**Developing AI Applications with Python and Flask**

# Python Coding Practices and Packaging Concepts

## Welcome

### Introduction to Course

Welcome to the course on the development of AI-based applications using Python and Flask. Python stands as one of the foremost programming languages globally, as evidenced by its inclusion in the top ten in a survey conducted by Stack Overflow (https://stackoverflow.blog/2023/06/13/developer-survey-results-are-in/). Flask, a Python micro-framework, demonstrates minimal reliance on external libraries, making it an ideal choice for crafting succinct (concise) applications, potentially incorporating AI elements, and seamlessly scaling solutions. Upon completing this course, participants will possess the following competencies:

* Articulating the steps and procedures involved in application creation
* Generating Python modules
* Conducting unit tests
* Packaging applications
* Deploying applications on the web using Flask
* Developing and deploying AI-based applications on a web server employing IBM Watson Embedded AI Libraries in conjunction with Flask.

This course caters to individuals with basic programming knowledge, a fundamental grasp of Python, and an interest in constructing reusable web applications integrating AI.

Module 1 serves as an introduction to application development fundamentals, encompassing lifecycle comprehension and coding best practices. Additionally, participants will engage in module activities such as module creation, unit test execution, and application packaging. Furthermore, they will acquire insights into optimal Python coding practices and static code analysis execution.

Module 2 introduces Flask, outlining its functionalities as a Python micro-framework for web development. Participants will delve into deployment concepts, encompassing route definition, request and response object handling, error mitigation, and decorator application. Practical implementation of Flask will include the creation and deployment of a web application.

In Module 3, participants will amalgamate knowledge acquired from preceding modules to craft functional web applications. This module will also delve into AI integration within applications using IBM Watson Embedded AI Libraries. Practical application of acquired skills will be demonstrated through practice and graded projects, allowing participants to showcase their proficiency in Flask development and AI integration.

Throughout the course, participants will engage in hands-on labs, quizzes, and projects to reinforce their learning. Active participation in the discussion forum is encouraged for troubleshooting assistance or query resolution.

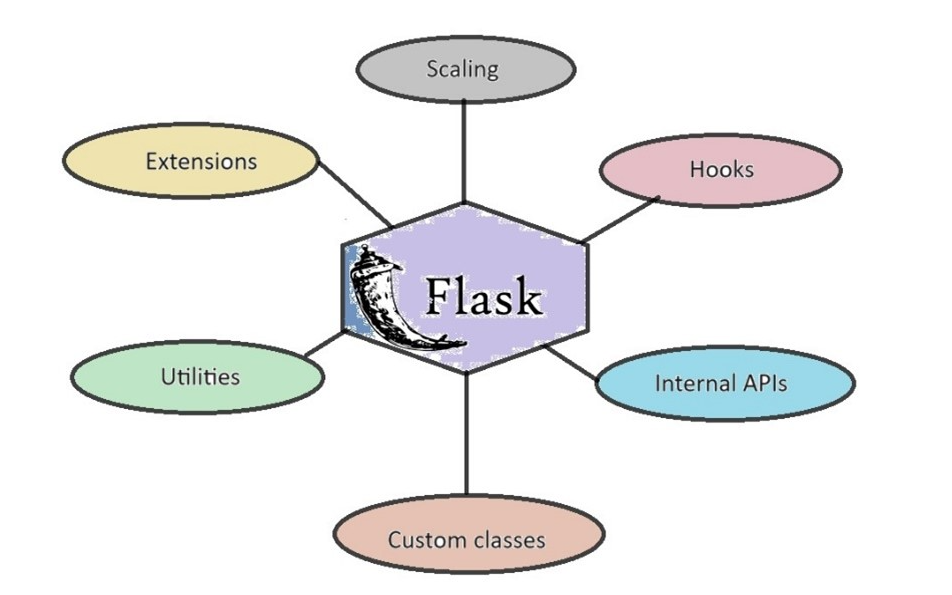
Upon course completion, participants will possess the aptitude to conceptualize and deploy web applications integrating AI components using Python and Flask. We extend our gratitude for your participation and anticipate your success in the course.

## Application Development and Packaging using Python

### Python with Flask for Large-Scale Projects

Python with Flask is widely recognized as a lightweight and versatile web application framework, esteemed for its simplicity, minimalism, and user-friendly nature. Tailored as a micro-framework, Flask empowers developers to swiftly construct web applications without compromising efficiency or scalability, facilitating seamless transitions from small-scale projects to more intricate endeavors (activities).

While Flask excels in smaller, straightforward applications, its scalability is not inherently restricted by its micro-framework nature. Rather, the framework's scalability hinges upon considerations of specific project requisites, prudent (careful) planning, robust architecture, and modular design. Large-scale Flask applications necessitate meticulous (thorough) management and scalability efforts compared to more comprehensive frameworks.



The framework's extensive ecosystem furnishes developers with a plethora (excess) of tools, libraries, and functionalities to streamline web development tasks such as routing, request handling, and template rendering. Leveraging techniques such as caching, load-balancing, replication, and scalable data storage can optimize performance for large-scale Flask applications.

Several techniques can be employed when developing large-scale Flask applications:

1. Extensibility: Flask's modular architecture allows for effortless addition or removal of features, fostering customization. Integration with various Python libraries and frameworks enhances Flask's capabilities, enabling developers to augment functionalities.
2. Documentation: Flask's comprehensive documentation facilitates access to internal APIs, utilities, hook points, overrides, and signals, empowering developers to tailor the framework to specific project requirements.
3. Customizations: Out-of-the-box customizations and custom classes can be utilized for aspects like request and response objects. Flask's class structure supports subclassing, enabling rapid addition or customization of behavior.
4. Scaling: Implement scaling strategies to ensure performance scalability. However, Flask's utilization of context local proxies introduces limitations, necessitating considerations regarding concurrency mechanisms.
5. Refactoring and Extensions: Refactor projects into utilities and Flask extensions to enhance modularity and maintainability. Explore community extensions and patterns for building custom extensions to cater to project needs.

Python with Flask has gathered popularity among renowned organizations due to its simplicity, flexibility, versatility, and ease of adoption. Notable companies such as Netflix, Reddit, Lyft, LinkedIn, Pinterest, and Uber leverage Flask for diverse backend services and functionalities. Flask's minimalistic design and extensibility render it a dependable choice for large-scale web development across various industries.

### Application Development Lifecycle

Consider a scenario where a client requests the development of an application to assist their employees in tracking daily tasks. Similarly, imagine another scenario where a client seeks a web application to manage hotel reservations, covering tasks from creation to customer check-out. However, can we immediately start building these applications upon receiving the request? Not quite. Certain preparatory activities such as requirement analysis, planning, and coding are imperative (necessary) before the applications can be deemed ready for user interaction.

