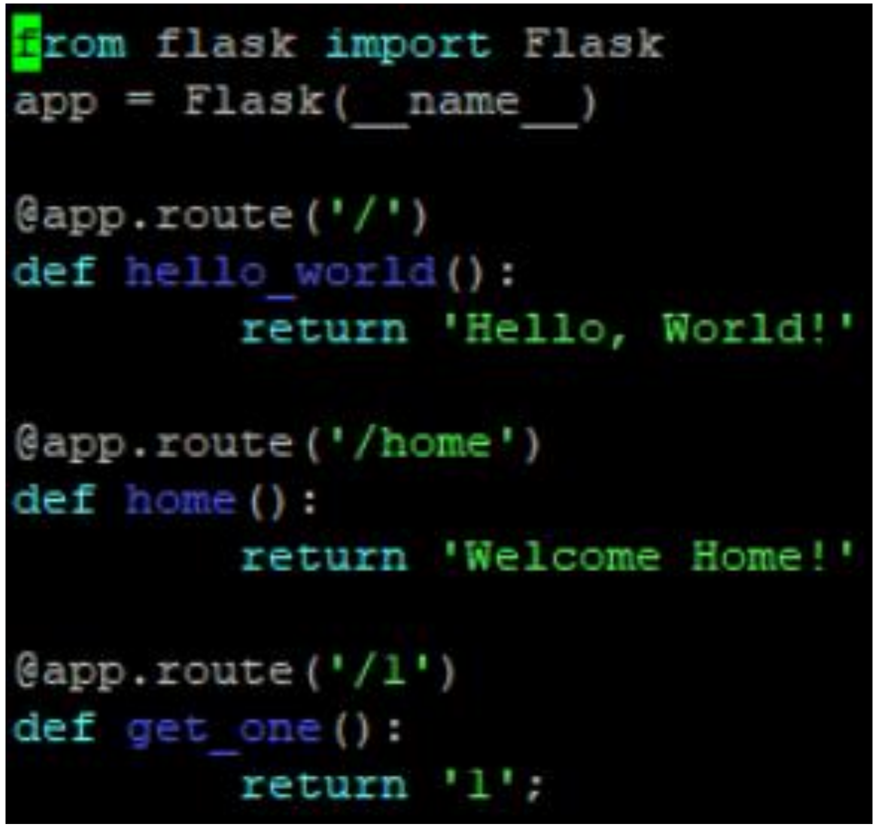
COSC 419 – Topics in Computer Science

Fall 2020

Recap: Getting Started With Flask

- Last lecture, we discussed how to install and configure the Flask web framework
- WSGI is used to interface the Python application with Apache
- We left off with a basic web application using pre-defined routes



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

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

@app.route('/home')
def home():
        return 'Welcome Home!'

@app.route('/1')
def get_one():
        return '1';
```

Routing in Flask

- ***Routing*** is the act of catching the URL of an input GET request, and then directing that request to the appropriate function to handle it
- All routes are declared using the `@app.route` directive, followed by a function to handle the route
- For example, if we wanted to have an about page on our site at www.mywebsite.com/about, we would include a directive for `@app.route(‘/about’)`

Variable Routing

- One powerful way to use routing is to use ***variable routing***, which allows you to have variables within the URL string
  - Variables are announced in the directive by surrounding them with angular brackets:

```
@app.route(‘/myName/<name>’)
```

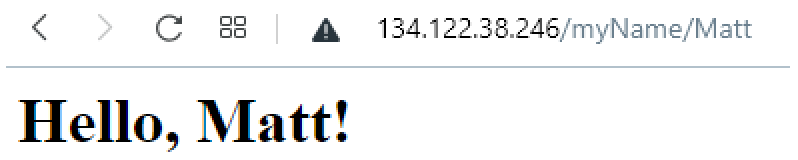
- We can then capture the variable in our function by including it as an input parameter:

