realpython.com

Primer on Jinja Templating – Real Python

Real Python

14-18 minutes

Flask comes packaged with the powerful Jinja templating language.

For those who have not been exposed to a templating language before, such languages essentially contain variables as well as some programming logic, which when evaluated (or rendered into HTML) are replaced with **actual values**.

The variables and/or logic are placed between tags or delimiters. For example, Jinja templates use `{% ... %}` for expressions or logic (like for loops), while `{{ ... }}` is used for outputting the results of an expression or a variable to the end user. The latter tag, when rendered, is replaced with a value or values, and is seen by the end user.

Note: Jinja Templates are just `.html` files. By convention, they live in the `/templates` directory in a Flask project. If you're familiar with string formatting or interpolation, templating languages follow a similar type of logic—just on the scale of an entire HTML page.

Quick Examples

Make sure you have Jinja installed before running these examples (`pip install jinja2`):

```
>>>
```

```
>>> from jinja2 import Template
>>> t = Template("Hello {{ something }}!")
>>> t.render(something="World")
u'Hello World!'

>>> t = Template("My favorite numbers: {%
for n in range(1,10) %}{{n}} " "{% endfor
%}")
>>> t.render()
u'My favorite numbers: 1 2 3 4 5 6 7 8 9 '
```

Notice how the actual output rendered to the user falls within the tags.

Flask Examples

The code can be found here.

Create the following project structure:

```
├── requirements.txt
├── run.py
└── templates
```

Activate a virtualenv, then install flask:

Add the following code to *run.py*:

```
from flask import Flask, render_template
```

```
app = Flask(__name__)


@app.route("/")
def template_test():
    return render_template('template.html',
my_string="Wheeeee!", my_list=[0,1,2,3,4,5])


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

Here, we are establishing the route `/`, which renders the template *template.html* via the function `render_template()`. This function must have a template name. Optionally, you can pass in keyword arguments to the template, like in the example with `my_string` and `my_list`.

Add the template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Flask Template Example</title>
    <meta name="viewport"
content="width=device-width, initial-
scale=1.0">
    <link
href="http://netdna.bootstrapcdn.com
/bootstrap/3.0.0/css/bootstrap.min.css"
```

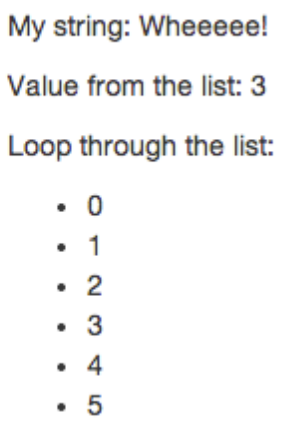
```
rel="stylesheet" media="screen">
    <style type="text/css">
      .container {
        max-width: 500px;
        padding-top: 100px;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <p>My string: {{my_string}}</p>
      <p>Value from the list:
{{my_list[3]}}</p>
      <p>Loop through the list:</p>
      <ul>
        {% for n in my_list %}
        <li>{{n}}</li>
        {% endfor %}
      </ul>
    </div>
    <script src="http://code.jquery.com
/jquery-1.10.2.min.js"></script>
    <script
src="http://netdna.bootstrapcdn.com
/bootstrap/3.0.0/js/bootstrap.min.js">
</script>
  </body>
</html>
```

Save this as *template.html* in the templates directory. Notice

the template tags. Can you guess the output before you run the app?

Run the following:

You should see the following:



Note: It's worth noting that Jinja only supports a few control structures: `if`-statements and `for`-loops are the two primary structures.

The syntax is similar to Python, differing in that no colon is required and that termination of the block is done using an `endif` or `endfor` instead of whitespace.

You can also complete the logic within your controller or views and then pass each value to the template using the template tags. However, it is much easier to perform such logic within the templates themselves.

Template Inheritance

Templates usually take advantage of inheritance, which includes a single base template that defines the basic structure of all subsequent child templates. You use the tags

`{% extends %}` and `{% block %}` to implement inheritance.

The use case for this is simple: as your application grows, and you continue adding new templates, you will need to keep common code (like an HTML navigation bar, Javascript libraries, CSS stylesheets, and so forth) in sync, which can be a lot of work. Using inheritance, we can move those common pieces to a parent/base template so that we can create or edit such code once, and all child templates will inherent that code.

