1. Flask Framework with Python

1.1 Setting up the Environment

1.1.1 Objectives:

It's good practice to have a different python environment for different development setup. Here we're learning about how to setup an virtual environment for Flask Framework for our web application development.

By the end of this section, you should be able to:

- Install `virtualenv` , `virtualwrapper` and create virtual environments
- Use Flask to setup a simple server and respond with text

1.1.2 Getting set up in a virtual environment

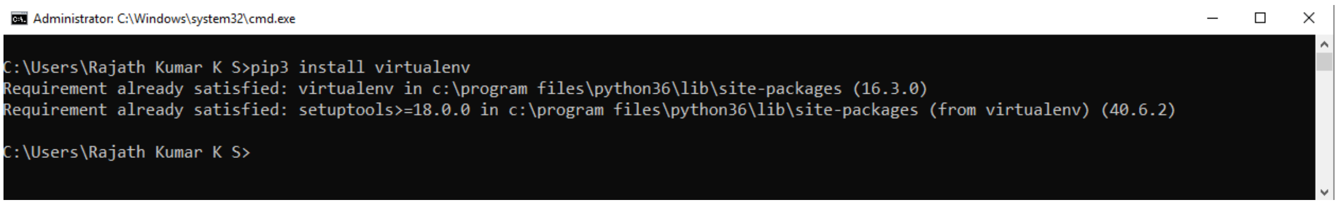
As you start working on more Python projects, you may find that managing dependencies becomes problematic. For example, you may start a project that uses version 3.5.x of some library, and later on you may start a new project and realize that version 3.7.x of the module is updated. But this new version breaks things in your old project, so you need to either deal with a broken old project, or a new project with an old version of the library.

It would be better if we could separate out distinct environments for each Python project we worked on. That way, we wouldn't have to worry about dependencies in one project potentially interfering with other projects.

Thankfully, there's a way to set this up without much trouble in Python. The tool we'll be using is called `virtualenv` and `virtualwrapper` . Before we get started with any significant Python development, let's set up `virtualenv` and `virtualwrapper` so we can create separate environments for each project we'll be working on.

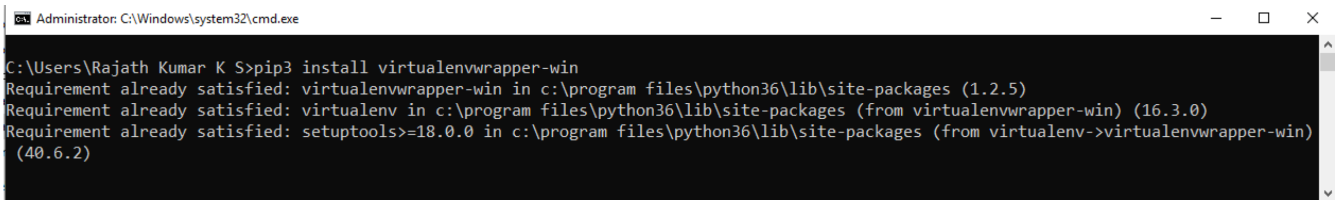
1.1.2.1 Installing `virtualenv` and `virtualwrapper`

```
pip3 install virtualenv
```



```
C:\Users\Rajath Kumar K S>pip3 install virtualenv
Requirement already satisfied: virtualenv in c:\program files\python36\lib\site-packages (16.3.0)
Requirement already satisfied: setuptools>=18.0.0 in c:\program files\python36\lib\site-packages (from virtualenv) (40.6.2)

C:\Users\Rajath Kumar K S>
```

```
pip3 install virtualenvwrapper-win
```



