FLASK TEMPLATES

Introduction

When you navigate through a website you may notice that many of the pages on the site have a similar look and feel. This aspect of a website can be achieved with the use of *templates*. In this lesson the term *template* refers to an HTML file that can represent multiple web pages with the same structure and functionality.

We will be using the Flask framework for our application in this lesson. Flask uses the Jinja2 template engine to render HTML files that include application variables and control structures. The Jinja2 template engine is a powerful tool that supports an organized and growth oriented application.

In this lesson we will look at:

- How to organize our site file structure
- Use our application data with our templates
- Leverage control structures within our templates
- Share common elements across many templates

The application we will be building in the following exercises is a cookbook site that consists of a main page and individual recipe pages. Currently our app consists of 2 routes that return HTML strings for the browser to display. Explore the application to begin your path to learning templates!

Instructions

1.
Let's explore the cookbook Flask application. Right now our app has two route functions, each returning a basic HTML string. Our routes are:

1. The `index` route function for a title page and recipe list
2. The `recipe` route function for details of a single recipe

Looking at the app's import statement we can see our app contains a file called **helper.py** which includes 4 dictionaries: `names`, `descriptions`, `ingredients` and `instructions`. Using a recipe `id` as a key, we can access the data in each dictionary.

Here is a description of our data given a specific recipe `id`:

- `recipes[id]` is a string and is the display name of the recipe.
- `descriptions[id]` is a string and is a brief description of the recipe.
- `ingredients[id]` is a list of ingredient strings.
- `instructions[id]` is a dictionary whose key correspond to the step number and the values are the step instructions.

We'll be using these variables from **helper.py** throughout the lesson. To access the correct recipe data we need to refresh our knowledge of passing variables between routes through the URL. Run the code when you're ready!

2.

The `index` route HTML there is a hyperlink where the URL points to the recipe route and the recipe `id` 1.

```
<a href="/recipe/1">Fried Egg</a>
```

The `id` is used as the key for all the data in **helper.py**, but we need to be able access it first.

In the recipe route, add the variable `id` as an integer to receive the correct recipe data.

- In the route decorator, add the variable to the URL making sure to specify the data type as an `int`.
- In the route function, add the variable as a parameter.

Run the code and navigate to the recipe page using the link. You can see the recipe data from **helper.py** is output on separate lines in the web browser. If you look back at the recipe route, each variable is contained inside the HTML string. The list and dictionary are string representations of the variables.

Proceed to the next exercise to begin working with templates!

Hint

Add variable `id` as an integer to the end of the URL string. Use the following syntax:

```
/recipe/<int:url_var>
```

Add the variable `id` as a parameter to the route function. Use the following syntax:

```
def recipe(url_var):
```

app.py

```
from flask import Flask
from helper import recipes, descriptions, ingredients, instructions

app = Flask(__name__)

@app.route('/')
def index():
  return '''
    <!DOCTYPE html>
    <html>
      <body>
        <h1>Cooking By Myself</h1>
        <p>Welcome to my cookbook. These are recipes I like.</p>
        <a href="/recipe/1">Fried Egg</a>
      </body>
    </html>
    '''

#### Add the variable `id` to the route URL
#### and make it the sole function parameter
@app.route("/recipe/<int:id>")
def recipe(id):
  return '''
    <!DOCTYPE html>
    <html>
      <body>
        <a href="/">Back To Recipe List</a>
        <p>names[id] = ''' + recipes[id] + '''</p>
        <p>descriptions[id] = ''' + descriptions[id] + '''</p>
        <p>ingredients[id] = ''' + str(ingredients[id]) + '''</p>
        <p>instructions[id] = ''' + str(instructions[id]) + '''</p>
      </body>
    </html>
    '''
```

Rendering Templates

Having routes return full web pages as strings is not a realistic way to build our site. Containing our HTML in files is the standard and more organized approach to structuring our web app.

To work with files, which we will call templates, we use the Flask function `render_template()`. Used in the return statement, this function takes a template file name as an argument and returns the content to send to the client. It uses the Jinja2 template engine to generate HTML using the template file as blueprint.

```
return render_template("my_template.html")
```

To use `render_template()` in our routes we need to import it from the `flask`. A simple app with an index route would look like this.

