**1. SDLC (Software Development Life Cycle) Models**

SDLC defines the process of planning, creating, testing, and deploying information systems or software applications. Different models exist to suit different project needs.

**🔹 Waterfall Model**

* **Definition**: A linear sequential model where each phase must be completed before moving to the next.
* **Phases**: Requirements → Design → Implementation → Testing → Deployment → Maintenance
* **Advantages**:
  + Simple and easy to understand.
  + Well-suited for projects with clear requirements.
* **Disadvantages**:
  + Difficult to accommodate changes once the process starts.
  + High risk if requirements are misunderstood.
* **Best Use Case**: Small projects with fixed, well-defined requirements (e.g., payroll systems).

**🔹 Agile Model**

* **Definition**: An iterative and incremental approach where requirements and solutions evolve through collaboration between cross-functional teams.
* **Principles**:
  + Customer collaboration over contract negotiation.
  + Responding to change over following a plan.
  + Working software over comprehensive documentation.
* **Advantages**:
  + Flexible, adaptive to changing requirements.
  + Frequent delivery of working software.
* **Disadvantages**:
  + Less predictable (budget, time).
  + Requires strong team collaboration.
* **Best Use Case**: Complex projects with dynamic requirements (e.g., web applications, SaaS).

**2. Phases of SDLC**

Regardless of model, most SDLC processes include these phases:

1. **Planning**
   * Define project scope, objectives, resources, and feasibility.
   * Deliverables: Project plan, cost estimation, timeline.
2. **Analysis (Requirements Gathering)**
   * Understand business needs and document functional & non-functional requirements.
   * Deliverables: SRS (Software Requirement Specification).
3. **Design**
   * Transform requirements into architecture and detailed design.
   * Deliverables: HLD (High-Level Design), LLD (Low-Level Design), ER diagrams, system architecture.
4. **Implementation (Coding)**
   * Developers write code based on the design.
   * Deliverables: Source code, build, documentation.
5. **Testing**
   * Verify and validate that software meets requirements.
   * Levels: Unit Testing, Integration Testing, System Testing, Acceptance Testing.
   * Deliverables: Test cases, test reports, bug reports.
6. **Deployment**
   * Release software to production environment.
   * Deliverables: Deployment plan, release notes.
7. **Maintenance**
   * Bug fixing, performance enhancement, updates, and upgrades.
   * Deliverables: Patches, new releases.

**3. Agile Frameworks**

Agile is not a single methodology but a mindset supported by frameworks.

**🔹 Scrum**

* **Definition**: A framework for managing work with an emphasis on iterative progress.
* **Key Roles**:
  + **Product Owner**: Defines product backlog, prioritizes features.
  + **Scrum Master**: Ensures Scrum principles are followed.
  + **Development Team**: Builds the product.
* **Ceremonies**:
  + Sprint Planning
  + Daily Standup (15-min updates)
  + Sprint Review
  + Sprint Retrospective
* **Artifacts**:
  + Product Backlog
  + Sprint Backlog
  + Increment (working product at end of sprint)

**🔹 Kanban**

* **Definition**: A visual workflow management system that focuses on continuous delivery.
* **Key Features**:
  + Tasks visualized on a **Kanban board** (To-Do → In Progress → Done).
  + WIP (Work-in-Progress) limits to prevent bottlenecks.
* **Advantages**:
  + Flexible, no fixed sprints.
  + Continuous delivery.
* **Best Use Case**: Teams with continuous flow of tasks (e.g., support, operations).

**4. Continuous Integration and Continuous Deployment (CI/CD)**

* **Continuous Integration (CI)**:
  + Developers frequently integrate code into a shared repository.
  + Automated build and test ensure code is always working.
  + Tools: Jenkins, GitHub Actions, GitLab CI/CD, CircleCI.
* **Continuous Deployment (CD)**:
  + Automatically deploys every code change to production once it passes testing.
  + Ensures faster delivery to users.
* **Benefits**:
  + Reduces integration issues.
  + Faster feedback loop.
  + Higher software quality and reliability.