Note: You should always add as much recurring code as possible to your base template to save yourself time in the future, which will far outweigh the initial time investment.

Let's add inheritance to our example.

Create the base (or parent) template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Flask Template Example</title>
    <meta name="viewport"
content="width=device-width, initial-
scale=1.0">
    <link
href="http://netdna.bootstrapcdn.com
/bootstrap/3.0.0/css/bootstrap.min.css"
rel="stylesheet" media="screen">
    <style type="text/css">
      .container {
```

```
        max-width: 500px;
        padding-top: 100px;
      }
      h2 {color: red;}
    </style>
  </head>
  <body>
    <div class="container">
      <h2>This is part of my base
template</h2>
      <br>
      {% block content %}{% endblock %}
      <br>
      <h2>This is part of my base
template</h2>
    </div>
    <script src="http://code.jquery.com
/jquery-1.10.2.min.js"></script>
    <script
src="http://netdna.bootstrapcdn.com
/bootstrap/3.0.0/js/bootstrap.min.js">
</script>
  </body>
</html>
```

Save this as *layout.html*.

Did you notice the `{% block %}` tags? This defines a block (or area) that child templates can fill in. Further, this just informs the templating engine that a child template may override the block of the template.

Note: Think of these as placeholders to be filled in by code from the child template(s).

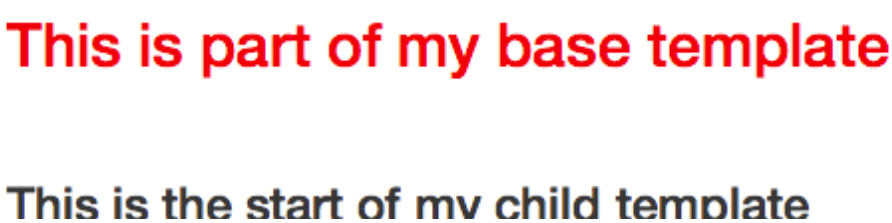
Let's do that.

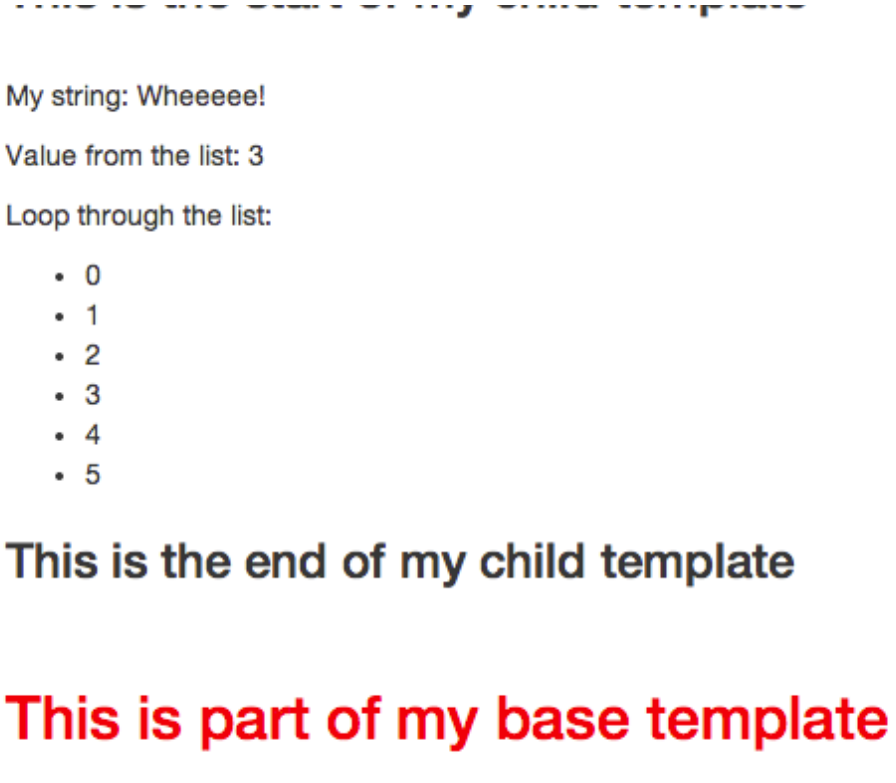
Update *template.html*:

