YZM1022

Advanced Programming

Week 15: Review and Project Presentations

Instructor: Ekrem Çetinkaya

Date: 02.06.2026

Today's Agenda

Final Session

Part 1: Course Review (~40 min)

- All topics recap
- Key concepts summary
- Common patterns

Part 2: Project Presentations (~60 min)

- Student projects showcase
- Q&A for each project
- Feedback and discussion

Part 3: Exam Preparation (~20 min)

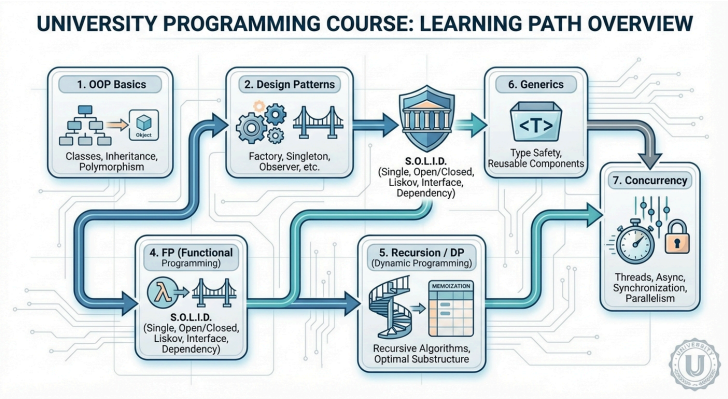
- What to expect
- Study tips
- Final questions

Course Review

Course Journey

What We Covered

Weeks	Topic	Focus
1-3	OOP Fundamentals	Classes, Inheritance, Polymorphism
4-6	Design Patterns	Creational, Structural, Behavioral
7	Software Engineering	SOLID, Clean Code, Architecture
9-10	Functional Programming	HOF, Closures, Immutability
11	Algorithms	Recursion, Dynamic Programming
12	Generics	Type Hints, TypeVar, Protocols
13-14	Concurrency	Threads, Processes, Async



Weeks 1-3: OOP Fundamentals

Key Concepts

- **Classes & Objects**: Blueprints and instances
- **Encapsulation**: Hide internal state
- **Inheritance**: Code reuse via hierarchy
- **Polymorphism**: Same interface, different behavior
- **Composition**: "Has-a" relationships
- **Abstract classes**: Define contracts

Code Example

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self) -> str:
        pass

class Dog(Animal):
    def speak(self) -> str:
        return "Woof!"

class Cat(Animal):
    def speak(self) -> str:
        return "Meow!"

# Polymorphism in action
animals: list[Animal] = [Dog(), Cat()]
for animal in animals:
    print(animal.speak())
```

Weeks 4-6: Design Patterns

Creational Patterns

- **Singleton**: One instance only
- **Factory**: Create without specifying class
- **Builder**: Step-by-step construction

Structural Patterns

- **Adapter**: Bridge incompatible interfaces
- **Decorator**: Add behavior dynamically
- **Facade**: Simplified interface

Behavioral Patterns

- **Observer**: Notify on state change
- **Strategy**: Swappable algorithms
- **Command**: Encapsulate actions

```
# Strategy Pattern
class PaymentStrategy(Protocol):
    def pay(self, amount: float) -> None: ...

class CreditCard:
    def pay(self, amount: float) -> None:
        print(f"Credit: ${amount}")

class PayPal:
    def pay(self, amount: float) -> None:
        print(f"PayPal: ${amount}")

def checkout(strategy: PaymentStrategy):
    strategy.pay(99.99)
```

Week 7: SOLID Principles

Principle	Meaning	Violation Example
Single Responsibility	One reason to change	Class does UI + DB + Logic
Open/Closed	Open for extension, closed for modification	Adding `if` for every new type
Liskov Substitution	Subtypes are substitutable	Square.setWidth breaks Rectangle
Interface Segregation	Small, specific interfaces	`IAnimal` with fly() method
Dependency Inversion	Depend on abstractions	Class creates its own database

```python
# DIP Example - Depend on abstraction
class OrderService:
    def __init__(self, repository: Repository):  # Abstraction
        self.repo = repository

# Can inject any implementation
service = OrderService(PostgresRepository())
service = OrderService(MongoRepository())
service = OrderService(MockRepository())  # Testing!
```

Weeks 9-10: Functional Programming

Core Concepts

- **Pure functions**: No side effects
- **Immutability**: Don't mutate data
- **Higher-order functions**: Functions as values
- **Closures**: Functions with environment
- **Map, Filter, Reduce**: Data transformations

Code Example

```python
from functools import reduce

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

def apply_twice(f, x):
    return f(f(x))

