



Turning a Python script into a website

One question we often hear from people starting out with PythonAnywhere is "how do I turn this script I've written into a website so that other people can run it?"

That's actually a bigger topic than you might imagine, and a complete answer would wind up having to explain almost everything about web development. So we won't do all of that in this blog post :) The good news is that simple scripts can often be turned into simple websites pretty easily, and in this blog post we'll work through a couple of examples.

Let's get started!

The simplest case: a script that takes some inputs and returns an output

Let's say you have this Python 3.x script:

```
number1 = float(input("Enter the first number: "))
number2 = float(input("Enter the second number: "))
solution = number1 + number2
print("The sum of your numbers is {}".format(solution))
```

Obviously that's a super-simple example, but a lot of more complicated scripts follow the same kind of form. For example, a script for a financial analyst might have these equivalent steps:

- Get data about a particular stock from the user.
- Run some kind of complicated analysis on the data.
- Print out a result saying how good the algorithm thinks the stock is as an investment.

The point is, we have three phases, input, processing and output.

(Some scripts have more phases -- they gather some data, do some processing, gather some more data, do more processing, and so on, and eventually print out a result. We'll come on to those later on.)

Let's work through how we would change our three-phase input-process-output script into a website.

Step 1: extract the processing into a function

In a website's code, we don't have access to the Python `input` or `print` functions, so the input and output phases will be different -- but the processing phase will be the same as it was in the original script. So the first step is to extract our processing code into a function so that it can be re-used. For our example, that leaves us with something like this:

```
def do_calculation(number1, number2):
    return number1 + number2

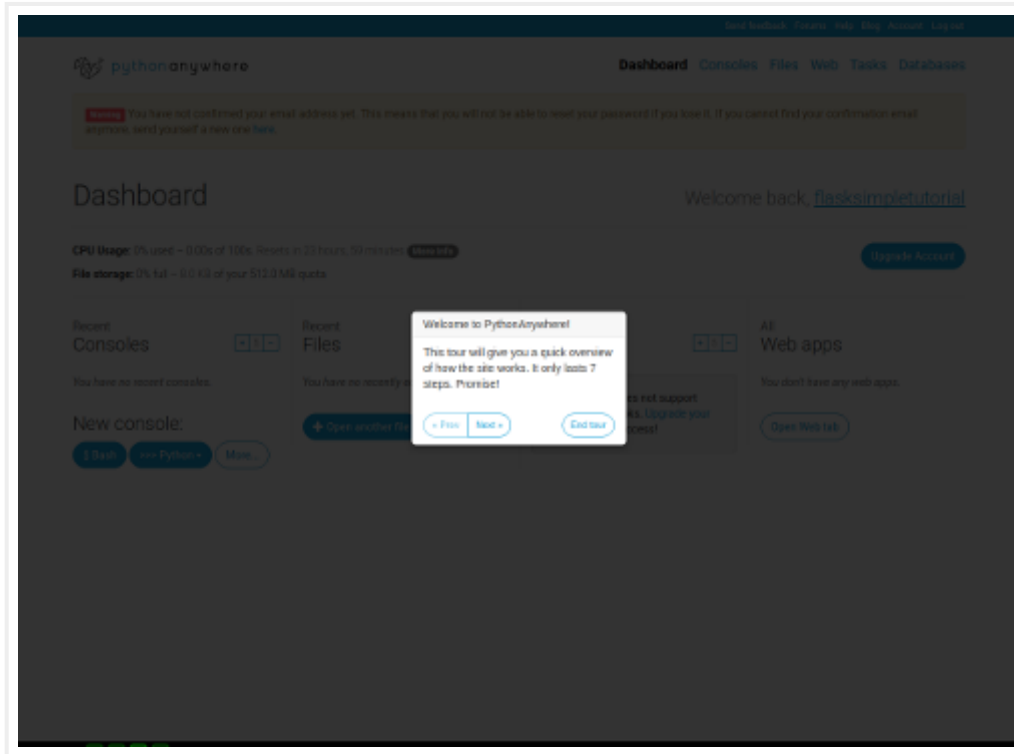
number1 = float(input("Enter the first number: "))
number2 = float(input("Enter the second number: "))
solution = do_calculation(number1, number2)
print("The sum of your numbers is {}".format(solution))
```

Simple enough. In real-world cases like the stock-analysis then of course there would be more inputs, and the `do_calculation` function would be considerably more complicated, but the theory is the same.

