
            column.name: getattr(self, column.name)
            for column in self.__table__.columns
        }

    def update(self, **kwargs):
        """Update model attributes"""
        for key, value in kwargs.items():
            if hasattr(self, key):
                setattr(self, key, value)
        db.session.commit()

class User(BaseModel):
    """User model with authentication"""
    __tablename__ = 'users'

    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password_hash = db.Column(db.String(255), nullable=False)
    is_active = db.Column(db.Boolean, default=True)
    role = db.Column(db.String(50), default='user')

    # Relationships
    posts = db.relationship('Post', backref='author', lazy='dynamic')

    def set_password(self, password):
        """Set user password"""
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        """Check user password"""
```

```python
        return check_password_hash(self.password_hash, password)

    def generate_token(self, expires_in=3600):
        """Generate JWT token"""
        return create_access_token(
            identity=self.id,
            expires_delta=timedelta(seconds=expires_in),
            additional_claims={'role': self.role}
        )

    def to_dict(self, include_email=False):
        """Convert user to dictionary"""
        data = {
            'id': self.id,
            'username': self.username,
            'created_at': self.created_at.isoformat()
        }
        if include_email:
            data['email'] = self.email
        return data

# Advanced decorators
def require_role(*allowed_roles):
    """Decorator to require specific roles"""
    def decorator(f):
        @wraps(f)
        @jwt_required()
        def decorated_function(*args, **kwargs):
            claims = get_jwt()
            user_role = claims.get('role', 'user')

            if user_role not in allowed_roles:
                return jsonify({'error': 'Insufficient permissions'}), 403

            return f(*args, **kwargs)
        return decorated_function
    return decorator

def cache_response(timeout=300):
    """Cache response decorator"""
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            # Create cache key from function name and arguments
            cache_key = f"{f.__name__}:{str(args)}:{str(kwargs)}"

            # Try to get from cache
```

```
            cached = redis_client.get(cache_key)
            if cached:
                return json.loads(cached)

            # Generate response
            response = f(*args, **kwargs)

            # Cache response
            redis_client.setex(
                cache_key,
                timeout,
                json.dumps(response.get_json())
            )

            return response
        return decorated_function
    return decorator

def validate_json(*required_fields):
    """Validate JSON request decorator"""
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not request.is_json:
                return jsonify({'error': 'Content-Type must be application/json'}), 400

            data = request.get_json()

            for field in required_fields:
                if field not in data:
                    return jsonify({'error': f'Missing required field: {field}'}), 400

            return f(*args, **kwargs)
        return decorated_function
    return decorator

# RESTful API with pagination
class PaginatedAPIMixin:
    """Mixin for paginated API responses"""

    @staticmethod
    def to_collection_dict(query, page, per_page, endpoint, **kwargs):
        """Convert query to paginated response"""
        resources = query.paginate(page, per_page, False)

        data = {
            'items': [item.to_dict() for item in resources.items],
```

```
            '_meta': {
                'page': page,
                'per_page': per_page,
                'total_pages': resources.pages,
                'total_items': resources.total
            },
            '_links': {
                'self': url_for(endpoint, page=page, per_page=per_page, **kwargs),
                'next': url_for(endpoint, page=page + 1, per_page=per_page, **kwargs) if resour
                'prev': url_for(endpoint, page=page - 1, per_page=per_page, **kwargs) if resour
            }
        }

        return data

# API endpoints
@api_bp.route('/users', methods=['GET'])
@jwt_required()
@limiter.limit("100 per hour")
def get_users():
    """Get paginated list of users"""
    page = request.args.get('page', 1, type=int)
    per_page = min(request.args.get('per_page', 10, type=int), 100)

    query = User.query.filter_by(is_active=True)

    return jsonify(
        PaginatedAPIMixin.to_collection_dict(
            query, page, per_page, 'api.get_users'
        )
    )

@api_bp.route('/users/<int:user_id>', methods=['GET'])
@jwt_required()
@cache_response(timeout=600)
def get_user(user_id):
    """Get specific user"""
    user = User.query.get_or_404(user_id)
    return jsonify(user.to_dict(include_email=get_jwt_identity() == user_id))

@api_bp.route('/users', methods=['POST'])
@require_role('admin')
@validate_json('username', 'email', 'password')
def create_user():
    """Create new user"""
    data = request.get_json()
```

```python
    # Validate unique fields
    if User.query.filter_by(username=data['username']).first():
        return jsonify({'error': 'Username already exists'}), 400

    if User.query.filter_by(email=data['email']).first():
        return jsonify({'error': 'Email already exists'}), 400