Irrespective of the application's nature, it undergoes distinct phases, collectively known as the application development lifecycle. This lifecycle comprises seven phases:

1. **Requirement gathering:** This initial phase involves capturing user, business, and technical requirements comprehensively. For instance, in the hotel reservation app scenario, user requirements may include viewing available rooms and amenities, while technical requirements may entail cross-browser and mobile device compatibility.
2. **Analysis:** Here, each requirement and constraint is meticulously (accurately) analyzed to devise a viable solution for the application's design. Multiple rounds of verification and revision may be necessary to refine the proposed solution.
3. **Design:** In this phase, a detailed solution model is formulated based on the analysis conducted. Clear and concise documentation is maintained throughout this process to guide subsequent phases effectively.
4. **Code and test:** The development team utilizes the design documentation to code, test, revise, and retest the application until all documented requirements are met. Unit testing at the programming level ensures adherence to specifications.
5. **User and system test:** The application undergoes a series of user and system-level tests to verify functionality, integration, and performance. These tests ensure that the application functions as expected within the broader framework and under varying workloads.
6. **Production:** Upon successful testing, the application is deployed into production, allowing end users to access and utilize it. The application must remain stable during this phase, with changes tightly controlled and rigorously tested before implementation.
7. **Maintenance:** The final phase involves ongoing maintenance, including upgrades and feature additions. New features must undergo the entire development lifecycle before integration into the production version of the application.

Lastly, organizing application code into multiple files is considered a best practice. Each functionality is coded in a separate file, facilitating efficient code maintenance and ease of adding new features. A central program calls these individual files and functions, streamlining the execution of specific actions within the application.

### Introduction to Web Applications and API’s

#### Web Applications (Web Apps):

* Definition: Programs stored on remote servers and delivered over the internet through a web browser.
* Examples: E-commerce sites, webmail, etc.
* Components: Web server, app server, database.
* Programming languages: Front-end (JavaScript, HTML, CSS), Back-end (Python, Java, Ruby).

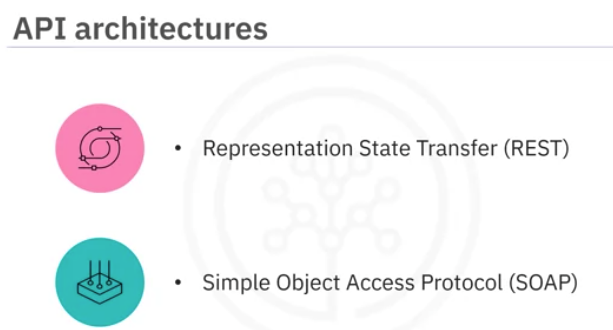
##### Advantages:

* Accessible from various platforms and browsers.
* No local installation required.
* Scalable updates for all users.

#### Application Programming Interfaces (APIs):

* Definition: Software components enabling communication between unconnected applications.
* Function: Provide standardized methods for data access and modification within an application.
* Example: Weather app retrieving data from a weather API.

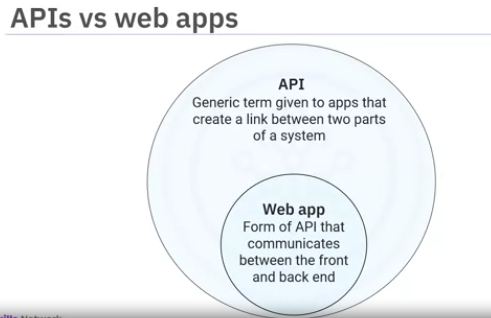
##### Advantages:

* Improved app connectivity.
* Supports CRUD (Create, Read, Update, Delete) operations.
* Utilizes HTTP verbs (PUT, POST, DELETE, GET).
* Customizable due to HTTP foundation.

Relationship to Web Apps:

Web apps are a specific type of API facilitating communication between front-end and back-end components.

#### Key Distinction:

* Web apps: Designed for user interaction through a browser.
* APIs: More general term for any software component facilitating communication between applications, including web-based and offline functionalities.

#### In essence:

* All web apps are APIs, but not all APIs are web apps.
* Both share data between applications, but web apps inherently require network connectivity.

### Demo: Working with a Cloud IDE

Cloud IDE is a web-based programming environment designed for learning and development. It allows users to write, run, debug, and execute code directly in a browser, eliminating the need for local software installation or configuration.

#### Key Features:

* Instructional Pane: Displays step-by-step instructions for completing projects.
* Programming Interface: Similar to VS Code, it offers an editor for writing code and a terminal for execution.
* AI-powered Chatbot (Tai): Provides assistance with using the lab environment for coding assignments.
* Skills Network Toolbox: Integrates various tools for database management, big data, cloud computing, and embedded AI.

#### Using Cloud IDE:

##### Navigate the Panes:

* Resize the instructional and code panes for better viewing.
* Use the "Table of Contents" button to navigate through instructional sections.

##### Interact with the Chatbot:

* Click the Tai icon to access the chatbot and ask questions related to coding tasks.

##### Utilize the Programming Interface:

* Write code in the Editor tab.
* Execute code in the Terminal tab.
* Install required libraries using the terminal.

##### Create and Execute Python Programs:

* Create a new file with a .py extension.
* Write or paste code into the file.
* Save the file.
* Navigate to the file location in the terminal.
* Run the program using python3 filename.py.

### Python Style Guide and Coding Practices

We will highlight the importance of writing clean and readable Python code, emphasizing the role of conventions and static code analysis in achieving this goal.

#### Key Points:

##### Readability:

###### PEP8 guidelines: (Python Enhancement Proposal 8)

* Use 4 spaces for indentation (avoid tabs due to inconsistency across editors).
* Separate functions and classes with blank lines.
* Use spaces around operators and after commas for better clarity.

##### Coding Conventions:

* Function creation: Break down large code blocks into well-defined functions for reusability and efficiency.

###### Naming:

* Functions and files: Lowercase with underscores (e.g., function\_name).
* Classes: CamelCase (e.g., ClassName).
* Packages: All lowercase without underscores or camelCase i.e., mypackage
* Constants: All uppercase with underscores (e.g., MAX\_VALUE).

##### Static Code Analysis:

* Purpose: Evaluate code against style guidelines and standards without execution.
* Benefits: Identify potential issues like syntax errors, coding standard violations, undefined values, and security vulnerabilities.
* Example tool: Pylint (checks PEP8 compliance)

### Unit Testing

#### What is unit testing?

* A method to verify if individual code units function as intended.
* A unit is a small, testable part of an application (e.g., a function).