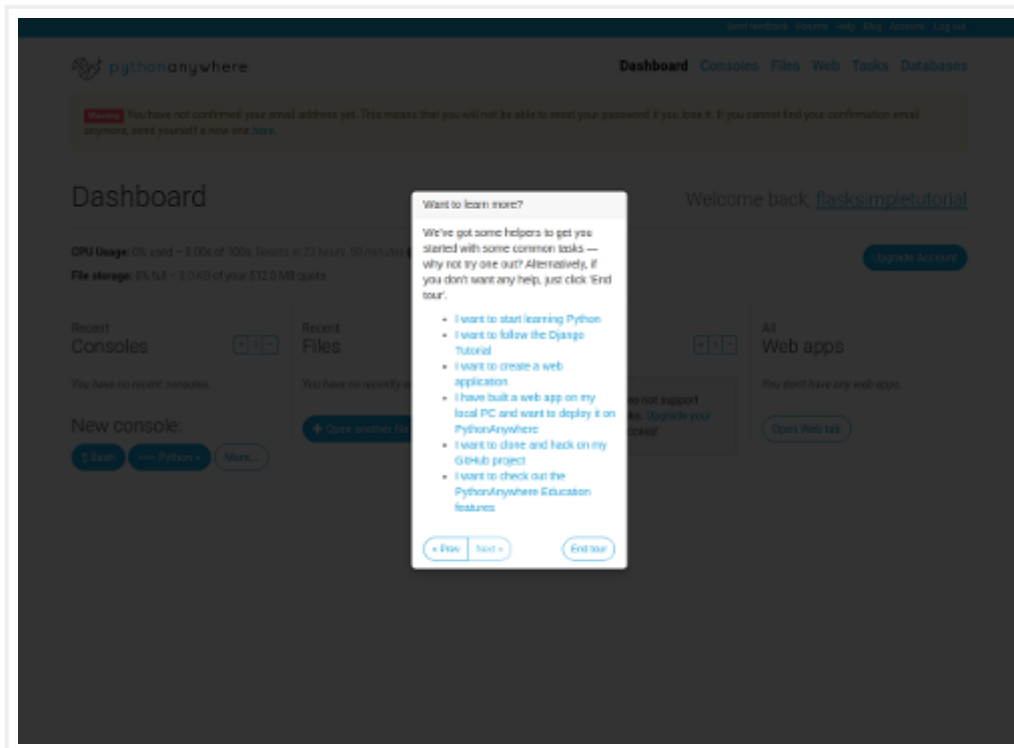
Step 2: create a website

Firstly, [create a PythonAnywhere account](#) if you haven't already. A free "Beginner" account is enough for this tutorial.