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")
```

Inside the application directory `render_template()` looks for templates inside a directory called **templates**. All template files should be kept inside this directory. To view the application file structure in this exercise click the folder icon in the top left corner of the code editor.

Instructions

1.
First thing to do is familiarize yourself with the directory structure. You can open the file picker by clicking the folder icon on the top left of the code editor.

Inside the templates directory open **index.html**. You'll notice the HTML from the previous exercise is now in the file. Run the code when you're ready.

2.
In the `index` route replace the empty string with `render_template()` to generate HTML from the **index.html** template.

Hint
Return `render_template()` with `"index.html"` as the parameter. Use the following syntax:

```
return render_template("my_template.html")
```

3.
Along with **index.html** there is a recipe template called **fried_egg.html**. In the `recipe` route's return statement replace the empty string with `render_template()` to generate the HTML from **fried_egg.html**.

When you run the code you can navigate the site between the 2 routes. The index HTML is the same as in the previous exercise except the HTML is contained in a file. The data in **fried_egg.html** has been hard coded in the file for this exercise.

Move on to the next exercise to see how we can start using variables to bring our data dynamically to our files.

Hint
Return `render_template()` with `"fried_egg.html"` as the parameter. Use the following syntax:

```
return render_template("my_template.html")
```

app.py

```
from flask import Flask, render_template
from helper import recipes, descriptions, ingredients, instructions

app = Flask(__name__)

@app.route('/')
def index():

  #### Return a rendered index.html file
  return render_template("index.html")

@app.route("/recipe/<int:id>")
def recipe(id):

  #### Return a rendered fried_egg.html file
  return render_template("fried_egg.html")
```

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Cooking By Myself</h1>
    <p>Welcome to my cookbook. These are recipes I like.</p>
    <a href="/recipe/1">Fried Egg</a>
  </body>
</html>
```

Template Variables

Instead of having an HTML file for each recipe, it would be a lot easier having one file for many recipes. Being able to pass data to template files is how we can begin to accomplish this goal.

After the filename argument in `render_template()` we can add keyword arguments to be used as variables within the template. These variables are assigned values or app data we would like to access within the template.

```
flask_variable = "Text for my template"

render_template("my_template.html",
                template_variable=flask_variable)
```

In this example we're assigning the value of `flask_variable` to `template_variable` which can be used in **my_template.html**. To add more than one variable separate each assignment with a comma.

```
render_template("my_template.html",
                template_var1="A string!",
                template_var2=100)
```

Our template now has access to the variables `template_var1` and `template_var2` which hold a string and an integer respectively.

App data can be passed as literal values or the values stored inside variables. We can pass strings, integers, lists, dictionaries or any other objects to our templates.

It is possible to give keyword arguments and the assignment variables the same name `var1=var1`. For these exercises all variables from our flask app will start with `flask` and all template variables will start with `template`.

To access the variables in our templates we need to use the expression delimiter: `{{ }}`.

```
{{ template_variable }}
```

The delimiter can be used inline with text and alongside HTML elements.

```
<h1>My Heading: {{ template_variable }}</h1>
```

Certain operations can be performed inside expression delimiters `{{ }}`.

With `template_variable = 20`.

```
<p>Template number plus ten: {{ template_variable + 10 }}</p>

OUTPUT
Template number plus ten: 30
```

List and dictionary elements can be accessed individually inside the expression delimiters `{{ }}`.

With `template_list = ["A", "B", "C"]`

```
<p>Element at index 1: {{ template_list[1] }}</p>
```

```
OUTPUT
Element at index 1: B
```

Instructions

1.
We've replaced the **fried_egg.html** template with a new template, **recipe.html**. This starts the process of using one template for many web pages.

In **recipe.html** find the first delimiter `{{ }}` and note the variable name `template_recipe`. The string inside this variable will be used as the header for the page.

Go to **app.py** and add a keyword argument to the `recipe` route's `render_template()` function. Assign the value of `recipes[id]` to `template_recipe`.

Hint
To assign a flask variable to a template variable, add a keyword argument to `render_template()`.

```
render_template("my_template.html", template_variable=flask_variable)
```

2.
Good job!!. Let's add 2 more variable assignments to `render_template()`.

- Assign `descriptions[id]` to `template_description`.
- Assign `ingredients[id]` to `template_ingredients`.

