

Version 1.32 (/updates) is now available! Read about the new features and fixes from February.



TOPICS

Flask Tutorial ▼

Flask



(<https://github.com/Microsoft/vscode-docs/blob/master/docs/python/tutorial-flask.md>)

Tutorial

## in Visual Studio Code

Flask (<http://flask.pocoo.org/>) is a lightweight Python framework for web applications that provides the basics for URL routing and page rendering.

Flask is called a "micro" framework because it doesn't directly provide features like form validation, database abstraction, authentication, and so on. Such features are instead provided by special Python packages called Flask extensions (<http://flask.pocoo.org/extensions/>). The extensions integrate seamlessly with Flask so that they appear as if they were part of Flask itself. For example, Flask doesn't provide a page template engine, but installing Flask includes the Jinja (<http://jinja.pocoo.org/>) templating engine by default. For convenience, we typically speak of these defaults as part of Flask.

In this Flask tutorial, you create a simple Flask app with three pages that use a common base template. Along the way, you experience a number of features of Visual Studio Code including using the terminal, the editor, the debugger, code snippets, and more.

The completed code project for this Flask tutorial can be found on GitHub: python-sample-vscode-flask-tutorial (<https://github.com/Microsoft/python-sample-vscode-flask-tutorial>).

If you have any problems, feel free to file an issue for this tutorial in the VS Code documentation repository (<https://github.com/Microsoft/vscode-docs/issues>).

## Prerequisites

To successfully complete this Flask tutorial, you must do the following (which are the same steps as in the general Python tutorial (/docs/python/python-tutorial)):

1. Install the Python extension (<https://marketplace.visualstudio.com/items?itemName=ms-python.python>).
2. Install a version of Python 3 (for which this tutorial is written). Options include:
  - (All operating systems) A download from python.org (<https://www.python.org/downloads/>); typically use the **Download Python 3.6.5** button that appears first on the page (or whatever is the latest version).
  - (Linux) The built-in Python 3 installation works well, but to install other Python packages you must run `sudo apt install python3-pip` in the terminal.
  - (macOS) An installation through Homebrew (<https://brew.sh/>) on macOS using `brew install python3` (the system install of Python on macOS is not supported).
  - (All operating systems) A download from Anaconda (<https://www.anaconda.com/download/>) (for data science purposes).

3. On Windows, make sure the location of your Python interpreter is included in your PATH environment variable. You can check the location by running `path` at the command prompt. If the Python interpreter's folder isn't included, open Windows Settings, search for "environment", select **Edit environment variables for your account**, then edit the **Path** variable to include that folder.

## Create a project environment for the Flask tutorial

In this section, you create a virtual environment in which Flask is installed. Using a virtual environment avoids installing Flask into a global Python environment and gives you exact control over the libraries used in an application. A virtual environment also makes it easy to Create a requirements.txt file for the environment.

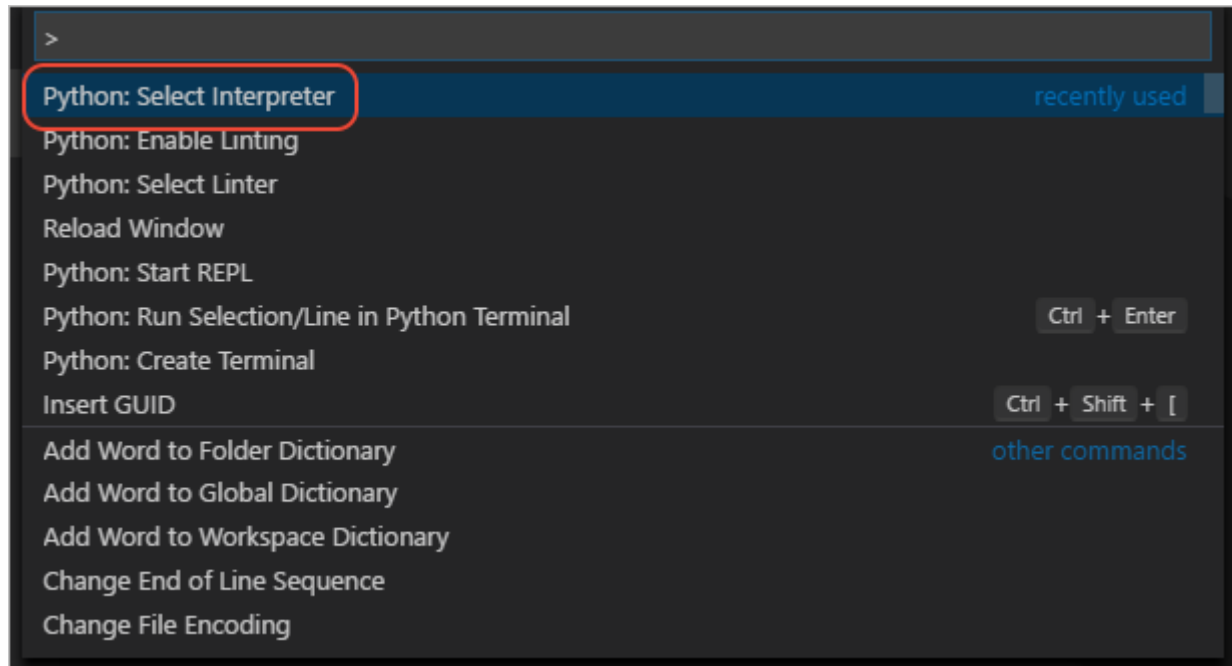
1. On your file system, create a project folder for this tutorial, such as `hello_flask`.
2. In that folder, use the following command (as appropriate to your computer) to create a virtual environment named `env` based on your current interpreter:

```
# macOS/Linux
sudo apt-get install python3-venv      # If needed
python3 -m venv env

# Windows
python -m venv env
```

**Note:** Use a stock Python installation when running the above commands. If you use `python.exe` from an Anaconda installation, you see an error because the `ensurepip` module isn't available, and the environment is left in an unfinished state.