```
def namePage(name):
      return name
```

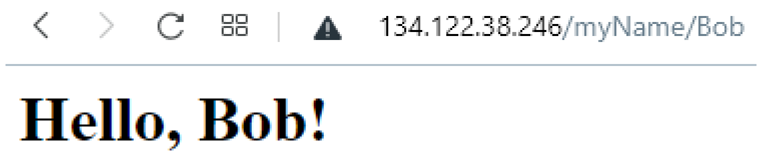
Why Use Variable Routing?

- Variable routing allows us to use a single function to handle multiple possible inputs
- A traditional use for this is for handling blog pages, for example:

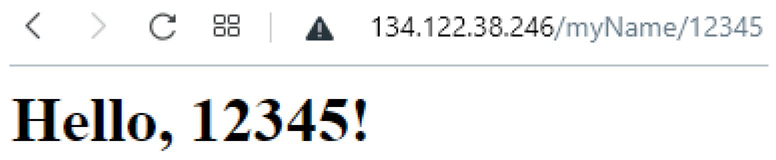
  www.myblog.com/article/5438
- We could then use the variable number (5438) to fetch the appropriate article text


134.122.38.246/myName/Matt

Hello, Matt!


134.122.38.246/myName/Bob

Hello, Bob!


134.122.38.246/myName/12345

Hello, 12345!

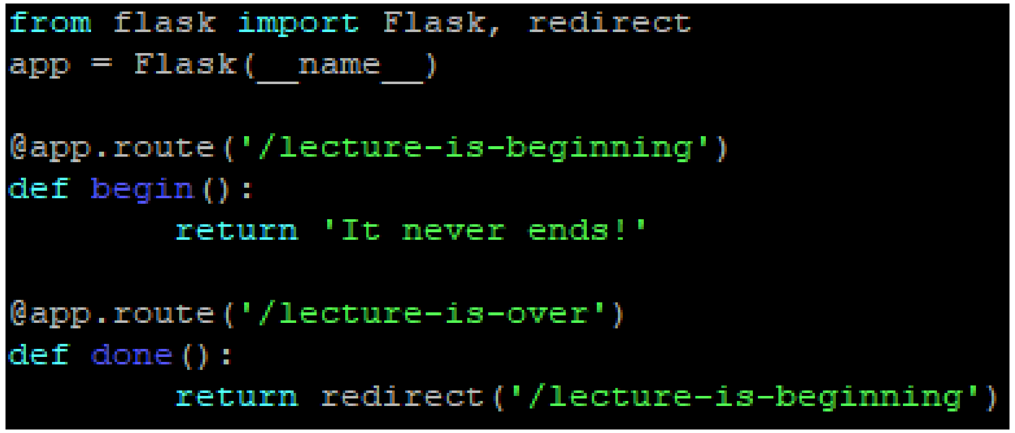
Variable Routing with Types

- By default, variables in a route are returned as strings
- If you want to handle them numerically, you can either cast the input to an integer, or you can use a specifier in the routing directive:

```
        @app.route(‘/myNumber/<int:number>’)
```

- This has two effects: first, it will ignore a variable that contains non-numbers (it will return 404 – not found), and second the number variable will be returned as an integer
- You can specify string, int, float, path, and UUID types

Handling Redirects



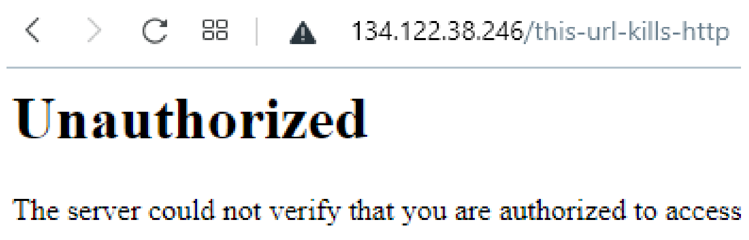
```
from flask import Flask, redirect
app = Flask(__name__)

@app.route('/lecture-is-beginning')
def begin():
        return 'It never ends!'

@app.route('/lecture-is-over')
def done():
        return redirect('/lecture-is-beginning')
```

- Flask includes a function called ***redirect***, which when returned will redirect the user to the specified route URL
- Import it from the flask package
- Useful when you need to force the user over to another page, for example redirecting a user to the login page

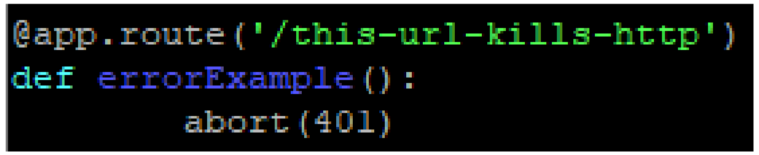
Returning Errors



134.122.38.246/this-url-kills-http

Unauthorized

The server could not verify that you are authorized to access

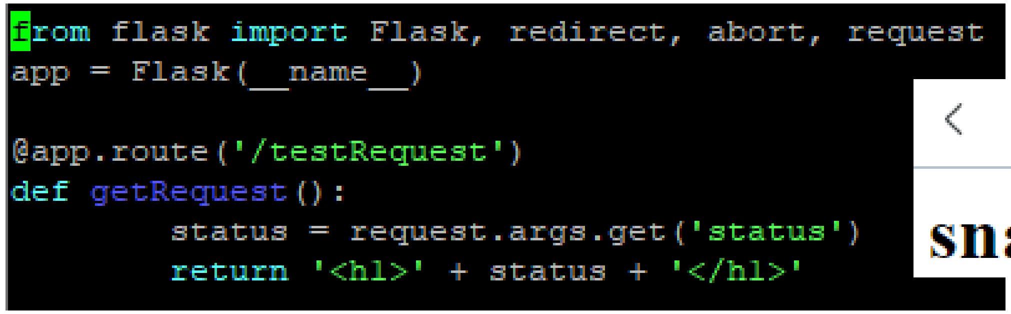