```
{% extends "layout.html" %}
{% block content %}
  <h3> This is the start of my child
template</h3>
  <br>
  <p>My string: {{my_string}}</p>
  <p>Value from the list: {{my_list[3]}}</p>
  <p>Loop through the list:</p>
  <ul>
    {% for n in my_list %}
    <li>{{n}}</li>
    {% endfor %}
  </ul>
  <h3> This is the end of my child
template</h3>
{% endblock %}
```

So, the `{% extends %}` informs the templating engine that this template "extends" another template, *layout.html*. This establishes the link between the templates.

Run it. You should see this:





One common use case is to add a navigation bar.

Add the following code to the base template, just after the opening <body> tag:

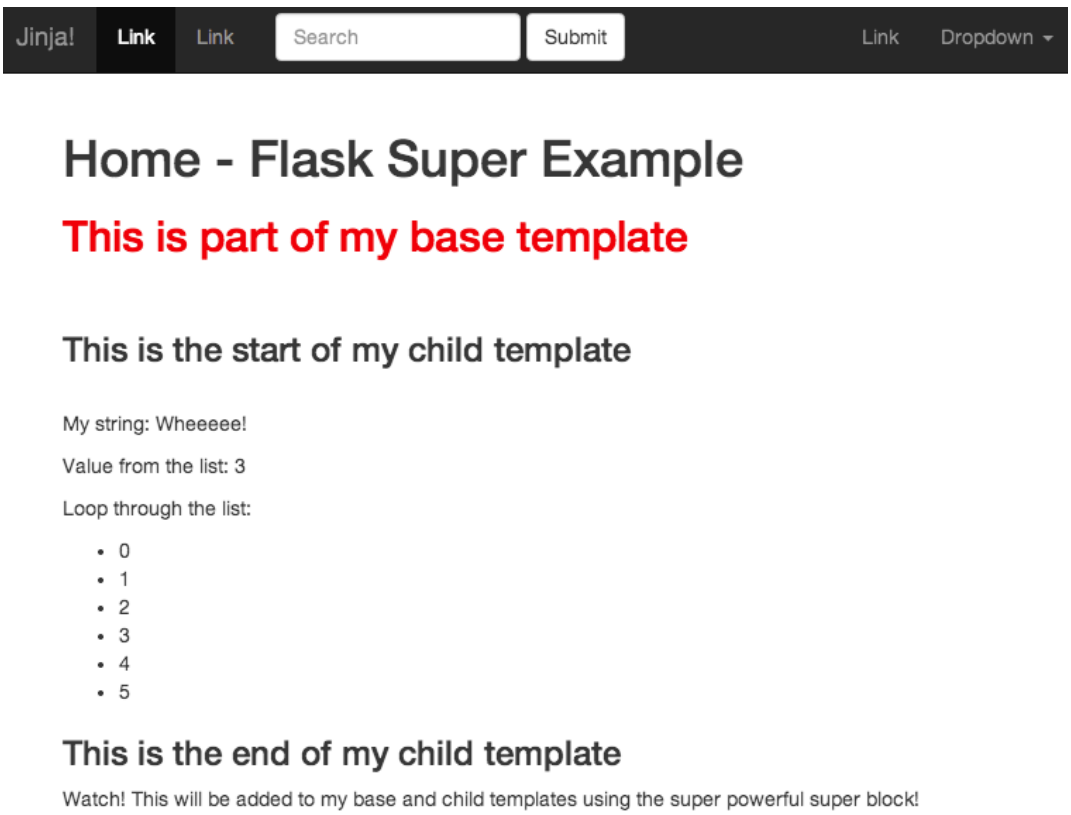
```
<nav class="navbar navbar-inverse"
role="navigation">
  <div class="container-fluid">
    <div class="navbar-header">
      <button type="button" class="navbar-
toggle" data-toggle="collapse" data-
target="#bs-example-navbar-collapse-1">
        <span class="sr-only">Toggle
navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
```

