# ****W8D3 -- Feature Integration: Component Integration****

JTC Program: Tech Pathways Cohort: S25 Lesson Plan: Feature Integration - Component Integration Type: Lesson Plan Week / Day: W8D3 Version Date: 05/23/2025

## ****Focus Concepts****

* Understanding how individual Python functions work together to solve complex problems
* Learning to design and implement classes that integrate multiple components
* Exploring module integration patterns with Python standard library
* Building advanced systems that combine multiple classes and functions effectively
* Developing enterprise-level integration patterns for real-world applications
* Applying component integration concepts to create maintainable and scalable code

## ****Learning Objectives****

By the end of this session, fellows will be able to:

* Explain the principles of feature integration and component-based design
* Create functions that work together to solve multi-step problems
* Design classes that integrate multiple components and responsibilities
* Implement systems that combine custom code with Python standard library modules
* Build advanced multi-component systems with proper data flow
* Apply enterprise-level integration patterns to create production-ready applications

## ****Out-of-Scope Objectives****

* Advanced design patterns (Factory, Observer, Strategy patterns)
* Database integration and ORM frameworks
* Web framework integration (Django, Flask)
* API development and microservices architecture
* Advanced testing frameworks and test-driven development
* Performance optimization and profiling techniques

## ****Required Competencies****

* Solid understanding of Python functions and parameters
* Familiarity with Python classes, objects, and basic OOP concepts
* Experience with Python data structures (lists, dictionaries, tuples)
* Basic understanding of Python modules and imports
* Comfort with control flow statements and error handling

## ****Technical Requirements****

* Python 3.x installed
* Code editor or IDE (VS Code, PyCharm, or Jupyter Notebook)
* Basic familiarity with command line/terminal
* Required libraries: random, datetime, math (standard library modules)

## ****Prerequisites****

* Completion of Python Fundamentals (Functions and Classes)
* Understanding of basic object-oriented programming concepts
* Experience with Python data types and control structures
* Familiarity with file I/O and basic error handling

## ****Assigned Reading & Pre-Class Learning****

Estimated Time: 25 minutes

Resources:

* [Python Module Documentation](https://docs.python.org/3/tutorial/modules.html) - Understanding Python modules and packages - 10 minutes
* [Object-Oriented Programming in Python](https://realpython.com/python3-object-oriented-programming/) - Review of OOP concepts and composition - 15 minutes

## ****Before-Class Mini Quiz Questions (5 questions)****

1. What is the main advantage of breaking code into smaller, reusable functions?
   * A) It makes the code run faster
   * \*B) It improves code organization, reusability, and maintainability
   * C) It reduces the total amount of code needed
   * D) It eliminates the need for comments
2. In object-oriented programming, what is composition?
   * A) Writing code in a specific order
   * \*B) Building complex objects by combining simpler objects
   * C) Inheriting from multiple parent classes
   * D) Using only built-in Python functions
3. Which Python statement is used to import functions from a module?
   * A) include
   * B) require
   * \*C) import
   * D) load
4. What is a key benefit of using classes to organize related functions and data?
   * A) Classes are always faster than functions
   * \*B) Classes encapsulate related functionality and data together
   * C) Classes eliminate the need for function parameters
   * D) Classes automatically handle all errors
5. When integrating multiple components, what is most important for maintainable code?
   * A) Using the shortest variable names possible
   * B) Writing everything in one large function
   * \*C) Clear interfaces and well-defined responsibilities between components
   * D) Avoiding the use of any external modules