3. Open the project folder in VS Code by running `code .` , or by running VS Code and using the **File > Open Folder** command.
4. In VS Code, open the Command Palette (**View > Command Palette** or ( `Ctrl+Shift+P` )). Then select the **Python: Select Interpreter** command:



5. The command presents a list of available interpreters that VS Code can locate automatically (your list will vary; if you don't see the desired interpreter, see [Configuring Python environments \(/docs/python/environments\)](/docs/python/environments)). From the list, select the virtual environment in your project folder that starts with `./env` or `.\env`:

```
current: C:\Python36-32\python.exe

Anaconda 4.4.0 Anaconda, Inc.
C:\Anaconda3\python.exe

Anaconda 5.0.0 Anaconda, Inc.
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Anaconda3_64\python.exe

Python 3.6 (32-bit) Python Software Foundation
C:\Python36-32\python.exe

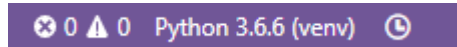
Python 3.6 (64-bit) Python Software Foundation
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe

Python 3.6.0 (venv)
.\env\Scripts\python.exe
```

6. Run **Terminal: Create New Integrated Terminal** ( `Ctrl+Shift+`` ) from the Command Palette, which creates a terminal and automatically activates the virtual environment by running its activation script.

**Note:** On Windows, if your default terminal type is PowerShell, you may see an error that it cannot run `activate.ps1` because running scripts is disabled on the system. The error provides a link for information on how to allow scripts. Otherwise, use **Terminal: Select Default Shell** to set "Command Prompt" or "Git Bash" as your default instead.

7. The selected environment appears on the left side of the VS Code status bar, and notice the "(venv)" indicator that tells you that you're using a virtual environment:



8. Install Flask in the virtual environment by running one of the following commands in the VS Code Terminal:

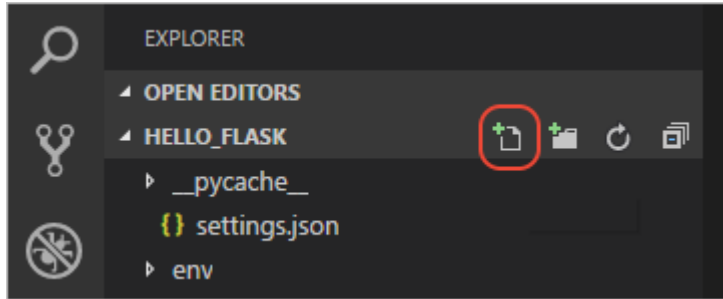
```
# macOS/Linux
pip3 install flask

# Windows
pip install flask
```

You now have a self-contained environment ready for writing Flask code. VS Code activates the environment automatically when you use **Terminal: Create New Integrated Terminal**. If you open a separate command prompt or terminal, activate the environment by running `source env/bin/activate` (Linux/macOS) or `env\scripts\activate` (Windows). You know the environment is activated when the command prompt shows **(env)** at the beginning.

## Create and run a minimal Flask app

1. In VS Code, create a new file in your project folder named `app.py` using either **File > New** from the menu, pressing `Ctrl+N`, or using the new file icon in the Explorer View (shown below).



2. In `app.py`, add code to import Flask and create an instance of the Flask object. If you type the code below (instead of using copy-paste), you can observe VS Code's IntelliSense and auto-completions ([/docs/python/editing#\\_autocomplete-and-intellisense](/docs/python/editing#_autocomplete-and-intellisense)):

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

3. Also in `app.py`, add a function that returns content, in this case a simple string, and use Flask's `app.route` decorator to map the URL route `/` to that function:

```
@app.route("/")
def home():
    return "Hello, Flask!"
```



**Tip:** You can use multiple decorators on the same function, one per line, depending on how many different routes you want to map to the same function.

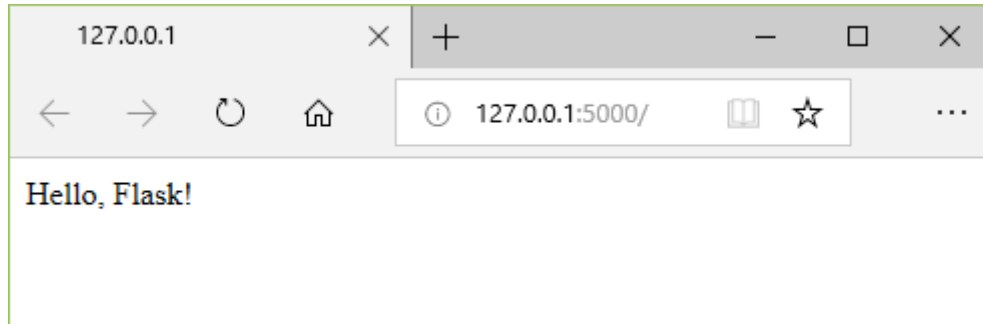
4. Save the `app.py` file ( `Ctrl+S` ).
5. In the terminal, run the app by entering `python3 -m flask run` (macOS/Linux) or `python -m flask run` (Windows), which runs the Flask development server. The development server looks for `app.py` by default. When you run Flask, you should see output similar to the following:

```
(env) D:\py\\hello_flask>python -m flask run
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

If you see an error that the Flask module cannot be found, make sure you've run `pip3 install flask` (macOS/Linux) or `pip install flask` (Windows) in your virtual environment as described at the end of the previous section.

Also, if you want to run the development server on a different IP address or port, use the host and port command-line arguments, as with `--host=0.0.0.0 --port=80`.

6. To open your default browser to the rendered page, `Ctrl+click` the `http://127.0.0.1:5000/` URL in the terminal.



7. Observe that when you visit a URL like `/`, a message appears in the debug terminal showing the HTTP request:

```
127.0.0.1 - - [11/Jul/2018 08:40:15] "GET / HTTP/1.1" 200 -
```

8. Stop the app by using `Ctrl+C` in the terminal.

**Tip:** If you want to use a different filename than `app.py`, such as `program.py`, define an environment variable named `FLASK_APP` and set its value to your chosen file. Flask's development server then uses the value of `FLASK_APP` instead of the default file `app.py`. For more information, see Flask command line interface (<http://flask.pocoo.org/docs/1.0/cli/>).

## Run the app in the debugger

Debugging gives you the opportunity to pause a running program on a particular line of code. When a program is paused, you can examine variables, run code in the Debug Console panel, and otherwise take advantage of the features described on [Debugging \(/docs/python/debugging\)](/docs/python/debugging). Running the debugger also automatically saves any modified files before the debugging session begins.

**Before you begin:** Make sure you've stopped the running app at the end of the last section by using `Ctrl+C` in the terminal. If you leave the app running in one terminal, it continues to own the port. As a result, when you run the app in the debugger using the same port, the original running app handles all the requests and you won't see any activity in the app being debugged and the program won't stop at breakpoints. In other words, if the debugger doesn't seem to be working, make sure that no other instance of the app is still running.

1. Replace the contents of `app.py` with the following code, which adds a second route and function that you can step through in the debugger:

```
from flask import Flask
from datetime import datetime
import re

app = Flask(__name__)

@app.route("/")
def home():
    return "Hello, Flask!"

