

Classes

`class MyClass:`

`#Define class attributes in the class block`

`my_class_attributes = "class attributes go here"`

`MY_CONSTANT = "they are often class-specific constants"`

`def __init__(self):`

`self.my_instance_attribute = "instance attributes here"`

```
class MyClass:
```

```
    b = "on class"
```

```
    def __init__(self):
```

```
        self.a = "on instance"
```

```
        print(self.a)
```

```
        print(MyClass.b)
```

```
        print(self.b)
```

```
        self.a = "re-bound"
```

```
        self.b = "new on instance"
```

```
        print(self.b)
```

◀ Accesses the **class attribute**

◀ **Re-binds** the existing instance attribute

◀ Instance attribute **hides** the class attribute

◀ Accesses the **instance attribute** via `self`

Programming using Python

Web Services

Standardized medium to propagate communication between client-server applications on the world wide web.

SOAP API

- SOAP (Simple Object Access Protocol) is an XML-based protocol for accessing web services over HTTP.
- intermediate language so that applications built in various programming languages can communicate with each other effectively.
- Web services use SOAP for exchange of XML data between applications.
- SOAP supports both stateful and stateless operations.
 - Stateful means that the server keeps the information that it receives from the client across multiple requests. These requests are chained together so that the server is aware about the previous requests. Examples are bank transactions, flight bookings, etc.
 - Stateless messaging has enough information about the state of the client so that the server does not have to bother.
- Envelope which contains information about what is to be done with web services.
 - Contains a header and body with a WSDL (Web Service Definition Language) file.
 - Sent to the service provider and that's why SOAP needs larger bandwidth.

REST API

- REST (Representational State Transfer) is an architectural approach for communication purposes often used in various web services development.
- It is a stateless client-server model.
- Web services that are defined on the concept of REST are RESTful web services.
- When a client makes a request via RESTful API, it transfers the representation of the state of the resources to the server.
- This information can be transferred in various formats via HTTP like, JSON, HTML, XLT, and Plain Text, but JSON is the most common language used due to its easy readability by machines and humans.
- Lightweight and scalable service built on REST architecture.
- Simpler to develop, requires less bandwidth
- Uses HTTP verbs like GET, POST, DELETE, PUT and PATCH for CRUD (Create, Read, Update and Delete) operations.

SOAP vs REST

SOAP vs. REST APIs

upwork



Server

SOAP is like using an envelope

Extra overhead, more bandwidth required, more work on both ends (sealing and opening).