## ****Key Terms****

* **Component Integration**: The process of combining multiple code components to work together effectively
* **Function Composition**: Using the output of one function as input to another function
* **Class Composition**: Building complex classes by combining simpler classes as components
* **Module Integration**: Incorporating functionality from Python's standard library or external packages
* **Interface**: The defined way that components communicate with each other
* **Encapsulation**: Bundling data and methods together within a class to hide internal details
* **Separation of Concerns**: Designing components so each has a single, well-defined responsibility
* **Data Flow**: How information moves between different components in a system
* **Configuration Management**: Organizing and managing system settings and parameters
* **Dependency Injection**: Providing dependencies to a component rather than having it create them internally
* **Factory Pattern**: A design approach for creating objects without specifying exact classes
* **Service Layer**: A layer that coordinates between different components to provide business logic
* **Integration Testing**: Testing how different components work together as a system
* **API Design**: Creating clear interfaces for components to interact with each other
* **System Architecture**: The overall structure and organization of a software system

## ****Lesson Schedule & Detailed Script****

### ****6:30 PM -- 6:45 PM: Interactive Check-In****

**Instructor Script:** "Welcome to Week 8, Day 3! Today we're diving into one of the most important concepts in software development: feature integration and component integration. Think about your smartphone - it's not just one giant piece of code, but thousands of components working together seamlessly. The camera app integrates with the photo library, which integrates with cloud storage, which integrates with sharing features. By the end of today, you'll understand how to build systems where individual pieces of code work together to create something much more powerful than the sum of their parts."

**Admin Tasks:**

* Take attendance
* Ensure everyone has Python installed and working
* Check that students can create and run basic Python files

**Prompting Questions:**

* "Can you think of an app or website where you can see multiple features working together?"
* "What happens when you order food delivery online? How many different systems do you think are involved?"

**Poll Questions:**

* "On a scale of 1-5, how comfortable are you with writing Python functions?"
* "Have you ever written a Python class before? Yes/No"

### ****6:45 PM -- 7:15 PM: Session 1 -- Basic Function Integration****

**Objective:** Understand how individual functions can work together to solve complex problems.

**Instructor Script:** "Let's start with the foundation of integration: making functions work together. Just like a recipe where you prepare ingredients separately but combine them to create the final dish, we can write functions that each do one thing well, then combine them to solve bigger problems."

#### ****Understanding Function Integration:****

# basic\_functions.py - Live coding demonstration

def greet\_user(name):

"""Function to greet a user by name"""

return f"Hello, {name}!"

def get\_age\_category(age):

"""Function to categorize age groups"""

if age < 13:

return "child"

elif age < 20:

return "teenager"

elif age < 60:

return "adult"

else:

return "senior"

def create\_user\_profile(name, age):

"""Integration function that combines greeting and age categorization"""

greeting = greet\_user(name)

category = get\_age\_category(age)

profile = {

"greeting": greeting,

"age\_category": category,

"full\_message": f"{greeting} You are classified as a {category}."

}

return profile

# Example usage

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

# Testing individual functions

print("Testing individual functions:")

print(greet\_user("Alice"))

print(get\_age\_category(16))

print("\nTesting integrated function:")

# Testing integrated function

user\_info = create\_user\_profile("Bob", 25)

print(user\_info["full\_message"])

**Key Concepts to Emphasize:**

1. **Single Responsibility**: Each function does one thing well
2. **Reusability**: Functions can be used independently or together
3. **Data Flow**: Output of one function becomes input to another
4. **Integration Function**: A function that coordinates other functions
5. **Testing**: Test both individual functions and their integration

**Live Coding Activity (10 minutes):** Students will modify the code to add a new function that validates email addresses and integrate it into the user profile creation.

### ****7:15 PM -- 7:45 PM: Session 2 -- Basic Class Integration****

**Objective:** Learn how to design classes that work together to solve problems.

**Instructor Script:** "Now let's level up from functions to classes. Classes allow us to bundle related functions and data together, and when we combine multiple classes, we can create powerful systems where each class has its own responsibility but they work together seamlessly."