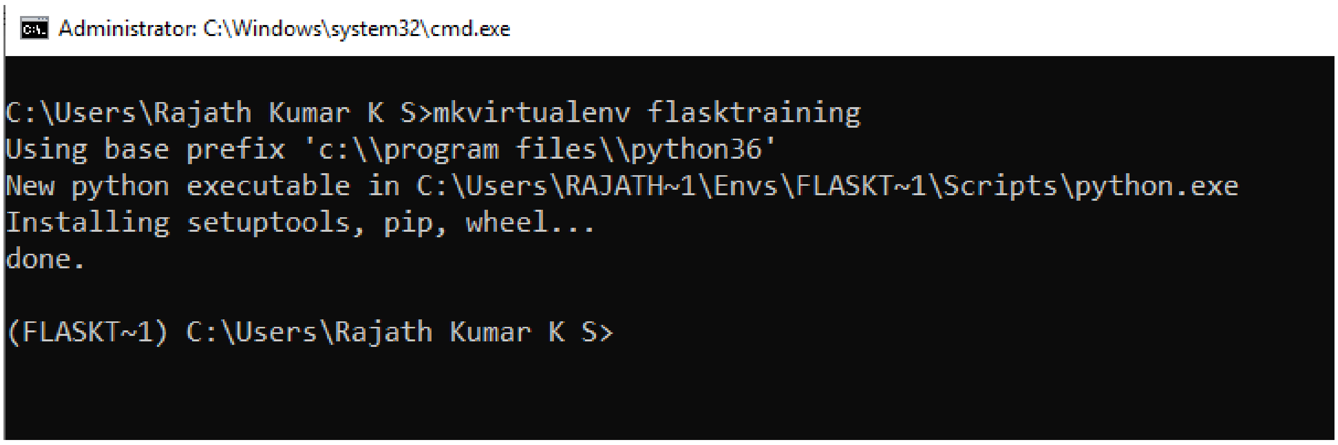
```
C:\Users\Rajath Kumar K S>pip3 install virtualenvwrapper-win
Requirement already satisfied: virtualenvwrapper-win in c:\program files\python36\lib\site-packages (1.2.5)
Requirement already satisfied: virtualenv in c:\program files\python36\lib\site-packages (from virtualenvwrapper-win) (16.3.0)
Requirement already satisfied: setuptools>=18.0.0 in c:\program files\python36\lib\site-packages (from virtualenv->virtualenvwrapper-win)
 (40.6.2)
```

1.1.2.2 Creating Virtual Environment

To create virtual environment, you'll use `mkvirtualenv` command

```
mkvirtualenv flasktraining
```

Here in this case i'm using my virtual environment name as **flasktraining**, you can use your own virtual environment name.

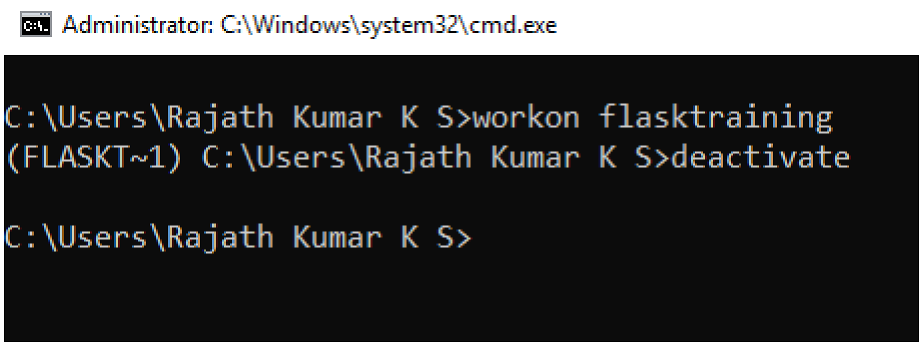


```
C:\Users\Rajath Kumar K S>mkvirtualenv flasktraining
Using base prefix 'c:\\program files\\python36'
New python executable in C:\Users\RAJATH~1\Envs\FLASKT~1\Scripts\python.exe
Installing setuptools, pip, wheel...
done.

(FLASKT~1) C:\Users\Rajath Kumar K S>
```

When it's created , it gets into virtual environment. To get out of your virtual environment use `deactivate`