```
@app.route('/this-url-kills-http')
def errorExample():
        abort(401)
```

- The ***abort*** function can also be imported from the flask package
- It does not need to be returned, but will pass along a provided HTTP error code to the browser (i.e. 404, 500, 401)
- Useful if you want to throw a proper error when something goes wrong
- By default uses the classic plain error pages

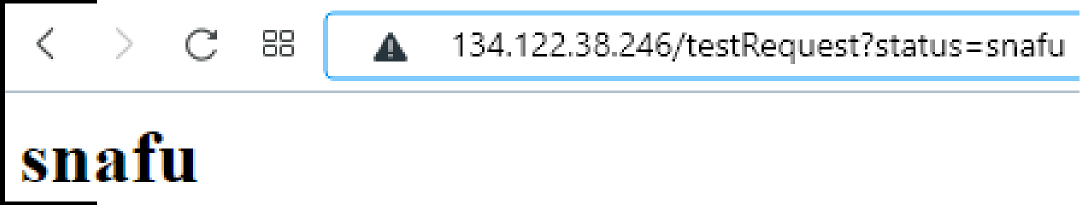
Handling GET Request Parameters

- The **request** function can be used to pull parameters out of a GET query string
- Use `request.args.get(variableName)` to fetch a specific named parameter from the GET request
- Returns *None* type (like null) if parameter does not exist in request – always catch this case



```
from flask import Flask, redirect, abort, request
app = Flask(__name__)

@app.route('/testRequest')
def getRequest():
        status = request.args.get('status')
        return '<h1>' + status + '</h1>'
```



```
134.122.38.246/testRequest?status=snafu
```

snafu

Dealing with Request Types

- We can specify what types of requests we want a specific route to ‘catch’ by specifying a **methods** list in the route directive:

```
@app.route(‘/myForm’, methods=[‘GET’, ‘POST’])
```

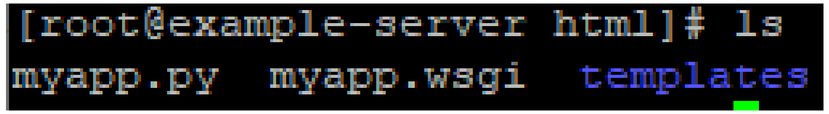
- You can have two identical route directives with different methods, allowing you to specify which function to use based on the type of request
- The `request.method` can also be used for determining the type of request that has been made

Using Templates

- ***Templates*** in Flask are HTML pages with special markup for inserting data or nesting templates together to create dynamic web pages
- These are handled using the Jinja2 templating engine, which is actually modelled after the templating engine in the Django framework
- Templates allow us to build our website in a modular fashion, creating dynamic “chunks” of HTML that can then be pieced together

Setting up the first Template

- By default, Flask looks for templates in a directory called ***templates***, in the same folder as your application Python file

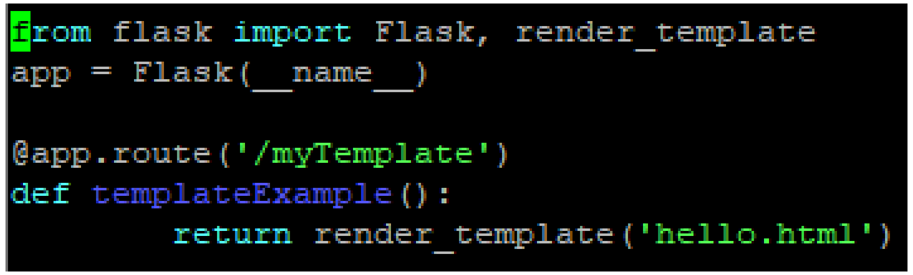