#### ****Building Integrated Class Systems:****

# simple\_classes.py - Live coding demonstration

class Calculator:

"""A simple calculator class with basic operations"""

def add(self, a, b):

return a + b

def subtract(self, a, b):

return a - b

def multiply(self, a, b):

return a \* b

def divide(self, a, b):

if b == 0:

return "Error: Cannot divide by zero"

return a / b

class NumberFormatter:

"""A class to format numbers in different ways"""

def format\_currency(self, amount):

return f"${amount:.2f}"

def format\_percentage(self, decimal):

return f"{decimal \* 100:.1f}%"

def format\_rounded(self, number, places=2):

return round(number, places)

class MathProcessor:

"""Integration class that uses Calculator and NumberFormatter together"""

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

self.calc = Calculator()

self.formatter = NumberFormatter()

def calculate\_tip(self, bill\_amount, tip\_percentage):

"""Calculate tip and format results"""

# Convert percentage to decimal

tip\_decimal = tip\_percentage / 100

# Calculate tip amount using Calculator

tip\_amount = self.calc.multiply(bill\_amount, tip\_decimal)

# Calculate total using Calculator

total = self.calc.add(bill\_amount, tip\_amount)

# Format results using NumberFormatter

formatted\_bill = self.formatter.format\_currency(bill\_amount)

formatted\_tip = self.formatter.format\_currency(tip\_amount)

formatted\_total = self.formatter.format\_currency(total)

formatted\_percentage = self.formatter.format\_percentage(tip\_decimal)

return {

"bill": formatted\_bill,

"tip": formatted\_tip,

"total": formatted\_total,

"tip\_rate": formatted\_percentage

}

**Key Class Integration Patterns:**

1. **Composition**: One class contains instances of other classes
2. **Delegation**: One class passes work to another class
3. **Coordination**: A master class coordinates multiple worker classes
4. **Clear Interfaces**: Each class has well-defined methods for interaction

**Student Exercise (10 minutes):** Students will add a new TaxCalculator class and integrate it into the MathProcessor to calculate tax on the bill total.

### ****7:45 PM -- 8:05 PM: Capstone Work Session****

**Activity:** Work on capstone project, applying integration concepts learned so far.

### ****8:05 PM -- 8:15 PM: Break****

10-minute break

### ****8:15 PM -- 8:45 PM: Session 3 -- Module Integration Patterns****

**Objective:** Learn to integrate custom code with Python's standard library modules.

**Instructor Script:** "Python's strength comes not just from the language itself, but from its vast ecosystem of modules. Today we'll learn how to integrate our custom classes with powerful built-in modules like random, datetime, and math to create real-world applications."

#### ****Integrating with Standard Library Modules:****

# module\_integration.py - Live coding demonstration

import random

import datetime

import math

class WeatherSimulator:

"""Simulates weather data using random module"""

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

self.weather\_types = ["sunny", "cloudy", "rainy", "snowy"]

def get\_random\_temperature(self, season="spring"):

"""Generate random temperature based on season"""

temp\_ranges = {

"spring": (50, 75),

"summer": (70, 95),

"fall": (45, 70),

"winter": (20, 50)

}

min\_temp, max\_temp = temp\_ranges.get(season, (50, 75))

return random.randint(min\_temp, max\_temp)

def get\_random\_weather(self):

"""Get random weather type"""

return random.choice(self.weather\_types)

class DateTimeHelper:

"""Helper class using datetime module"""

def get\_current\_info(self):

"""Get current date and time information"""

now = datetime.datetime.now()

return {

"date": now.strftime("%Y-%m-%d"),

"time": now.strftime("%H:%M:%S"),

"day\_of\_week": now.strftime("%A"),

"month": now.strftime("%B")

}

def get\_season(self, month=None):

"""Determine season based on month"""

if month is None:

month = datetime.datetime.now().month

if month in [12, 1, 2]:

return "winter"

elif month in [3, 4, 5]:

return "spring"

elif month in [6, 7, 8]:

return "summer"

else:

return "fall"

class WeatherReport:

"""Integration class that combines all weather-related functionality"""

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

self.weather\_sim = WeatherSimulator()

self.date\_helper = DateTimeHelper()

def generate\_daily\_report(self):

"""Generate a complete weather report"""