App

REST is like a postcard

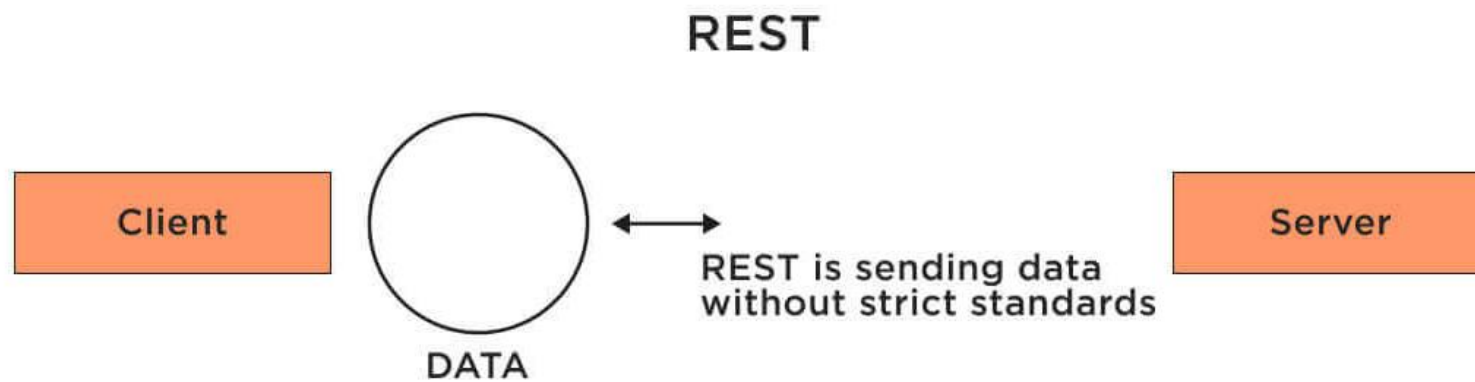
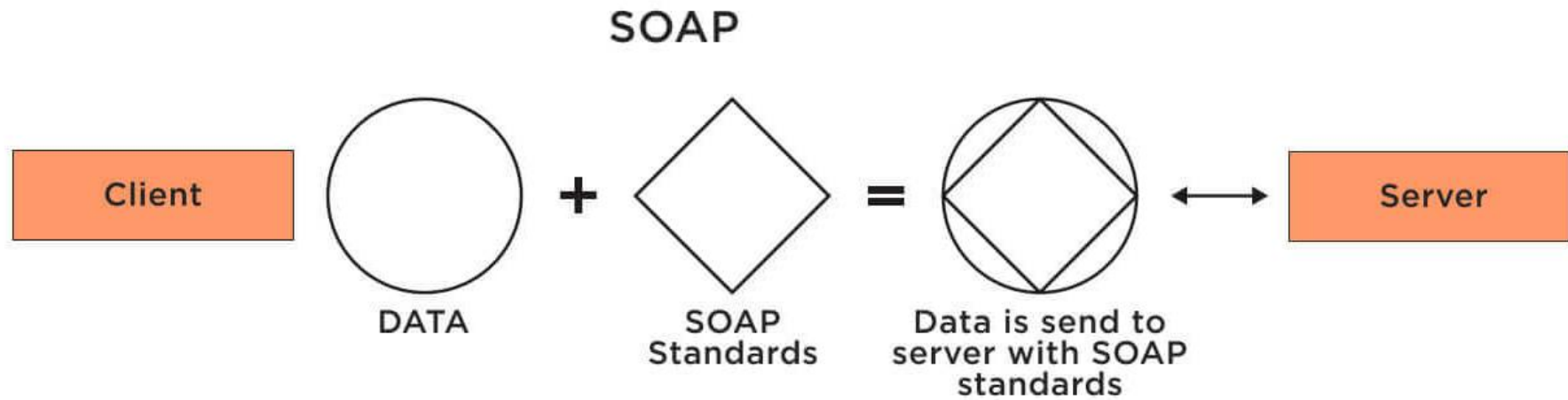
Lighterweight, can be cached, easier to update.

SOAP

- Standardized protocol for pre-defined rules to follow.
- By default stateless, but it is possible to make this API stateful.
- WSDL to expose supported methods and technical details.
- Only supports XML format.
- Requires more bandwidth and computing power
- Preferred for enterprise-level
- SOAP cannot use REST because SOAP is a protocol and REST has an architectural style.

REST

- Architectural style with loose guidelines.
- Stateless, i.e. no server-side sessions occur.
- Exposes methods through URIs, there are no technical details.
- Supports various formats like HTML, XML and JSON.
- Requires less resources, and this makes it more powerful.
- Preferred for public APIs
- REST can use SOAP as a protocol for web services.





- Micro web framework written in Python.
- Provides libraries to build lightweight web applications in python
- Developed by Armin Ronacher
- Based on the Werkzeug WSGI toolkit and the Jinja2 template engine. Both are Pocco projects.
- Why micro?
 - keep its core functionality small yet typically extensible to cover an array of small and large applications
- Features:
 - Built-in development server, fast debugger.
 - Integrated support for unit testing.
 - RESTful request dispatching.
 - Jinja2 Templating.
 - Support for secure cookies.
 - Lightweight and modular design allows for a flexible framework.

Components

- WSGI
 - standard for python web application development
 - specification of a common interface between web servers and web applications.
 - Web Server Gateway Interface
- **Werkzeug**
 - WSGI toolkit that implements requests, response objects, and utility functions.
 - basis for a custom software framework and supports Python
- Jinja
 - popular template engine for Python.
 - A web template system combines a template with a specific data source to render a dynamic web page.



Flask

web development,
one drop at a time

Installation

The following command installs **virtualenv**.

```
pip install virtualenv
```

This command needs administrator privileges. Add **sudo** before **pip** on Linux/Mac OS. If you are on Windows, log in as Administrator. On Ubuntu **virtualenv** may be installed using its package manager.

```
Sudo apt-get install virtualenv
```

Once installed, new virtual environment is created in a folder.

```
mkdir newproj  
cd newproj  
virtualenv venv
```

To activate corresponding environment, on **Linux/OS X**, use the following:

```
venv/bin/activate
```

On **Windows**, following can be used:

```
venv\scripts\activate
```

We are now ready to install Flask in this environment.

```
pip install Flask
```

Hello World! App

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

Variables to HTML

```
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
    <h1>Hello {{ username }}</h1>
  </body>
</html>
```

