**Introduction to Application Development Lifecycle**

* **Video Overview**: Aims to explain the steps and phases of the application development lifecycle and the importance of writing code in multiple files.

**Application Development Lifecycle Overview**

* **Steps in Lifecycle**: Requirement gathering, analysis, design, code and test, user and system test, production, and maintenance.

**Requirement Gathering**

* **Capture Requirements**: User, business, and technical requirements.
  + **Example**: For a hotel reservation app:
    - User requirement: Viewing rooms and amenities.
    - Business requirement: Determining room charges.
    - Technical requirement: Compatibility across browsers and devices.
* **Identify Constraints**: Business model viability, such as server updates and associated costs.

**Analysis and Design**

* **Analyze Requirements**: To create a solution design.
* **Documentation**: Maintain records of all updates.

**Code and Test Phase**

* **Code and Test**: Based on design documentation.
* **Unit Testing**: Ensure code meets specifications.
* **Application Version**: Generate an acceptable version post-unit testing.

**User and System Testing**

* **User Tests**: Verify functionality from the user perspective.
* **System Tests**: Include integration and performance testing.
* **Ensure Functionality**: Within a larger framework and meets performance standards.

**Production Phase**

* **Availability**: Application made available to end users.
* **Ensure Functionality**: Accurate functionality and availability.
* **Steady State**: Maintain steady state but allow controlled and tested changes if errors occur.

**Maintenance Phase**

* **Upgrades and New Features**: Address upgrades or new features.
* **Integration**: New features go through all phases before integration into production.

**Importance of Multiple Files for Coding**

* **Best Practice**: Code different functionalities in separate files.
* **Central Program**: Use a central program to run the application and call individual files.
* **Benefits**: Aids in efficient code maintenance and integration of new functionalities.

**Introduction to Web Applications and APIs**

* **Video Overview**: Aims to explain web applications (web apps), Application Programming Interfaces (APIs), and compare and contrast them.

**Web Applications (Web Apps)**

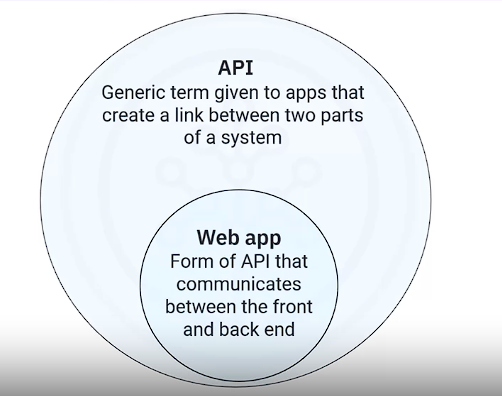
* **Definition**: Programs stored on a remote server, delivered over the internet, and interacted with via a browser.
* **Examples**: E-commerce sites, webmail, etc.
* **Compatibility**: Most web apps work on all modern web browsers.
* **Components**:
  + Web server: Manages client requests.
  + App server: Executes tasks.
  + Database: Stores information.
* **Front-End Programming**: Uses JavaScript, HTML, CSS.
* **Back-End Programming**: Uses Python, Java, Ruby.
* **Advantages**:
  + Same version available to multiple users simultaneously.
  + Flexibility in platform choice (desktop, laptop, mobile).
  + Accessible through any browser.
  + No need for local installation.

**Application Programming Interfaces (APIs)**

* **Definition**: Software component enabling communication between two unconnected apps.
* **Functionality**:
  + Standardized rules and functions for data requests and modifications.
  + Examples: Phone apps requiring permission for location, camera, etc.
  + Example: Weather app using a weather API to provide forecasts.
* **Architectures**:
  + REST (Representational State Transfer)
  + SOAP (Simple Object Access Protocol)
* **Benefits**:
  + Improves connectivity between apps.
  + Supports CRUD (Create, Read, Update, Delete) actions.
  + Works with HTTP verbs: PUT, POST, DELETE, GET.
  + Customizable as it is based on HTTP.