To work in virtual environment you need to type in as follows,

```
workon flasktraining
```


Administrator: C:\Windows\system32\cmd.exe

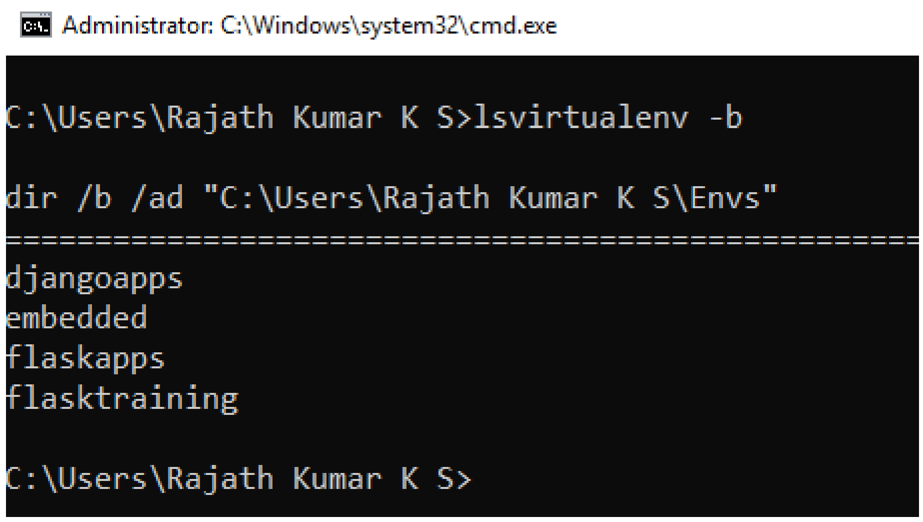
```
C:\Users\Rajath Kumar K S>workon flasktraining
(FLASKT~1) C:\Users\Rajath Kumar K S>deactivate

C:\Users\Rajath Kumar K S>
```

```
deactivate
```

to exit the virtual environment to your base machine.

To list the virtualenvironments that we've created in our system, you can use the `lsvirtualenv -b` command (`-b` stands for "brief").

```
lsvirtualenv -b
```


Administrator: C:\Windows\system32\cmd.exe

```
C:\Users\Rajath Kumar K S>lsvirtualenv -b

dir /b /ad "C:\Users\Rajath Kumar K S\Envs"
====================================================
djangoapps
embedded
flaskapps
flasktraining

C:\Users\Rajath Kumar K S>
```

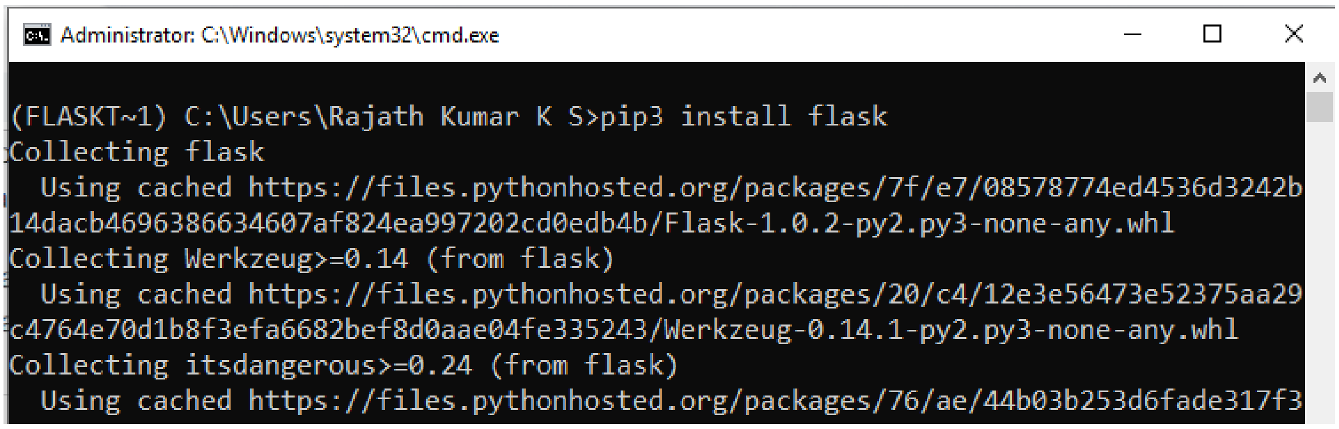
In my case i've installed with four virtual environments viz.,