Render Template

```
from flask import render_template
```

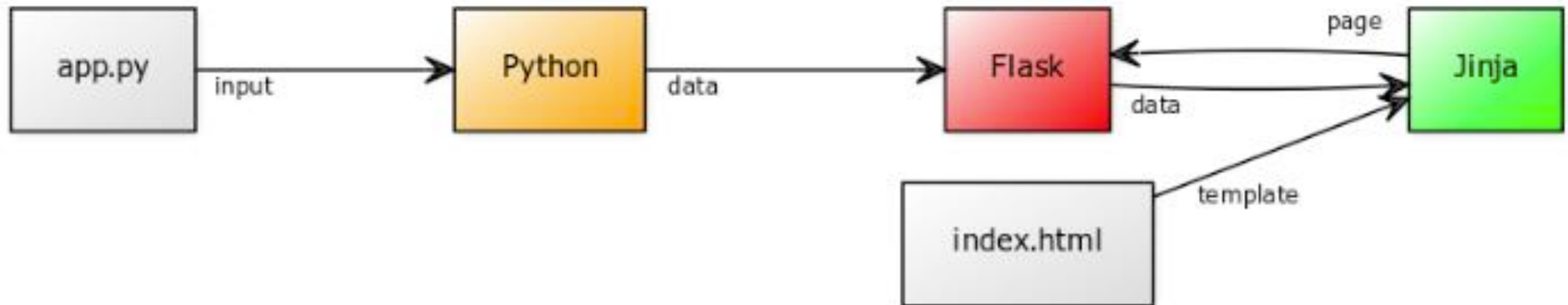
Change the original code:

```
@app.route('/')
def index():
    return 'Web App with Python Flask!'
```

Into one that renders the template and passes variables:

```
@app.route('/')
@app.route('/index')
def index():
    name = 'Rosalia'
    return render_template('index.html', title='Welcome', username=n
```

Render Template



Loops

```
@app.route('/')
@app.route('/index')
def index():
    users = [ 'Rosalia', 'Adrianna', 'Victoria' ]
    return render_template('index.html', title='Welcome', members=users)
```

The code includes a list (users). That list is passed to the render_template function. In the template, you can use a for loop to iterate over the list.

```
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
    <ul>
      {% for member in members: %}
      <li>{{ member }}</li>
      {% endfor %}
    </ul>
  </body>
</html>
```


If-else

```
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
    {% if username == "Rosalia": %}
    <h1>Hello my love</h1>
    {% else %}
    <h1>Hello {{ username }}</h1>
    {% endif %}
  </body>
</html>
```

Routing

flask route params

Parameters can be used when creating routes. A parameter can be a string (text) like this:

`/product/cookie` .

That would have this route and function:

```
@app.route('/product/<name>')
def get_product(name):
    return "The product is " + str(name)
```

So you can pass parameters to your Flask route, can you pass numbers?

The example here creates the route `/sale/<transaction_id>` , where `transaction_id` is a number.

```
@app.route('/sale/<transaction_id>')
def get_sale(transaction_id=0):
    return "The transaction is "+str(transaction_id)
```

flask route multiple arguments

If you want a flask route with multiple parameters that's possible. For the route

`/create/<first_name>/<last_name>` you can do this:

```
@app.route('/create/<first_name>/<last_name>')
def create(first_name=None, last_name=None):
    return 'Hello ' + first_name + ', ' + last_name
```