Hint
To assign multiple variable assignments, add more keyword arguments to `render_template`.

```
render_template("my_template.html", template_variable=flask_variable,
template_variable2=flask_variable2,
template_variable3=flask_variable3)
```

3.
Great!! Navigate to the recipe page in the web browser and you'll see the description is now showing, but the ingredients list is still empty.

Go to **recipe.html** and populate the ingredients list using explicit indexes for `template_ingredients`. Add one element per `<li>` statement in the template. Be sure to start your index at 0 and list all three ingredients from the list.

Don't forget to use the delimiters `{{ }}`.

Once you run the code you may start to see that even though there's currently one recipe, **recipe.html** can be used to display multiple recipes.

Hint

Using the list `template_ingredients`, access each of the three elements by their index. Put each element inside the `<li></li>` using the following syntax.

```
{{ list_variable[0] }}
```

recipe.html

```
<!DOCTYPE html>
<html>
  <body>
    <a href="/">Back To Recipe List</a>
    <h1>{{ template_recipe }}</h1>
    <p>{{ template_description }}</p>
    <h3>Ingredients</h3>
    <ul>
      <!-- Ingredients list elements
      should fill the <li> tags -->
      <li>{{ template_ingredients[0] }}</li>
      <li>{{ template_ingredients[1] }}</li>
      <li>{{ template_ingredients[2] }}</li>
    </ul>
  </body>
</html>
```

app.py

```
from flask import Flask, render_template
from helper import recipes, descriptions, ingredients

app = Flask(__name__)

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

@app.route("/recipe/<int:id>")
def recipe(id):
  #### Add template variables as
```

```
  #### variable assignment arguments
  return render_template("recipe.html", template_recipe=recipes[id], template_des
cription=descriptions[id], template_ingredients=ingredients[id])
```

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Cooking By Myself</h1>
    <p>Welcome to my cookbook. These are recipes I like.</p>
    <a href="/recipe/1">Fried Egg</a>
  </body>
</html>
```

Variable Filters

Now that we can use variables in our templates, let's look at different ways we can perform actions on them.

Filters are used by the template engine to act on template variables. To use them simply follow the variable with the filter name inside the delimiter and separate them with the `|` character.

```
{{ variable | filter_name }}
```

The character `|` separating the variable and the filter is called a *pipe* or *vertical bar*.

The filter `title` acts on a string variable and capitalizes the first letter in every word. This is good for using as formatting on heading strings. Given the variable assignment `template_heading = "my very interesting website"`.

```
{{ template_heading |  title }}

OUTPUT
My Very Interesting Website
```

Notice that the first letter of every word is now capitalized.

Filters can also take arguments. The `default` filter will output the text in its argument when a variable isn't passed to the template. Consider if `no_template_variable` is missing from the `render_template()` arguments.

```
{{ no_template_variable | default("I am not from a variable.") }}

OUTPUT
I am not from a variable.
```

The `default` filter does not work on empty strings `""` or `None` values. We will look at this scenario in the next exercise.

While filters perform more complex functions than simple operators, they are still small, focused actions. Here is a list of commonly applied filters and their descriptions. More information can be found in the Jinja2 documentation

- `title`: Capitalizes the first letter of each word in a string, known as titlecase
- `capitalize`: Capitalizes the first character of a string, such as in a sentence
- `lower`/`uppercase`: Makes **all** the characters in a string lowercase/uppercase
- `int`/`float`: Changes any number variable to an integer/float
- `default`: Defines a default string if the variable is not defined
- `length`: Calculates the length of a string, list or dictionary variable
- `dictsort`: Sorts a dictionary by its keys

Instructions

1.

Since the recipe names are stored in all lowercase characters you need to capitalize the first letter of each word for the recipe heading.

In **recipe.html** apply a filter to `template_recipe` that capitalizes the first letter of each word.

Hint

The `title` filter can be applied using the following syntax.

```
{{ template_variable | filter }}
```

2.

The description argument in **app.py** has been removed. Use a filter to make sure a proper description is displayed.

Use a filter on `template_description` that will display text even if the variable isn't passed through `render_template`.

If you want to utilize the recipe name in your message, you can pass the filter a string such as:

```
"A " + template_recipe + " recipe."
```

Once you run the code go to **app.py** and put `template_description` variable assignment back in the `render_template()` function of the recipe route.

Hint

The `default` filter and its argument can be applied using the following syntax.