    # Create user
    user = User(
        username=data['username'],
        email=data['email']
    )
    user.set_password(data['password'])

    db.session.add(user)
    db.session.commit()

    return jsonify(user.to_dict()), 201

# WebSocket support with Flask-SocketIO
from flask_socketio import SocketIO, emit, join_room, leave_room

socketio = SocketIO(cors_allowed_origins="*")

@socketio.on('connect')
@jwt_required()
def handle_connect():
    """Handle WebSocket connection"""
    user_id = get_jwt_identity()
    join_room(f"user_{user_id}")
    emit('connected', {'status': 'Connected'})

@socketio.on('disconnect')
def handle_disconnect():
    """Handle WebSocket disconnection"""
    user_id = get_jwt_identity()
    leave_room(f"user_{user_id}")

@socketio.on('message')
@jwt_required()
def handle_message(data):
    """Handle WebSocket message"""
    user_id = get_jwt_identity()

    # Process message
    processed_data = {
        'user_id': user_id,
```

```
        'message': data['message'],
        'timestamp': datetime.utcnow().isoformat()
    }

    # Broadcast to room
    emit('new_message', processed_data, room=f"user_{user_id}")

# Middleware
def register_middleware(app):
    """Register application middleware"""

    @app.before_request
    def before_request():
        """Before request middleware"""
        g.start_time = time.time()

        # Log request
        app.logger.info(f"{request.method} {request.path}")

    @app.after_request
    def after_request(response):
        """After request middleware"""
        # Add custom headers
        response.headers['X-Request-ID'] = str(uuid.uuid4())

        # Log response time
        if hasattr(g, 'start_time'):
            elapsed = time.time() - g.start_time
            response.headers['X-Response-Time'] = f"{elapsed:.3f}s"

        return response

# Error handlers
def register_error_handlers(app):
    """Register error handlers"""

    @app.errorhandler(404)
    def not_found_error(error):
        return jsonify({'error': 'Not found'}), 404

    @app.errorhandler(500)
    def internal_error(error):
        db.session.rollback()
        return jsonify({'error': 'Internal server error'}), 500

    @app.errorhandler(ValidationError)
    def validation_error(error):
```

```
        return jsonify({'error': str(error)}), 400

# Background tasks with Celery
from celery import Celery

celery = Celery(__name__)

@celery.task
def send_email_task(recipient, subject, body):
    """Send email asynchronously"""
    # Email sending logic
    pass

@celery.task
def process_data_task(data_id):
    """Process data asynchronously"""
    # Data processing logic
    pass

# API versioning
class APIVersion:
    """API versioning handler"""

    def __init__(self, app=None):
        self.app = app
        if app:
            self.init_app(app)

    def init_app(self, app):
        """Initialize API versioning"""
        app.before_request(self._check_api_version)

    def _check_api_version(self):
        """Check API version in request"""
        version = request.headers.get('API-Version', 'v1')

        if version not in ['v1', 'v2']:
            return jsonify({'error': 'Invalid API version'}), 400

        g.api_version = version
```

Chapter 16: Data Processing and Analysis {#data-processing}

Advanced Data Processing

python

```python
import pandas as pd
import numpy as np
from typing import List, Dict, Any, Callable, Optional
import dask.dataframe as dd
from functools import reduce
import pyarrow.parquet as pq
import h5py
from scipy import stats
from sklearn.preprocessing import StandardScaler, LabelEncoder
import multiprocessing as mp

# Data pipeline framework
class DataPipeline:
    """Advanced data processing pipeline"""

    def __init__(self):
        self.steps = []
        self.data = None
        self.metadata = {}

    def add_step(self, func: Callable, name: str = None, **kwargs):
        """Add processing step to pipeline"""
        step = {
            'function': func,
            'name': name or func.__name__,
            'kwargs': kwargs
        }
        self.steps.append(step)
        return self

    def run(self, data):
        """Execute pipeline on data"""
        self.data = data

        for step in self.steps:
            print(f"Executing: {step['name']}")
            try:
                self.data = step['function'](self.data, **step['kwargs'])
            except Exception as e:
                print(f"Error in step {step['name']}: {e}")
                raise

        return self.data

    def parallel_run(self, data_chunks: List[Any], n_workers: int = None):
        """Run pipeline in parallel on data chunks"""
```

```
        n_workers = n_workers or mp.cpu_count()

        with mp.Pool(n_workers) as pool:
            results = pool.map(self.run, data_chunks)

        return results

    def save_pipeline(self, filename: str):
        """Save pipeline configuration"""
        import pickle
        with open(filename, 'wb') as f:
            pickle.dump(self.steps, f)