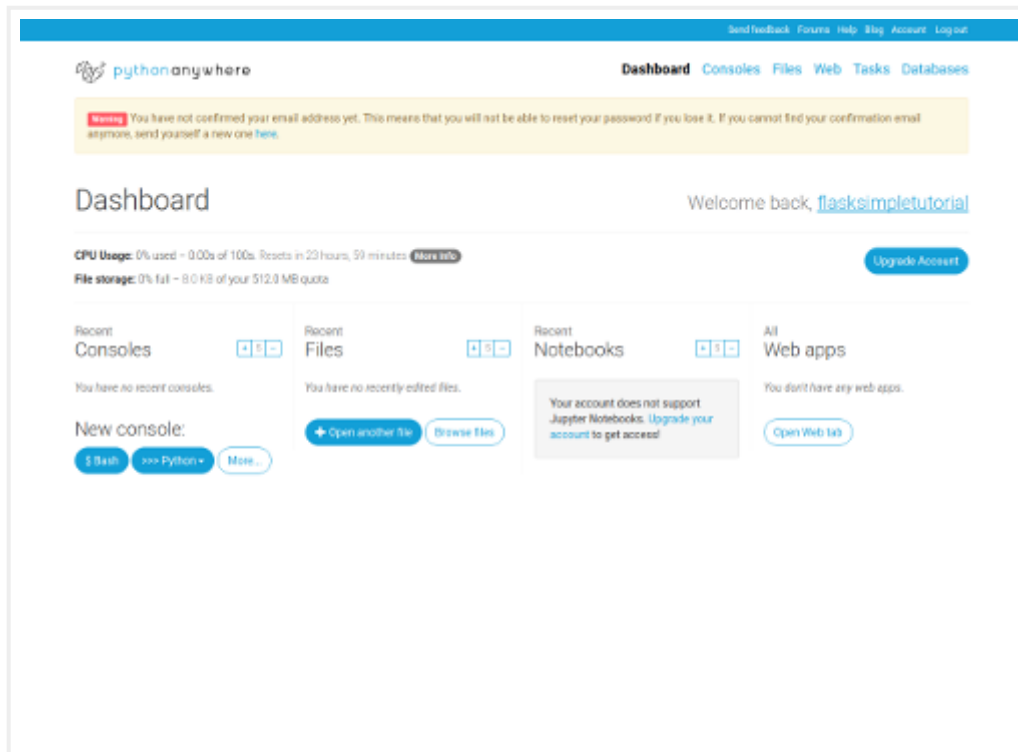
Once you've signed up, you'll be taken to the dashboard, with a tour window. It's worth going through the tour so that you can learn how the site works -- it'll only take a minute or so.



At the end of the tour you'll be presented with some options to "learn more". You can just click "End tour" here, because this tutorial will tell you all you need to know.

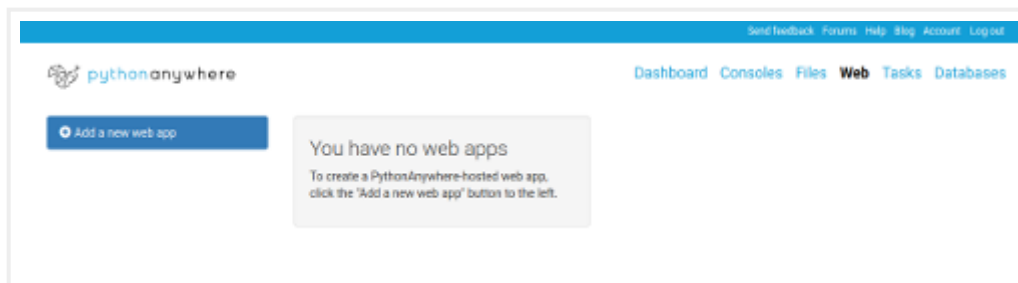


Now you're presented with the PythonAnywhere dashboard. I recommend you check your email and confirm your email address -- otherwise if you forget your password later, you won't be able to reset it.

