Python Jinja tutorial

Jinja tutorial shows how to create templates in Python with Jinja module.

Python Jinja module

Jinja is a template engine for Python. It is similar to the Django template engine.

A template engine or template processor is a library designed to combine templates with a data model to produce documents. Template engines are often used to generate large amounts of emails, in source code preprocessing, or producing dynamic HTML pages.

We create a template engine, where we define static parts and dynamic parts. The dynamic parts are later replaced with data. The rendering function later combines the templates with data.

Jinja installation

```
$ sudo pip3 install jinja2
```

We use the pip3 tool to install Jinja.

Jinja delimiters

Jinja uses various delimiters in the template strings.

- {% %} - statements
- {{ }} - expressions to print to the template output
- {# #} - comments which are not included in the template output
- # ## - line statements

Jinja simple example

In the first example, we create a very simple template.

simple.py

```
#!/usr/bin/env python3
```

```
from jinja2 import Template

name = input("Enter your name: ")

tm = Template("Hello {{ name }}")
msg = tm.render(name=name)

print(msg)
```

The example asks for a user name and generates a message string, which is printed to the user. The templat engine is similar to the Python format() method; but template engines are more powerful and have many more features.

```
from jinja2 import Template
```

We import the Template object from the jinja2 module. Template is the central template object. It represents compiled template and is used to evaluate it.

```
tm = Template("Hello {{ name }}")
```

In our template, we have the {{ }} syntax which is used to print the variable. The variable is passed in the render() method.

```
msg = tm.render(name=name)
```

With the render() method, we generate the final output. The method joins the template string with the data passed as argument. The variable that is passed to the render() method is called the *context variable*.

```
$ ./simple.py
Enter your name: Paul
Hello Paul
```

This is a sample output.

In the next example, we use two variables.

simple2.py

```
#!/usr/bin/env python3

from jinja2 import Template

name = 'Peter'
age = 34

tm = Template("My name is {{ name }} and I am {{ age }}")
msg = tm.render(name=name, age=age)

print(msg)
```

The template string renders two variables: name and age. This time the variables are hard-coded.

```
$ ./simple2.py
My name is Peter and I am 34
```

This is the output.

Jinja objects

We can work with objects in our template strings.

objects.py

```
#!/usr/bin/env python3

from jinja2 import Template

class Person:

    def __init__(self, name, age):

        self.name = name
        self.age = age

    def getAge(self):
        return self.age

    def getName(self):
        return self.name


person = Person('Peter', 34)

tm = Template("My name is {{ per.getName() }} and I am {{ per.getAge() }}")
msg = tm.render(per=person)

print(msg)
```

In the example, we define a Person object. We get the name and age via the two getters.

Dictionaries

Jinja allows a convenient dot notation to access data in Python dictionaries.

dicts.py

```
#!/usr/bin/env python3

from jinja2 import Template

person = { 'name': 'Person', 'age': 34 }
```

```
tm = Template("My name is {{ per.name }} and I am {{ per.age }}")
# tm = Template("My name is {{ per['name'] }} and I am {{ per['age'] }}")
msg = tm.render(per=person)

print(msg)
```

We have a person dictionary. We access the dictionary keys with a dot operator.

```
tm = Template("My name is {{ per.name }} and I am {{ per.age }}")
# tm = Template("My name is {{ per['name'] }} and I am {{ per['age'] }}")
```

Both the active and the commented way are valid. The dot notation is more convenient.

Jinja raw data

We can use raw, endraw markers to escape Jinja delimiters.

raw_data.py

```
#!/usr/bin/env python3

from jinja2 import Template

data = '''
{% raw %}
His name is {{ name }}
{% endraw %}
'''

tm = Template(data)
msg = tm.render(name='Peter')

print(msg)
```

By using the raw, endraw block, we escape the Jinja {{ }} syntax. It is printed in its literal meaning.

Jinja escape data

To escape data such as < or > characters, we can use a filter or the escape() function.

escape_data.py

```
#!/usr/bin/env python3

from jinja2 import Template, escape

data = '<a>Today is a sunny day</a>'

tm = Template("{{ data | e}}")
msg = tm.render(data=data)
```

```
print(msg)
print(escape(data))
```

The example escapes < and > characters.

```
tm = Template("{{ data | e}}")
```

Using the e filter, the data is escaped. Filters are applied with the | character.

```
print(escape(data))
```

The escape function does the same.

Jinja for expressions

The for expression is used to iterate over a data collection in a template.

Now we do not use a simple string template anymore. We use a text file which is loaded with FileSystemLoader.

for_expr.py

```
#!/usr/bin/env python3

from jinja2 import Environment, FileSystemLoader

persons = [
    {'name': 'Andrej', 'age': 34},
    {'name': 'Mark', 'age': 17},
    {'name': 'Thomas', 'age': 44},
    {'name': 'Lucy', 'age': 14},
    {'name': 'Robert', 'age': 23},
    {'name': 'Dragomir', 'age': 54}
]

file_loader = FileSystemLoader('templates')
env = Environment(loader=file_loader)

template = env.get_template('showpersons.txt')

output = template.render(persons=persons)
print(output)
```

In this example, the template is the showpersons.txt file. The file is located in the templates directory.

```
persons = [
    {'name': 'Andrej', 'age': 34},
    {'name': 'Mark', 'age': 17},
    {'name': 'Thomas', 'age': 44},
    {'name': 'Lucy', 'age': 14},
```

```
    {'name': 'Robert', 'age': 23},
    {'name': 'Dragomir', 'age': 54}
]
```

The data is a list of dictionaries.

```
file_loader = FileSystemLoader('templates')
env = Environment(loader=file_loader)
```

We define a FileSystemLoader. The template is retrieved from the templates directory.

```
template = env.get_template('showpersons.txt')
```

We get the template with the get_template() method.

templates/showpersons.txt

```
{% for person in persons -%}
    {{ person.name }} {{ person.age }}
{% endfor %}
```

In the template file, we use the for expression to iterate over the collection. We show the person's name and age. The dash character next to the % characters is used to control white space.