```
[root@example-server html]# ls
myapp.py  myapp.wsgi  templates
```

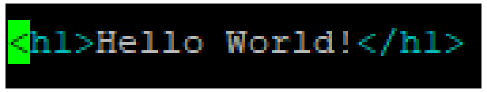
- Templates are stored as regular HTML files using the .html file extension
- We can then import the ***render_template*** function from flask, and pass it the name of the template to render:

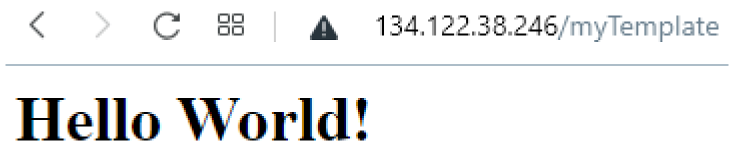
```
render_template('hello.html')
```

Minimum Working Example


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

@app.route('/myTemplate')
def templateExample():
        return render_template('hello.html')
```


```
<h1>Hello World!</h1>
```


134.122.38.246/myTemplate

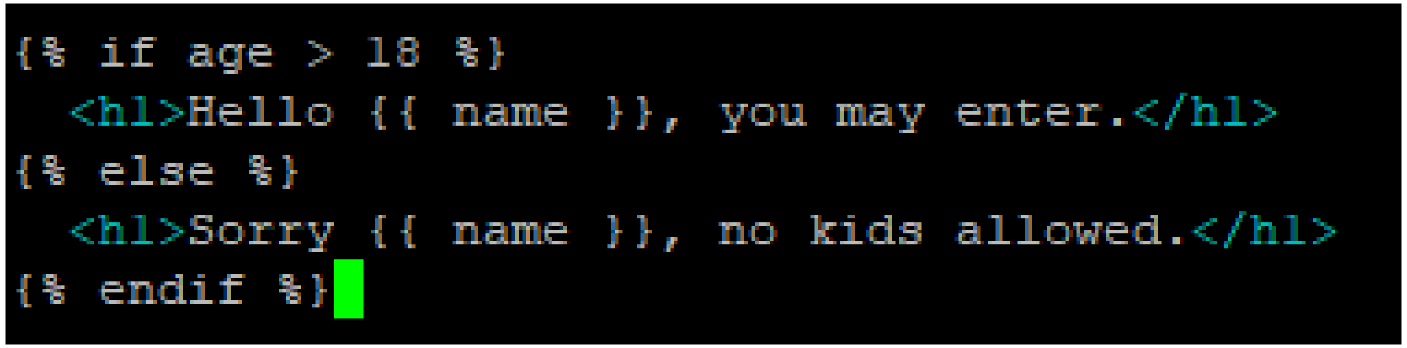
Hello World!

- Here’s some example code showing the rendering of a very simple template
- Templates can be as small or as large as you want – they might be entire HTML pages, or they could be individual elements
- The output of `render_template` is directly returned

Making Templates more Dynamic

- We can pass data to, and even insert logic within our template files – this is often very useful, as it allows us to make dynamic pages without forcing us to rewrite a lot of HTML
- Double braces **{{ }}** will print a variable at that location. The variable will automatically be escaped to prevent injection (HTML, JS, etc will not work when inserted)
- Percentile braces **{% %}** are used to denote logic within the template file

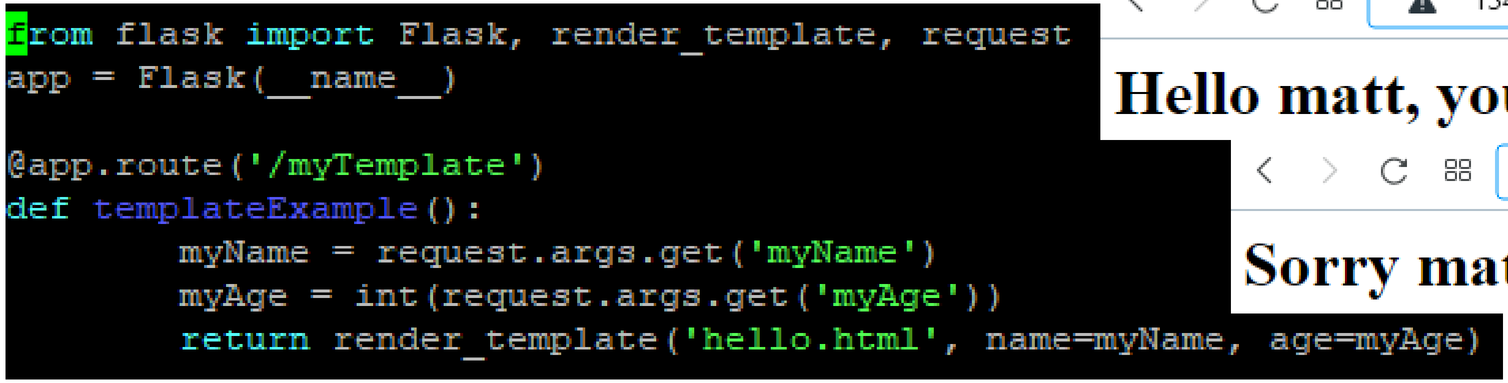
A Simple Data Template



```
{% if age > 18 %}
  <h1>Hello {{ name }}, you may enter.</h1>
{% else %}
  <h1>Sorry {{ name }}, no kids allowed.</h1>
{% endif %}
```