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

double = multiplier(2)
print(double(5))  # 10
numbers = [1, 2, 3, 4, 5]
result = reduce(
    lambda acc, x: acc + x,
    filter(lambda x: x % 2 == 0,
           map(lambda x: x ** 2, numbers))
)  # 4 + 16 = 20
```

Week 11: Recursion & Dynamic Programming

Recursion

- **Base case**: Stop condition
- **Recursive case**: Call itself
- **Call stack**: Memory consideration

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    return n * factorial(n - 1)
```

Tree Traversal

```
def traverse(node):
    if node is None:
        return
    print(node.value)
    traverse(node.left)
    traverse(node.right)
```

Dynamic Programming

- **Memoization**: Cache results (top-down)
- **Tabulation**: Build table (bottom-up)

```
# Memoization
from functools import cache

@cache
def fib(n: int) -> int:
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

# Tabulation
def fib_tab(n: int) -> int:
    if n < 2:
        return n
    dp = [0, 1]
    for i in range(2, n + 1):
        dp.append(dp[-1] + dp[-2])
    return dp[n]
```

Week 12: Generics & Type Systems

Type Hints

```python
from typing import TypeVar, Generic

# Basic annotations
def greet(name: str) -> str:
    return f"Hello, {name}!"

# Collection types
def first(items: list[int]) -> int:
    return items[0]

# Optional and Union
def find(id: int) -> str | None:
    return db.get(id)
```

Generics

```python
T = TypeVar('T')

def first(items: list[T]) -> T:
    return items[0]

class Stack(Generic[T]):
    def __init__(self):
        self._items: list[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> T:
        return self._items.pop()

# Protocol for structural typing
class Comparable(Protocol):
    def __lt__(self, other) -> bool: ...
```

Weeks 13-14: Concurrency

Threading (Week 13)

- **Thread creation**: `threading.Thread`
- **Race conditions**: Shared state bugs
- **Synchronization**: Lock, Semaphore
- **GIL**: Python's limitation
- Best for: I/O-bound tasks

```
import threading

lock = threading.Lock()

def worker():
    with lock:
        # Critical section
        pass

thread = threading.Thread(target=worker)
thread.start()
thread.join()
```

Async & Parallel (Week 14)

```
# asyncio - I/O concurrency
import asyncio

async def fetch(url):
    await asyncio.sleep(1)
    return f"Data from {url}"

async def main():
    results = await asyncio.gather(
        fetch("url1"),
        fetch("url2"),
    )

# multiprocessing - CPU parallelism
from multiprocessing import Pool

def compute(x):
    return x ** 2

with Pool(4) as pool:
    results = pool.map(compute, range(10))
```

When to Use What?

Task Type	Best Approach	Why
I/O-bound, few tasks	Threading	Simple, shared memory
I/O-bound, many tasks	asyncio	Lightweight, scalable
CPU-bound	multiprocessing	True parallelism
Mixed workload	Combine approaches	Best of both worlds

The combined approach pairs asyncio for fast I/O fetching with multiprocessing for CPU-heavy processing — each tool doing what it does best.

```
# Combined approach
async def process_data(urls: list[str]):
    # Fetch with asyncio
    async with aiohttp.ClientSession() as session:
        pages = await asyncio.gather(*[fetch(session, u) for u in urls])

    # Process with multiprocessing
    loop = asyncio.get_event_loop()
    with ProcessPoolExecutor() as pool:
        results = await asyncio.gather(*[
            loop.run_in_executor(pool, parse, page)
            for page in pages
        ])
    return results
```

Key Programming Paradigms

Programming Paradigms		
OOP	Functional	Concurrent
Classes Inheritance Polymorphism Encapsulation Composition	Pure functions Immutability HOF Closures Map/Filter	Threads Processes Async/Await Synchronization Message passing
Python supports ALL paradigms! Choose based on the problem.		

Essential Patterns Summary

Design

- **Singleton** - Global state
- **Factory** - Object creation
- **Observer** - Event systems
- **Strategy** - Algorithm selection
- **Decorator** - Extend behavior

Code Quality

- **SOLID principles**
- **Clean code practices**
- **Type hints for clarity**
- **Test-driven development**

Concurrency

- **Producer-Consumer**
- **Worker Pool**
- **Rate Limiting**
- **Circuit Breaker**

Functional

- **Pipeline processing**
- **Memoization**
- **Immutable data structures**
- **Pure functions**

Project Presentations

Project Presentation Guidelines

Time: 5-7 minutes per project

What to Cover

1. **Problem statement** - What problem does it solve?
2. **Architecture** - High-level design
3. **Key features** - What makes it interesting?
4. **Technical highlights** - Patterns/techniques used
5. **Demo** - Show it working!
6. **Challenges** - What was hard?
7. **Learnings** - What did you learn?

