## Create a project environment for Flask[#](https://code.visualstudio.com/docs/python/tutorial-flask#_create-a-project-environment-for-flask)

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](https://code.visualstudio.com/docs/python/tutorial-flask#_create-a-requirementstxt-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

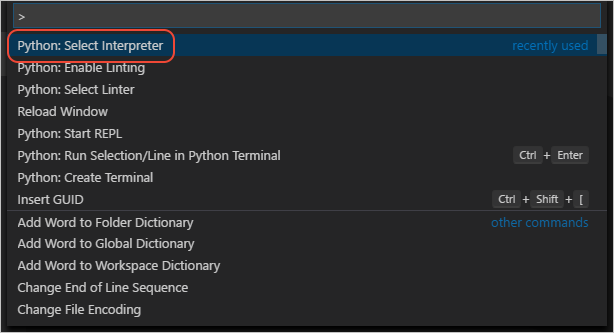
python3 -m venv env

# Windows

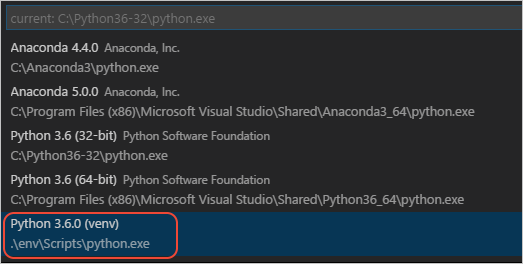
py -3 -m venv env

**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.

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



1. 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](https://code.visualstudio.com/docs/python/environments)). From the list, select the virtual environment in your project folder that starts with ./env or .\env:



1. Run **Python: Create Terminal** 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.

1. 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:

Selected environment showing in the VS Code status bar

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

# macOS/Linux

pip3 install flask

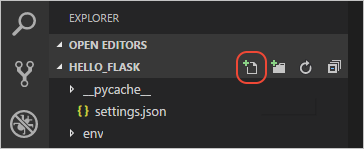
# Windows

pip install flask

You now have an self-contained environment ready for writing Flask code. VS Code activates the environment automatically when you use **Python: Create 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[#](https://code.visualstudio.com/docs/python/tutorial-flask#_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).



1. 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](https://code.visualstudio.com/docs/python/editing#_autocomplete-and-intellisense):

from flask import Flask

app = Flask(\_\_name\_\_)

1. 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.

1. Save the app.py file (Ctrl+S).
2. 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.

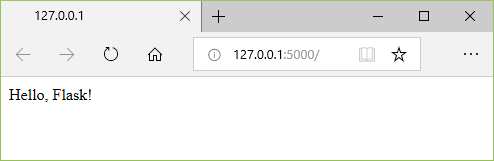
\* Debug mode: off

\* 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.

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



1. 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 -

1. 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[#](https://code.visualstudio.com/docs/python/tutorial-flask#_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](https://code.visualstudio.com/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

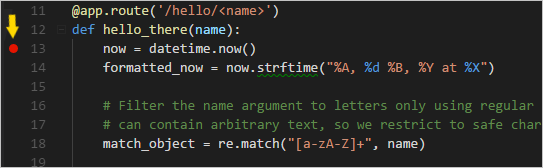
The decorator used for the new URL route, /hello/<name>, defines an endpoint /hello/ that can accept any additional value. The identifier inside < and > in the route defines a variable that is passed to the function and can be used in your code.

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.

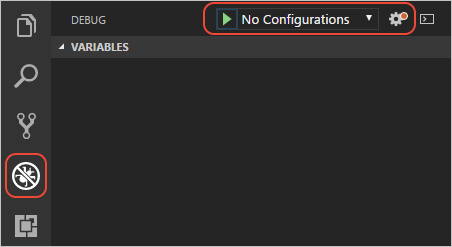
As described in the code comments, always filter arbitrary user-provided information to avoid various attacks on your app. In this case, the code filters the name argument to contain only letters, which avoids injection of control characters, HTML, and so forth. (When you use templates in the next section,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

1. 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:



1. 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:



1. 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.
2. 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 flaskwhen it starts the debugger. It also defines the FLASK\_APP environment variable in the envproperty 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",

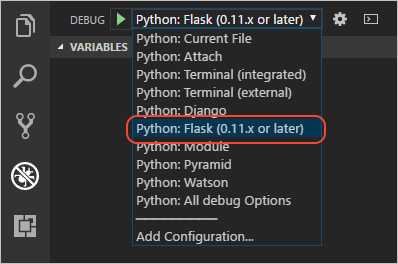
"--no-reload"

]

},

**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.

1. 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 .