```
{{ template_variable | filter(some_argument) }}
```

3.

In the ingredients heading it might be important to display the number of ingredients. This could help a prospective chef know how complicated a recipe is.

In **recipe.html** create a separate delimiter block at the end of, but still inside, the ingredients heading. Inside this block display the length of `template_ingredients`.

For example the heading output can look like this: **Ingredients - 3**

Hint

Be sure to create a separate delimiter `{{ }}` in the Ingredients heading.

The `length` filter can be applied using the following syntax.

```
{{ template_variable | filter }}
```

4.

In the instructions section the dictionary has unsorted keys.

Use a filter to make sure the dictionary is output in sorted order.

Hint

The `dictsort` filter can be applied using the following syntax.

```
{{ template_variable | filter }}
```

recipe.html

```
<!DOCTYPE html>
<html>
  <body>
    <a href="/">Back To Recipe List</a>
    <!-- Make template_recipe title case -->
```

```
    <h1>{{ template_recipe | title }}</h1>
    <!-- Ensure a default description -->
    <p>{{ template_description | default("A " + template_recipe + " recipe.") }}<
/p>
    <!-- Output number of ingredients -->
    <h3>Ingredients {{ template_ingredients | length }}</h3>
    <ul>
      <li>{{ template_ingredients[0] }}</li>
      <li>{{ template_ingredients[1] }}</li>
      <li>{{ template_ingredients[2] }}</li>
    </ul>
    <h3>Instructions</h3>
    <ol>
      <!-- Ensure sorted instruction dictionary -->
      <li>{{ template_instructions | dictsort }}</li>
    </ol>
  </body>
</html>
```

app.py

```
from flask import Flask, render_template
from helper import recipes, descriptions, ingredients, instructions

app = Flask(__name__)

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

@app.route("/recipe/<int:id>")
def recipe(id):
  return render_template("recipe.html", template_recipe=recipes[id], template_ing
redients=ingredients[id], template_instructions=instructions[id])
```

Cooking By Myself

Welcome to my cookbook. These are recipes I like.

Fried Egg

If Statements

Including *conditionals* such as if and if/else statements in our templates allows us to control how data is handled.

Let's say we have a string variable passed to our template. When the variable contains an empty string will you want to output it or will you want to output another string? Remember the `default` filter doesn't work in this situation so an if statement is needed.

Using if statements in a template happens inside a statement delimiter block: `{% %}`.

```
{% if condition %}
  <p>This text will output if condition is True</p>
{% endif %}
```

Notice the `{% endif %}` delimiter is necessary to close the `if` statement.

The *condition* can include a variable that is tested using standard comparison operators, `<`, `>`, `<=`, `>=`, `==`, `!=`.

```
{% if template_variable == "Hello" %}
  <p>{{template_variable}}, World!</p>
{% endif %}
```

While inside statement delimiters `{% %}` we can access variables without using the usual expression delimiter `{{ }}`.

Variables can also be tested on their own. A variable defined as `None` or `False` or equates to `0` or contains an empty sequence such as `""` or `[]` will test as `False`.

The `{% else %}` and `{% elif %}` delimiters can be included to create multi-branch if statements.

Given the assignment `template_number = 20`.

```
{% if template_number < 20 %}
  <p>{{ template_number }} is less than 20.</p>
{% elif template_number > 20 %}
   <p>{{ template_number }} is greater than 20.</p>
{% else %}
   <p>{{ template_number }} is equal to 20.</p>
{% endif %}

OUTPUT
20 is equal to 20.
```

As expected the `{% else %}` branch is the one that is followed.

Instructions

1.
Let's note the few changes made to our app:

- The recipe list has expanded to 2 recipes.
- The new recipe has a description of `None` which doesn't engage the `default` filter.
- The list of ingredients is set up for 3 ingredients, but the new recipe only has 2.

With these points in mind run the code to explore the app and move on when you are ready.

Hint
Move to the next instruction.

2.

In **recipe.html** we have 2 paragraphs for our description. We want one paragraph to output `template_description` and the other to output a default statement.

Start by surrounding BOTH paragraphs with an if statement that tests the variable on its own.

Remember testing a variable on its own will be `False` if it is an empty string or is `NoneType`.

When you run the code you will see that the recipe with a description displays both paragraphs while the recipe with no description displays nothing.

Hint

Use `template_description` as the *condition* using the following syntax.