@app.route("/hello/<name>")
def hello_there(name):
    now = datetime.now()
    formatted_now = now.strftime("%A, %d %B, %Y at %X")

    # Filter the name argument to letters only using regular expressions. URL arguments
    # can contain arbitrary text, so we restrict to safe characters only.
    match_object = re.match("[a-zA-Z]+", name)

    if match_object:
        clean_name = match_object.group(0)
    else:
        clean_name = "Friend"
```

```
content = "Hello there, " + clean_name + "! It's " + formatted_now
return content
```

The decorator used for the new URL route, `/hello/<name>`, defines an endpoint `/hello/` that can accept any additional value. The identifier inside `<` and `>` in the route defines a variable that is passed to the function and can be used in your code.

URL routes are case-sensitive. For example, the route `/hello/<name>` is distinct from `/Hello/<name>`. If you want the same function to handle both, use decorators for each variant.

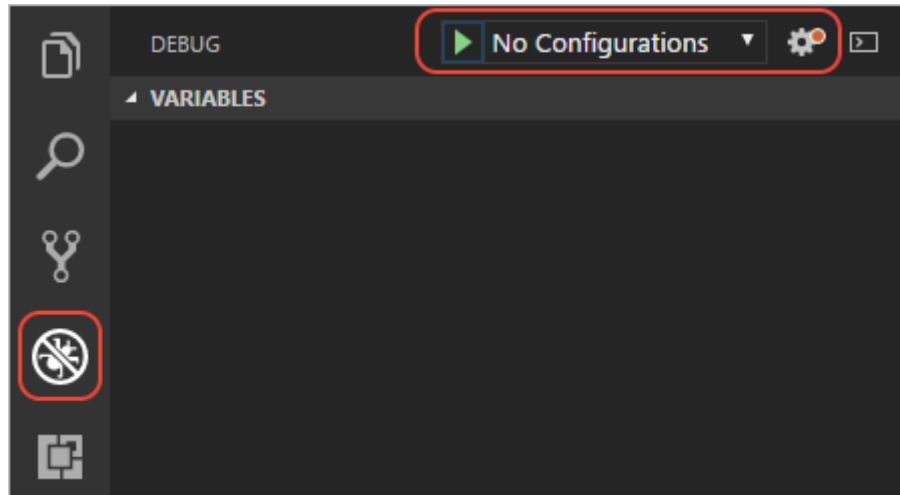
As described in the code comments, always filter arbitrary user-provided information to avoid various attacks on your app. In this case, the code filters the name argument to contain only letters, which avoids injection of control characters, HTML, and so forth. (When you use templates in the next section, Flask does automatic filtering and you won't need this code.)

2. Set a breakpoint at the first line of code in the `hello_there` function (`now = datetime.now()`) by doing any one of the following:
  - With the cursor on that line, press `F9`, or,
  - With the cursor on that line, select the **Debug > Toggle Breakpoint** menu command, or,
  - Click directly in the margin to the left of the line number (a faded red dot appears when hovering there).

The breakpoint appears as a red dot in the left margin:

```
11 @app.route('/hello/<name>')
12 def hello_there(name):
13     now = datetime.now()
14     formatted_now = now.strftime("%A, %d %B, %Y at %X")
15
16     # Filter the name argument to letters only using regular
17     # can contain arbitrary text, so we restrict to safe char
18     match_object = re.match("[a-zA-Z]+", name)
19
```

3. Switch to **Debug** view in VS Code (using the left-side activity bar). Along the top of the Debug view, you may see "No Configurations" and a warning dot on the gear icon. Both indicators mean that you don't yet have a `launch.json` file containing debug configurations:



4. Select the gear icon and select **Python** from the list that appears. VS Code creates and opens a `launch.json` file. This JSON file contains a number of debugging configurations, each of which is a separate JSON object within the `configuration` array.
5. Scroll down to and examine the configuration with the name "Python: Flask (0.11.x or later)". This configuration contains `"module": "flask"`, which tells VS Code to run Python with `-m flask` when it starts the debugger. It also defines the `FLASK_APP` environment variable in the `env` property to identify the startup file, which is `app.py` by default, but allows you to easily specify a different file. If you want to change the host and/or port, you can use the `args` array.