1. djangoapps
2. embedded
3. flaskapps
4. flasktraining -- Virtual Environment that created now.

Let's install flask python package for the virtual environment that we've created . To install flask package i'm using PyPi (Python Package Index)

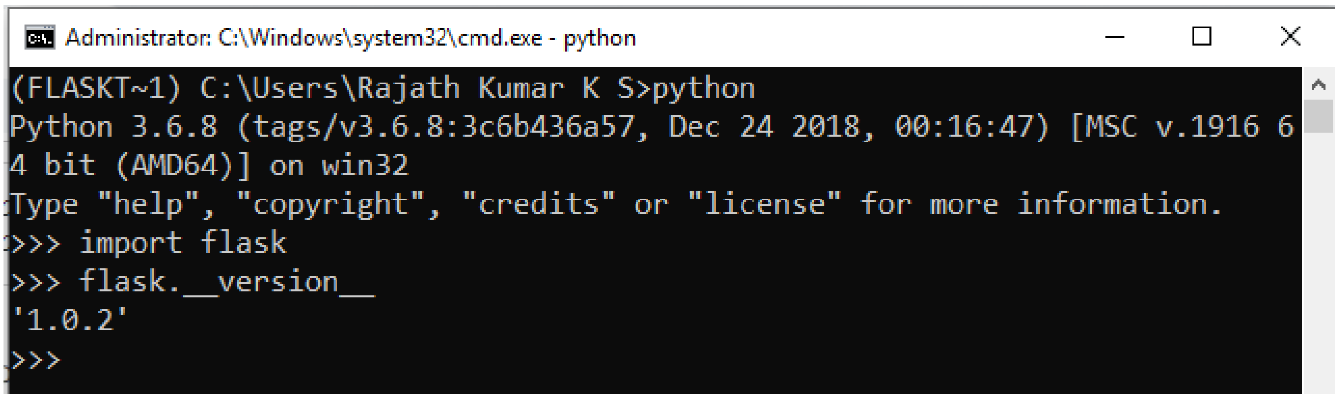
```
pip3 install flask
```

make sure that you're working inside the virtual environment you've created.



```
(FLASKT~1) C:\Users\Rajath Kumar K S>pip3 install flask
Collecting flask
  Using cached https://files.pythonhosted.org/packages/7f/e7/08578774ed4536d3242b
14dacb4696386634607af824ea997202cd0edb4b/Flask-1.0.2-py2.py3-none-any.whl
Collecting Werkzeug>=0.14 (from flask)
  Using cached https://files.pythonhosted.org/packages/20/c4/12e3e56473e52375aa29
c4764e70d1b8f3efa6682bef8d0aae04fe335243/Werkzeug-0.14.1-py2.py3-none-any.whl
Collecting itsdangerous>=0.24 (from flask)
  Using cached https://files.pythonhosted.org/packages/76/ae/44b03b253d6fade317f3
```

Let's check our installation happend we made has done properly, to check go to python interactive shell by typing `python` in your command prompt. Then type in `import flask`, if you don't see any error then you're installed flask on your virtual environment



```
(FLASKT~1) C:\Users\Rajath Kumar K S>python
Python 3.6.8 (tags/v3.6.8:3c6b436a57, Dec 24 2018, 00:16:47) [MSC v.1916 6
4 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import flask
>>> flask.__version__
'1.0.2'
>>>
```