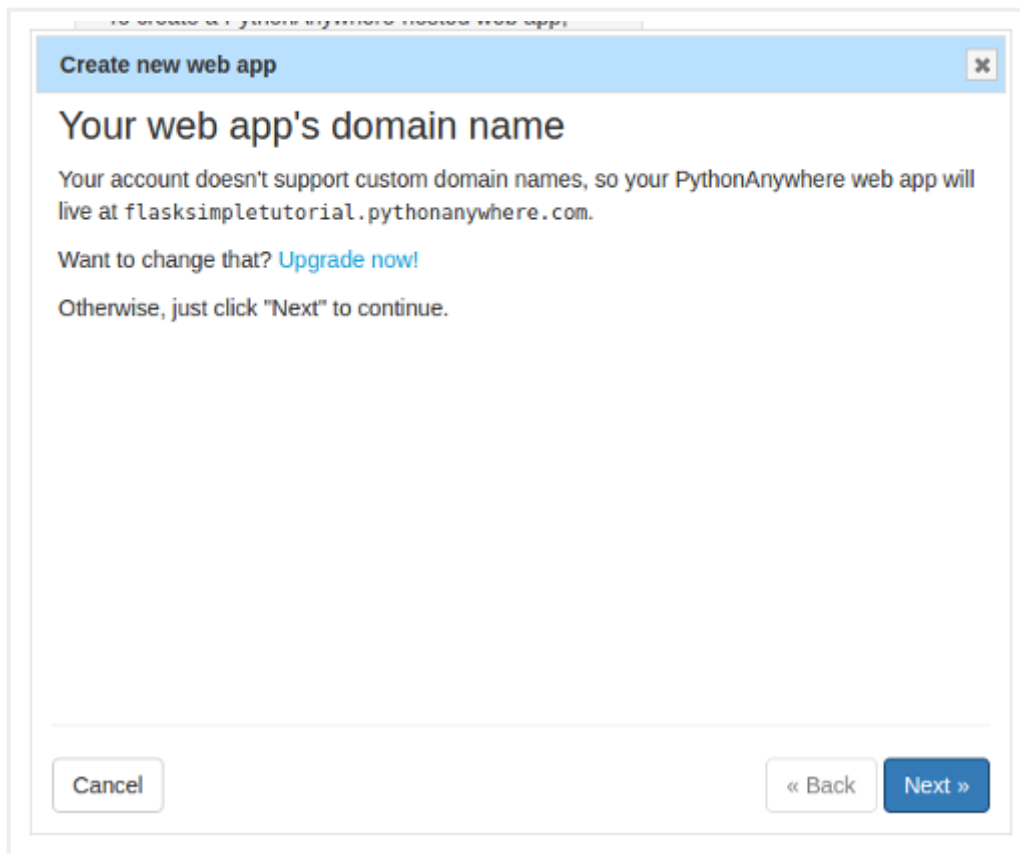


Now you need to create a website, which requires a web framework. The easiest web framework to get started with when creating this kind of thing is [Flask](#); it's very simple and doesn't have a lot of the built-in stuff that other web frameworks have, but for our purposes that's a good thing.

To create your site, go to the "Web" page using the tab near the top right:



Click on the "Add a new web app" button to the left. This will pop up a "Wizard" which allows you to configure your site. If you have a free account, it will look like this:



Create new web app

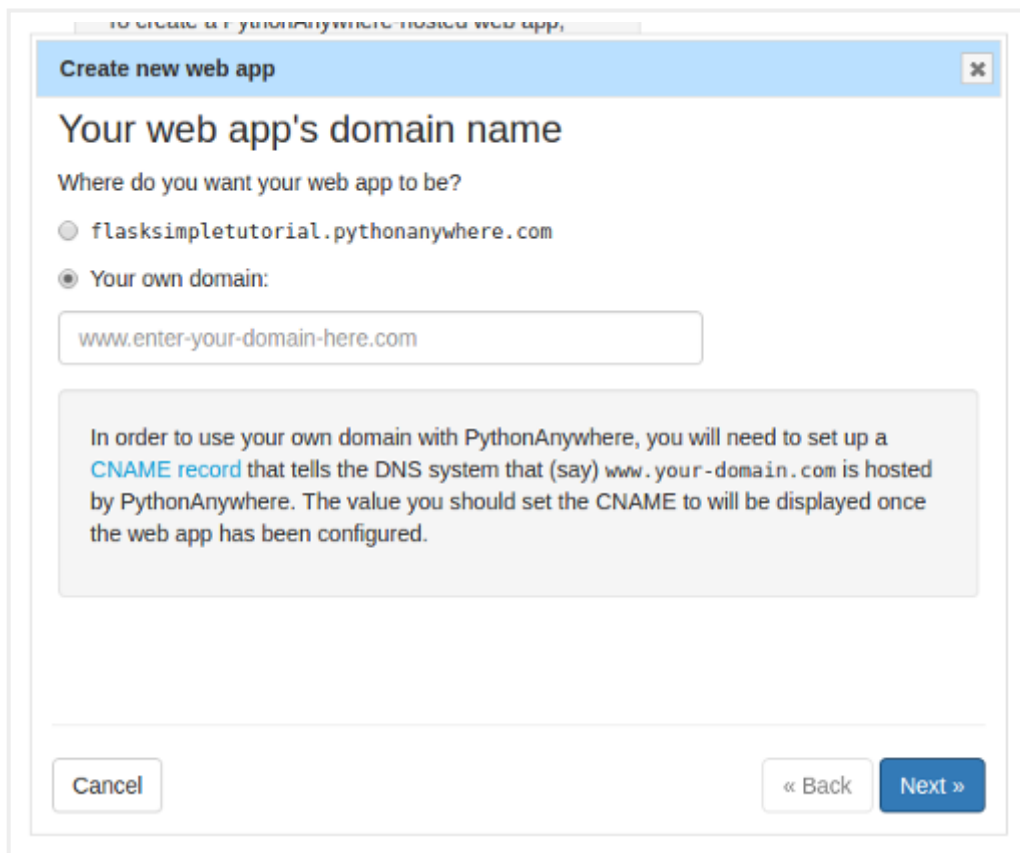
Your web app's domain name

Your account doesn't support custom domain names, so your PythonAnywhere web app will live at `flasksimpletutorial.pythonanywhere.com`.

Want to change that? [Upgrade now!](#)

Otherwise, just click "Next" to continue.

If you decided to go for a paid account (thanks :-), then it will be a bit different:



Create new web app

Your web app's domain name

Where do you want your web app to be?

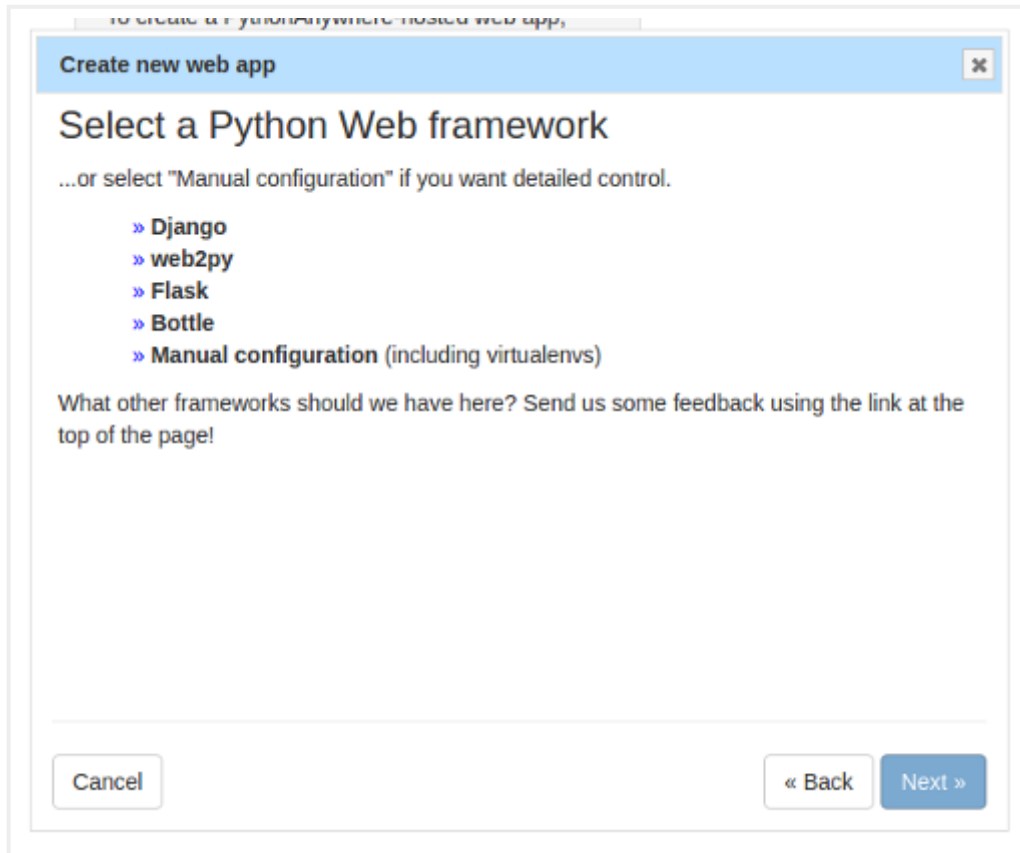
☐ `flasksimpletutorial.pythonanywhere.com`

☒ Your own domain:

In order to use your own domain with PythonAnywhere, you will need to set up a [CNAME record](#) that tells the DNS system that (say) `www.your-domain.com` is hosted by PythonAnywhere. The value you should set the CNAME to will be displayed once the web app has been configured.