```
{% if condition %}
  <p> ... </p>

  <p> ... </p>
{% endif %}
```

3.

Now separate the two paragraphs with an else statement.

This will now display only one message per recipe: `template_description` or a default message.

Hint

Insert `{% else %}` between the two paragraphs.

4.

Now let's use an if statement to fix the blank 3rd bullet in the list when displaying the new recipe.

In the **Ingredients** section surround the 3rd ingredient `<li>` line with an if statement. For the condition, test that the `length` of `template_ingredients` is equal, `==`, to 3.

After completing this think about how limiting this task is. What if we have an ingredients list greater than 3? What if there's only one ingredient? Run the code and move to the next exercise to find the answer.

Hint

Use `template_ingredients | length == 3` as the *condition* using the following syntax.

```
{% if condition %}
  <li> ... </li>
{% endif %}
```

recipe.html

```
<!DOCTYPE html>
<html>
  <body>
    <a href="/">Back To Recipe List</a>
    <h1>{{ template_recipe | title }}</h1>
    <!-- Insert description if statement here -->
    {% if template_description %}
      <p>{{ template_description }}</p>
      <!-- Include else here -->
      {% else %}
        <p>A {{ template_recipe }} recipe.</p>
        <!-- Be sure to close with an endif block -->
    {% endif %}
    <h3>Ingredients - {{ template_ingredients | length}}</h3>
    <ul>
      <li>{{ template_ingredients[0] }}</li>
      <li>{{ template_ingredients[1] }}</li>
      <!-- Insert ingredient if statement -->
      {% if template_ingredients | length == 3 %}
        <li>{{ template_ingredients[2] }}</li>
        <!-- Be sure to close with an endif block -->
      {% endif %}
    </ul>
    <h3>Instructions</h3>
    <ol>
      <li>{{ template_instructions | dictsort }}</li>
    </ol>
  </body>
</html>
```

app.py

```
from flask import Flask, render_template
from helper import recipes, descriptions, ingredients, instructions

app = Flask(__name__)

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

@app.route("/recipe/<int:id>")
def recipe(id):
  return render_template("recipe.html", template_recipe=recipes[id], template_des
cription=descriptions[id], template_ingredients=ingredients[id], template_instruc
tions=instructions[id])
```

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Cooking By Myself</h1>
    <p>Welcome to my cookbook. These are recipes I like.</p>
    <p><a href="/recipe/1">Fried Egg</a></p>
    <p><a href="/recipe/2">Buttered Toast</a></p>
  </body>
</html>
```

helper.py

```
recipes = {1: "fried egg", 2: "buttered toast"}
descriptions = {1: 'Egg fried in butter', 2: None}
ingredients = {1: ['1 pad of butter', '1 Egg', 'A pinch of salt'], 2: ['1 pad of
salted butter', '1 slice of bread']}
instructions = {1: {'Step 2': 'Crack the egg into the buttered pan', 'Step 5': 'S
erve egg after about a minute and a half', 'Step 1': 'Melt butter in pan over med
ium-
low heat', 'Step 4': 'Flip egg after about a minute and a half', 'Step 3': 'Sprin
kle the pinch of salt onto cooking egg',},
```

```
              2: {'Step 3': 'Put the pad of butter on the toasted bread', 'Step
4': 'After a minute spread the melted butter onto the bread', 'Step 1': 'Put the
bread in the toaster', 'Step 2': 'Take the toast out of the toaster'}}
```

For Loops

Repetitive tasks are standard in most computer applications and template rendering is no different. Creating lists, tables or a group of images are all repetitive tasks that can be solved using *for loops*.

Using the same statement delimiter block as if statements `{% %}`, for loops step through a range of numbers, lists and dictionaries.

The following code will create an ordered list where each line will output the index of the sequence:

```
<ol>
{% for x in range(3) %}
  <li>{{ x }}</li>
{% endfor%}
</ol>

OUTPUT
1. 0
2. 1
3. 2
```

The syntax is similar to a Python for loop where we define a loop variable `x` to step through a series of numbers using `range(3)`. The local loop variable can be used inside our loop with the expression delimiter `{{x}}`.

Similar to the if statements we need to close the loop with an `{%endfor%}` block.

The following are a few more applications of a for loop.