#### Unit testing process:

During code developement, we will test each unit, the test is performed in two phases:

##### Local testing:

* In the first phase, we will test each unit on our local system, if the test fails, we will determine the reason for the failure and fix the issue, then we will test the unit again. If the test passes, we will test the unit in server environment

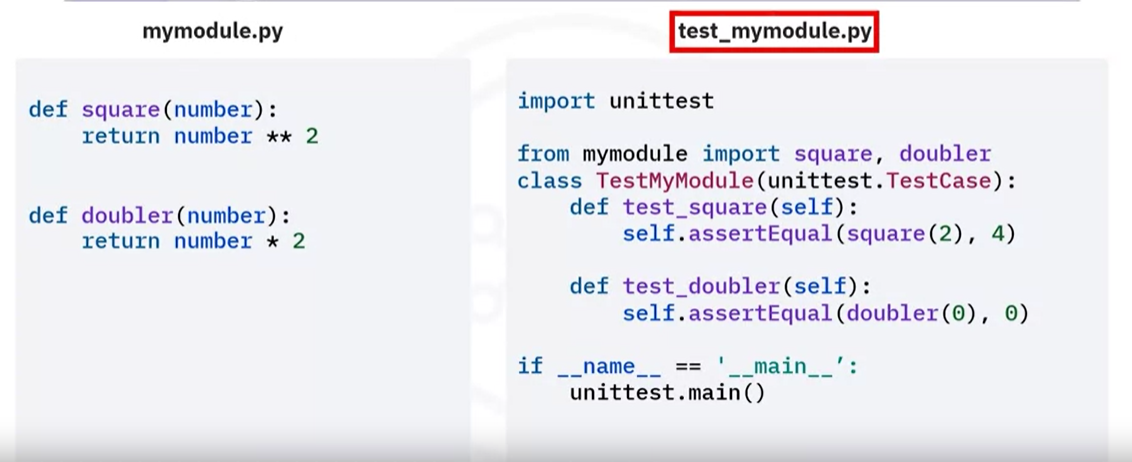
##### Server testing:

* Run tests in a server environment (e.g., CI/CD test server).
* Fix any failures identified by the server tests.

##### Integration:

* If the unit passes both the tests, it will be intergrated into final code base.

#### Building unit tests:





Inherit TesCast

##### Create a unit test class:

* Inherit from unittest.TestCase.
* Name the class with Test prefix (e.g., TestMyModule).

##### Create test functions:

* Each function corresponds to a unit to be tested.
* Prepend function names with test (e.g., test\_square).

##### Add assertions:

* Use assertEqual() to compare actual vs. expected values.
* Actual value: Call the function you want to test.
* Expected value: Specify the expected output of the function.

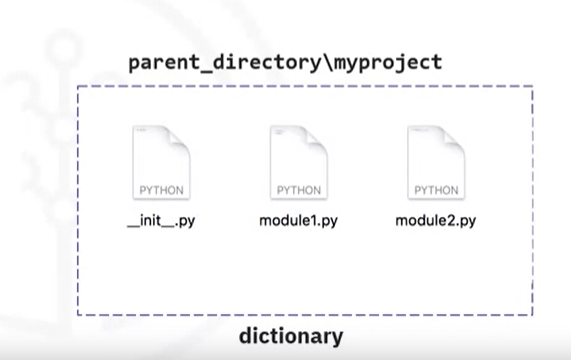
#### Test output:

* Displays the number of tests run and execution time.
* "OK" indicates successful tests.
* Failure details are provided for failed tests, including:
* Function where the test failed.
* Specific assertion that failed.
* Mismatched values.

### Packaging

#### Key points:

* Module: A .py file containing Python code (definitions, statements, functions, classes). You can import modules into other scripts and notebooks.
* Package: A collection of Python modules organized into a directory with an \_\_init\_\_.py file. This file can optionally contain initialization code for the package.



When we import a module or a package, the corresponding object created by python is always of type module. Which means that the distinction between module and package is at the filesystm level.

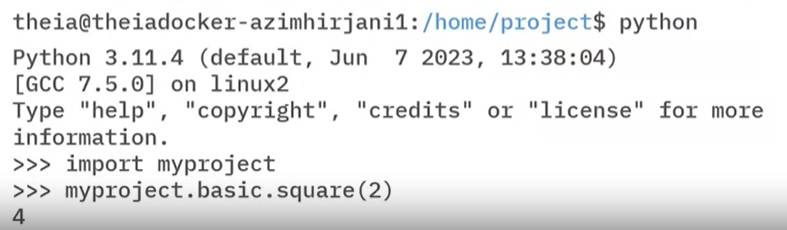
* Library: A collection of packages or a single package. Examples include NumPy, PyTorch, and Pandas. The terms "package" and "library" are often used interchangeably.

#### Creating a Python package:

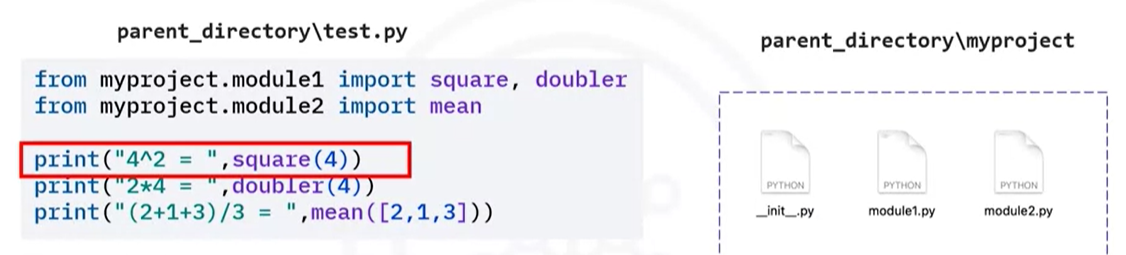
1. Create a folder with the package name.
2. Create an empty \_\_init\_\_.py file inside the folder.
3. Create the required modules (.py files) within the folder.
4. In the \_\_init\_\_.py file, add code to import the modules needed in the package using from . import module\_name.

#### Verifying and using the package:

1. Open a terminal and navigate to the directory containing the package.
2. Use the Python interpreter to import the package using import package\_name. If successful, no errors will occur.
3. Use the package functions using package\_name.module\_name.function\_name(arguments).



#### Using Package:



#### Additional notes:

* The distinction between modules and packages is mainly at the filesystem level.
* You can use a package in other scripts if the package folder is in the same directory as the script.