```
      <a class="navbar-brand"
href="/">Jinja!</a>
    </div>

    <div class="collapse navbar-collapse"
id="bs-example-navbar-collapse-1">
      <ul class="nav navbar-nav">
        <li class="active"><a
href="#">Link</a></li>
        <li><a href="#">Link</a></li>
      </ul>
      <form class="navbar-form navbar-left"
role="search">
        <div class="form-group">
          <input type="text" class="form-
control" placeholder="Search">
        </div>
        <button type="submit" class="btn
btn-default">Submit</button>
      </form>
      <ul class="nav navbar-nav navbar-
right">
        <li><a href="#">Link</a></li>
        <li class="dropdown">
          <a href="#" class="dropdown-
toggle" data-toggle="dropdown">Dropdown <b
class="caret"></b></a>
          <ul class="dropdown-menu">
            <li><a href="#">Action</a></li>
```

```
            <li><a href="#">Another
action</a></li>
            <li><a href="#">Something else
here</a></li>
            <li class="divider"></li>
            <li><a href="#">Separated
link</a></li>
          </ul>
        </li>
      </ul>
    </div><!-- /.navbar-collapse -->
  </div><!-- /.container-fluid -->
</nav>
```

Now, every single child template that extends from the base will have the same navigation bar. To steal a line from Java philosophy: “Write once, use anywhere.”



Super Blocks

If you need to render a block from the base template, use a super block:

Add a footer to the base template:

```
<div class="footer">
  {% block footer %}
    Watch! This will be added to my base and
child templates using the super powerful
super block!
    <br>
    <br>
  {% endblock %}
</div>
```

Here's the updated code:

```
<!DOCTYPE html>
  <head>
    <title>Flask Template Example</title>
    <meta name="viewport"
content="width=device-width, initial-
scale=1.0">
    <link
href="http://netdna.bootstrapcdn.com
/bootstrap/3.0.0/css/bootstrap.min.css"
rel="stylesheet" media="screen">
    <style type="text/css">
      .container {
```

```
        max-width: 500px;
        padding-top: 100px;
      }
      h2 {color: red;}
    </style>
  </head>
  <body>
    <div class="container">
      <h2>This is part of my base
template</h2>
      <br>
      {% block content %}{% endblock %}
      <br>
      <h2>This is part of my base
template</h2>
      <br>
      <div class="footer">
        {% block footer %}
          Watch! This will be added to my
base and child templates using the super
powerful super block!
          <br>
          <br>
          <br>
        {% endblock %}
      </div>
    </div>
    <script src="http://code.jquery.com
/jquery-1.10.2.min.js"></script>
```