HTTP Methods: POST

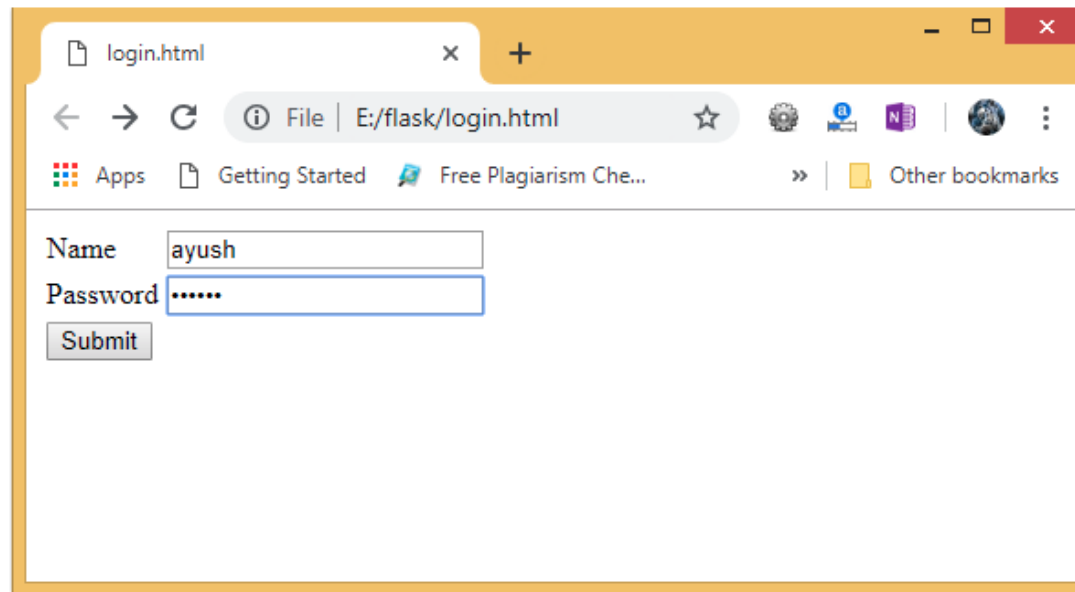
post_example.py

```
from flask import *
app = Flask(__name__)

@app.route('/login', methods = ['POST'])
def login():
    uname=request.form['uname']
    passwd=request.form['pass']
    if uname=="ayush" and passwd=="google":
        return "Welcome %s" %uname

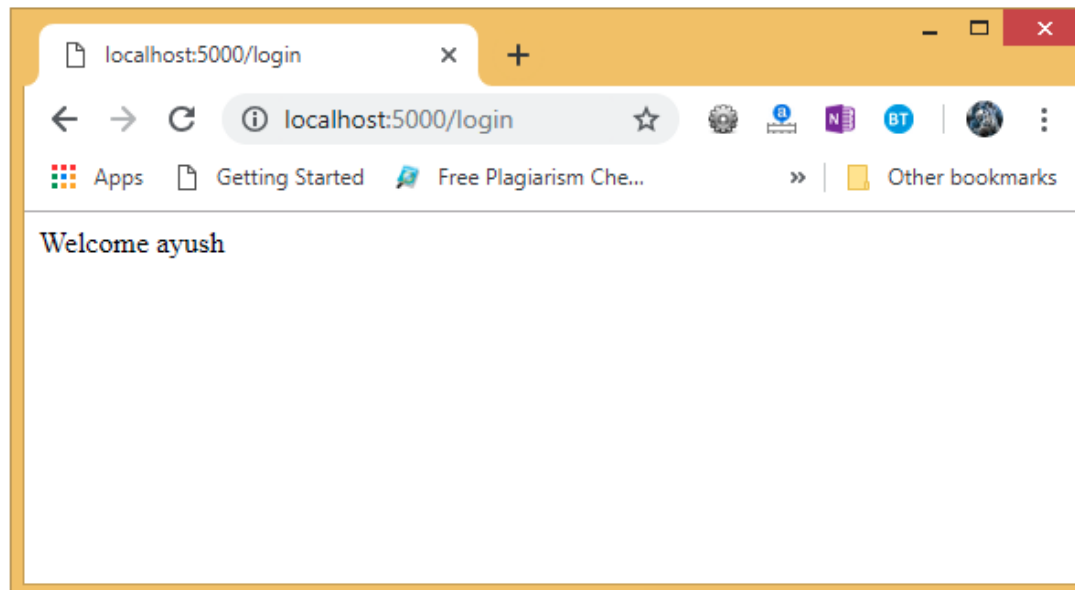
if __name__ == '__main__':
    app.run(debug = True)
```

HTTP Methods: POST



A screenshot of a web browser window displaying a login form. The browser's address bar shows the file path `E:/flask/login.html`. The form contains two input fields: "Name" with the text "ayush" and "Password" with masked characters ".....". Below the password field is a "Submit" button. The browser's tab is labeled "login.html".

Give the required input and click Submit, we will get the following result.