# Web App Deployment Using Flask

## Web Application Deployment Using Flask

### Python Libraries and Frameworks For Application Development



#### Introduction to Python Libraries

* Python libraries serve as toolkits, providing specialized functions to expedite (accelerate) programming tasks.
* Examples include NumPy for mathematical calculations, Pandas for data manipulation, and Matplotlib for data visualization.
* These libraries save significant time and effort for developers by offering pre-written functionalities.

##### Essential Libraries for Web Application Development

For web application development, certain Python libraries are indispensable.

* The *Requests* library simplifies HTTP requests handling, while *BeautifulSoup* facilitates web scraping tasks.
* *SQLAlchemy* serves as a SQL toolkit and Object-Relational Mapping (ORM) tool, empowering developers with SQL functionalities.

#### Introducing Frameworks for Application Development

* Frameworks provide predefined structures and guidelines for application development.
* They facilitate good coding practices by offering well-defined structures for code organization and reuse of code libraries.
* Examples include Django, Flask, and Web2Py, each catering to different needs and preferences.

##### Benefits of Using Frameworks

* Frameworks streamline the development process by providing pre-written code libraries and developer guidelines.
* They simplify debugging with built-in tools and enable developers to add more functionality with less code.
* Frameworks enhance database management efficiency through built-in integration tools and enforce security through predefined guidelines.

#### Contrasting Frameworks and Libraries

* Frameworks encompass the entire application architecture and flow, enabling developers to build complete applications.
* Libraries, on the other hand, focus on specific functionalities and assist with particular tasks within the application.

### Introduction To Flask

#### Understanding Flask:

Flask is a micro framework designed for creating web applications in Python. Unlike some larger frameworks, Flask is not opinionated (intolerant) and does not enforce a specific set of tools or conventions upon the developer.

#### Installation Process:

To install Flask and pin the version number of its dependencies while ensuring best practices for environment reproducibility, follow these steps:

##### Create a Virtual Environment:

* Before installing Flask, it's recommended to create a virtual environment using the venv module (or virtualenv if you prefer). This helps isolate your project's dependencies from other Python projects and system-wide packages.
* Run the following command in your terminal or command prompt to create a virtual environment named myenv:

python -m venv myenv

Activate the virtual environment:

* On Windows:

// In widows, running scripts is disabled by default,so first use the following script to enable it.

Set-ExecutionPolicy -Scope Process Bypass

//Now run the following

myenv\Scripts\activate

The above command will activate the virtual environment, Now you can install Flask by following the upcomming step

* On Unix or MacOS:

source myenv/bin/activate

##### Install Flask:

With the *virtual environment activated*, use:

pip install Flask

This command installs Flask and its dependencies into the virtual environment.

##### Pin Dependency Versions:

To ensure reproducibility and prevent unintended updates to dependencies, you can generate a *requirements.txt* file containing the specific versions of all installed packages, including Flask and its dependencies.

Run the following command to generate the *requirements.txt* file:

pip freeze > requirements.txt

This command creates a requirements.txt file listing all installed packages and their versions.

##### Using the requirements.txt File:

Whenever you need to recreate the same environment, such as in a different development environment, staging, or production, you can use the requirements.txt file to install the exact same versions of packages.

To install the dependencies listed in the requirements.txt file, run:

pip install -r requirements.txt

This command installs all the packages listed in the requirements.txt file, ensuring that you have the same environment as when the file was generated.

By following these steps, you can create a reproducible environment for your Flask application, install Flask along with its dependencies, and ensure consistency across different environments.

#### Main Features of Flask:

* Web Server and Debugger: Flask comes with a built-in web server for running applications in development mode. Additionally, it includes a debugger that provides interactive traceback and stack trace in the browser to aid in debugging.
* Logging: Flask utilizes standard Python logging for application logs, enabling developers to log custom messages about their applications.
* Testing Support: Flask provides a testing feature that enables developers to follow a test-driven approach. Frameworks like PyTest and **coverage** can be used for testing Flask applications.
* Request and Response Handling: Developers can access request and response objects in Flask to retrieve arguments and customize responses.

##### Additional Features:

* Static Assets and Templating: Flask supports static assets like CSS, JavaScript, and images, and utilizes the **Jinja** templating framework for dynamic page rendering.
* Routing and URL Handling: Flask supports routing and dynamic URLs, making it suitable for RESTful services. Developers can define routes for different HTTP methods and handle redirection within their applications.
* Error Handling and Session Management: Flask allows developers to write global error handlers at the application level and provides support for user session management.

#### Community Extensions:

Flask's extensibility is augmented by numerous community extensions, including:

* Flask-SQLAlchemy: Adds support for SQLAlchemy ORM for working with database objects.
* Flask-Mail: Facilitates setting up an SMTP mail server.
* Flask-Admin: Simplifies adding admin interfaces to Flask applications.
* Flask-Uploads: Enables customized file uploading capabilities.
* Flask-Cors: Allows application to handle Cross Origin Resource Sharing, making cross origin Javascript requests possible.
* Flask-Migrate: Adds database migration to SQLAchemy ORM.
* Flask-User: Adds User Authentication, authorization and other user management activities.
* Marshmallow: Adds extensive object serialization and deserialization.
* Celery: A powerful task queue that can be used for simple background tasks and complex multi-storage programs and schedules.

#### Dependency Management:

Flask relies on several built-in dependencies to enable its features, including **Werkzeug** for WSGI (Web Server Gateway Interface) implementation, Jinja for template rendering, **MarkupSafe** (comes with Jinja) for secure rendering and avoiding injection attacks, **ItsDangerous** for secure data assignment and it is a signing library for session cookies, and **Click** for command-line application support.

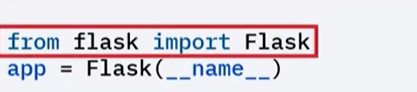
#### Comparison with Django:

* Flask is a lightweight micro framework, whereas Django is a full-stack framework providing all the necessary components for building web applications out-of-the-box.
* Flask offers flexibility, allowing developers to add or remove components as needed. In contrast, Django is opinionated and makes decisions for the developer to streamline application development.

### Flask - Basic Applications and Routes

#### Creating a Flask Application:

* Start by ensuring Flask is installed.
* Create a Python file named app.py.



Flask(\_\_name\_\_) instantiates a Flask object in app.The constructor takes a single argument of class scaffold. We set the name in the scaffold by passing the name of the app module in the built-in \_\_name\_\_ variable. This name is used to find the resources on the file system and by extentions to provide debugging information

* Import the Flask class from the flask module.
* Instantiate an object of the Flask class to create your app.

In the Flask code snippet app = Flask(\_\_name\_\_), \_\_name\_\_ is a special Python variable that represents the name of the current module.