Iterate through a list variable:

```
{% for element in template_list %}
```

Iterate through a string:

```
{% for char_in_string in “Hello!” %}
```

Iterate through the keys of a dictionary variable:

```
{% for key in template_dict %}
```

Iterate through keys AND values of a dictionary with `items()`:

```
{% for key, value in template_dict.items() %}
```

Using the filter `dictsort` in a loop outputs the key/value pairs just like `items()`

Instructions

1.

Loops can help create lists based on the length of the data we have to work with.

In **recipe.html** the ingredients section now has one `<li>` element. Apply a for loop to create an HTML list based on the Python list `template_ingredients`.

Note the variable name `ingredient` to be used as the loop variable.

Hint

Apply the for loop using `ingredient` as the local loop variable and `template_ingredients` as the *iterable*. Use the following syntax.

```
{% for local_var in iterable %}
<li> ... </li>
{% endfor %}
```

2.

In the instructions section, remember that we are using the `dictsort` filter on `template_instructions`. When used in a for loop `dictsort` will give you access to both the key and value of the dictionary.

Using the `key` and `instruction` variables create a simple string inside the `<p>` tag. The output should resemble something like this:

Step 1: Put the bread in the toaster

Hint

Use two delimiters next to each other with `key` and `instruction` in each.

```
<p>{{ delimiter 1 }}: {{ delimiter 2 }}</p>
```

3.

Now let's move to **index.html** and support the expansion of our recipe list. The dictionary of ids and recipes names is now accessible in the template through the variable `template_recipes`.

Create a `for` loop around the hyperlink and iterate through `template_recipes.items()`. Name the key and value loop variables `id` and `name` respectively.

Hint

Create `id` and `name` as the local loop variables and use `template_recipes.items()` as the iterable. Use the following syntax.

```
{% for local_var1, local_var2 in iterable %}
  <p><a href="/recipe/"></a></p>
{% endfor %}
```

4.

Use the `id` and `name` variables to complete the hyperlink URL and the display string.

Once the code is run you can now add recipes to the application data and automatically get a link on the index page that connects to a custom recipe page.

recipe.html

```
<!DOCTYPE html>
<html>
  <body>
    <a href="/">Back To Recipe List</a>
    <h1>{{ template_recipe | title }}</h1>
    {% if template_description %}
      <p>{{ template_description }}</p>
    {%else%}
      <p>A {{ template_recipe }} recipe.</p>
    {% endif %}
    <h3>Ingredients - {{ template_ingredients | length}}</h3>
    <ul>
      <!-- Implement a for loop to iterate through
      `template_ingredients`-->
      {% for ingredient in template_ingredients %}
        <li>{{ ingredient }}</li>
      {% endfor%}
    </ul>
    <h3>Instructions</h3>
    <ul>
    {% for key, instruction in template_instructions|dictsort %}
      <!-- Add the correct dictionary element to list
      the instructions -->
      <p>{{ key }}: {{ instruction }}</p>
    {% endfor %}
    </ul>
  </body>
</html>
```

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Cooking By Myself</h1>
    <p>Welcome to my cookbook. These are recipes I like.</p>
    <!-- Implement a for loop using `template_recipes`-->
    {% for id, name in template_recipes.items() %}
      <p><a href="/recipe/{{ id }}">{{ name }}</a></p>
```

```
    {% endfor %}
  </body>
</html>
```

app.py

```
from flask import Flask, render_template
from helper import recipes, descriptions, ingredients, instructions

app = Flask(__name__)

@app.route('/')
def index():
  return render_template("index.html", template_recipes=recipes)

@app.route("/recipe/<int:id>")
def recipe(id):
  return render_template("recipe.html", template_recipe=recipes[id], template_des
cription=descriptions[id], template_ingredients=ingredients[id], template_instruc
tions=instructions[id])
```

Inheritance

If you go to any website you may notice certain elements exist across different web pages.

The navigation bar is a good example of a common page element. This is the banner at the top of most sites that has links to different pages. No matter what page you're on the navigation bar is there.

Imagine having separate files for each web page and wanting to make a change to the navigation bar. Would you have to change the content of every template of the site? No, that would take too long.

To solve this problem template files are used to share content across multiple templates. The simplest case is a file that includes the top portion of the templates through the `<body>` tag and then the closing `</body>` and `</html>` tags. Jinja2 statement delimiters are then used to identify the area of the template where specific content will be substituted in.