```
{
  "name": "Python: Flask (0.11.x or later)",
  "type": "python",
  "request": "launch",
  "module": "flask",
  "env": {
    "FLASK_APP": "app.py"
  },
  "args": [
    "run",
    "--no-debugger",
    "--no-reload"
  ]
},
```

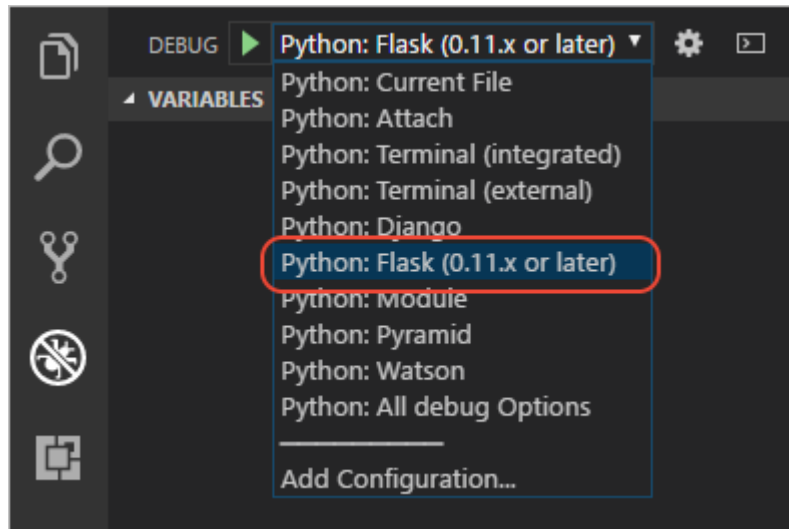
**Note:** If the `env` entry in your configuration contains `"FLASK_APP":`

`"${workspaceFolder}/app.py"`, change it to `"FLASK_APP": "app.py"` as shown above.

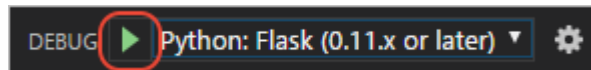
Otherwise you may encounter error messages like "Cannot import module C" where C is the drive letter where your project folder resides.

6. Save `launch.json` (Ctrl+S). In the debug configuration drop-down list (which reads **Python: Current File**) select the **Python: Flask (0.11.x or later)** configuration.





7. Start the debugger by selecting the **Debug > Start Debugging** menu command, or selecting the green **Start Debugging** arrow next to the list ( `F5` ):



Observe that the status bar changes color to indicate debugging:



A debugging toolbar (shown below) also appears in VS Code containing commands in the following order: Pause (or Continue, `F5`), Step Over (`F10`), Step Into (`F11`), Step Out (`Shift+F11`), Restart (`Ctrl+Shift+F5`), and Stop (`Shift+F5`). See VS Code debugging (</docs/editor/debugging>) for a description of each command.

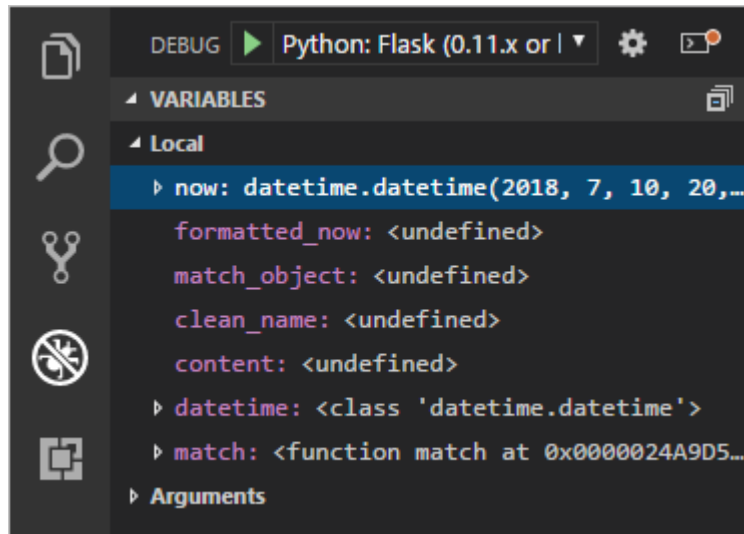


8. Output appears in a "Python Debug Console" terminal. `Ctrl+click` the `http://127.0.0.1:5000/` link in that terminal to open a browser to that URL. In the browser's address bar, navigate to `http://127.0.0.1:5000/hello/VSCode`. Before the page renders, VS Code pauses the program at the breakpoint you set. The small yellow arrow on the breakpoint indicates that it's the next line of code to run.

```
11 @app.route('/hello/<name>')
12 def hello_there(name):
13     now = datetime.now()
14     formatted_now = now.strftime("%A, %d %B, %Y at %X")
15
16     # Filter the name argument to letters only using regular
17     # can contain arbitrary text, so we restrict to safe char
18     match_object = re.match("[a-zA-Z]+", name)
19
```

9. Use Step Over to run the `now = datetime.now()` statement.

10. On the left side of the VS Code window, you see a **Variables** pane that shows local variables, such as `now`, as well as arguments, such as `name`. Below that are panes for **Watch**, **Call Stack**, and **Breakpoints** (see VS Code debugging (/docs/editor/debugging) for details). In the **Locals** section, try expanding different values. You can also double-click values (or use `F2`) to modify them. Changing variables such as `now`, however, can break the program. Developers typically make changes only to correct values when the code didn't produce the right value to begin with.



11. When a program is paused, the **Debug Console** panel (which is different from the "Python Debug Console" in the Terminal panel) lets you experiment with expressions and try out bits of code using the current state of the program. For example, once you've stepped over the line `now = datetime.now()`, you might experiment with different date/time formats. In the editor, select the code that reads `now.strftime("%A, %d %B, %Y at %X")`, then right-click and select **Debug: Evaluate** to send that code to the debug console, where it runs:

```
now.strftime("%A, %d %B, %Y at %X")
'Wednesday, 31 October, 2018 at 18:13:39'
```

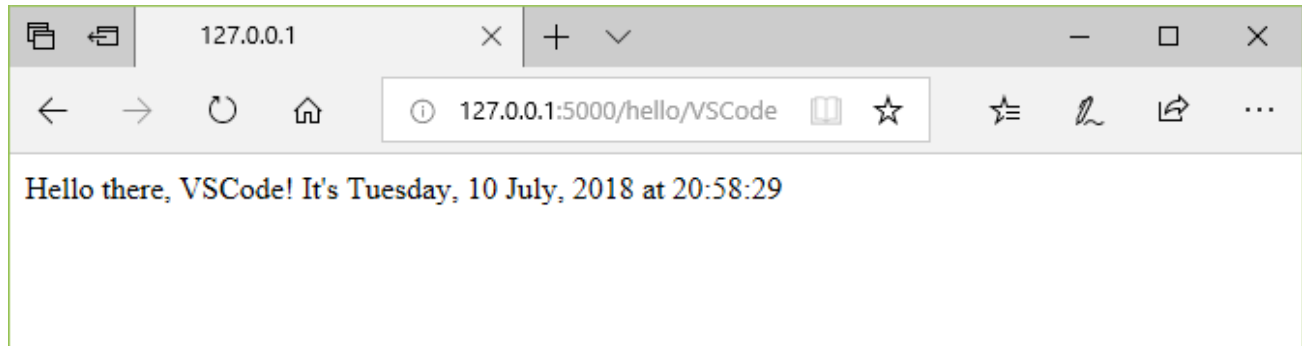
**Tip:** The **Debug Console** also shows exceptions from within the app that may not appear in the terminal. For example, if you see a "Paused on exception" message in the **Call Stack** area of Debug View, switch to the **Debug Console** to see the exception message.

12. Copy that line into the `>` prompt at the bottom of the debug console, and try changing the formatting:

```
now.strftime("%a, %d %B, %Y at %X")
'Wed, 31 October, 2018 at 18:13:39'
now.strftime("%a, %d %b, %Y at %X")
'Wed, 31 Oct, 2018 at 18:13:39'
now.strftime("%a, %d %b, %y at %X")
'Wed, 31 Oct, 18 at 18:13:39'
```

**Note:** If you see a change you like, you can copy and paste it into the editor during a debugging session. However, those changes aren't applied until you restart the debugger.

13. Step through a few more lines of code, if you'd like, then select Continue ( `F5` ) to let the program run. The browser window shows the result:



14. Close the browser and stop the debugger when you're finished. To stop the debugger, use the Stop toolbar button (the red square) or the **Debug > Stop Debugging** command ( `Shift+F5` ).

**Tip:** To make it easier to repeatedly navigate to a specific URL like `http://127.0.0.1:5000/hello/VSCoDe`, output that URL using a `print` statement. The URL appears in the terminal where you can use `Ctrl+click` to open it in a browser.

## Go to Definition and Peek Definition commands

During your work with Flask or any other library, you may want to examine the code in those libraries themselves. VS Code provides two convenient commands that navigate directly to the definitions of classes and other objects in any code:

- **Go to Definition** jumps from your code into the code that defines an object. For example, in `app.py`, right-click on the `Flask` class (in the line `app = Flask(__name__)`) and select **Go to Definition** (or use `F12`), which navigates to the class definition in the Flask library.
- **Peek Definition** (`Ctrl+Shift+F10`, also on the right-click context menu), is similar, but displays the class definition directly in the editor (making space in the editor window to avoid obscuring any code). Press `Escape` to close the Peek window or use the `x` in the upper right corner.

```
6 app = Flask(__name__)

app.py env\lib\site-packages\flask
class Flask(_PackageBoundObject):
    """The flask object implements a WSGI application and acts as
    the central registry. It is passed the name of the module or package of the
    application. Once it is created it will act as a central registry for
    the view functions, the URL rules, template configuration and
    much more.

    """
    modules,
    'database models and everything related at a central
    place '
    'before the application starts serving requests.')
    return f(self, *args, **kwargs)
    return update_wrapper(wrapper_func, f)

70 class Flask(_PackageBoundObject):
71     """The flask object implements a WSGI application and acts as
72     the central
73     object. It is passed the name of the module or package of the
74     application. Once it is created it will act as a central
75     registry for
76     the view functions, the URL rules, template configuration and
77     much more.
78
79
80 @app.route('/')
81 def home():
82     return 'Hello, Flask!'

10
```

Use a template to render a page

The app you've created so far in this tutorial generates only plain text web pages from Python code. Although it's possible to generate HTML directly in code, developers avoid such a practice because it opens the app to cross-site scripting (XSS) attacks (<http://flask.pocoo.org/docs/1.0/security/#cross-site-scripting-xss>). In the `hello_there` function of this tutorial, for example, one might think to format the output in code with something like `content = "<h1>Hello there, " + clean_name + "!</h1>`, where the result in `content` is given directly to a browser. This opening allows an attacker to place malicious HTML, including JavaScript code, in the URL that ends up in `clean_name` and thus ends up being run in the browser.

A much better practice is to keep HTML out of your code entirely by using **templates**, so that your code is concerned only with data values and not with rendering.

- A template is an HTML file that contains placeholders for values that the code provides at run time. The templating engine takes care of making the substitutions when rendering the page. The code, therefore, concerns itself only with data values and the template concerns itself only with markup.
- The default templating engine for Flask is Jinja (<http://jinja.pocoo.org/>), which is installed automatically when you install Flask. This engine provides flexible options including automatic escaping (to prevent XSS attacks) and template inheritance. With inheritance, you can define a base page with common markup and then build upon that base with page-specific additions.

In this section, you create a single page using a template. In the sections that follow, you configure the app to serve static files, and then create multiple pages to the app that each contains a nav bar from a base template.



1. Inside the `hello_flask` folder, create a folder named `templates`, which is where Flask looks for templates by default.
2. In the `templates` folder, create a file named `hello_there.html` with the contents below. This template contains two placeholders named "name" and "date", which are delineated by pairs of curly braces, `{{` and `}}`. As you can see, you can also include formatting code in the template directly:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Hello, Flask</title>
  </head>
  <body>
    {%if name %}
      <strong>Hello there, {{ name }}!</strong> It's {{ date.strftime("%A,
%d %B, %Y at %X") }}.
    {% else %}
      What's your name? Provide it after /hello/ in the URL.
    {% endif %}
  </body>
</html>
```

**Tip:** Flask developers often use the flask-babel (<https://pythonhosted.org/Flask-Babel/>) extension for date formatting, rather than `strftime`, as flask-babel takes locales and timezones into consideration.

3. In `app.py`, import Flask's `render_template` function near the top of the file:

```
from flask import render_template
```

4. Also in `app.py`, modify the `hello_there` function to use `render_template` to load a template and apply the named values (and add a route to recognize the case without a name). `render_template` assumes that the first argument is relative to the `templates` folder. Typically, developers name the templates the same as the functions that use them, but matching names are not required because you always refer to the exact filename in your code.

```
@app.route("/hello/")
@app.route("/hello/<name>")
def hello_there(name = None):
    return render_template(
        "hello_there.html",
        name=name,
        date=datetime.now()
    )
```

You can see that the code is now much simpler, and concerned only with data values, because the markup and formatting is all contained in the template.

5. Start the program (inside or outside of the debugger, using `Ctrl+F5`), navigate to a `/hello/name` URL, and observe the results.

6. Also try navigating to a `/hello/name` URL using a name like

`<a%20value%20that%20could%20be%20HTML>` to see Flask's automatic escaping at work. The "name" value shows up as plain text in the browser rather than as rendering an actual element.