**Comparison: Web Apps vs. APIs**

* **Web Apps as APIs**:
  + Web apps facilitate communication between front-end and back-end.
  + Example: E-commerce shopping service accessed via browser or mobile device acts as an API.



* **Distinction**:
  + All web apps are APIs.
  + Not all APIs are web apps.
  + Web apps are "online" APIs requiring a network.
  + APIs can be both "online" and "offline".

**Summary**

* **Web Apps**: Programs delivered over the internet through browsers, running on various platforms without installation.
* **APIs**: Software components enabling communication between apps.
* **Relationship**: All web apps are APIs, but not all APIs are web apps. Web apps require networks, while APIs can function offline.

**ython Style Guide and Coding Practices**

**Video Overview**: Explains the importance of writing readable code, Python coding conventions, and static code analysis.

**Importance of Writing Readable Code**

* **Team Readability**: Ensures team members can easily read and understand the code.
* **Coding Standards and Conventions**: Essential for maintaining code readability and consistency.

**Python Coding Conventions (PEP8)**

1. **Indentation**
   * **Use Spaces Instead of Tabs**: Avoids non-uniformity due to different text editors interpreting tabs differently.
   * **Guideline**: Use four spaces for each indentation level for consistent readability.
2. **Blank Lines**
   * **Function and Class Separation**: Use blank lines to separate functions and classes for clear code structure.
3. **Spaces Around Operators and After Commas**
   * **Improves Readability**: Spaces around operators and after commas make commands more readable.
   * **Example**: Write **A = b + c** instead of **A=b+c**.

**Coding Conventions for Consistency and Manageability**

1. **Functions for Larger Code Blocks**
   * **Reuse and Maintainability**: Create separate functions for functionalities to enhance execution speed and code reuse.
2. **Naming Conventions**
   * **Functions and Files**: Use lowercase with underscores (e.g., **comp\_surface\_radiation**).
   * **Classes**: Use CamelCase (e.g., **LamSquirrelCage**).
   * **Constants**: Use all capital letters with underscores (e.g., **MAX\_FILE\_UPLOAD\_SIZE**).

**Static Code Analysis**

* **Definition**: Evaluates code against predefined styles and standards without executing it.
* **Purpose**: Identifies programming errors, coding standard violations, undefined values, syntax violations, and security vulnerabilities.
* **Tool**: Use the PyLint library to check code compliance with PEP8 guidelines.

**Summary**

* **Consistent Code**: Helps team members read and understand code easily.
* **PEP8 Guidelines**:
  + Use four spaces for indentation.
  + Use blank lines to separate functions and classes.
  + Use spaces around operators and after commas.
* **Coding Conventions**:
  + Create functions for larger blocks of code.
  + Name functions and files in lowercase with underscores.
  + Use CamelCase for class names.
  + Name constants in all capital letters with underscores.
* **Static Code Analysis**: Evaluates code compliance with style guidelines using tools like PyLint.

**What is Unit Testing?**

* **Definition**: Unit testing validates if units of code operate as designed. A unit is a small, testable part of an application.
* **Example Units**:
  + Function **square** in **mymodule.py**: **def square(number): return number \*\* 2**
  + Function **doubler** in **mymodule.py**: **def doubler(number): return number \* 2**

**Unit Testing Process**

1. **Local Testing**:
   * Test each unit on your local system.
   * If a test fails, identify and fix the issue, then retest.
2. **Server Testing**:
   * Test the unit in a server environment (e.g., CI/CD test server).
   * If it fails, identify and fix the issue, then retest.
   * Once it passes, integrate the unit into the final codebase.

**Building Unit Tests**

1. **Naming Conventions**:
   * Name unit test files by appending or prepending "test" to the unit file name (e.g., **test\_mymodule.py**).
2. **Creating Unit Test File**:
   * **Import unittest**: **import unittest**
   * **Import Functions**: **from mymodule import square, doubler**
   * **Build Testing Class**:

python

Copy code

class TestMyModule(unittest.TestCase):

* + **Inherit TestCase**: Allows use of existing methods in the **TestCase** class.