# Get current date info

date\_info = self.date\_helper.get\_current\_info()

current\_month = datetime.datetime.now().month

season = self.date\_helper.get\_season(current\_month)

# Generate weather data

temperature = self.weather\_sim.get\_random\_temperature(season)

weather\_type = self.weather\_sim.get\_random\_weather()

# Create comprehensive report

report = f"""

Weather Report for {date\_info['day\_of\_week']}, {date\_info['date']}

Generated at: {date\_info['time']}

Current Season: {season.title()}

Weather Condition: {weather\_type.title()}

Temperature: {temperature}°F

"""

return report.strip()

**Key Module Integration Concepts:**

1. **Import Strategy**: Choosing what to import and how
2. **Module Responsibilities**: Understanding what each module provides
3. **Integration Patterns**: How to combine module functionality with custom classes
4. **Error Handling**: Managing potential module-specific errors

**Hands-on Activity (15 minutes):** Students will add a MathCalculator class that uses the math module to calculate wind chill factors and integrate it into the weather report system.

### ****8:45 PM -- 9:15 PM: Session 4 -- Advanced System Integration****

**Objective:** Build complex systems that integrate multiple components with proper architecture.

**Instructor Script:** "Now we're ready for the big leagues. We'll build a complete student management system that demonstrates enterprise-level integration patterns. This system will show you how real-world applications are structured, with proper separation of concerns and clean interfaces between components."

#### ****Enterprise-Level Integration Architecture:****

# advanced\_system.py - Guided implementation

class DataProcessor:

"""Handles all data processing logic"""

@staticmethod

def calculate\_grade(score, max\_score=100):

"""Calculate letter grade from numeric score"""

percentage = (score / max\_score) \* 100

if percentage >= 90:

return 'A'

elif percentage >= 80:

return 'B'

elif percentage >= 70:

return 'C'

elif percentage >= 60:

return 'D'

else:

return 'F'

@staticmethod

def calculate\_statistics(scores):

"""Calculate statistical measures for a list of scores"""

if not scores:

return {"error": "No scores provided"}

avg = sum(scores) / len(scores)

return {

"average": round(avg, 2),

"highest": max(scores),

"lowest": min(scores),

"count": len(scores)

}

class Student:

"""Student entity with integrated data processing"""

def \_\_init\_\_(self, student\_id, first\_name, last\_name, grade\_level):

self.student\_id = student\_id

self.first\_name = first\_name

self.last\_name = last\_name

self.grade\_level = grade\_level

self.assignments = {}

self.processor = DataProcessor() # Integration!

def add\_assignment(self, assignment\_name, score, max\_score=100):

"""Add an assignment score"""

self.assignments[assignment\_name] = {

"score": score,

"max\_score": max\_score,

"letter\_grade": self.processor.calculate\_grade(score, max\_score)

}

def get\_grade\_summary(self):

"""Get comprehensive grade summary using DataProcessor"""

if not self.assignments:

return {"message": "No assignments recorded"}

scores = [assignment["score"] for assignment in self.assignments.values()]

stats = self.processor.calculate\_statistics(scores)

return {

"student": f"{self.last\_name}, {self.first\_name}",

"statistics": stats,

"assignment\_count": len(self.assignments)

}

class Classroom:

"""Classroom management system integrating multiple components"""

def \_\_init\_\_(self, class\_name, teacher\_name):

self.class\_name = class\_name

self.teacher\_name = teacher\_name

self.students = {}

self.processor = DataProcessor()

def enroll\_student(self, student):

"""Enroll a student in the classroom"""

self.students[student.student\_id] = student

def generate\_class\_report(self):

"""Generate comprehensive class report"""

report\_lines = [

f"CLASS REPORT: {self.class\_name}",

f"Teacher: {self.teacher\_name}",

f"Students Enrolled: {len(self.students)}",

"",

"STUDENT PERFORMANCE:"

]

for student in self.students.values():

summary = student.get\_grade\_summary()

if "statistics" in summary:

report\_lines.append(f"• {summary['student']}: Avg {summary['statistics']['average']}%")

return "\n".join(report\_lines)

**Advanced Integration Concepts:**

1. **Layered Architecture**: Separating data processing, business logic, and presentation
2. **Dependency Injection**: Providing dependencies rather than creating them internally
3. **Service Objects**: Classes that coordinate between multiple other classes
4. **Interface Design**: Creating clear contracts between components

**Team Coding Exercise (20 minutes):** Students work in pairs to extend the system by adding a GradeCalculator service and a ReportGenerator class that integrates with all existing components.