```
    <script
src="http://netdna.bootstrapcdn.com
/bootstrap/3.0.0/js/bootstrap.min.js">
</script>
  </body>
</html>
```

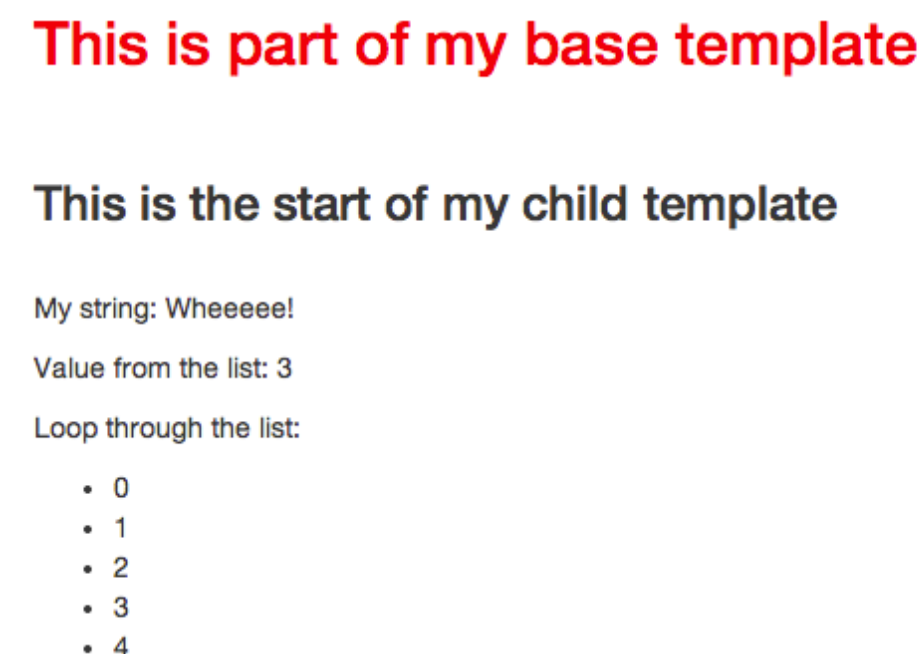
Run the app. You should see that the footer is just part of the base:

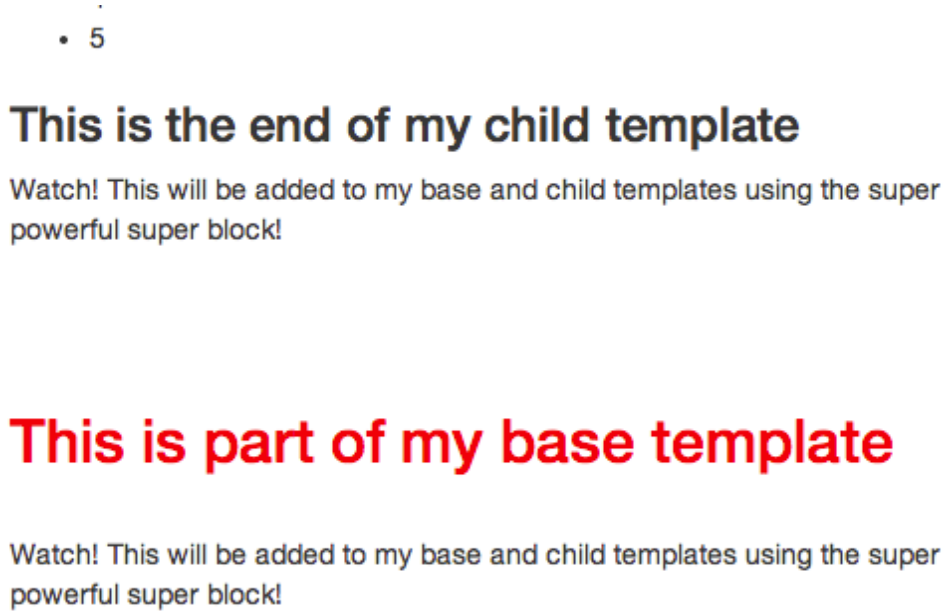
Now, add the super block to *template.html*:

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

```
{% block content %}
  <h3> This is the start of my child
template</h3>
  <br>
  <p>My string: {{my_string}}</p>
  <p>Value from the list: {{my_list[3]}}</p>
  <p>Loop through the list:</p>
  <ul>
    {% for n in my_list %}
    <li>{{n}}</li>
    {% endfor %}
  </ul>
  <h3> This is the end of my child
template</h3>
  {% block footer %}
  {{super()}}
  {% endblock %}
{% endblock %}
```

Check it out in your browser:





The super block is used for common code that both the parent and child templates share, such as the `<title>`, where both templates share part of the title. Then, you would just need to pass in the other part. It could also be used for a heading.

Here's an example:

Parent

```
{% block heading %}
  <h1>{% block page %}{% endblock %} - Flask
Super Example</h1>
{% endblock %}
```

Child

```
{% block page %}Home{% endblock %}
{% block heading %}
  {{ super() }}
{% endblock %}
```

Let's see that in action:

See what happens when you remove `{% block page %}Home{% endblock %}` from the child template.

Challenge: Try to update the `<title>` using the same method with the super block. Check out my code if you need help.

Instead of hard coding the name of the template, let's make it dynamic.

Update the two code snippets in *template.html*:

```
{% block title %}{{title}}{% endblock %}
```

```
{% block page %}{{title}}{% endblock %}
```

Now, we need to pass in a `title` variable to our template

from our controller, *run.py*:

```
@app.route("/")
def template_test():
    return render_template(
        'template.html',
my_string="Wheeeee!",
        my_list=[0,1,2,3,4,5], title="Home")
```

Test this out.

Macros

In Jinja, we can use macros to abstract commonly used code snippets that are used over and over to not repeat ourselves. For example, it's common to highlight the link of the current page on the navigation bar (active link). Otherwise, we'd have to use `if`/`elif`/`else` statements to determine the active link. Using macros, we can abstract out such code into a separate file.

Add a *macros.html* file to the `templates` directory:

```
{% macro nav_link(endpoint, name) %}
{% if request.endpoint.endswith(endpoint) %}
  <li class="active"><a href="{{
url_for(endpoint) }}">{{name}}</a></li>
{% else %}
  <li><a href="{{ url_for(endpoint)
}}">{{name}}</a></li>
{% endif %}
{% endmacro %}
```

Here, we're using Flask's request object, which is part of Jinja by default, to check the requested endpoint, and then assigning the `active` class to that endpoint.

Update the unordered list with the `nav navbar-nav` class in the base template:

```
<ul class="nav navbar-nav">
  {{ nav_link('home', 'Home') }}
  {{ nav_link('about', 'About') }}
  {{ nav_link('contact', 'Contact Us') }}
</ul>
```

Also, make sure to add the import at the top of the template: `{% from "macros.html" import nav_link with context %}`.

Notice how we're calling the `nav-link` macro and passing it two arguments: the endpoint (which comes from our controller) and the text we want displayed.

Finally, let's add three new endpoints to the controller:

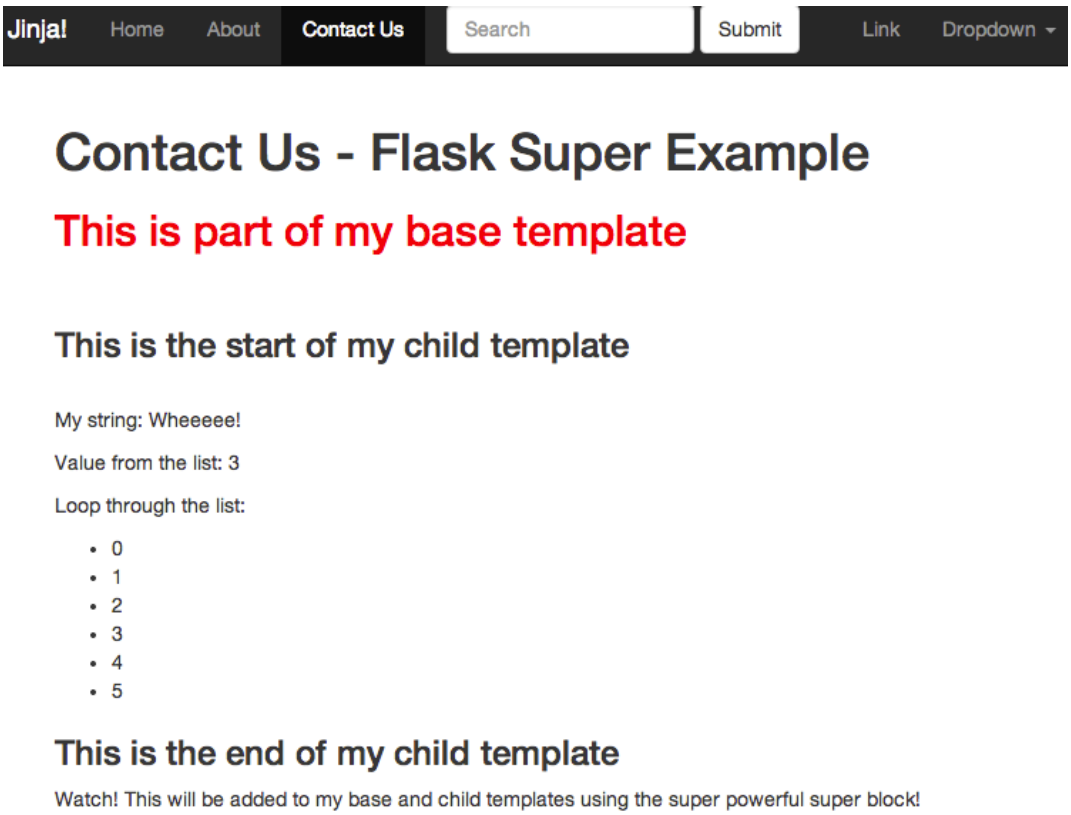
```
@app.route("/home")
def home():
    return render_template(
        'template.html',
my_string="Wheeeee!",
        my_list=[0,1,2,3,4,5], title="Home")