    @classmethod
    def load_pipeline(cls, filename: str):
        """Load pipeline configuration"""
        import pickle
        pipeline = cls()
        with open(filename, 'rb') as f:
            pipeline.steps = pickle.load(f)
        return pipeline

# Custom data transformers
class DataTransformers:
    """Collection of data transformation functions"""

    @staticmethod
    def remove_outliers(df: pd.DataFrame, columns: List[str], method: str = 'iqr', threshold: f
        """Remove outliers from dataframe"""
        df_clean = df.copy()

        for column in columns:
            if method == 'iqr':
                Q1 = df_clean[column].quantile(0.25)
                Q3 = df_clean[column].quantile(0.75)
                IQR = Q3 - Q1

                lower_bound = Q1 - threshold * IQR
                upper_bound = Q3 + threshold * IQR

                df_clean = df_clean[
                    (df_clean[column] >= lower_bound) &
                    (df_clean[column] <= upper_bound)
                ]

            elif method == 'zscore':
                z_scores = np.abs(stats.zscore(df_clean[column]))
```

```
            df_clean = df_clean[z_scores < threshold]

    return df_clean

@staticmethod
def handle_missing_values(df: pd.DataFrame, strategy: Dict[str, str]):
    """Handle missing values with different strategies per column"""
    df_filled = df.copy()

    for column, method in strategy.items():
        if column not in df_filled.columns:
            continue

        if method == 'mean':
            df_filled[column].fillna(df_filled[column].mean(), inplace=True)
        elif method == 'median':
            df_filled[column].fillna(df_filled[column].median(), inplace=True)
        elif method == 'mode':
            df_filled[column].fillna(df_filled[column].mode()[0], inplace=True)
        elif method == 'forward_fill':
            df_filled[column].fillna(method='ffill', inplace=True)
        elif method == 'backward_fill':
            df_filled[column].fillna(method='bfill', inplace=True)
        elif method == 'interpolate':
            df_filled[column].interpolate(inplace=True)
        elif method == 'drop':
            df_filled.dropna(subset=[column], inplace=True)

    return df_filled

@staticmethod
def feature_engineering(df: pd.DataFrame, features: Dict[str, Callable]):
    """Create new features based on existing ones"""
    df_enhanced = df.copy()

    for feature_name, feature_func in features.items():
        df_enhanced[feature_name] = feature_func(df_enhanced)

    return df_enhanced

@staticmethod
def encode_categorical(df: pd.DataFrame, columns: List[str], method: str = 'onehot'):
    """Encode categorical variables"""
    df_encoded = df.copy()

    for column in columns:
        if method == 'onehot':
```

```
                dummies = pd.get_dummies(df_encoded[column], prefix=column)
                df_encoded = pd.concat([df_encoded, dummies], axis=1)
                df_encoded.drop(column, axis=1, inplace=True)

            elif method == 'label':
                le = LabelEncoder()
                df_encoded[column] = le.fit_transform(df_encoded[column])

            elif method == 'ordinal':
                # Requires mapping dictionary
                pass

        return df_encoded

# Large-scale data processing
class LargeDataProcessor:
    """Process large datasets efficiently"""

    def __init__(self, chunk_size: int = 10000):
        self.chunk_size = chunk_size

    def process_csv_in_chunks(self, filename: str, processor: Callable):
        """Process large CSV file in chunks"""
        results = []

        for chunk in pd.read_csv(filename, chunksize=self.chunk_size):
            result = processor(chunk)
            results.append(result)

        return pd.concat(results, ignore_index=True)

    def process_with_dask(self, filename: str, processor: Callable):
        """Process data using Dask for parallel computing"""
        ddf = dd.read_csv(filename)

        # Apply processor to each partition
        result = ddf.map_partitions(processor)

        # Compute result
        return result.compute()

    def stream_process(self, data_source, processor: Callable, buffer_size: int = 1000):
        """Stream processing for real-time data"""
        buffer = []

        for record in data_source:
            buffer.append(record)
```

```
            if len(buffer) >= buffer_size:
                # Process buffer
                processed = processor(buffer)
                yield processed
                buffer = []

        # Process remaining records
        if buffer:
            yield processor(buffer)

# Time series analysis
class TimeSeriesAnalyzer:
    """Advanced time series analysis"""

    @staticmethod
    def detect_seasonality(series: pd.Series, periods: List[int] = None):
        """Detect seasonality in time series"""
```