**5. Version Control Systems (VCS)**

* **Definition**: Tools that track changes to source code over time, enabling collaboration and rollback if needed.
* **Types**:
  + **Centralized VCS**: Single server stores all versions (e.g., Subversion - SVN).
  + **Distributed VCS**: Each developer has a local copy (e.g., Git, Mercurial).
* **Git Basics**:
  + git init → Initialize repo
  + git clone → Clone repo
  + git add → Stage changes
  + git commit → Save changes
  + git push → Upload to remote repo
  + git pull → Fetch and merge changes from remote
* **Platforms**:
  + GitHub, GitLab, Bitbucket

**Summary Flow**:

* **SDLC Models** → Waterfall (sequential) vs Agile (iterative).
* **SDLC Phases** → Planning → Analysis → Design → Implementation → Testing → Deployment → Maintenance.
* **Agile Frameworks** → Scrum (sprints, roles, ceremonies) & Kanban (visual boards, continuous flow).
* **CI/CD** → Automates build, test, and deployment for faster delivery.
* **Version Control (Git)** → Manages code history and collaboration.

**Waterfall vs Agile**

| **Aspect** | **Waterfall Model** | **Agile Model** |
| --- | --- | --- |
| **Approach** | Sequential (step-by-step, one phase must finish before next begins) | Iterative & incremental (small working pieces delivered continuously) |
| **Flexibility** | Rigid – changes are hard to accommodate once development starts | Flexible – adapts easily to changing requirements |
| **Requirements** | Must be fully defined at the start | Can evolve during the project |
| **Delivery** | Final product delivered at the end | Working product delivered in short cycles (sprints/iterations) |
| **Customer Involvement** | Low – customer involved mainly at the beginning & end | High – customer is involved throughout the project |
| **Testing** | Happens after development is complete | Happens in every iteration (continuous testing) |
| **Risk Management** | Higher risk (issues found late in the cycle) | Lower risk (early feedback reduces risks) |
| **Documentation** | Heavy documentation-driven | Lightweight, focuses on working software |
| **Best Suited For** | Small projects with fixed, clear requirements (e.g., government projects, payroll systems) | Complex, evolving projects with uncertain requirements (e.g., SaaS, mobile apps, startups) |
| **Project Size** | Better for small, well-defined projects | Better for medium to large, dynamic projects |
| **Cost & Time** | More costly if requirements change mid-way | More cost-effective in dynamic environments |
| **Example** | Building a **bridge** (fixed design, no changes once started) | Building a **mobile app** (frequent updates, feedback-driven) |

**Waterfall vs Agile Flow**

**Waterfall:**

Requirements → Design → Implementation → Testing → Deployment → Maintenance

**Agile:**

Iteration 1: Plan → Design → Build → Test → Deploy → Feedback

Iteration 2: Plan → Design → Build → Test → Deploy → Feedback

... (continuous until final product)

**Scrum vs Kanban**

| **Aspect** | **Scrum** | **Kanban** |
| --- | --- | --- |
| **Definition** | A structured Agile framework with fixed-length iterations (sprints). | A visual workflow system focusing on continuous delivery. |
| **Work Cycle** | Time-boxed sprints (usually 2–4 weeks). | Continuous flow (no fixed time-box). |
| **Roles** | Defined roles: **Scrum Master**, **Product Owner**, **Development Team**. | No mandatory roles (teams manage work themselves). |
| **Meetings** | Formal ceremonies: Sprint Planning, Daily Stand-up, Sprint Review, Retrospective. | No required ceremonies; teams may hold standups if needed. |
| **Planning** | Work planned at the start of each sprint (fixed scope until sprint ends). | Work pulled continuously from backlog when capacity is available. |
| **Board** | Sprint board resets every sprint (tasks must be completed within sprint). | Kanban board is continuous (tasks flow across columns until done). |
| **WIP Limits** (Work in Progress) | Not enforced directly (focus on sprint scope instead). | Strict WIP limits per column to avoid bottlenecks. |
| **Flexibility** | Less flexible during sprint (scope locked once sprint starts). | Highly flexible (new tasks can be added anytime). |
| **Delivery** | Product increments delivered at the end of each sprint. | Continuous delivery (whenever a task is done, it can be released). |
| **Metrics** | Velocity (measures completed story points per sprint). | Lead time & cycle time (time taken for tasks to move across board). |
| **Best For** | Teams needing structure, predictability, and iterative improvements. | Teams needing flexibility, continuous flow, and minimal overhead. |
| **Example** | A software product team delivering features every 2 weeks. | A support/maintenance team handling tickets continuously. |

**🔹 Workflow Example**

**Scrum Workflow**

1. Product Backlog → Sprint Planning → Sprint Backlog
2. Sprint (2-4 weeks) → Daily Standups → Development & Testing
3. Sprint Review & Retrospective → Deliver Increment

[ Product Backlog ] → [ Sprint Backlog ] → [ Sprint (2–4 weeks) ] → [ Increment Delivered ]

**Kanban Workflow**

1. Tasks placed on a **Kanban Board** (To-Do → In Progress → Done).
2. Team pulls tasks when ready (respecting WIP limits).
3. Continuous flow → Deployment when ready.