1.2 Introduction to Flask Framework

> **Pirates use Flask, the Navy uses Django**

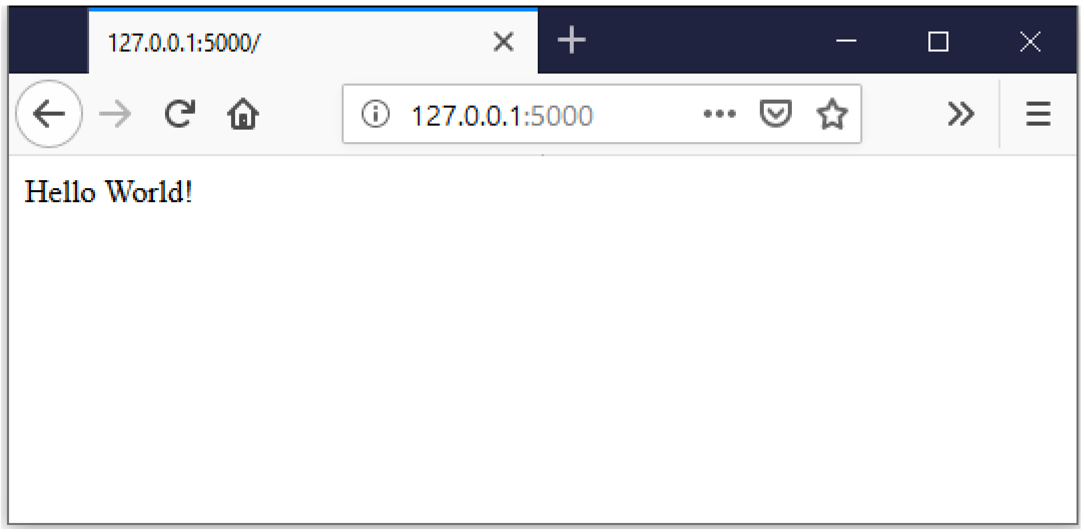
1.2.1 What is Flask ?

Flask is a micro-framework in Python. It allows us to easily start a server, and, when combined with other modules, build sophisticated applications. Flask is very easy to get started with so let's jump in!

Now let's create an main.py file in the desired location. To begin, we'll just use the below code.

```python
# from the flask library import a class named Flask
from flask import Flask

# create an instance of the Flask class
app = Flask(__name__)

# listen for a route to `/` - this is known as the root route
@app.route('/')
def home():
    return "Hello World!"

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

Make sure we saved our file. Next, in the terminal let's run `python main.py` and head over to `localhost:5000`/`127.0.0.1:5000` in the browser. You should see the text "Hello World" appear on the page!.

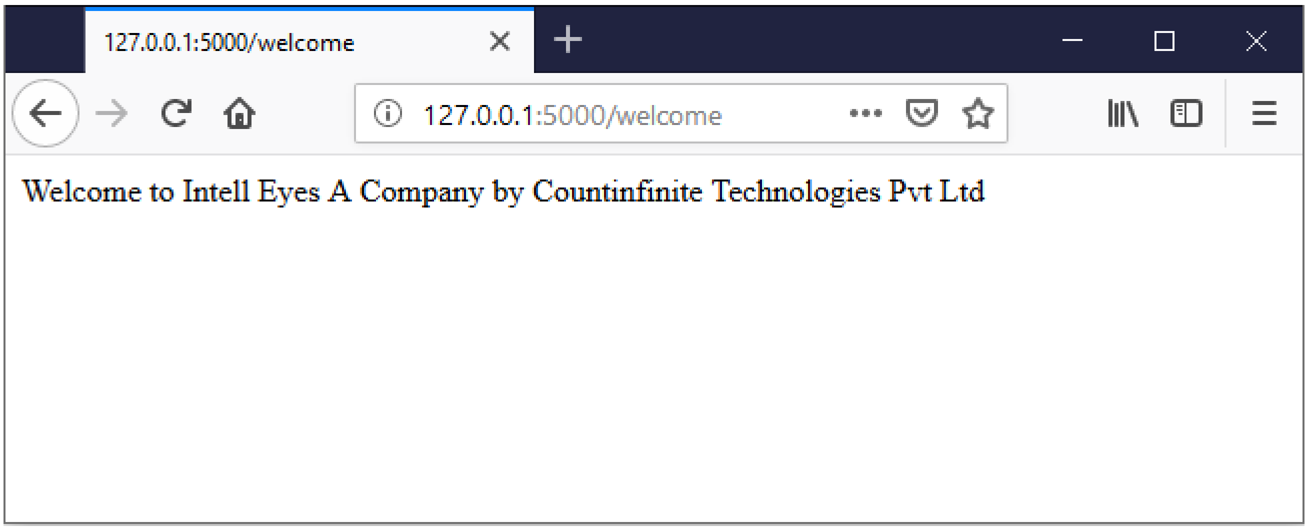


Hello World!

Now let's try to add another route to our application. When a user goes to `localhost:5000\welcome` we should return the text "Welcome to Intell Eyes A Company by Countinfinite Technologies Pvt Ltd." . We will need to include another route as well as a function to run when that route is reached.