## Serve static files

Static files are of two types. First are those files like stylesheets to which a page template can just refer directly. Such files can live in any folder in the app, but are commonly placed within a `static` folder.

The second type are those that you want to address in code, such as when you want to implement an API endpoint that returns a static file. For this purpose, the Flask object contains a built-in method, `send_static_file`, which generates a response with a static file contained within the app's `static` folder.

The following sections demonstrate both types of static files.

Refer to static files in a template

1. In the `hello_flask` folder, create a folder named `static`.

2. Within the `static` folder, create a file named `site.css` with the following contents. After entering this code, also observe the syntax highlighting that VS Code provides for CSS files, including a color preview:

```
.message {  
    font-weight: 600;  
    color: blue;  
}
```

3. In `templates/hello_there.html`, add the following line before the `</head>` tag, which creates a reference to the stylesheet.

```
<link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='site.css')}}" />
```

Flask's `url_for` tag ([http://flask.pocoo.org/docs/0.12/api/#flask.url\\_for](http://flask.pocoo.org/docs/0.12/api/#flask.url_for)) that's used here creates the appropriate path to the file. Because it can accept variables as arguments, `url_for` allows you to programmatically control the generated path, if desired.

4. Also in `templates/hello_there.html`, replace the contents `<body>` element with the following markup that uses the `message` style instead of a `<strong>` tag (and also displays a message if you just use a hello/ URL without a name):

```
{%if name %}  
    <span class="message">Hello there, {{ name }}!</span> It's {{ date.strftime  
("%A, %d %B, %Y at %X") }}.  
{% else %}  
    <span class="message">What's your name? Provide it after /hello/ in the URL.  
</span>  
{% endif %}
```

5. Run the app, navigate to a `/hello/name` URL, and observe that the message renders in blue. Stop the app when you're done.

Serve a static file from code

1. In the `static` folder, create a JSON data file named `data.json` with the following contents (which are meaningless sample data):

```
{  
    "01": {  
        "note" : "This data is very simple because we're demonstrating only the  
mechanism."  
    }  
}
```

2. In `app.py`, add a function with the route `/api/data` that returns the static data file using the `send_static_file` method:

```
@app.route("/api/data")
def get_data():
    return app.send_static_file("data.json")
```

3. Run the app and navigate to the `/api/data` endpoint to see that the static file is returned. Stop the app when you're done.

## Create multiple templates that extend a base template

Because most web apps have more than one page, and because those pages typically share many common elements, developers separate those common elements into a base page template that other page templates can then extend. (This is also called template inheritance.)

Also, because you'll likely create many pages that extend the same template, it's helpful to create a code snippet in VS Code with which you can quickly initialize new page templates. A snippet helps you avoid tedious and error-prone copy-paste operations.

The following sections walk through different parts of this process.

Create a base page template and styles

A base page template in Flask contains all the shared parts of a set of pages, including references to CSS files, script files, and so forth. Base templates also define one or more **block** tags that other templates that extend the base are expected to override. A block tag is delineated by `{% block <name> %}` and `{% endblock %}` in both the base template and extended templates.

The following steps demonstrate creating a base template.

1. In the `templates` folder, create a file named `layout.html` with the contents below, which contains blocks named "title" and "content". As you can see, the markup defines a simple nav bar structure with links to Home, About, and Contact pages, which you create in a later section. Each link again uses Flask's `url_for` tag to generate a link at runtime for the matching route.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>{% block title %}{% endblock %}</title>
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='site.css')}}" />
  </head>

  <body>
    <div class="navbar">
      <a href="{{ url_for('home') }}" class="navbar-brand">Home</a>
      <a href="{{ url_for('about') }}" class="navbar-item">About</a>
      <a href="{{ url_for('contact') }}" class="navbar-item">Contact</a>
    </div>

    <div class="body-content">
      {% block content %}
      {% endblock %}
      <hr/>
      <footer>
        <p>&copy; 2018</p>
      </footer>
    </div>
  </body>
</html>
```



- 
2. Add the following styles to `static/site.css` below the existing "message" style, and save the file. (This walkthrough doesn't attempt to demonstrate responsive design; these styles simply generate a reasonably interesting result.)

```
.navbar {
  background-color: lightslategray;
  font-size: 1em;
  font-family: 'Trebuchet MS', 'Lucida Sans Unicode', 'Lucida Grande', 'Lucida
Sans', Arial, sans-serif;
  color: white;
  padding: 8px 5px 8px 5px;
}

.navbar a {
  text-decoration: none;
  color: inherit;
}

.navbar-brand {
  font-size: 1.2em;
  font-weight: 600;
}

.navbar-item {
  font-variant: small-caps;
  margin-left: 30px;
}

.body-content {
  padding: 5px;
```

```
font-family:'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}
```

You can run the app at this point, but because you haven't made use of the base template anywhere and haven't changed any code files, the result is the same as the previous step. Complete the remaining sections to see the final effect.

Create a code snippet

Because the three pages you create in the next section extend `layout.html`, it saves time to create a **code snippet** to initialize a new template file with the appropriate reference to the base template. A code snippet provides a consistent piece of code from a single source, which avoids errors that can creep in when using copy-paste from existing code.

1. In VS Code, select the **File** (Windows/Linux) or **Code** (macOS), menu, then select **Preferences > User snippets**.
2. In the list that appears, select **html**. (The option may appear as "html.json" in the **Existing Snippets** section of the list if you've created snippets previously.)
3. After VS code opens `html.json`, add the following entry within the existing curly braces (the explanatory comments, not shown here, describe details such as how the `$0` line indicates where VS Code places the cursor after inserting a snippet):