When a Python script is executed as the main program (i.e., directly run by the Python interpreter), the value of \_\_name\_\_ is set to '\_\_main\_\_'. This indicates that the script is the main module being executed.

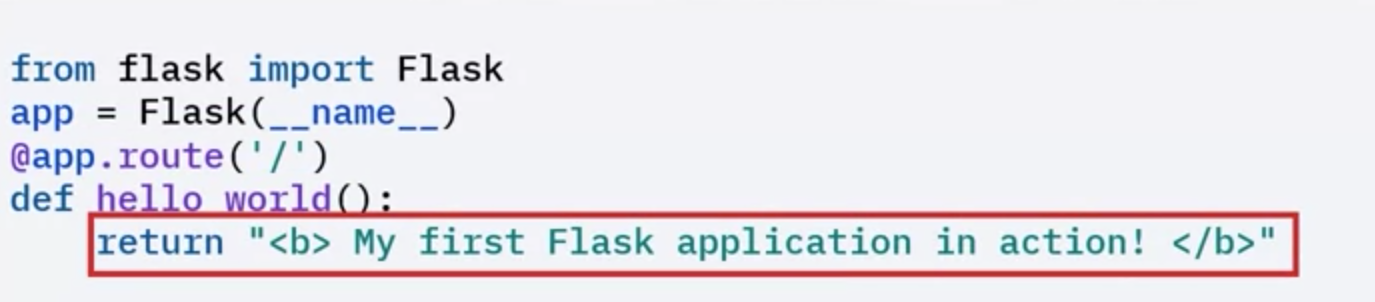
However, when a Python script is imported as a module into another script, the value of \_\_name\_\_ is set to the name of the module itself. In this case, it will be set to the name of the Python file without the .py extension.

In the context of a Flask application, passing \_\_name\_\_ to the Flask constructor as an argument helps Flask determine the root path of the application. This is crucial for Flask to locate static files, templates, and other resources associated with the application.

By passing \_\_name\_\_, Flask can set up the proper paths and configurations based on the current module's name, ensuring that the application works correctly regardless of how it is executed or imported.

#### Defining Routes:

* Use the @app.route() decorator to define routes.
* Routes specify the URL path where the client can access specific functionality.
* Return text or HTML from the route methods.



#### Running the Application:

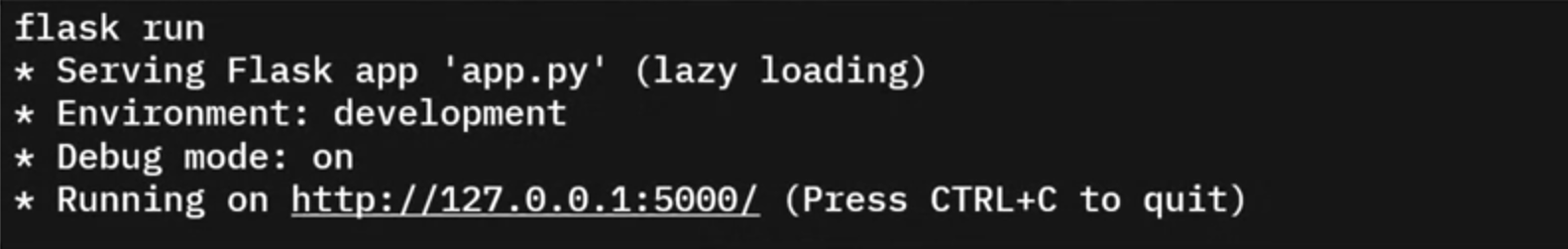
##### Set system environment variables:

* FLASK\_APP: Name of the main server file (e.g., app.py).
* FLASK\_ENV: Define the environment (development or production). This will be deprecated after Flask 3.3