### ****9:15 PM -- 9:25 PM: Breakout #2: Integration Challenge****

**Activity:** Students work individually or in pairs to solve an integration challenge: "Build a simple e-commerce system that integrates a Product catalog, Shopping cart, and Payment processor using the patterns learned today."

**Deliverable:** A working Python system with at least 3 integrated classes.

### ****9:25 PM -- 9:30 PM: Wrap-Up & Final Questions****

**Instructor Script:** "Today you've learned the fundamental patterns of component integration that power every major software system. From simple function composition to enterprise-level architecture, these concepts will serve you throughout your programming career. Remember: good integration isn't about complex code—it's about clear interfaces, single responsibilities, and components that work together seamlessly."

**Review Key Points:**

* Functions can be composed to solve complex problems
* Classes provide encapsulation and can be integrated through composition
* Python's standard library provides powerful modules for integration
* Enterprise systems use layered architectures with clear separation of concerns
* Good integration patterns make code more maintainable, testable, and scalable

**Prompting Question:** "What's one integration pattern you learned today that you could apply to your capstone project?"

**Next Steps:** "Tomorrow we'll build on these integration concepts to learn about data structures and algorithms, showing how different data storage patterns integrate with the processing logic we've learned today."

## ****After-Class Quiz (5 questions)****

1. What is the main benefit of function composition in programming?
   * A) It makes functions run faster
   * \*B) It allows you to solve complex problems by combining simple functions
   * C) It reduces the number of variables needed
   * D) It eliminates the need for classes
2. In class composition, what does it mean when one class "has-a" relationship with another?
   * A) One class inherits from another
   * B) One class calls functions from another module
   * \*C) One class contains an instance of another class as a component
   * D) One class has the same methods as another
3. When integrating with Python standard library modules, what is the best practice?
   * A) Import everything with from module import \*
   * B) Only use modules in the main function
   * \*C) Import only what you need and organize module usage in appropriate classes
   * D) Avoid using standard library modules in classes
4. What is a key characteristic of good component integration?
   * A) All components should know about all other components
   * B) Components should share as much data as possible
   * \*C) Components should have clear, well-defined interfaces
   * D) Components should be as large as possible
5. In enterprise-level integration, what is the purpose of a "service layer"?
   * A) To store all the data for the application
   * B) To handle user interface interactions
   * \*C) To coordinate between different components and provide business logic
   * D) To manage database connections only

## ****Homework Assignment****

**Project:** "Personal Finance Tracker Integration System"

**Objective:** Build a personal finance tracker that integrates multiple components to manage income, expenses, and financial goals.

**Requirements:**

1. Create at least 4 integrated classes:
   * Transaction (for individual income/expense records)
   * BudgetCalculator (for budget analysis)
   * GoalTracker (for savings goals)
   * FinanceManager (main coordinator class)
2. Integrate with at least 2 standard library modules:
   * datetime for transaction dates
   * json for saving/loading data
3. Implement proper integration patterns:
   * Single responsibility for each class
   * Clear interfaces between components
   * One main coordinator class that integrates others
4. Include error handling and basic validation

**Deliverables:**

* Complete Python code with all classes
* Demonstration script showing the integrated system working
* Brief documentation explaining how your components integrate

**Due:** Before next class session

**Evaluation Criteria:**

* Correct implementation of integration patterns (40%)
* Proper use of classes and methods (30%)
* Code organization and documentation (20%)
* Creative features and error handling (10%)