```
"Flask Tutorial: template extending layout.html": {
  "prefix": "flectlayout",
  "body": [
    "{% extends \"layout.html\" %}",
    "{% block title %}",
    "$0",
    "{% endblock %}",
    "{% block content %}",
    "{% endblock %}"
  ],
  "description": "Boilerplate template that extends layout.html"
},
```

4. Save the `html.json` file (Ctrl+S).

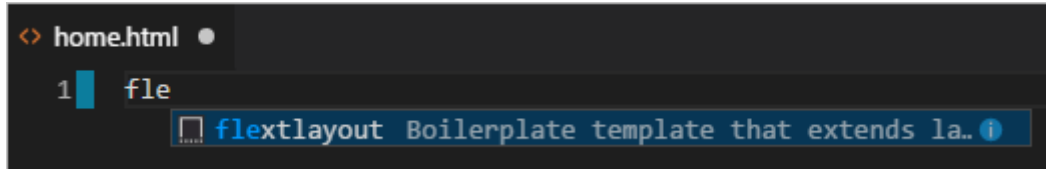
5. Now, whenever you start typing the snippet's prefix, such as `flect`, VS Code provides the snippet as an autocomplete option, as shown in the next section. You can also use the **Insert Snippet** command to choose a snippet from a menu.

For more information on code snippets in general, refer to [Creating snippets \(/docs/editor/userdefinedsnippets\)](/docs/editor/userdefinedsnippets).

Use the code snippet to add pages

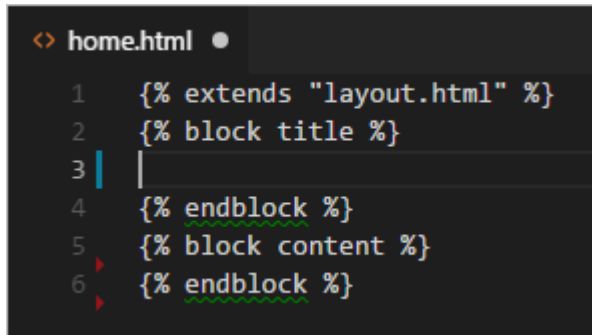
With the code snippet in place, you can quickly create templates for the Home, About, and Contact pages.

1. In the `templates` folder, create a new file named `home.html` , Then start typing `flex` to see the snippet appear as a completion:



A screenshot of the Visual Studio Code editor. The top bar shows a file named 'home.html'. The editor window shows line 1 with the text 'fle'. A dropdown menu is open, showing a snippet 'flexlayout Boilerplate template that extends la..' with a small icon to the left and an information icon to the right.

When you select the completion, the snippet's code appears with the cursor on the snippet's insertion point:



A screenshot of the Visual Studio Code editor. The top bar shows a file named 'home.html'. The editor window shows the following code:

```
1  {% extends "layout.html" %}
2  {% block title %}
3  |
4  {% endblock %}
5  {% block content %}
6  {% endblock %}
```

The cursor is positioned at the end of line 3, which is the insertion point for the snippet.

2. At the insertion point in the "title" block, write `Home` , and in the "content" block, write `<p>Home page for the Visual Studio Code Flask tutorial.</p>` , then save the file. These lines are the only unique parts of the extended page template:

3. In the `templates` folder, create `about.html` , use the snippet to insert the boilerplate markup, insert `About us` and `<p>About page for the Visual Studio Code Flask tutorial.</p>` in the "title" and "content" blocks, respectively, then save the file.
4. Repeat the previous step to create `templates/contact.html` using `Contact us` and `<p>Contact page for the Visual Studio Code Flask tutorial.</p>` in the two content blocks.
5. In `app.py` , add functions for the `/about/` and `/contact/` routes that refer to their respective page templates. Also modify the `home` function to use the `home.html` template.

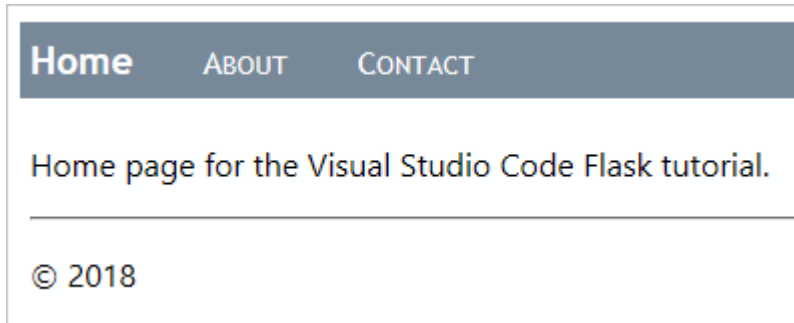
```
# Replace the existing home function with the one below
@app.route("/")
def home():
    return render_template("home.html")

# New functions
@app.route("/about/")
def about():
    return render_template("about.html")

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

Run the app

With all the page templates in place, save `app.py`, run the app, and open a browser to see the results. Navigate between the pages to verify that the page templates are properly extending the base template.



## Optional activities

The following sections describe additional steps that you might find helpful in your work with Python and Visual Studio Code.

Create a `requirements.txt` file for the environment

When you share your app code through source control or some other means, it doesn't make sense to copy all the files in a virtual environment because recipients can always recreate the environment themselves.

Accordingly, developers typically omit the virtual environment folder from source control and instead describe the app's dependencies using a `requirements.txt` file.

Although you can create the file by hand, you can also use the `pip freeze` command to generate the file based on the exact libraries installed in the activated environment:

1. With your chosen environment selected using the **Python: Select Interpreter** command, run the **Terminal: Create New Integrated Terminal** command ( `Ctrl+Shift+`` ) to open a terminal with that environment activated.
2. In the terminal, run `pip freeze > requirements.txt` to create the `requirements.txt` file in your project folder.

Anyone (or any build server) that receives a copy of the project needs only to run the `pip install -r requirements.txt` command to reinstall the packages in the original the environment. (The recipient still needs to create their own virtual environment, however.)

**Note:** `pip freeze` lists all the Python packages you have installed in the current environment, including packages you aren't currently using. The command also lists packages with exact version numbers, which you might want to convert to ranges for more flexibility in the future. For more information, see Requirements files ([https://pip.readthedocs.io/en/stable/user\\_guide/#requirements-files](https://pip.readthedocs.io/en/stable/user_guide/#requirements-files)) in the pip command documentation.

Refactor the project to support further development



Throughout this Flask tutorial, all the app code is contained in a single `app.py` file. To allow for further development and to separate concerns, it's helpful to refactor the pieces of `app.py` into separate files.