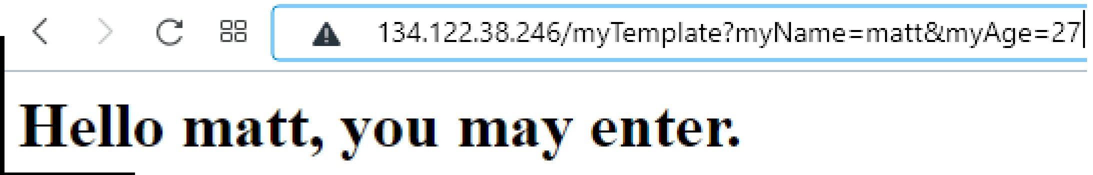
- Two variables: *age* and *name*
- Template will return one of two <h1> header objects depending on the value of *age* – note the **if, else, endif** structure that is used
- The template also prints the *name* in both cases

Passing Data into Templates



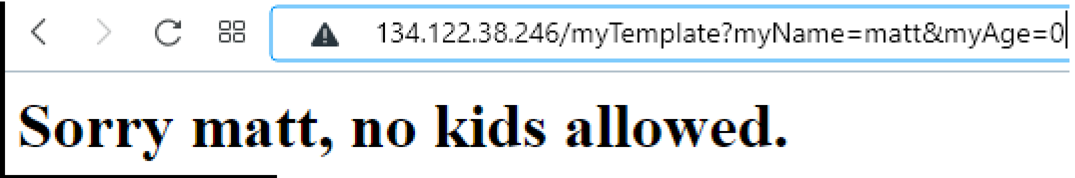
```
from flask import Flask, render_template, request
app = Flask(__name__)

@app.route('/myTemplate')
def templateExample():
        myName = request.args.get('myName')
        myAge = int(request.args.get('myAge'))
        return render_template('hello.html', name=myName, age=myAge)
```



134.122.38.246/myTemplate?myName=matt&myAge=27

Hello matt, you may enter.



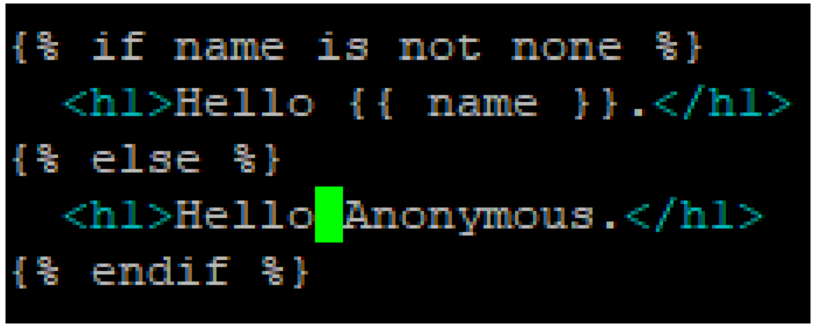
134.122.38.246/myTemplate?myName=matt&myAge=0

Sorry matt, no kids allowed.

- Templates will throw errors if they expect data but don’t receive it – to do so, we’ll need to pass it in as part of the render_template function
- When we pass the data in, we need to assign it to the variables as they are named in the template file

A Handy Trick

- Templates will return a 500 error if you try to print out or otherwise use a variable that doesn't exist
- We can use an if/else/endif to check if a variable exists by testing against the *none* value
- In this case, we print out two different output lines, depending on whether the *name* variable is set or not



```
{% if name is not none %}
  <h1>Hello {{ name }}.</h1>
{% else %}
  <h1>Hello Anonymous.</h1>
{% endif %}
```

Next Lecture

- Next lecture, we’ll continue talking about templates in the form of ***template inheritance**,* which will allow us to combine and “stack” our templates together
- We’ll look at how to handle POST request data in Flask, and when to use GET, POST, or routing variables depending on the situation
- We’ll also learn how to store session data, which will allow us to maintain user-specific data between requests → finally, we can maintain a state with our application

Any Questions?