[ To Do ] → [ In Progress (WIP limit: 3) ] → [ Testing ] → [ Done ]

**Summary**

* **Scrum** = Iterative, sprint-based, structured, good for product development.
* **Kanban** = Continuous, flow-based, flexible, good for support & operations.

**Object-Oriented Programming (OOP) in Python**

OOP is a programming paradigm that organizes code into **objects** that bundle **data (attributes)** and **behaviors (methods)** together. Python fully supports OOP.

**1. Core OOP Concepts in Python**

**🔹 Class**

* A blueprint for creating objects.
* Defines attributes (variables) and methods (functions).

class Car:

def \_\_init\_\_(self, brand, model):

self.brand = brand # attribute

self.model = model # attribute

def start(self): # method

print(f"{self.brand} {self.model} is starting...")

# Creating an object

car1 = Car("Toyota", "Corolla")

car1.start()

**🔹 Object**

* An instance of a class.
* Each object has its own copy of attributes but shares methods of the class.

car2 = Car("Honda", "Civic")

print(car2.brand) # Honda

**🔹 Encapsulation**

* Bundling data (attributes) and methods that operate on that data inside a class.
* Access modifiers:
  + Public: default (e.g., self.name)
  + Protected: single underscore \_name (convention only)
  + Private: double underscore \_\_name

class BankAccount:

def \_\_init\_\_(self, owner, balance):

self.owner = owner

self.\_\_balance = balance # private attribute

def deposit(self, amount):

self.\_\_balance += amount

def get\_balance(self):

return self.\_\_balance

acc = BankAccount("Alice", 1000)

acc.deposit(500)

print(acc.get\_balance()) # 1500

**🔹 Inheritance**

* Allows one class to inherit attributes and methods from another.

class Animal:

def speak(self):

print("This is an animal.")

class Dog(Animal): # Dog inherits from Animal

def speak(self):

print("Woof Woof!")

dog = Dog()

dog.speak() # Woof Woof!

**🔹 Polymorphism**

* The same method name behaves differently depending on the object.

class Cat(Animal):

def speak(self):

print("Meow!")

animals = [Dog(), Cat()]

for a in animals:

a.speak() # Woof Woof! / Meow!

**🔹 Abstraction**

* Hiding implementation details and showing only essential features.
* Achieved using **abstract base classes (ABC)** in Python.

from abc import ABC, abstractmethod

class Vehicle(ABC):

@abstractmethod

def move(self):

pass

class Bike(Vehicle):

def move(self):

print("Bike is moving on 2 wheels")

b = Bike()

b.move()

**2. Special Methods (Dunder Methods)**

Python classes can use built-in special methods.

class Person:

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

self.name = name

def \_\_str\_\_(self): # String representation

return f"Person(name={self.name})"

p = Person("John")

print(p) # Person(name=John)

Common dunder methods:

* \_\_init\_\_ → Constructor
* \_\_str\_\_ → String representation
* \_\_len\_\_ → Length support
* \_\_add\_\_ → Operator overloading

**3. OOP Features in Real Life Example**

class Shape:

def area(self):

pass

class Rectangle(Shape):

def \_\_init\_\_(self, length, width):

self.length = length

self.width = width

def area(self):

return self.length \* self.width

class Circle(Shape):

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

self.radius = radius

def area(self):

return 3.14 \* self.radius \* self.radius

shapes = [Rectangle(4, 5), Circle(3)]

for s in shapes:

print(s.area()) # 20, 28.26

**4. Benefits of OOP in Python**

* **Code reusability** (inheritance).
* **Encapsulation** (secure data handling).
* **Modularity** (class-based design).
* **Polymorphism** (flexible methods).
* **Abstraction** (focus on essential details).

**Quick Summary**

* **Class & Object** → Blueprint vs Instance.
* **Encapsulation** → Data hiding with access control.
* **Inheritance** → Reuse existing classes.
* **Polymorphism** → Same method behaves differently.
* **Abstraction** → Hides unnecessary details, shows essential.