1. In your project folder, create a folder for the app, such as `hello_app`, to separate its files from other project-level files like `requirements.txt` and the `.vscode` folder where VS Code stores settings and debug configuration files.
2. Move the `static` and `templates` folders into `hello_app`, because these folders certainly contain app code.
3. In the `hello_app` folder, create a file named `views.py` that contains the routings and the view functions:

```
from flask import Flask
from flask import render_template
from datetime import datetime
from . import app

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

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

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

@app.route("/hello/")
@app.route("/hello/<name>")
def hello_there(name):
    return render_template(
        "hello_there.html",
        name=name,
        date=datetime.now()
    )
```

```
@app.route("/api/data")
def get_data():
    return app.send_static_file("data.json")
```

4. Optional: Right-click in the editor and select the **Sort Imports** command, which consolidates imports from identical modules, removes unused imports, and sorts your import statements. Using the command on the code above in `views.py` changes the imports as follows (you can remove the extra lines, of course):

```
from datetime import datetime

from flask import Flask, render_template

from . import app
```

5. In the `hello_app` folder, create a file `__init__.py` with the following contents:

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

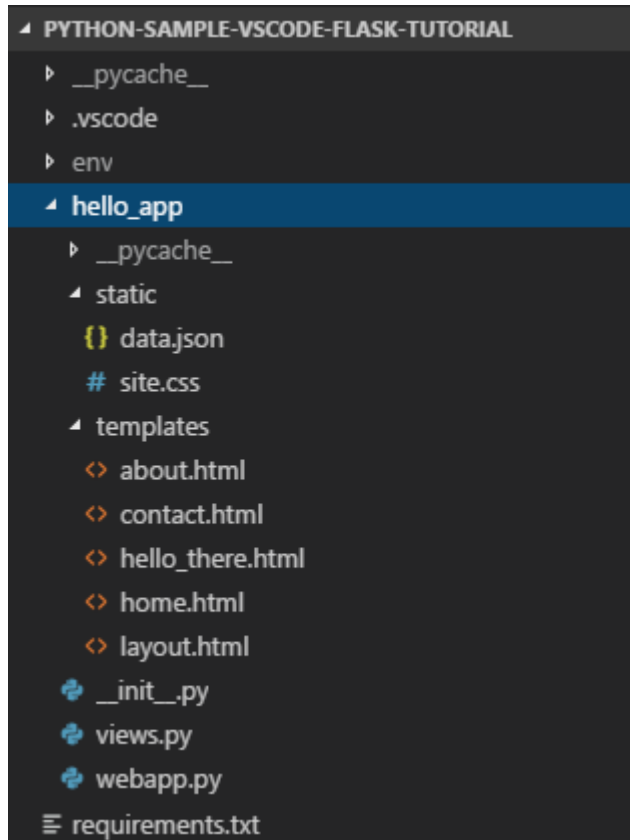
6. In the `hello_app` folder, create a file `webapp.py` with the following contents:

```
# Entry point for the application.  
from . import app    # For application discovery by the 'flask' command.  
from . import views  # For import side-effects of setting up routes.
```

7. Open the debug configuration file `launch.json` and update the `env` property as follows to point to the startup object:

```
"env": {  
    "FLASK_APP": "hello_app.webapp"  
},
```

8. Delete the original `app.py` file in the project root, as its contents have been moved into other app files.
9. Your project's structure should now be similar to the following:



10. Run the app in the debugger again to make sure everything works. To run the app outside of the VS Code debugger, use the following steps:

1. Set an environment variable for `FLASK_APP`. On Linux and macOS, use `export set FLASK_APP=webapp`; on Windows use `set FLASK_APP=webapp`.
2. In the `hello_app` folder, launch the program using `python3 -m flask run` (Linux/macOS) or `python -m flask run` (Windows).

If you have any problems, feel free to file an issue for this tutorial in the VS Code documentation repository (<https://github.com/Microsoft/vscode-docs/issues>).

## Next steps

Congratulations on completing this walkthrough of working with Flask in Visual Studio Code!

The completed code project from this tutorial can be found on GitHub: `python-sample-vscode-flask-tutorial` (<https://github.com/Microsoft/python-sample-vscode-flask-tutorial>).

Because this tutorial has only scratched the surface of page templates, refer to the Jinja2 documentation (<http://jinja.pocoo.org/docs/>) for more information about templates. The Template Designer Documentation (<http://jinja.pocoo.org/docs/templates/#synopsis>) contains all the details on the template language. You might also want to review the official Flask tutorial (<http://flask.pocoo.org/docs/1.0/tutorial/>).

To try your app on a production website, check out the tutorial `Deploy Python apps to Azure App Service using Docker Containers` (</docs/python/tutorial-deploy-containers>). Azure also offers a standard container, `App Service on Linux (Preview)` (</docs/python/tutorial-deploy-app-service-on-linux>), to which you deploy web apps from within VS Code.

You may also want to review the following articles in the VS Code docs that are relevant to Python:

- Editing Python code (/docs/python/editing)
- Linting (/docs/python/linting)
- Managing Python environments (/docs/python/environments)
- Debugging Python (/docs/python/debugging)
- Unit testing (/docs/python/unit-testing)

If you encountered any problems in the course of this tutorial, feel free to file an issue in the VS Code documentation repository (<https://github.com/Microsoft/vscode-docs/issues>).

**Was this documentation helpful?**

Yes

No

Hello from Seattle. Follow @code ()

Star

71,227

Support ([https://support.microsoft.com/en-us/getsupport?](https://support.microsoft.com/en-us/getsupport?wf=0&tenant=ClassicCommercial&oaspworkflow=start_1.0.0.0&locale=en-us&supportregion=en-us&pesid=16064&ccsid=636196895839595242)

[wf=0&tenant=ClassicCommercial&oaspworkflow=start\\_1.0.0.0&locale=en-us&supportregion=en-us&pesid=16064&ccsid=636196895839595242](https://support.microsoft.com/en-us/getsupport?wf=0&tenant=ClassicCommercial&oaspworkflow=start_1.0.0.0&locale=en-us&supportregion=en-us&pesid=16064&ccsid=636196895839595242))

Privacy (<https://privacy.microsoft.com/en-us/privacystatement>)

Terms of Use (<https://www.microsoft.com/en-us/legal/intellectualproperty/copyright/default.aspx>)

License (/License)

 Microsoft (<https://www.microsoft.com>)

© 2019 Microsoft