```python
# from the flask library import a class named Flask
from flask import Flask

# create an instance of the Flask class
app = Flask(__name__)

# listen for a route to `/` - this is known as the root route
@app.route('/')
def home():
    return "Hello World!"

@app.route('/welcome')
def welcome():
    return "Welcome to Intell Eyes A Company by Countinfinite Technologies Pvt Ltd"

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

Check back your link that you created has a new route by hitting `127.0.0.1:5000\welcome` on your favourite browser.



Welcome to Intell Eyes A Company by Countinfinite Technologies Pvt Ltd

1.2.2 Flask Exercises

For thsi assignemtn you will be creating a very small flask appkication,. Your application should:

- have a route for `/welcome` , which responds with the string "Welcome to Flask from Intell Eyes".
- have a route for `/welcome/home`, which responds with the string "Welcome Home"

Bonus Exercise

- have a route for `/welcome/<yourname>`, which responds with string "Welcome `<yourname>`"

2. Templating with Jinja2

2.1 What is a Template ?

Very commonly, we want to send data from our server back to our client. In the olden days, servers often just held a collection of files; the client would request, say, an HTML file, and the server would send it over. Because these files were static and unchanged by the server, such sites are often referred to as *static sites*.

In more modern web development, instead of sending static data all the time, we often want to send *dynamic* data, which may depend on which user is signed in, whether a user's account has expired, and so on. To do this we will be using the templating engine built into Flask, which is called **Jinja2**.

2.1.1 Objectives:

By the end of this section, you should be able to:

- Use Flask to respond by rendering HTML instead plain text
- Use Jinja2 as a server side templating engine
- Pass values to a server side template with Flask and evaluate them with Jinja

2.1.2 Jinja2

Since Jinja2 comes with Flask we do not need to `pip install` anything and we can get started right away. To evaluate data from our server in our templates we use the `{% %}` notation and to print data we use `{{ }}`. (Not clear on the difference between evaluating and printing data in a template ? Don't worry, we'll see an example soon)