Jinja conditionals

Conditionals are expressions that are evaluated when a certain condition is met.

conditionals.py

```
#!/usr/bin/env python3

from jinja2 import Environment, FileSystemLoader

persons = [
    {'name': 'Andrej', 'age': 34},
    {'name': 'Mark', 'age': 17},
    {'name': 'Thomas', 'age': 44},
    {'name': 'Lucy', 'age': 14},
    {'name': 'Robert', 'age': 23},
    {'name': 'Dragomir', 'age': 54},
]

file_loader = FileSystemLoader('templates')
env = Environment(loader=file_loader)
env.trim_blocks = True
env.lstrip_blocks = True
env.rstrip_blocks = True

template = env.get_template('showminors.txt')
```

```
output = template.render(persons=persons)
print(output)
```

The example prints only minor persons; a minor is someone younger than 18.

```
env.trim_blocks = True
env.lstrip_blocks = True
env.rstrip_blocks = True
```

White space in output can be controllew with environment attributes.

templates/showminors.txt

```
{% for person in persons %}
    {% if person.age < 18 %}
        {{- person.name }}
    {% endif %}
{%- endfor %}
```

In the template, we output only persons younger than 18 using if expression.

```
$ ./conditionals.py
Mark
Lucy
```

This is the output.

Jinja sum filter

Filters can be applied to data to modify them. For instance, the sum filter can sum data, escape filter escape them, and sort filter sorts them.

sum_filter.py

```
#!/usr/bin/env python3

from jinja2 import Environment, FileSystemLoader

cars = [
    {'name': 'Audi', 'price': 23000},
    {'name': 'Skoda', 'price': 17300},
    {'name': 'Volvo', 'price': 44300},
    {'name': 'Volkswagen', 'price': 21300}
]

file_loader = FileSystemLoader('templates')
env = Environment(loader=file_loader)

template = env.get_template('sumprices.txt')
```

```
output = template.render(cars=cars)
print(output)
```

In the example, we use the sum filter to calculate the sum of all car prices.

```
cars = [
    {'name': 'Audi', 'price': 23000},
    {'name': 'Skoda', 'price': 17300},
    {'name': 'Volvo', 'price': 44300},
    {'name': 'Volkswagen', 'price': 21300}
]
```

We have a list of car dictionaries. Each dictionary has a price key. It will be used to calculate the sum.

templates/sumprices.txt

```
The sum of car prices is {{ cars | sum(attribute='price') }}
```

In the template file, we apply the filter on the cars collection object. The sum is calculated from the price attribute.

```
$ ./sum_filter.py
The sum of car prices is 105900
```

This is the output.

Jinja template inheritance

Template inheritance is a powerful feature that reduces code duplication and improves code organization. We define a base template from which we inherit in other template files. These template files overwrite specific blocks of the base template file.

ineritance.py

```
#!/usr/bin/env python3

from jinja2 import Environment, FileSystemLoader

content = 'This is about page'

file_loader = FileSystemLoader('templates')
env = Environment(loader=file_loader)

template = env.get_template('about.html')

output = template.render(content=content)
print(output)
```

We render the about.html file. It inherits from the base.html file.

base.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    {% block content%}

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

In the base.html file, we declare two blocks: title and content. These blocks are going to be filled with specifi tags and text in the child templates.

about.html

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

{% block title%}About page{% endblock %}

{% block content %}
<h1>About page</h1>
<p>
    This is about page
</p>
{% endblock %}
```

The about.html template file inherits from base.html. It adds data specific to this page. We avoid code repetition; we do not repeat tags that are same for both pages, such as body and html and meta tags.

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

The inheritance is done with the extends directive.

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

We define a title.

```
{% block content %}
<h1>About page</h1>
<p>
    This is about page
</p>
{% endblock %}
```

And we define content.

Jinja Flask example

In the next example, we create a simple Flask application that uses Jinja.

app.py

```
#!/usr/bin/env python3

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

@app.route("/greet")
def greet():
    username = request.args.get('name')
    return render_template('index.html', name=username)

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

In this Flask application, we get the name of a user and pass it as a parameter to the render_template() method. The greet() function reacts to the /greet path.

templates/index.html

```
<!doctype html>

<html lang="en">

<head>
    <meta charset="utf-8">
    <title>Greeting</title>
</head>

<body>
    <p>
        Hello {{ name }}
    </p>
</body>

</html>
```

This is the template file, located in the templates directory. We add the name of the user to the template file with {{ name }} syntax.

```
$ python3 app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

We start the server.

```
$ curl http://127.0.0.1:5000/greet?name=Peter
<!doctype html>

<html lang="en">

<head>
    <meta charset="utf-8">
    <title>Greeting</title>
</head>

<body>
    <p>
        Hello Peter
    </p>
</body>

</html>
```

We connect to the application with the curl tool. We add a name parameter.

In this tutorial, we have covered Python Jinja module.

You might also be interested in the following related tutorials: PyMongo tutorial, Python logging tutorial, pyDAL tutorial, and Python tutorial.

Home Top of Page