1. **Creating Test Functions**:
   * Define functions in the test class for each function to be tested, prepending "test" (e.g., **def test\_square(self)** and **def test\_doubler(self)**).
2. **Writing Test Cases**:
   * Use assertion methods like **assertEqual()** to ensure test conditions are met.
     + **Example**:

python

Copy code

def test\_square(self): self.assertEqual(square(2), 4)

* + - **assertEqual()** compares two values and determines if they are equal.

**Reviewing Unit Test Output**

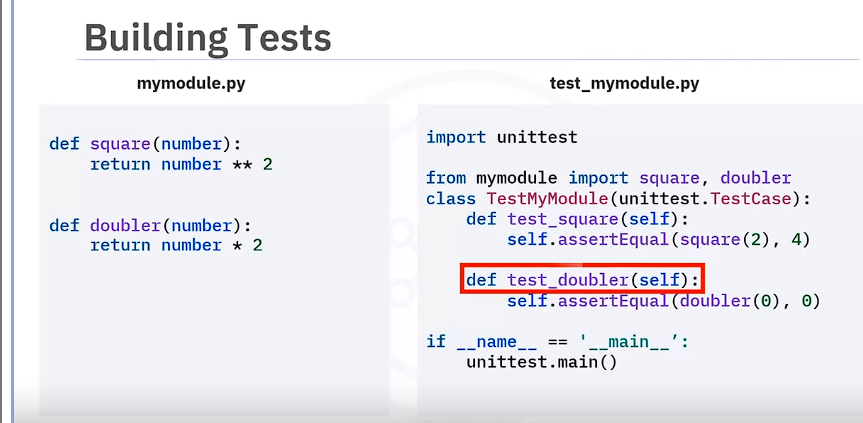
* **Pass Scenario**:
  + Output indicates the number of tests run and time taken.
  + **OK** means all tests passed.
* **Fail Scenario**:
  + Output shows which tests failed and provides details of the failure.
  + Example:

plaintext

Copy code

FAIL: test\_square (\_\_main\_\_.TestMyModule) self.assertEqual(square(2), 4) AssertionError: 8 != 4

* + Detailed output helps identify and correct mistakes before deployment.