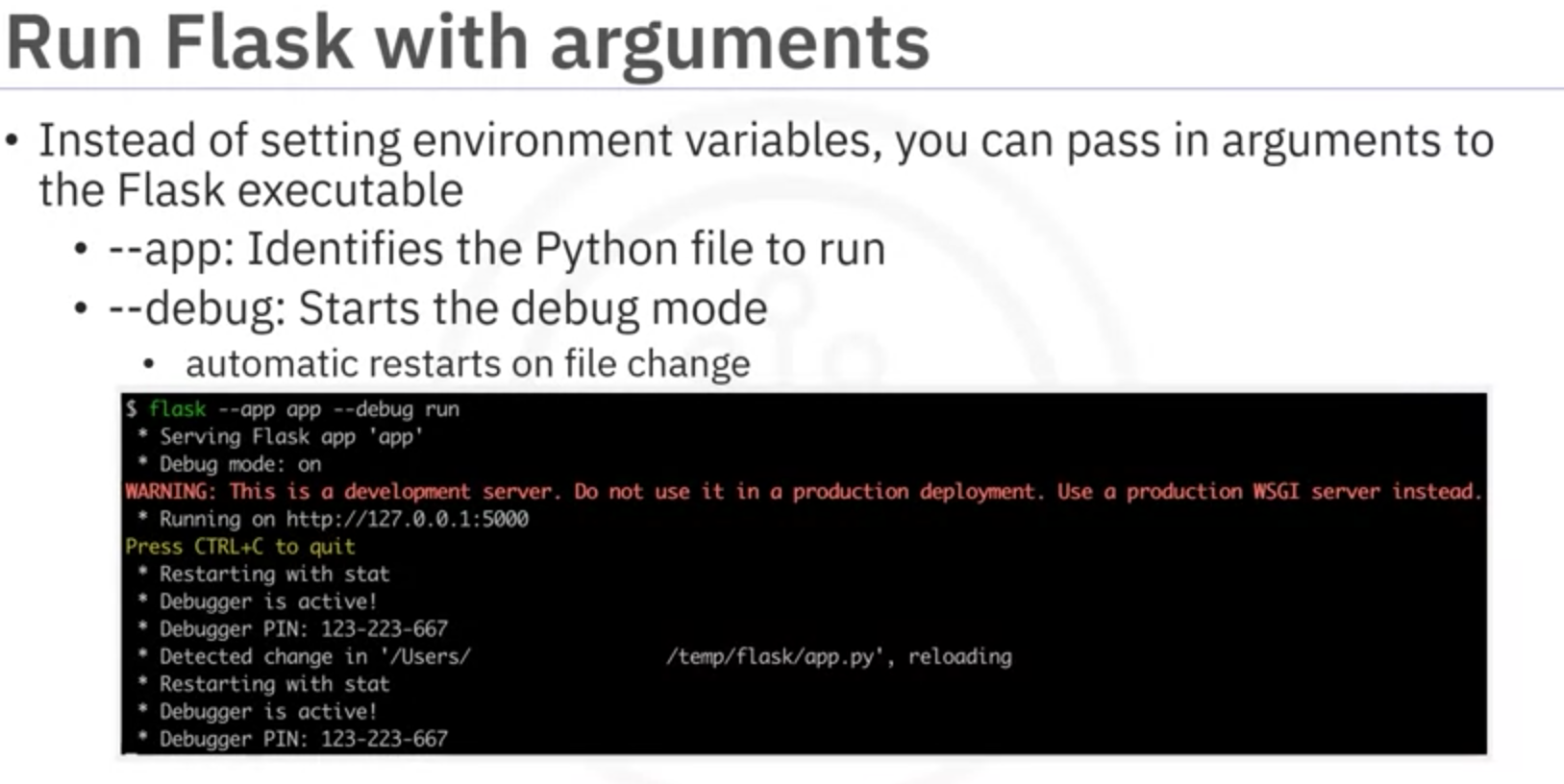


##### Run the application:

* flask run



##### Access the application at http://localhost:5000 in a browser.



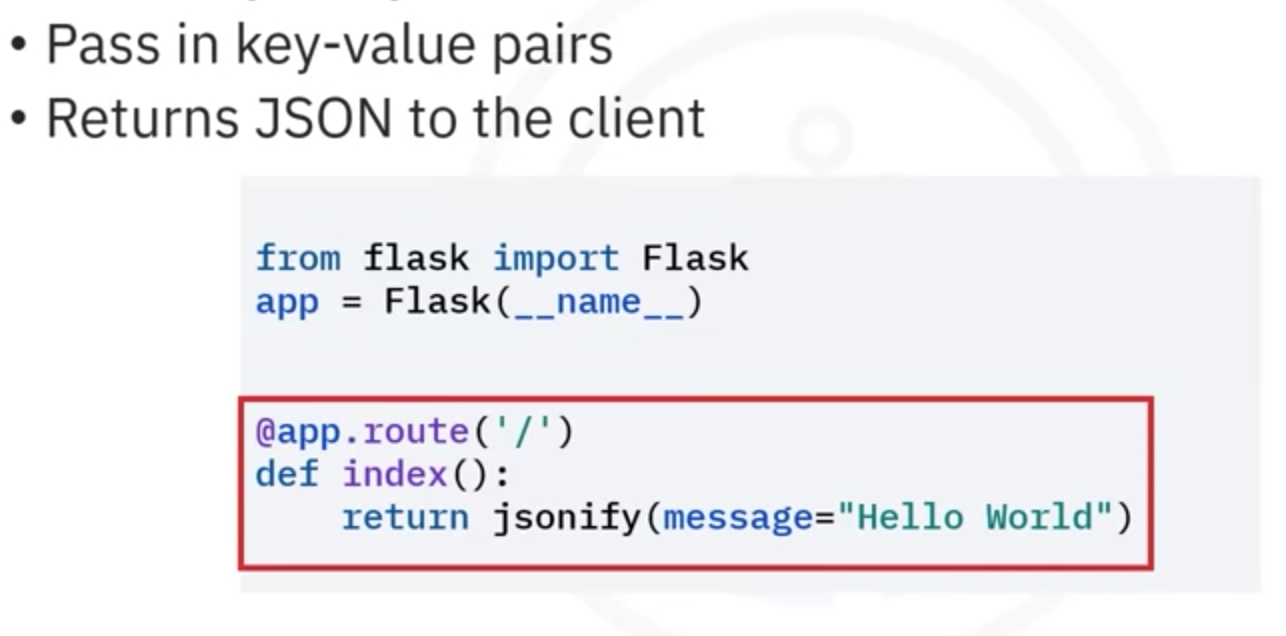
#### Returning JSON Responses:

Flask allows returning JSON responses in various ways:

* Return a serializable object like a dictionary or list. If we return a more complex object-like a class- then ensure that it is serializable.
* Serialization is the process of converting complex data structures, such as objects or classes, into a format that can be easily stored or transmitted. In the case of web applications, JSON (JavaScript Object Notation) is a common format used for serializing data because it is lightweight, human-readable, and widely supported.



* Use the jsonify() method provided by Flask.