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

```
        'template.html',
my_string="Wheeeee!",
        my_list=[0,1,2,3,4,5],
title="About")

@app.route("/contact")
def contact():
    return render_template(
        'template.html',
my_string="Wheeeee!",
        my_list=[0,1,2,3,4,5],
title="Contact Us")
```

Refresh the page. Test out the links at the top. Does the current page get highlighted? It should.



Custom Filters

Jinja uses filters to modify variables, mostly for formatting purposes.

Here's an example:

This will round the `num` variable. So, if we pass the argument `num=46.99` into the template, then `47.0` will be outputted.

As you can tell, you specify the variable and then a pipe (|), followed by the filter. Check out this link for the list of filters already included within Jinja. In some cases, you can specify optional arguments in parentheses.

Here's an example:

This will join a list by the comma delimiter.

Test this out. Add the following line to *template.html*

```
<p>Same list with a filter: {{
my_list|join(', ') }}</p>
```

Now, besides the built-in filters, we can create our own.

Let's add one of our own. One common example is a custom datetime filter.

Add the following code to our controller after we create the app, `app = Flask(__name__)`:

```
@app.template_filter()
def datetimefilter(value, format='%Y/%m/%d
%H:%M'):
    """Convert a datetime to a different
format."""
    return value.strftime(format)
```

```
app.jinja_env.filters['datetimefilter'] =
datetimefilter
```

Using the `@app.template_filter()` decorator, we are registering the `datetimefilter()` function as a filter.

Note: The default name for the filter is just the name of the function. However, you can customize it by passing in an argument to the function, such as `@app.template_filter(formatdate)`.

Next, we are adding the filter to the Jinja environment, making it accessible. Now it's ready for use.

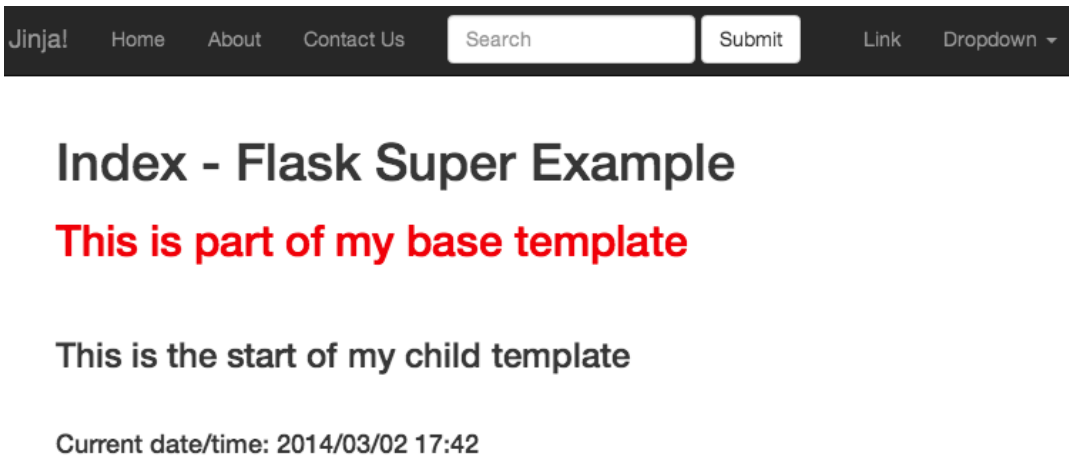
Add the following code to our child template:

```
<h4>Current date/time: {{ current_time |
datetimefilter }}</h4>
```

Finally, just pass in the datetime to our template:

```
current_time = datetime.datetime.now()
```

Test it.



Conclusion

That’s it. Grab the sample code here. Did I miss anything? Leave a comment below.