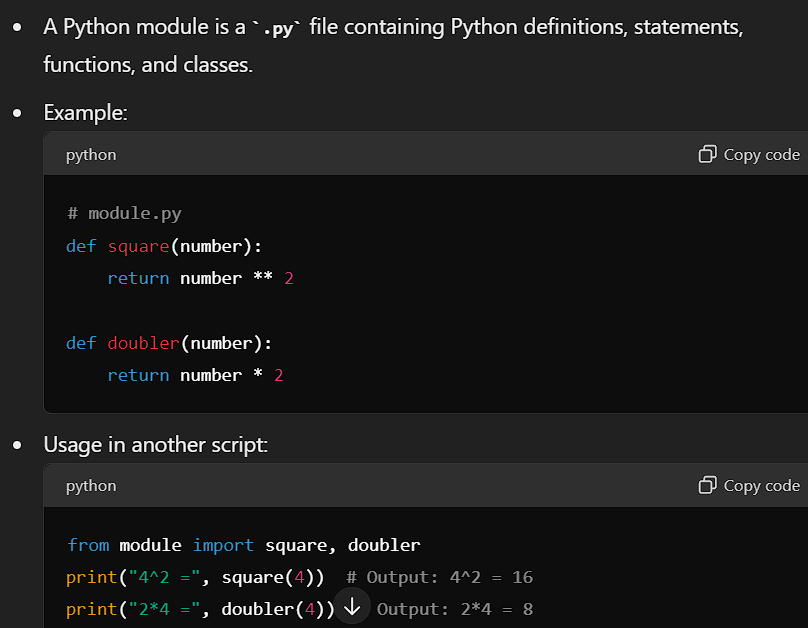


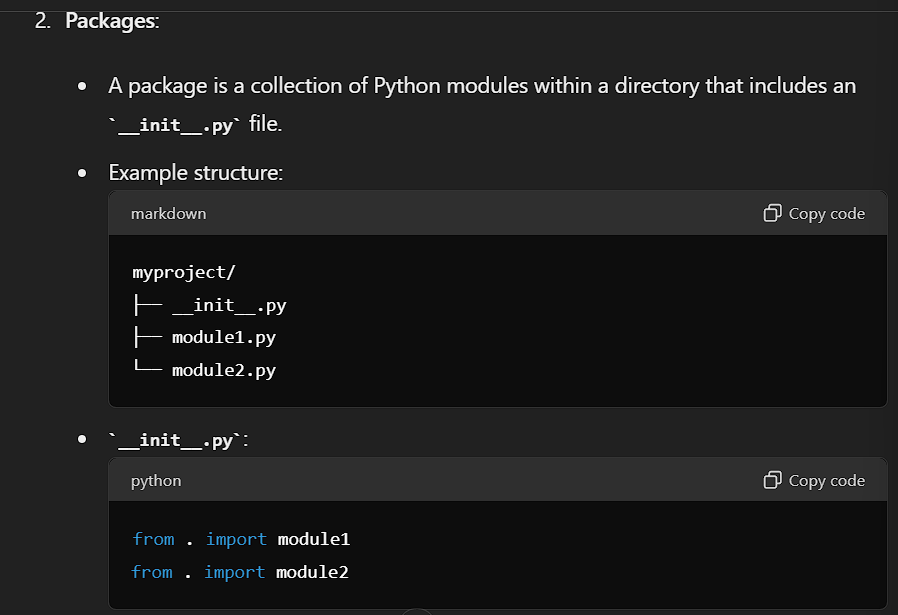
**Summary**

* **Unit Testing**: Validates code units operate as intended.
* **Testing Process**: Includes local and server testing phases.
* **Naming Conventions**: Append/prepend "test" to unit test file names.
* **Building Tests**: Import **unittest**, functions, create test class, define test functions, and write test cases with assertions.
* **Reviewing Output**: Check for pass/fail results and detailed error messages for corrections.

By following these guidelines, you can effectively create, execute, and review unit tests to ensure your code functions correctly before deployment.

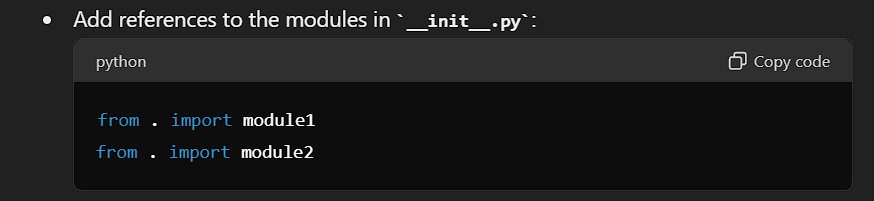
**Key Concepts**

****



1. **Libraries**:
   * A library is a collection of packages or a single package.
   * Examples: NumPy, PyTorch, Pandas.
   * Note: The terms "package" and "library" are often used interchangeably.

**Steps to Create a Python Package**

1. **Create a Folder**:
   * Create a folder named after your package (e.g., **myproject**).
2. **Create \_\_init\_\_.py File**:
   * Create an empty **\_\_init\_\_.py** file in the folder to mark it as a package.
   * Add references to the modules in **\_\_init\_\_.py**:
3. **Create Modules**:
   * Create required module files (**module1.py**, **module2.py**).
4. **Verify the Package**:
   * Open a bash terminal and navigate to the package directory.
   * Run the Python interpreter:

bash

Copy code

python

* + Import the package:

python

Copy code

import myproject

* + Test functions from the package:

python

Copy code

print(myproject.module1.square(4)) # Output: 16 print(myproject.module2.mean([2, 1, 3])) # Output: 2.0 (Assuming mean function is implemented)

