Previous Next

Templating in Flask using Jinja

Estimated Time: 1-2 hours

Developed by Thinkful Mentor Michael Herman. Originally posted on Real Python.

In the previous assignment you returned HTML as a string using Flask. In this assignment you'll learn how to use Jinja templates to construct the HTML you want to display. Templating allows you to separate HTML from the view functions and construct complex HTML pages in a simpler, more modular way.

Right out of the box, Flask includes the powerful Jinja templating language. It's modeled after Django templates (but it renders much faster) and, although Flask does not force you to use any templating language, it assumes that you'll be using Jinja since it does come pre-installed.

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 `{{ ... }}` are 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 are seen by the end user.

> 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")
'Hello World!'
>>>
>>> t = Template("My favorite numbers: {% for n in range(1,10) %}{{n}}
>>> t.render()
'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

First you will need to create a new project:

- Make a new project directory – `mkdir jinja_example`
- Move into the directory – `cd jinja_example`
- Make a templates directory – `mkdir templates`
- Set up a virtualenv – `python3 -m venv env`
- Activate the virtualenv – `source env/bin/activate`
- Install Flask – `pip install flask`

Then make a new file called *run.py* and add the following code:

```
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, host="0.0.0.0", port=8080)
```

Here you 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 arguments to the template, like in the example – `my_string` and `my_list`.

Next make a template and save it as *template.html* in the *templates* directory:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Flask Template Example</title>
        <meta name="viewport" content="width=device-width, initial-sca
        <style>
            div#main {
                max-width: 500px;
                padding: 20px;
            }

            nav .title {
                font-weight: bold;
            }

            a.active {
                font-weight: bold;
            }
        </style>
    </head>
    <body>
        <div id="main">
```

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

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

Run the app using `python3 run.py`, and view the page in your browser (as per the technique in the last section). You should see the following:

My string: Wheeeee!

Value from the list: 3

Loop through the list:

- 0
- 1
- 2
- 3
- 4
- 5

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 by whitespace. You can also complete the logic within your Python code 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.

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, you can move those common pieces to a parent template so that you can create or edit such code once and all child templates will inherit that code.

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.

Next you'll add inheritance to the example. First create the base (or parent) template in a file called *base.html*:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Flask Template Example</title>
        <meta name="viewport" content="width=device-width, initial-sca
        <style>
            div#main {
                max-width: 500px;
                padding: 20px;
            }

            nav .title {
```

```
                font-weight: bold;
            }

            a.active {
                font-weight: bold;
            }
        </style>
    </head>
    <body>
        <div id="main">
            <h2>This is part of my base template</h2>
            <br>
            {% block content %}{% endblock %}
            <br>
            <h2>This is part of my base template</h2>
        </div>
    </body>
</html>
```

Did you notice the `{% block %}` tag? This defines a block (or area) that child templates can fill in. Think of these as placeholders to be filled in by code from the child template(s).