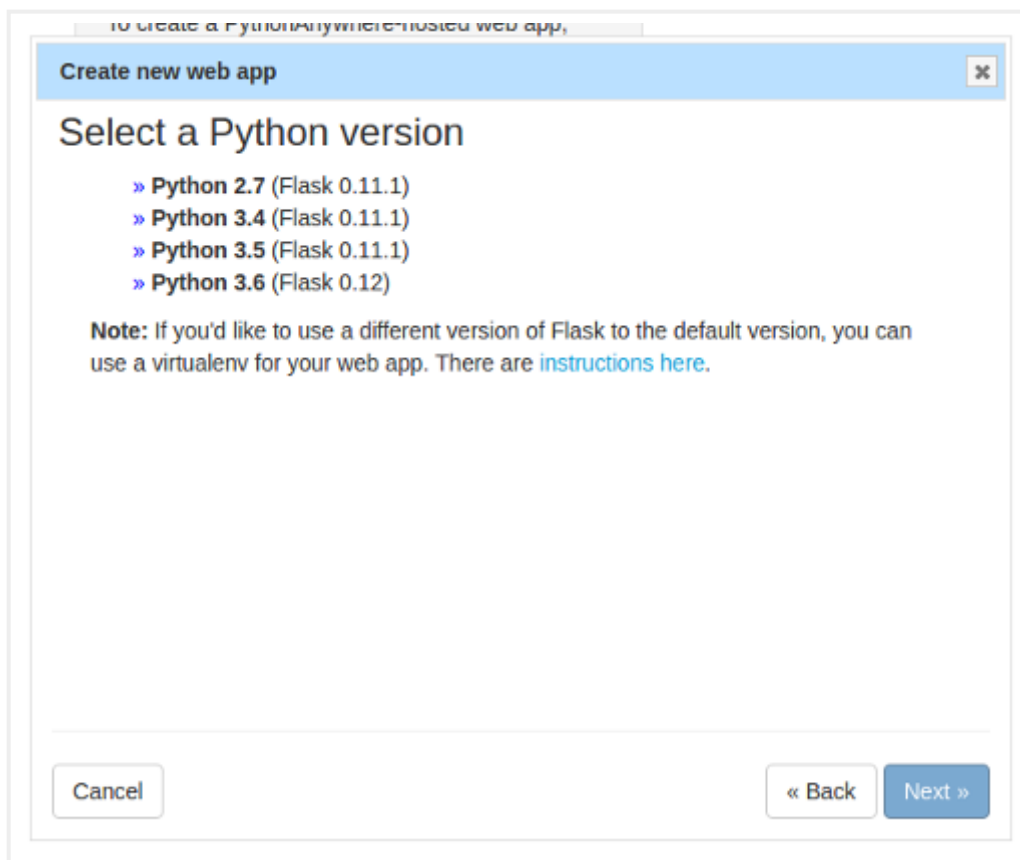
What we're doing on this page is specifying the host name in the URL that people will enter to see your website. Free accounts can have one website, and it must be at `yourusername.pythonanywhere.com`. Paid accounts have the option of using their own custom host names in their URLs.

For now, we'll stick to the free option. If you have a free account, just click the "Next" button, and if you have a paid one, click the checkbox next to the *yourusername* [.pythonanywhere.com](https://pythonanywhere.com), then click "Next". This will take you on to the next page in the wizard.