HTTP Methods: GET

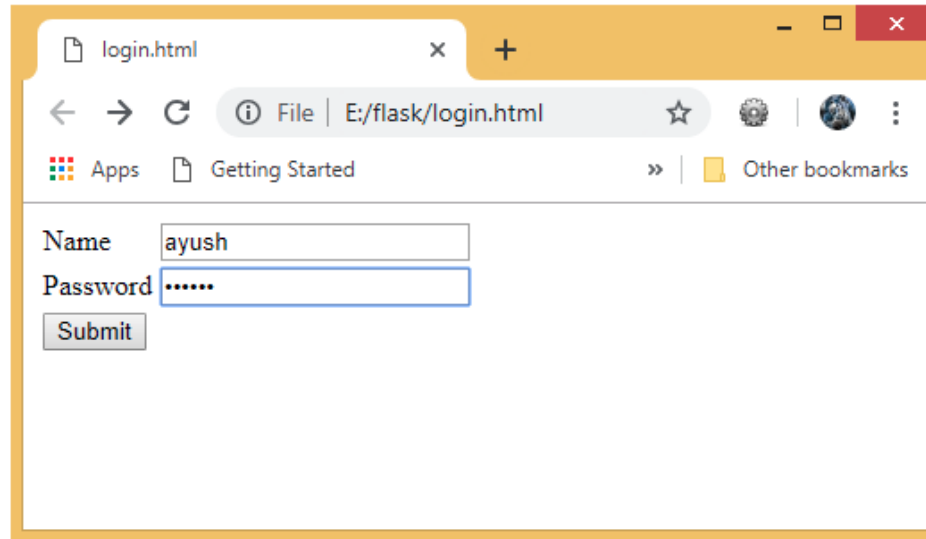
get_example.py

```
from flask import *
app = Flask(__name__)

@app.route('/login', methods = ['GET'])
def login():
    uname=request.args.get('uname')
    passwr=request.args.get('pass')
    if uname=="ayush" and passwr=="google":
        return "Welcome %s" %uname

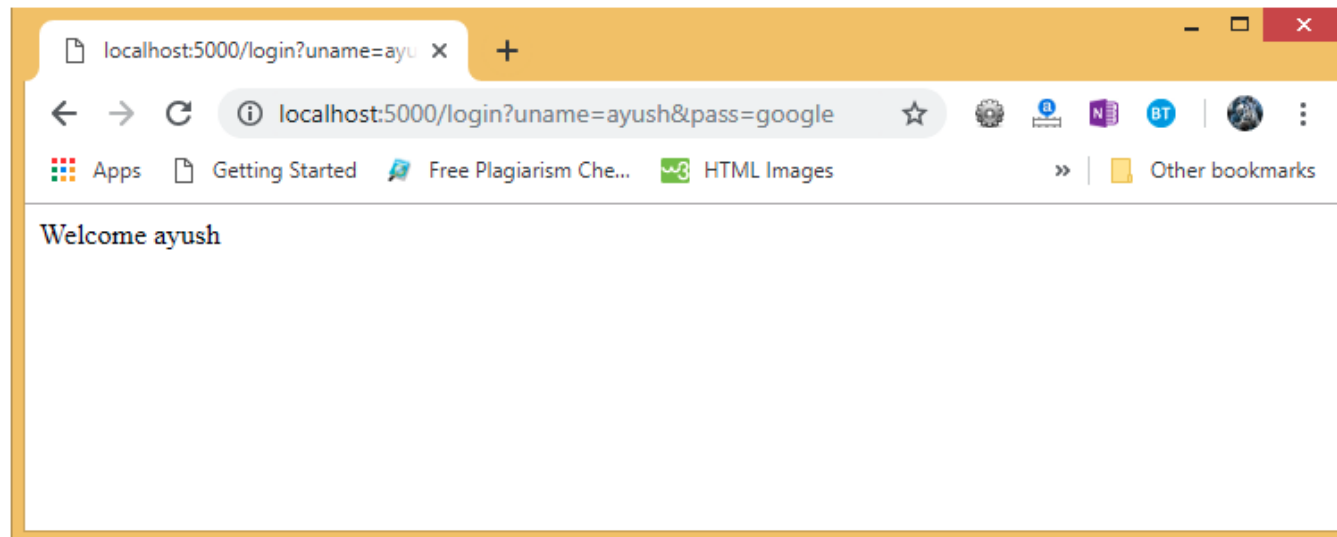
if __name__ == '__main__':
    app.run(debug = True)
```

HTTP Methods: GET



A screenshot of a web browser window with a single tab titled 'login.html'. The address bar shows the file path 'E:/flask/login.html'. The page content includes a form with two input fields: 'Name' containing the text 'ayush' and 'Password' containing seven dots. Below the password field is a 'Submit' button.

Now, click the submit button.



A screenshot of a web browser window after a successful login. The address bar shows 'localhost:5000/login?uname=ayush&pass=google'. The page content displays the text 'Welcome ayush'.