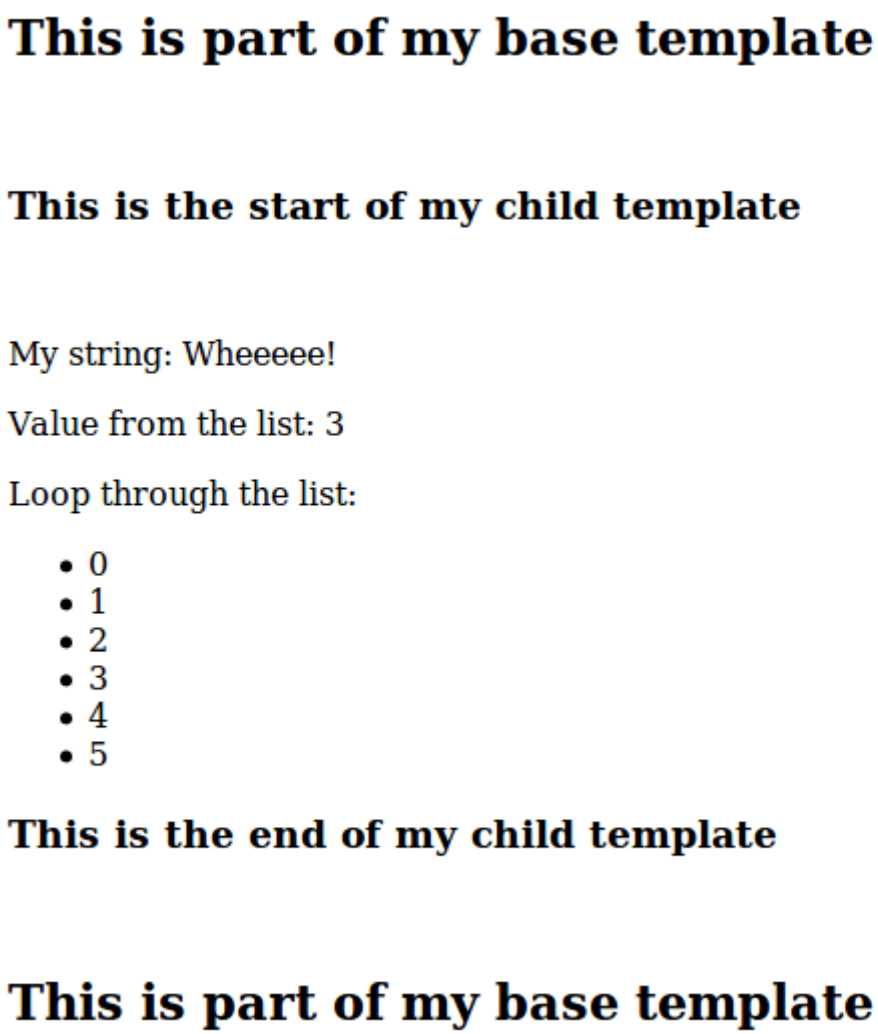
Update your *template.html* to inherit from *base.html*:

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

The `{% extends %}` informs the templating engine that this template "extends" another template, *base.html*. This establishes the link between the templates, in other words.

Try visiting your page again. 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>
    <span class="title">Jinja!</span>
```

```
    <a href="#">Home</a> |
    <a href="#">About</a> |