```
<html>
  <head>
    <title>MY WEBSITE</title>
  </head>
  <body>
  {% block content %}{% endblock %}
  </body>
</html>
```

For this exercise we will name the above template **base.html**.

To inherit this content in another template we will use the `extends` statement. The code to be substituted should then be surrounded by `{%block content%}` and `{%endblock%}`. All together this looks like the following template:

```
{% extends "base.html"  %}

{% block content %}
    <p>This is my paragraph for this page.</p>
{% endblock %}
```

This template is named **index.html**.

When a route returns `render_template("index.html")` the rendered page will have this content.

```
<html>
  <head>
    <title>MY WEBSITE</title>
```

```
  </head>
  <body>
    <p>This is my paragraph for this page.</p>
  </body>
</html>
```

Instructions

1.
Look in **index.html** and **recipe.html** and identify the common HTML at the top and bottom of the templates.

Copy the common HTML from one of those files and paste it into the blank **base.html** file. Don't forget to include the closing tags, `</body>` and `</html>`.

Hint
The common HTML is:

```
<!DOCTYPE html>
<html>
  <body>

  </body>
</html>
```

2.
Now in **base.html** identify where you want to put `{%block content%}{%endblock%}`. It is your choice whether `{%block content%}` and `{%endblock%}` are on the same line or separate lines. Both work as long as there is nothing in between.

Hint
Inside the `<body>` tags insert these two delimiters.

```
{%block content%}{%endblock%}
```

3.
Now go to **index.html** and:

- Delete the duplicate text
- Extend **base.html** at the top
- Surround the **index.html** specific template with an `{%block content%}` and `{%endblock%}`

Remember you start the block with `{%block content%}` and close it with `{%endblock%}` with the HTML in between.

Hint
Make sure to delete the following from **index.html**.

```
<!DOCTYPE html>
<html>
  <body>

  </body>
</html>
```

At the top of the file insert the following code.

```
{% extends "base.html" %}
{% block content %}
```

At the bottom of the file insert the following code.

```
{% endblock %}
```

4.

Do the same in **recipe.html**. Be sure to extend **base.html** and replace the duplicate text with the inheritance blocks.

Hint

Make sure to delete the following from **recipe.html**.

```
<!DOCTYPE html>
<html>
  <body>

  </body>
</html>
```

At the top of the file insert the following code.

```
{% extends "base.html" %}
{% block content %}
```

At the bottom of the file insert the following code.

```
{% endblock %}
```

base.html

```
{% extends "base.html" %}
{% block content %}
    <a href="/">Back To Recipe List</a>
    <h1>{{ template_recipe | title }}</h1>

    {% if template_description %}
      <p>{{ template_description }}</p>
    {%else%}
      <p>A {{ template_recipe }} recipe.</p>
    {% endif %}
```

```
    <h3>Ingredients - {{ template_ingredients | length}}</h3>
    <ul>
    {% for ingredient in template_ingredients %}
      <li>{{ ingredient }}</li>
    {% endfor %}
    </ul>

    <h3>Instructions</h3>
    <ul>
    {% for key, instruction in template_instructions|dictsort %}
      <li>{{ instruction }}</li>
    {% endfor %}
    </ul>
  {% endblock %}
```

index.html

```
{% extends "base.html" %}
{% block content %}
    <h1>Cooking By Myself</h1>
    <p>Welcome to my cookbook. These are recipes I like.</p>
    {% for id, name in template_recipes.items() %}
      <p><a href="/recipe/{{ id }}">{{ name | title }}</a></p>
    {% endfor %}
{% endblock %}
```

recipe.html

```
{% extends "base.html" %}
{% block content %}
    <a href="/">Back To Recipe List</a>
    <h1>{{ template_recipe | title }}</h1>

    {% if template_description %}
      <p>{{ template_description }}</p>
    {%else%}
      <p>A {{ template_recipe }} recipe.</p>
    {% endif %}

    <h3>Ingredients - {{ template_ingredients | length}}</h3>
    <ul>
```

```
  {% for ingredient in template_ingredients %}
    <li>{{ ingredient }}</li>
  {% endfor %}
  </ul>