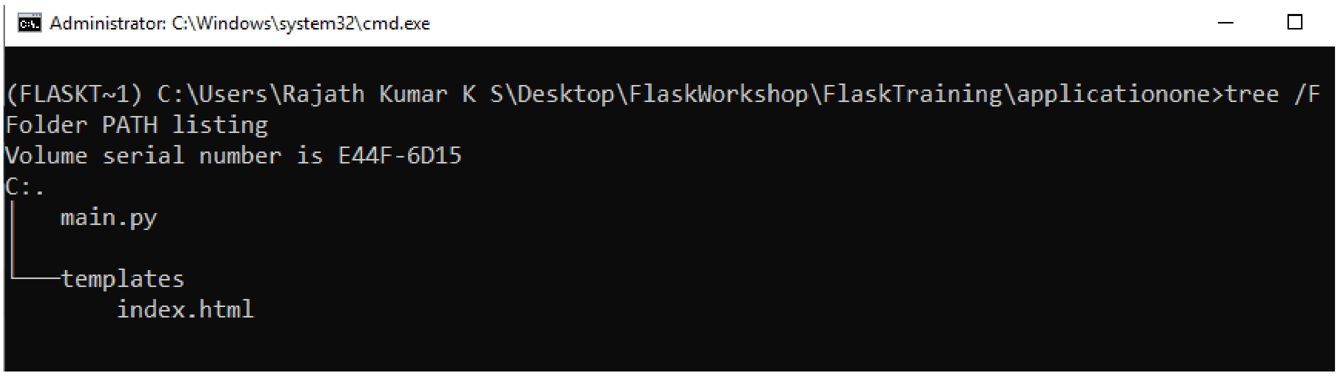
Let's start by first having Flask render HTML. To do so we must include `render_template` when importing from flask. Let's make an `flask`. Let's make an `main.py` and include the following:

```
# from the flask library import a class named Flask
from flask import Flask, render_template # we are now importing just more than Flask!

app = Flask(__name__)

@app.route('/')
def welcome():
    return render_template('index.html')

@app.route('/second')
def second():
    return "WELCOME TO THE SECOND PAGE!"

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

So what's happening above ? Instead of just rendering plain text, we are rendering a template called `index.html`. Now in order to make sure that Flask can find our templates, we need to create a folder called `templates` and inside let's create our `index.html` file.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Intell Eyes</title>
</head>
<body>
   <h1>Hello World</h1>
   <h2>Flask Framework</h2>
</body>
</html>
```

Make sure you've made a correct directory structure, as shown below


Administrator: C:\Windows\system32\cmd.exe

```
(FLASKT~1) C:\Users\Rajath Kumar K S\Desktop\FlaskWorkshop\FlaskTraining\applicationone>tree /F
Folder PATH listing
Volume serial number is E44F-6D15
C:.
│   main.py
│
└───templates
        index.html
```

```
root
  |- main.py
  |- templates
  |     |- index.html
  |     |
```

When you run the above application it's just rendering the static Webpage as we created in a template directory as index.html, But as jinja says it has to work with the dynamic content . so here is the second example to understand the power of Jinja2

Python Main.py

```
# from the flask library import a class named Flask
from flask import Flask, render_template # we are now importing just more than Flask!

app = Flask(__name__)

@app.route('/')
def welcome():
    names_of_instructors = ["Rajath", "Amarnath", "Sudarshan"]
    random_name = "Rajath"
    return render_template('index.html', names=names_of_instructors, name=random_name)

@app.route('/second')
def second():
    return "WELCOME TO THE SECOND PAGE!"

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

index.html in templates directory which has some logic here,

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Intell Eyes</title>
</head>
<body>
    {% for instructor_name in names %}
        <p>{{ instructor_name }}</p>
    {% endfor %}

    {% if name == 'Rajath' %}
        <p>Hello {{ name }}!</p>
    {% endif %}
</body>
</html>
```

Well when we see `main.py`,`index.html` now it looks different than the other one let's see what we've done here, we're passing some variables into the template. These variables are called `names` and `name` and their values are the `names_of_instructors` variable (`["Rajath", "Amarnath", "Sudarshan"]`) and `random_name` variable `("Rajath")`.Very commonly these keys and values will be named as the same, but we are using different names to show the difference in the example.

We can iterate over lists using `for` with Jinja and use conditional logic with `if` statements! This means we can render the same HTML page and produce different views for different users. Our pages are now much more dynamic! Think about some applications for something like this: templating allows you to listing all of your friends on a social network, displaying information only if you are an admin, show a logout button if you are logged in, and more, all with just a small number of templates living on the server.