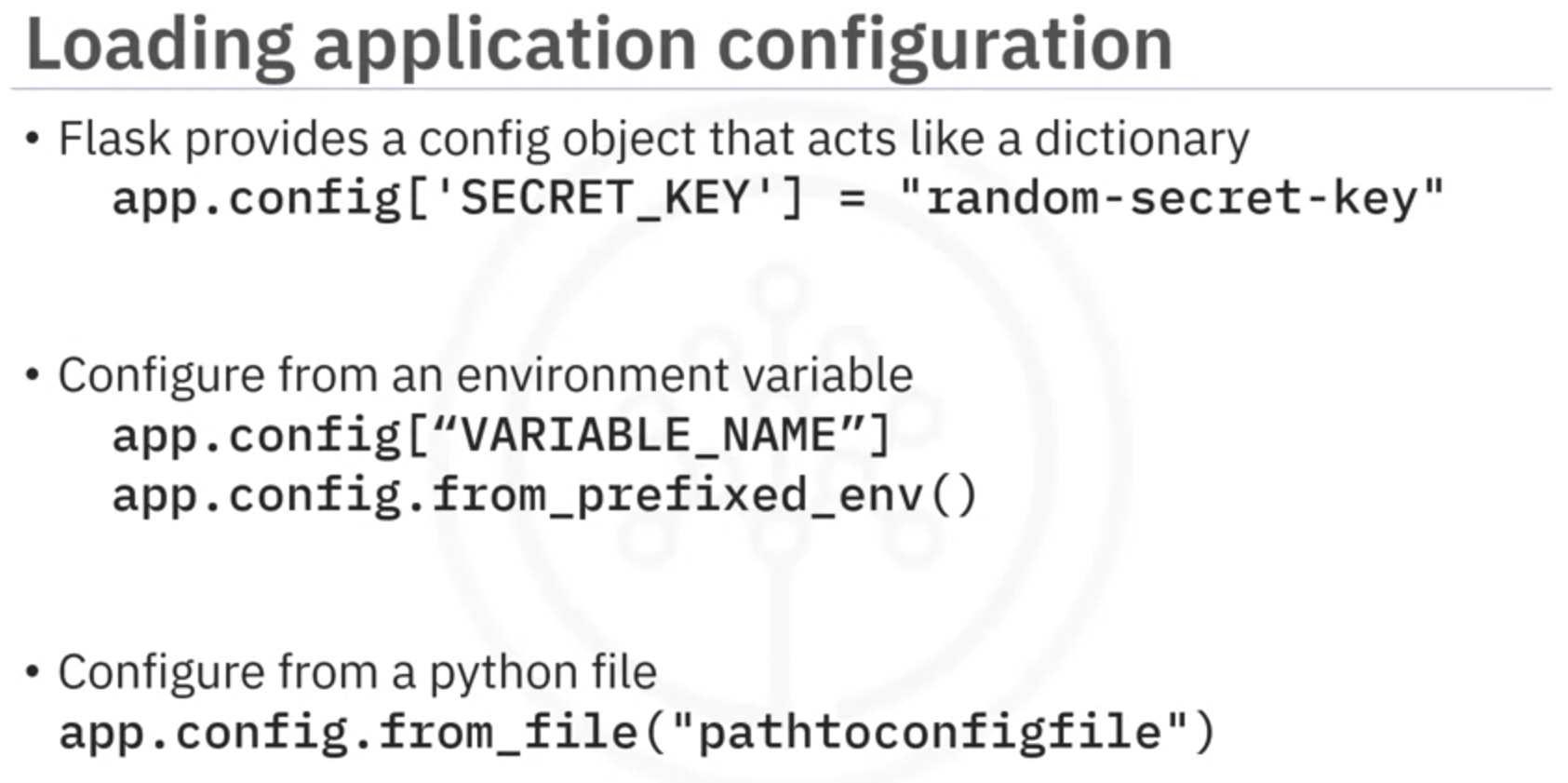


#### Configuration Options in Flask:

Flask provides various configuration options:

* ENV: Indicates the environment (production or development).
* DEBUG: Enables debug mode.
* TESTING: Enables testing mode.
* SECRET\_KEY: Used to sign the session cookie.
* SESSION\_COOKIE\_NAME: Name of the session cookie.
* SERVER\_NAME: Binds the host and port.
* JSONIFY\_MIMETYPE: Defaults to 'application/json'.

Configuration options can be set using environment variables, directly in Python files, or through a separate JSON file.



#### Best Practices for Application Structure:

Organize your application into a directory structure:

* Main source code in the root directory.
* Configurations are also stored in a separate file in root directory.
* Static assets (images, JavaScript, CSS) in a dedicated directory.
* Dynamic content in a template directory.
* Test files in a separate directory.
* Use a virtual environment to manage dependencies.

#### Use the following command to access the website using cmd (not powershell)

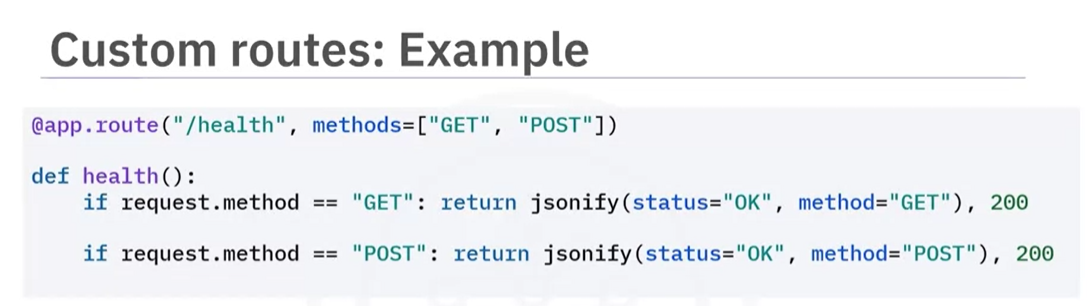
curl -X GET -i -w '\n' localhost:5000

### Request and Response Objects – Using Get and Post Modes

#### Changing the default ‘method’ (Get method is default) of routes

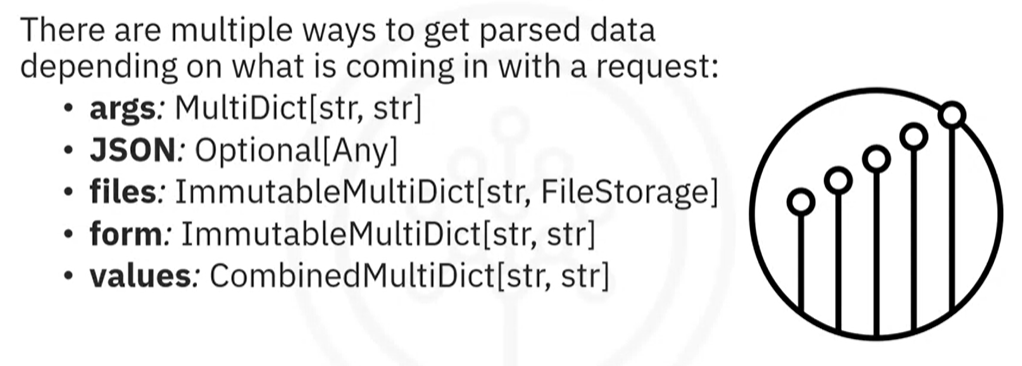


Responding to more than one type of methods as shown below

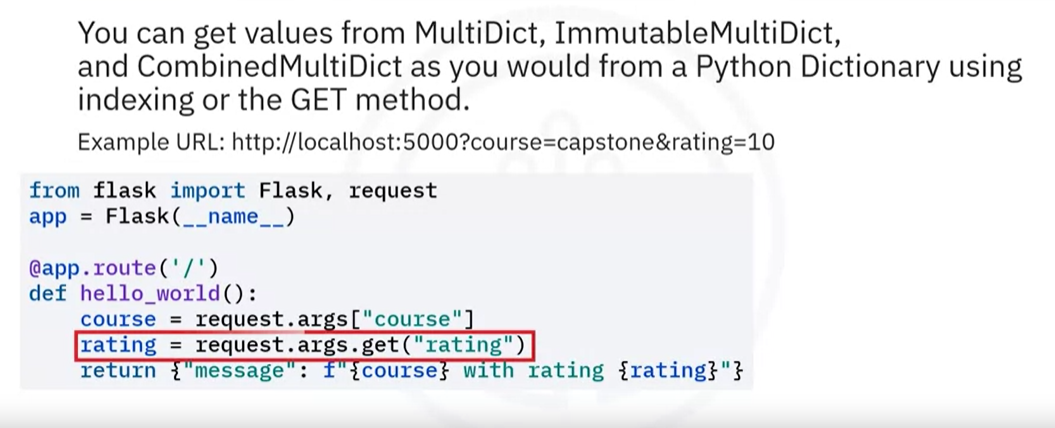


#### Request Object:

* Created from Flask.Request class for every client request.
* Holds information about the request:
* Client address (IP, port)
* Headers (e.g., User-Agent, Accept)
* Requested URL and path
* Query parameters (args)
* Request body data (accessed with different methods based on content type: .get\_data(), .get\_json())
* Uploaded files (files)
* Form data (form)
* Methods to access data:



The return types in the above are the same as python dictionary and we can use either indexing or .get() method just like a normal dictionary to access data as shown below:



The advantage of .get() method is that it would return ‘none’ if the data does not exist, whereas in case of indexing it will generate error and stop the server.