    <a href="#">Contact Us</a>
</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.”

Jinja! Home | About | Contact Us |

This is part of my base template

This is the start of my child template

My string: Wheeeee!

Value from the list: 3

Loop through the list:

- 0
- 1
- 2
- 3
- 4
- 5

This is the end of my child template

This is part of my base template

Super Blocks

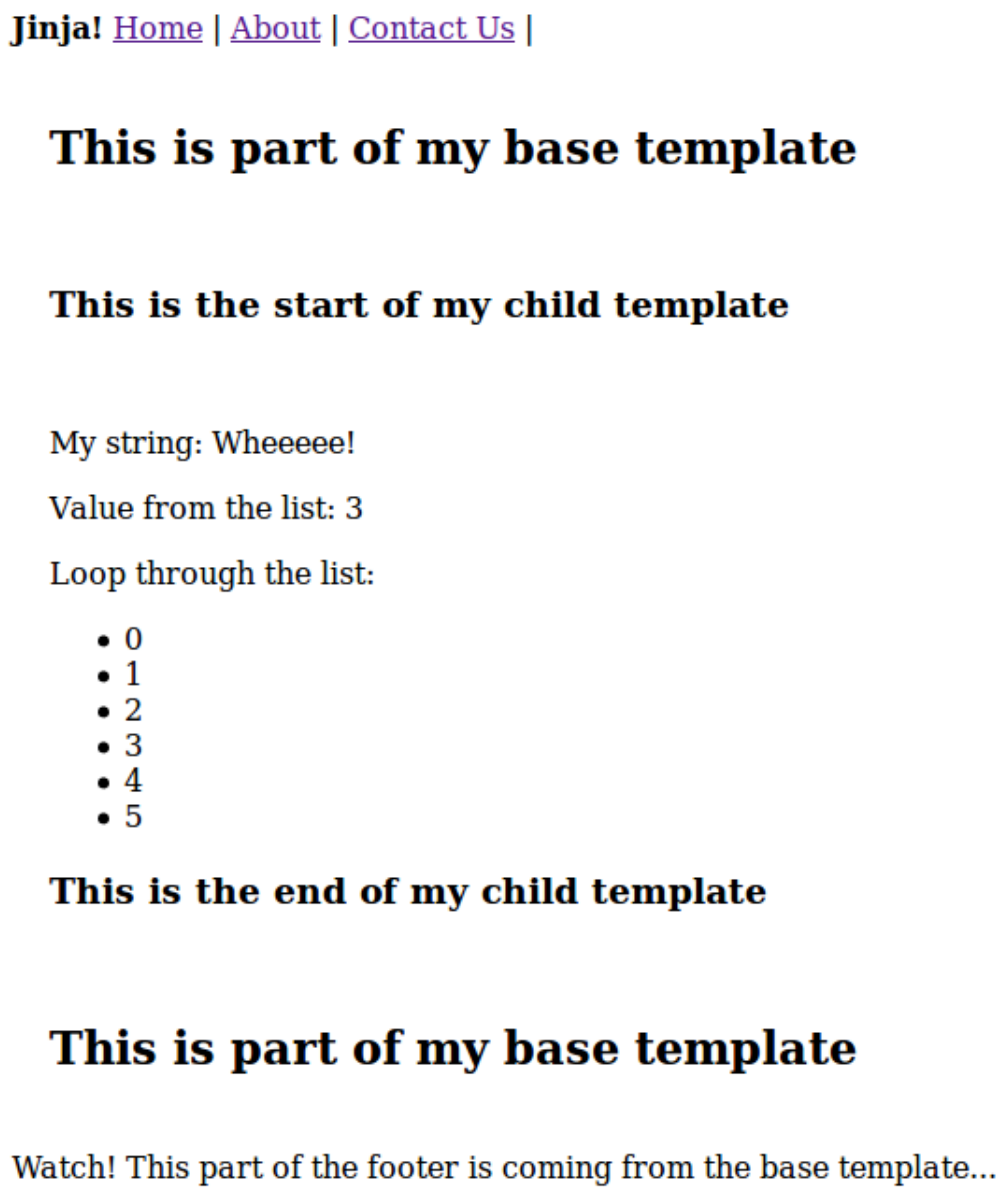
If you need to render a block from an inherited template then you can use a super block:

```
{{ super() }}
```