**Using the Package in Other Scripts**

* If the package folder is in the same directory as your script, you can import and use the functions.
* Example script (**test.py**):

python

Copy code

from myproject.module1 import square, doubler from myproject.module2 import mean print("4^2 =", square(4)) # Output: 4^2 = 16 print("2\*4 =", doubler(4)) # Output: 2\*4 = 8 print("(2+1+3)/3 =", mean([2, 1, 3])) # Output: (2+1+3)/3 = 2.0

**Summary**

* **Module**: A **.py** file with definitions, functions, and classes.
* **Package**: A collection of modules within a directory with an **\_\_init\_\_.py** file.
* **Library**: A collection of packages or a single package.
* **Creating a Package**:
  1. Create a folder with the package name.
  2. Create an **\_\_init\_\_.py** file.
  3. Create the required modules.
  4. Reference the modules in **\_\_init\_\_.py**.
* **Verifying the Package**: Use a bash terminal and Python interpreter to check if the package loads correctly.
* **Using the Package**: Import and use the package in other scripts if the package folder is in the same directory.

**"Python Libraries and Frameworks for Application Development"**

**Introduction**

In this video, you will learn to:

1. Describe the Python libraries for app development
2. Define a framework for app development
3. Identify the benefits of using a framework for app development
4. Compare frameworks and libraries

**Python Libraries for Application Development**

**What are Python Libraries?**

Python libraries are like toolkits. Each library contains specific tools to simplify and expedite certain programming tasks. Here are some key libraries used in various domains of app development:

* **NumPy**: Facilitates advanced mathematical calculations.
* **Pandas**: Offers data manipulation and analysis capabilities.
* **Matplotlib**: Simplifies data visualization.
* **Requests**: Simplifies the process of sending HTTP requests, essential for web development.
* **BeautifulSoup**: Makes web scraping easier by iterating, searching, and modifying the parse tree.
* **SQLAlchemy**: A SQL toolkit and Object-Relational Mapping (ORM) tool that gives developers the full power and flexibility of SQL.
* **PyTest**: A testing framework for creating simple to complex functional tests for applications and libraries.

**Frameworks for Application Development**

**What is a Framework?**

Frameworks are predefined structures that provide a set of guidelines for application development. They help in organizing code, adhering to good coding practices, and reusing code libraries. Some popular Python frameworks for web development include:

* **Django**
* **Flask**
* **Web2Py**

**Benefits of Using a Framework for App Development**

1. **Eases the Development Process**: Frameworks come with pre-written code libraries, modules, and developer guidelines that streamline the development process.
2. **Simplifies Debugging**: Pre-built debugging tools in web frameworks make it easier to debug web applications.
3. **Enables More Functionality with Less Code**: Frameworks come with pre-built libraries and modules, reducing the need to create functionalities from scratch.
4. **Improves Database Management Efficiency**: Built-in database integration tools help seamlessly integrate database endpoints for data transfer.
5. **Enhances Security**: Built-in security features and guidelines in frameworks help enforce security, which is crucial for application users.

**Comparing Frameworks and Libraries**

* **Frameworks**: Provide the basic flow and architecture of an application, enabling you to build complete applications.
* **Libraries**: Are sets of packages designed to perform specific functions, aiding in various aspects of application development without providing a complete application structure.

**Summary**

In this video, you learned:

* Python libraries are toolkits with specific tools to simplify and expedite programming tasks.
* Frameworks are predefined structures that provide guidelines and support for developing complete applications.
* Using frameworks eases development, simplifies debugging, enables more functionality with less code, improves database management, and enhances security.
* Frameworks provide the structure for building complete applications, whereas libraries aid in specific functionalities.

By understanding the differences and benefits of Python libraries and frameworks, you can choose the appropriate tools to streamline your application development process and build robust, efficient applications.