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

Start debugging/continue arrow on the debug toolbar

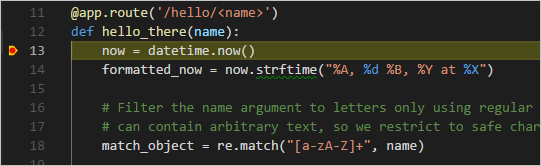
Observe that the status bar changes color to indicate debugging:

Appearance of the debugging status bar

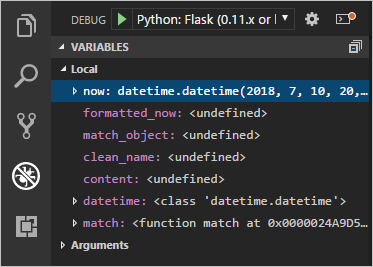
A debugging toolbar (shown below) also appears in VS Code containing commands in the following order: Pause (or Continue, F5), Step Over (F10), Step Into (F11), Step Out (Shift+F11), Restart (Ctrl+Shift+F5), and Stop (Shift+F5). See [VS Code debugging](https://code.visualstudio.com/docs/editor/debugging)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

The VS Code debug toolbar

1. 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.



1. Use Step Over to run the now = datetime.now() statement.
2. 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](https://code.visualstudio.com/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.



1. 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")

'Thursday, 24 May, 2018 at 14:35:27'

**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.

1. 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")

'Thu, 24 May, 2018 at 14:35:27'

now.strftime("%a, %d %b, %Y at %X")

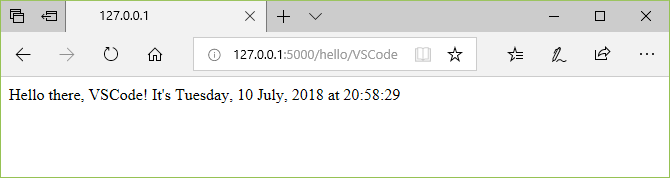
'Thu, 24 May, 2018 at 14:35:27'

now.strftime("%a, %d %b, %y at %X")

'Thu, 24 May, 18 at 14:35:27'

**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.

1. 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:



1. 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.

## Use a template to render a page[#](https://code.visualstudio.com/docs/python/tutorial-flask#_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 typically avoid such a practice because it's vulnerable to cross-site scripting (XSS) attacks. Instead, developers separate HTML markup from the code-generated data that gets inserted into that markup. **Templates** are a common approach to achieve this separation.

* 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, which is installed automatically when you install Flask. This engine provides flexible options including 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 contain 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 "title" and "content", which are delineated by pairs of curly braces, {{ and }}.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>{{ title }}</title>

</head>

<body>

{{ content }}

</body>

</html>

1. In app.py, import Flask's render\_template function near the top of the file:

from flask import render\_template

1. Also in app.py, modify the hello\_there function to use render\_template to load a template and apply the named values. 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/<name>")

def hello\_there(name):

now = datetime.now()

formatted\_now = now.strftime("%A, %d %B, %Y at %X")

# BAD CODE! Avoid inline HTML for security reason, plus templates automatically escape HTML content.

content = "<strong>Hello there, " + name + "!</strong> It's " + formatted\_now

return render\_template(

"hello\_there.html",

title='Hello, Flask',

content=content

)

1. Start the program (inside or outside of the debugger, using Ctrl+F5), navigate to a /hello/name URL, and observe the results. Notice that the inline HTML, if you happen to write bad code like this, doesn't get rendered as HTML because the templating engine automatically escapes values used in placeholders. Automatic escaping prevent accidental vulnerabilities to injection attacks: developers often gather input from one page, or the URL, and use it as a value in another page through a template placeholder. Escaping also serves as a reminder that it's again best to keep HTML out of the code entirely.

For this reason, modify the template and view function as follows to make each piece of content more specifically. While you're at it, also move more of the text (including the title) and formatting concerns into the template:

In templates/hello\_there.html:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Hello, Flask</title>

</head>

<body>

<strong>Hello there, {{ name }}!</strong> It's {{ date.strftime("%A, %d %B, %Y at %X") }}.

</body>

</html>

In app.py:

@app.route("/hello/<name>")

def hello\_there(name):

return render\_template(

"hello\_there.html",

name=name,

date=datetime.now()

)

**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.

1. Run the app again and navigate to a /hello/name URL to observe the expected result, then stop the app when you're done.

## Serve static files[#](https://code.visualstudio.com/docs/python/tutorial-flask#_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 staticfolder.

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 provide for CSS files, including a color preview:

.message {

font-weight: 600;

color: blue;

}

1. 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/" \l "flask.url_for" \t "_blank) 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.

1. 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:

<span class="message">Hello, there {{ name }}!</span>. It's {{ date.strftime("%A, %d %B, %Y at %X") }}.

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