Add a footer to the end of the body in *base.html*:

```
<footer>
    {% block footer %}
        Watch! This part of the footer is coming from the base templat
        <br>
        <br>
    {% endblock %}
</footer>
```

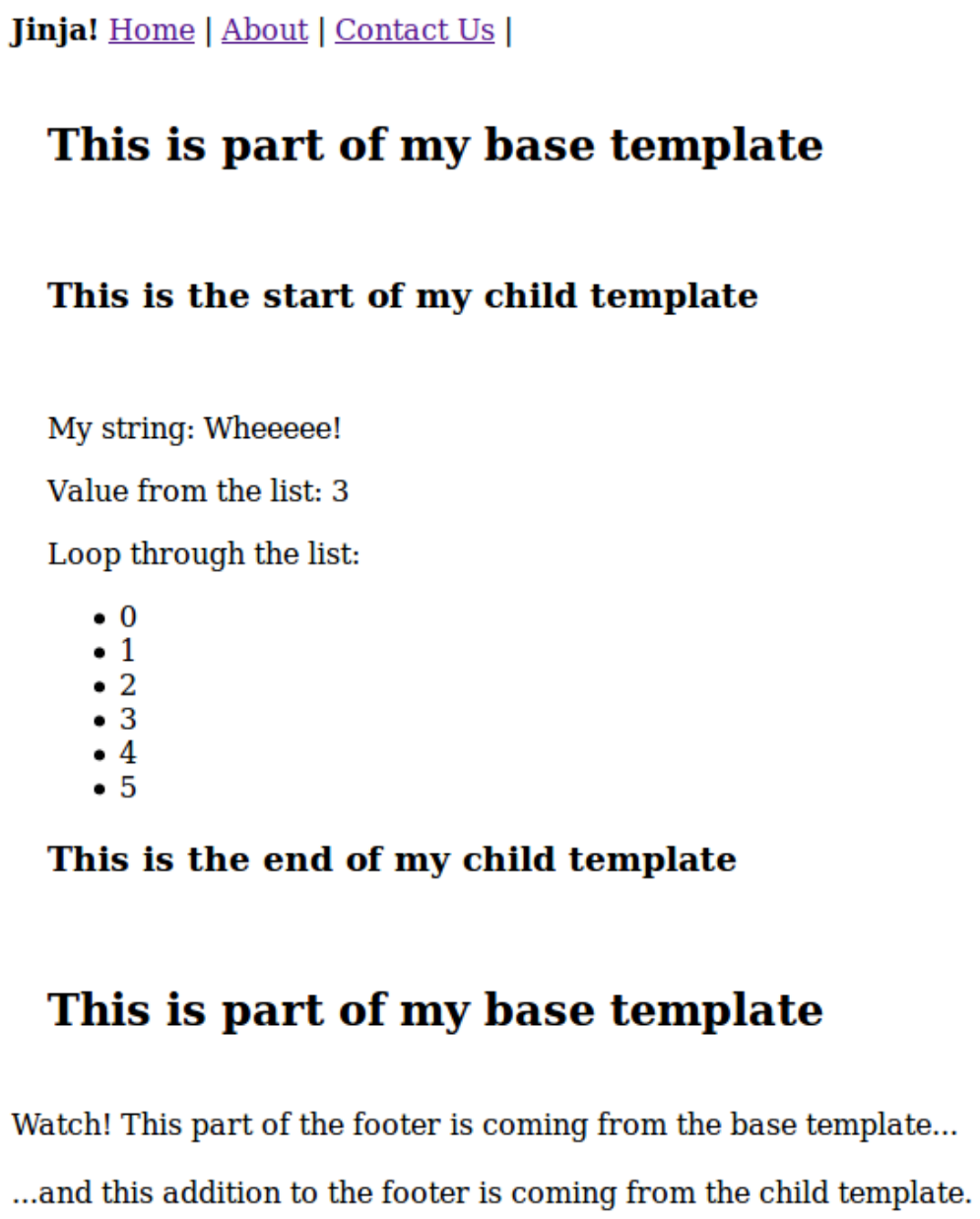
When you visit your page you should see that the footer is just part of the base:



Now, redefine the footer in the child template, using the `super` function so that you extend the base's content rather than completely overriding it. Add the following code to the end of *template.html*:

```
{% block footer %}
{{super()}}
...and this addition to the footer is coming from the child template.
{% endblock %}
```

Check it out in your browser:



Notice how the child template uses the base template's footer block – the super block is used for the common code that both the parent and child templates share.

Macros

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

Make a new file called *macros.html*, and add the following content.

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

Here, you're using Flask's request object, which is part of Jinja by default, to check whether the link points to the current page. If so then you'll add the active class to the `<a>` tag.

Next, add three new routes to *run.py*:

```
@app.route("/home")
def home():
    return render_template('template.html',
                           my_string="I'm the home page",
                           my_list=[0,1,2,3,4,5])

@app.route("/about")
def about():
    return render_template('template.html',
                            my_string="I'm the about page",
                            my_list=[0,1,2,3,4,5])

@app.route("/contact")
def contact():
    return render_template('template.html',
                           my_string="I'm the contact page",
                           my_list=[0,1,2,3,4,5])
```

Finally, update the links in the base template to use our new macro:

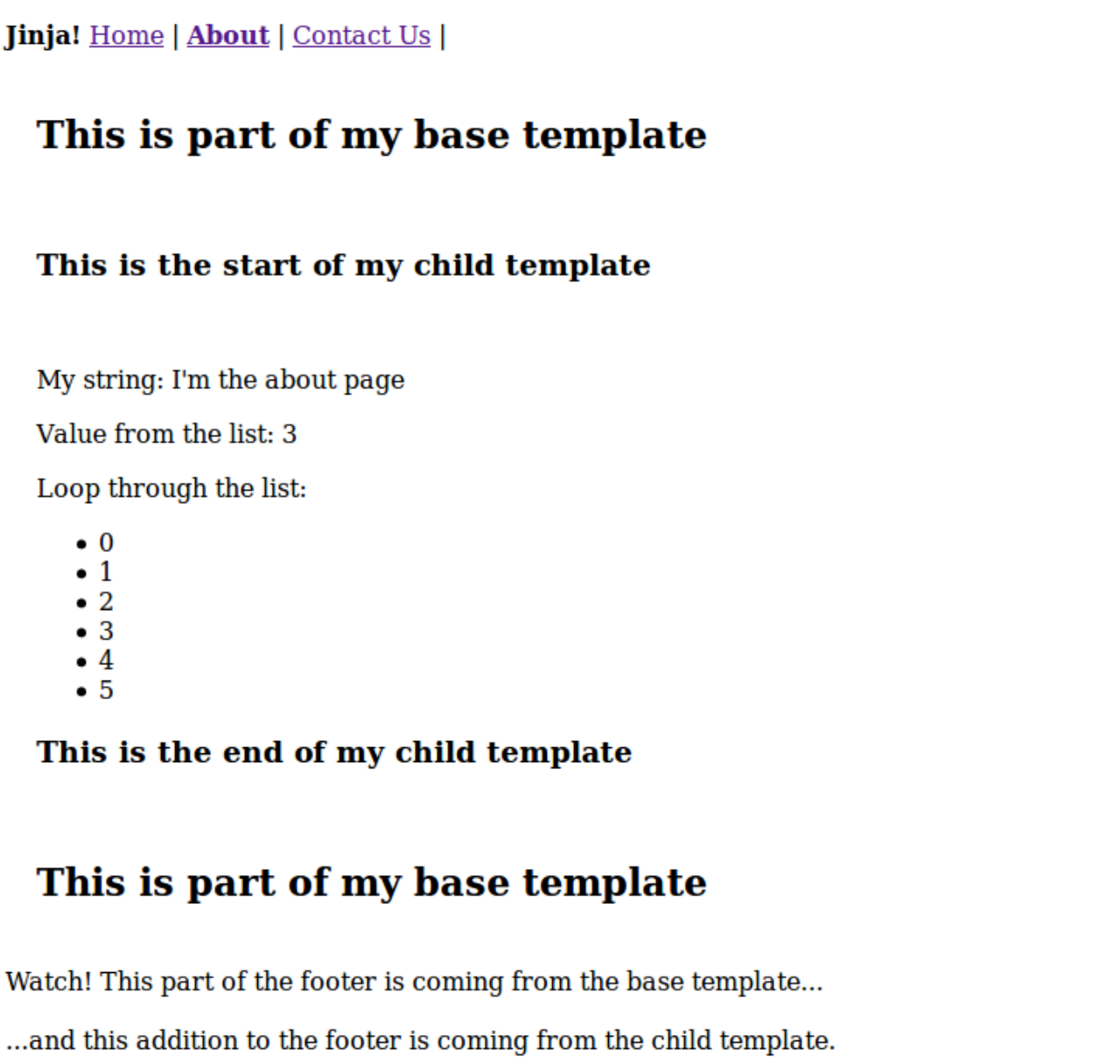
```
<ul class="nav navbar-nav">
  {{ nav_link('home', 'Home') }}
  {{ nav_link('about', 'About') }}
```

```
    {{ 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 you're calling the `nav-link` macro and passing it two arguments, the link location and the text you want displayed.

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 purpose.

For example:

```
{{ num | round }}
```

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

As you can tell, you specify the variable 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.

For example:

```
{{ list|join(', ') }}
```

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, you can create your own.

One common example is a custom datetime filter.

Add the following code to *run.py* after you create the app variable:

```
@app.template_filter()
def datetimefilter(value, format='%Y/%m/%d %H:%M'):
```