Evaluation Criteria

- Code quality and organization
- Use of course concepts
- Creativity and problem-solving
- Presentation clarity

Project Showcase

🎤 Presentations Begin!

Each presenter:

- Present your project (5-7 min)
- Q&A from class (2-3 min)
- Feedback from instructor

Remember:

- Focus on the interesting parts
- Don't apologize for what's missing
- Be proud of what you built!

Exam Preparation

Exam Format

What to Expect

Format: Open-book, open-internet
Duration: 90 minutes
Questions: Mix of code writing and analysis

Question Types

1. **Concept questions** - Explain why/when/how
2. **Code analysis** - Find bugs, predict output
3. **Code writing** - Implement solutions
4. **Design questions** - Choose patterns, justify

Allowed Resources

- Notes, slides, textbooks
- Internet search, documentation
- AI tools (ChatGPT, Claude, etc.)

⚠️ **Not allowed:** Communicating with others

Topics to Review

High Priority

- OOP fundamentals
- SOLID principles
- Common design patterns
- Type hints and generics
- Threading basics
- async/await syntax

Medium Priority

- Functional programming
- Dynamic programming
- Protocol typing
- ProcessPoolExecutor
- Synchronization primitives

Sample Questions

> Implement Observer pattern for a stock price tracker

> Why would this code cause a deadlock?

> Convert this synchronous code to async

> Which SOLID principle is violated? How to fix?

> Write a thread-safe counter class

> Use memoization to optimize this recursive function

Study Tips

Do ✅

- **Practice coding** by hand
- **Explain concepts** out loud
- Review **your own code** from labs
- Create **cheat sheets**
- Focus on **understanding**, not memorizing
- Get comfortable with **Python docs**

Don't ❌

- Don't memorize syntax
- Don't cram the night before
- Don't ignore practice problems
- Don't skip the slides
- Don't rely solely on AI during prep
  - Use it for learning, not shortcuts

Common Mistakes to Avoid

These five bugs appear in almost every early Python codebase — recognizing the pattern is faster than debugging from scratch.

```
# ❌ Forgetting super().__init__()
class Child(Parent):
    def __init__(self):
        # super().__init__() missing!
        pass
# ❌ Mutable default arguments
def add_item(item, items=[]):  # Bug!
    items.append(item)
    return items
# ❌ Not using context managers
lock.acquire()
do_work()
lock.release()  # If do_work raises, never released!

# ❌ Modifying list while iterating
for item in items:
    if condition:
        items.remove(item)  # Bug!

# ❌ Forgetting to await
async def main():
    result = fetch_data()  # Missing await!
```

Quick Reference Card — Part 1

OOP class, __init__, self, super(), @property, @abstractmethod
Patterns Singleton, Factory, Observer, Strategy, Decorator
SOLID Single Resp, Open/Closed, Liskov, Interface Seg, Dep Inv
Functional map, filter, reduce, lambda, @cache, functools

Quick Reference Card — Part 2

```
Types
  TypeVar, Generic[T], Protocol, list[int], str | None
```

```
Threading
  Thread, Lock, with lock:, Queue, ThreadPoolExecutor
```

```
Async
  async def, await, asyncio.gather, asyncio.create_task
```

```
Multiprocessing
  Process, Pool, ProcessPoolExecutor, Queue, Value
```

Final Questions?

Topics for Discussion

- Any concept that's still unclear?
- Specific exam concerns?
- Career advice?
- Further learning resources?

Course Takeaways

What You've Learned

1. **Think in objects** - Model real-world entities
2. **Design with patterns** - Proven solutions to common problems
3. **Write clean code** - Future you will thank you
4. **Use types** - Catch bugs before they run
5. **Think functionally** - Immutability and purity
6. **Handle concurrency** - Right tool for the job

What's Next?

- **Practice, practice, practice**
- Build real projects
- Contribute to open source
- Never stop learning!

Resources for Continued Learning

Books

- *Clean Code* - Robert Martin
- *Design Patterns* - Gang of Four
- *Fluent Python* - Luciano Ramalho
- *Effective Python* - Brett Slatkin

Online

- Real Python
- Python Docs
- PEP 8
- mypy docs

Practice

- LeetCode
- HackerRank
- Exercism
- Advent of Code

Projects

- Build a web scraper
- Create a CLI tool
- Contribute to open source
- Build something you'd use!

Thank You!

It's Been a Great Semester! 🎉

- **Email:** ekrem.cetinkaya@yildiz.edu.tr
- **Office Hours:** Tuesday 14:00-16:00 - Room F-B21
- **Course Repository:** GitHub Link

Good Luck on Your Exams!

Remember: The goal isn't to pass the exam, it's to become a better programmer.

> "The best time to plant a tree was 20 years ago. The second best time is now."
>
> — Chinese Proverb

Keep coding! 🐍