#### Response Object:

* Used to send data back to the client.
* Common attributes of response Object:
* status\_code
* headers: Additional information about the response
* content\_type: Media type of the response (e.g., text/html)
* content\_length: Size of the response body
* content\_encoding:
* mime\_type
* expires
* Methods of response object:
* .set\_cookie(): Set a cookie on the client
* .delete\_cookie(): Delete a cookie on the client

#### Additional Points:

* Flask automatically creates a basic response object with status code of 200 and mime\_type of html/text when you return data from a route handler.
* JSONify() also creates response object automatically.
* make\_response is used to create custom response.
* Flask provides a redirect() method to return a 302 status code and redirect the client to a different URL.
* abort() method is used to return a response with an error condition.
* You can customize responses using methods like **make\_response(), .redirect(), and .abort().**

### Dynamic Routes

We will learn how to call external APIs and describe how to pass parameters to routes in flask.

Dynamic Routes: URLs that can contain variable parts to handle different resources.

RESTful API Endpoints: URLs that follow a standard structure to access and manipulate resources.

#### Calling External APIs

* Utilize the Python requests library to call external APIs within Flask.
* Process the API response before sending it back to the client.
* Example:

//It is assumed that we have installed the ‘requests’ library

from flask import Flask, escape

import requests

@app.route('/search\_author')

def search\_author():

res = requests.get('https://openlibrary.org/search.json?author=Michael+Crichton')

if res.status\_code == 200:

return res.json()

elif res.status\_code == 404:

return "Something went wrong."

else:

return "Server error", 500

#### Passing Parameters to Routes

* Use dynamic routing in Flask to handle parameters passed as part of the URL.
* Set parameter type for validation of incoming requests.
* Examples:
* Use dynamic parts of the URL, such as ISBN, to fetch book information from the OpenLibrary API.
* Set parameter types, such as string, int, float, path, or UUID, to validate requests effectively.
* Endpoint with ISBN Parameter:

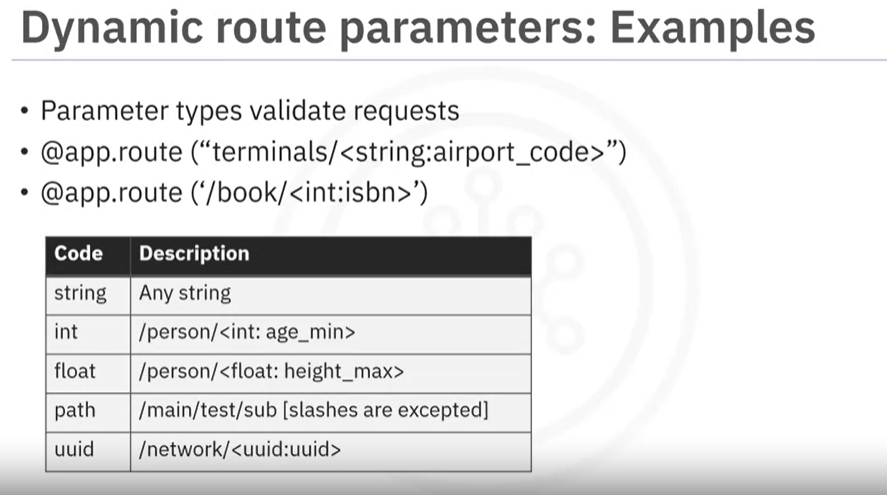
@app.route('/book/<isbn>')

def get\_book\_info(isbn):

res = requests.get(f'https://openlibrary.org/isbn/{isbn}.json')

return res.json()

#### Parameter Types:



* string, int, flaot, path, and UUID
* Example with UUID:

@app.route('/network/<uuid:network\_id>')

def get\_network\_info(network\_id):

if network\_id == 'example\_uuid':

return "Success: Network found."

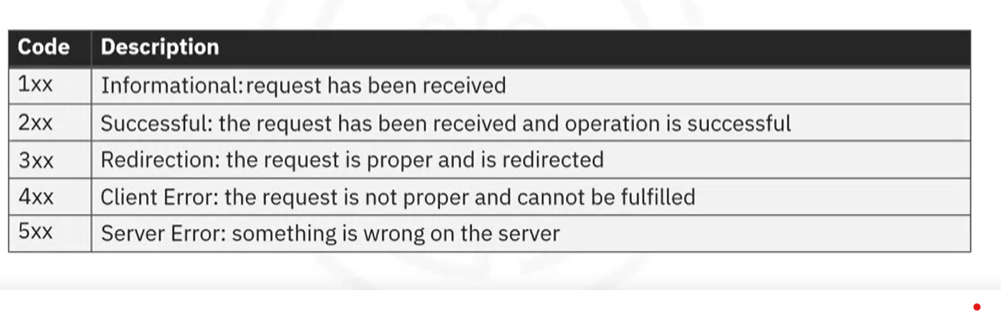
else:

return "Error: Network not found.", 404

### Error Handling

We will describe different http status, explain error handling in flask and explain how to return errors from APIs in flask.

#### HTTP Status Codes



* HTTP responses include a three-digit status code indicating different error and success statuses.
* Status codes range from 100 to 599, categorized into informational, success, redirection, client error, and server error codes.
* Examples of Status Codes:

1. 200: Request successful.
2. 201: Resource created.
3. 202: Request accepted and in process.
4. 204: Request completed successfully with no content returned.
5. 400: Invalid request.
6. 401: Missing or invalid credentials.
7. 403: Insufficient client credentials.
8. 404: Resource not found.
9. 405: Requested operation not supported.
10. 500: Server error.

#### Returning Status Codes in Flask

* Flask automatically returns a status code of 200 OK by default.
* Explicitly set status codes using tuples () or the make\_response() method.
* Example:

@app.route('/')

def index():

return ("My first application in action!", 200)

@app.route('/create\_resource')

def create\_resource():

response = make\_response("Resource created successfully!", 201)

return response

#### Handling Errors

* Flask allows defining application-level error handlers to handle specific HTTP error codes.
* Customize error messages and status codes for better user experience.
* Example:

@app.errorhandler(404)

def not\_found\_error(error):

return "API not found", 404

@app.errorhandler(500)

def internal\_server\_error(error):

return "Something went wrong on the server", 500