```
    """Convert a datetime to a different format."""
    return value.strftime(format)
```

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

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 – e.g, `@app.template_filter("formatdate")` .

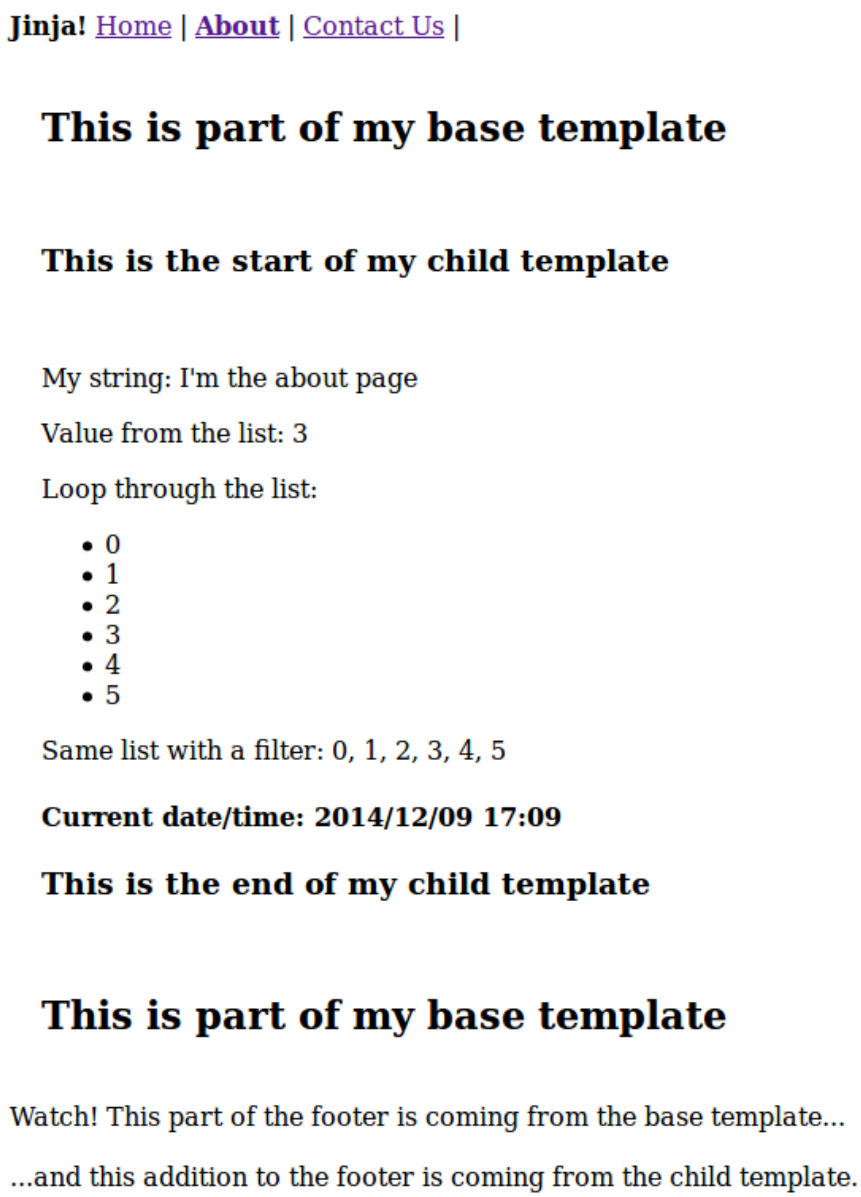
Next, you are adding the filter to the Jinja environment, making it accessible. Now it's ready to use.

Add the following code to the child template:

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

Finally, in *run.py* pass in the datetime to your template along with `my_list` and `my_string` (remembering to import the `datetime` module at the top of the file:

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



Try It!

Take your Hello World app from the previous exercise, and modify it so that it uses Jinja as a template engine instead of Python string formatting.

Where does the `render_template` function look for templates by default?

> *In a folder called "templates" in the current directory.*

What arguments can you pass to `render_template`, and how are these accessed from within a template?

> *First the name of the template file, followed by optional keyword arguments.*

What is the difference between `{{ }}` tags and `{% %}` tags?

> `{{ }}` *is used for outputting the results of an expression or variable, while* `{% %}` *is used for logic (for instance, for loops).*

How do you inherit from a parent template? Why is this helpful?

> *You use* `{% extends %}` *and* `{% block %}`*. You define a parent template and insert placeholders for sections of content that will vary between pages. The advantage is that display code that is shared across pages only gets defined in one place and is therefore more maintainable.*

What is the `super` block used for?

> `super` *is used for bits of code that are defined in a parent template but need to be accessed in one or more child templates.* `super()` *will render the contents of the block defined in the parent.*

How do you create a template macro? How do you call it from a template?

> `jinja {% macro macro_name(param1='default_value', param2='default_value',...) %} <p id="{{ param1 }}" class="{{ param2 }}">HTML content goes here</p> {% endmacro %}` *If the macro was defined in the same template, you can just call it like a function:* `jinja <div>{{ macro_name("foo", "bar") }}</div>` *If the macro was defined in a different template, you must* `import` *it before using it. See http://jinja.pocoo.org/docs/templates/#macros for more information.*

What does the pipe (|) symbol do inside a tag?

> *It pumps the output of one filter into another.*

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

Mark as completed

Previous Next