Python for Software Engineering

- **Python** – an interpreted, high-level, general-purpose programming language known for its readable syntax. Python's design philosophy emphasizes code clarity and a simple syntax that helps developers write clear, logical code.
- Python supports multiple paradigms (procedural, object-oriented, functional). In this course we use Python to illustrate object-oriented programming (OOP) concepts because of its simplicity and widespread use.
- Key Python features include dynamic typing (you don't declare variable types), a rich standard library, and cross-platform portability. This allows rapid prototyping and experimentation while learning design principles.

Object-Oriented Programming Basics

- **Object-Oriented Programming (OOP)** is a paradigm centered on *objects* (instances of classes) that encapsulate data and behavior. OOP promotes organizing software as a collection of interacting objects.
- **Four Pillars of OOP:**
  - *Abstraction* – modeling real-world entities by focusing on relevant features while ignoring unnecessary details.
  - *Encapsulation* – hiding internal state and implementation details of an object behind a public interface.
  - *Inheritance* – creating new classes by extending existing ones, enabling reuse of code and establishing hierarchies.
  - *Polymorphism* – treating objects of different subclasses through a common interface, and automatically calling the correct overridden behaviors at runtime.

Abstraction & Inheritance in Practice

Abstraction Example:

```
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def send_notification(self):
        # Abstract the complex email sending logic
        pass
```

Inheritance Example:

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Manager(Employee):  # Inherits from Employee
```

Abstraction & Inheritance Concepts

- **Abstraction:** In design, identify the essential attributes of an entity and model a class for it, omitting extraneous details. For instance, a "User" class might abstract common user properties (name, email) relevant to the system, leaving out personal details not needed. Abstraction allows focusing on *what* an object does instead of *how* it does it.
- **Inheritance:** Establishes an "is-a" relationship between classes. A base (parent) class defines general attributes and methods, and a derived (child) class inherits those and extends or overrides behavior. Inheritance promotes code reuse – new classes reuse existing code instead of writing from scratch.
- Using inheritance wisely can reduce duplication, but be cautious: subclasses should truly satisfy the relationship (Liskov Substitution Principle). If a subclass violates expectations of the parent class, the design can break. Prefer composition over inheritance when appropriate to avoid rigid hierarchies.

UML Diagrams for Design

- **Unified Modeling Language (UML)** provides a standard way to visualize system design. UML diagrams are blueprints for understanding software structure and behavior without diving into code.
- **Structure Diagrams:** e.g. *Class Diagrams* show classes, their attributes and methods, and relationships (associations, inheritance, composition). This helps in designing the static architecture of the system.
- **Behavior Diagrams:** e.g. *Sequence Diagrams* illustrate interactions over time between objects (method call flows), and *State Diagrams* show state transitions of an object. *Use Case Diagrams* (covered next week) depict the functional requirements by showing actors and their interactions with the system.
- By using UML, we can communicate designs clearly. For instance, a class diagram can clarify how classes relate (a **composition** is depicted with a filled diamond, meaning a strong "has-a" relationship). UML is a common language for developers to collaborate

Software Requirements Fundamentals

- **Requirements** capture what the software should *do* (functional requirements) and how it should *be* (non-functional requirements). A functional requirement might be "the system shall **send a password reset email** to the user upon request," whereas a non-functional requirement could be "the email is sent **within 5 seconds** of request (performance)".
- *Functional vs Non-Functional:* Broadly, functional requirements define **what** a system must accomplish (features and behaviors), and non-functional requirements define **how** the system performs or the constraints on it (quality attributes like reliability, usability, security). For example, functional requirements are expressed as specific actions or outputs ("system shall do X"), while non-functional are expressed as properties or criteria ("system shall be Y" in terms of speed, security, etc.).
- Gathering clear requirements is critical to project success. Techniques include stakeholder interviews, user stories, and use cases. A well-written requirement is