  <h3>Instructions</h3>
  <ul>
  {% for key, instruction in template_instructions|dictsort %}
    <li>{{ instruction }}</li>
  {% endfor %}
  </ul>
{% endblock %}
```

Cooking By Myself

Welcome to my cookbook. These are recipes I like.

Fried Egg

Buttered Toast

Back To Recipe List

Fried Egg

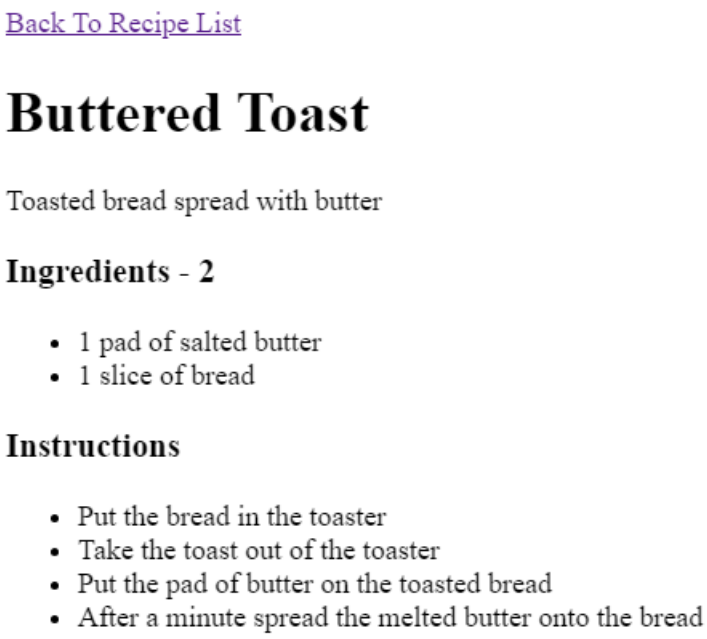
Egg fried in butter

Ingredients - 3

- 1 pad of butter
- 1 Egg
- A pinch of salt

Instructions

- Melt butter in pan over medium-low heat
- Crack the egg into the buttered pan
- Sprinkle the pinch of salt onto cooking egg
- Flip egg after about a minute and a half
- Serve egg after about a minute and a half



Review

Congratulations, this concludes the lesson on Flask templates. In this lesson we:

- Created a file structure that works with the Jinja2 template engine
- Rendered pages in our browser using files called templates
- Shared our application data for use within templates
- Applied filters to our data within our templates
- Utilized if statements to bring decision making to our templates
- Implemented for loops to perform repetitive tasks in our templates
- Moved common content to separate files to be shared by many templates

To show the power of what we have learned let’s add a simple navigation bar to the app.

Instructions

1.
Things added to our app are:

- an "about" route in **app.py**
- an **about.html** template inheriting from **base.html**
- the 'Back To Recipe List' link has been removed from **recipe.html**

Let's create a navigation bar that will link to the about page and the index page. In **base.html** add the following content just inside the `<body>` tag.

```
<div>
  <a href="/">Recipes</a>
   |
  <a href="/about">About</a>
</div>
```

Once complete run the code and you'll see that each page in the site now has a simple navigation bar including the new 'About' page.

Great work!

Hint
Paste the following code just inside the opening `<body>` tag.

```
<div>
  <a href="/">Recipes</a>
   |
  <a href="/about">About</a>
</div>
```

base.html

```
<!DOCTYPE html>
<html>
  <body>
    <!-- Insert navigation bar HTML below -->
    <div>
      <a href="/">Recipes</a>
      |
      <a href="/about">About</a>
    </div>
    {% block content %}


    {% endblock %}
```

```
  </body>
</html>
```

app.py

```
from flask import Flask, render_template
from helper import recipes, descriptions, ingredients, instructions

app = Flask(__name__)

@app.route('/')
def index():
  return render_template("index.html", template_recipes=recipes)

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

@app.route("/recipe/<int:id>")
def recipe(id):
  return render_template("recipe.html", template_recipe=recipes[id], template_des
cription=descriptions[id], template_ingredients=ingredients[id], template_instruc
tions=instructions[id])
```

about.html

```
{% extends "base.html" %}
{% block content %}
  <h1>About Myself</h1>
  <p>
    I like to cook, save recipes and share them with you.
  </p>
  <p>
    Thank you for visiting!!!
  </p>
{% endblock %}
```

About Myself

I like to cook, save recipes and share them with you.

Thank you for visiting!!!