Note also that that only Python code inside of `{{ }}` gets printed to the page. You'll often see `for` loops and `if` statements defined inside of `{% %}`, since we want to evaluate that code, but it's only when we're inside the `if` or `for` blocks that we want to display something on the page (using `{{ }}`).

2.2 Template Inheritance

One of the more powerful features of Jinja is the ability to use template inheritance, which means that one template can inherit from another. Let's see an example! First, let's create a file called `base.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Intell Eyes</title>
</head>
<body>
   {% block content %}
   {% endblock %}
</body>
</html>
```

In another template (in the same directory), create a file called `title.html`

```
{% extends "base.html" %}
{% block content %}
<h1>This page has everything our base.html has!</h1>
{% endblock %}
```

We can inherit from other templates using `extends` keyword. This tremendously reduces our code duplication especially for things like headers and footers. To see this inheritance in action, add another route to your `main.py`

```
# from the flask library import a class named Flask
from flask import Flask, render_template # we are now importing just more than Flask!

app = Flask(__name__)

@app.route('/')
```

```
def welcome():
    names_of_instructors = ["Rajath", "Amarnath", "Sudarshan"]
    random_name = "Rajath"
    return render_template('index.html', names=names_of_instructors, name=random_name)

@app.route('/second')
def second():
    return "WELCOME TO THE SECOND PAGE!"

@app.route('/title')
def title():
    return render_template('title.html')

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

2.3 URL Helpers

Another great helper that Jinja has is the `url_for` helper which eliminates the need for hard coding a URL. This is very helpful when you have dynamically created URLs or don't always know the exact path. Let's imagine we are working in our `index.html` file and we would like an anchor tag to link to the route `/second`. We also know that in our `app.py` the function that is used to send a response to our server is called `second`. This is what we can place as the value for our `url_for()`!

```
{% extends "base.html" %}
{% block content %}
    {% for name in names %}
        {{name}}
    {% endfor %}

    {% if name == 'Rajath' %}
        <p>Hello Rajath!</p>
    {% endif %}

    Tired of this page? Head over to <a href="{{url_for('second')}}">the second page!</a>
{% endblock %}
```

2.4 Getting data from the query string

When a user submits a form via a GET request, that form data can be captured from the query string. Flask makes this process a bit easier with a method called `request`. Let's see it in action. To begin, start with a simple form in a page called `first-form.html`:

```
{% extends "base.html" %}
{% block content %}
    <form action="/data">
        <input type="text" name="first">
        <input type="text" name="last">
        <input type="submit" value="Submit Form">
    </form>
{% endblock %}
```

To capture this data we can use `request` which we need to import from `flask` and access the `args` property inside request.

```
# from the flask library import a class named Flask
from flask import Flask, render_template,request # we are now importing just more than
Flask!

app = Flask(__name__)

@app.route('/')
def welcome():
    return render_template('index.html')

@app.route('/second')
def second():
    return "WELCOME TO THE SECOND PAGE!"

@app.route('/title')
def title():
    return render_template('title.html')

@app.route('/show-form')
def show_form():
    return render_template('first-form.html')

@app.route('/data')
def print_name():
    first = request.args.get('first')
    last = request.args.get('second')
    return first

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

2.4.1 Flask Exercises

Flask Templating

Create a new Flask application with the following:

Part 1

- A `base.html` template for others to inherit from

- A route for `/person/<name>/<age>` which renders a template that displays the name and age entered for the URL. That template should inherit from `base.html`

Part 2

Refactor your calculator application from before!

- have a route for `/calculate` which renders a template called `calc.html`
- in `calc.html`, build a form which has two inputs (one with a name of `num1` and another with the name of `num2`for numbers and a select field with the name of `calculation` with options for "add", "subract", "multiply" and "divide".
- When the form is submitted it should make a request to a route called `/math`
- In your python file, accept the values from the form and depending on what the request contains, respond with the sum, difference, product or quotient.

3. CRUD with Flask