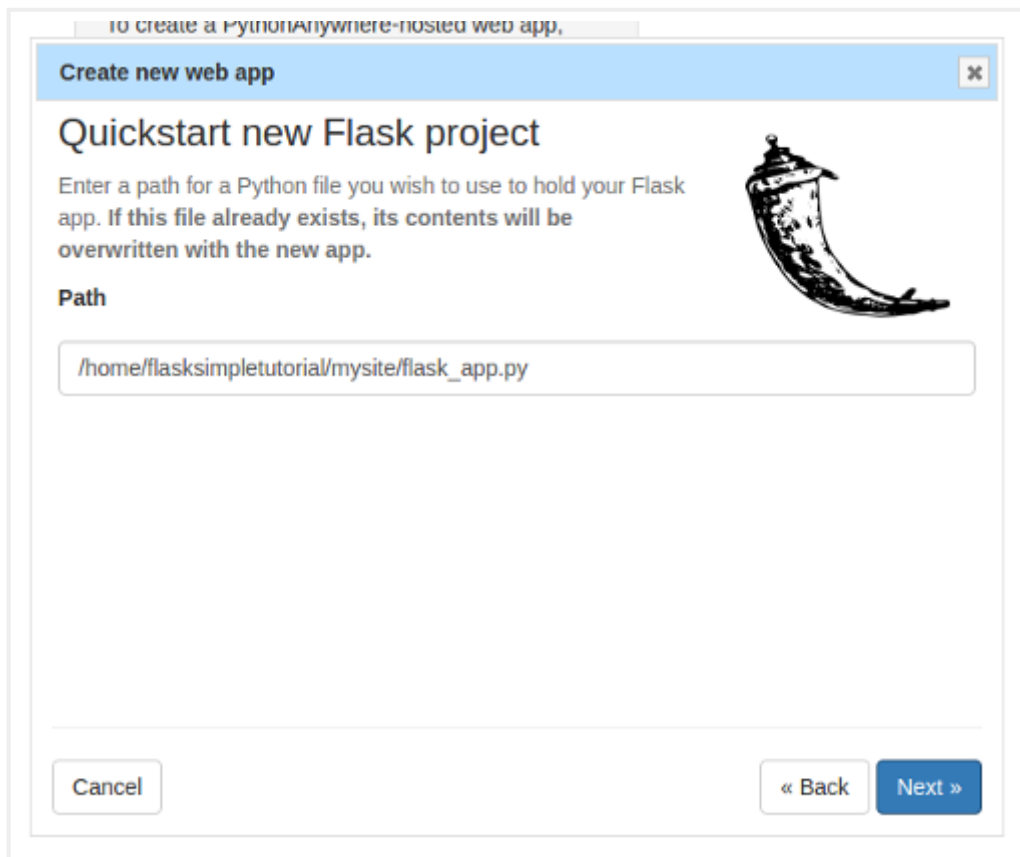
The screenshot shows a web browser window with a tab titled "to create a PythonAnywhere-hosted web app,". The page has a blue header bar with the text "Create new web app" and a close button (X). The main heading is "Select a Python Web framework". Below it, a subtext says "...or select 'Manual configuration' if you want detailed control." There is a list of options, each preceded by a right-pointing arrow (»): Django, web2py, Flask, Bottle, and Manual configuration (including virtualenvs). Below the list, a paragraph asks: "What other frameworks should we have here? Send us some feedback using the link at the top of the page!". At the bottom, there are three buttons: "Cancel", "« Back", and "Next »".

This page is where we select the web framework we want to use. We're using Flask, so click that one to go on to the next page.



The screenshot shows the same web browser window, but the page content has changed. The header bar remains the same. The main heading is now "Select a Python version". Below it, a list of options is shown, each preceded by a right-pointing arrow (»): Python 2.7 (Flask 0.11.1), Python 3.4 (Flask 0.11.1), Python 3.5 (Flask 0.11.1), and Python 3.6 (Flask 0.12). Below the list, a "Note" section states: "Note: If you'd like to use a different version of Flask to the default version, you can use a virtualenv for your web app. There are [instructions here](#)." At the bottom, the same three buttons are present: "Cancel", "« Back", and "Next »".

PythonAnywhere has various versions of Python installed, and each version has its associated version of Flask. You can use different Flask versions to the ones we supply by default, but it's a little more tricky (you need to use a thing called a *virtualenv*), so for this tutorial we'll create a site using Python 3.6, with the default Flask version. Click the option, and you'll be taken to the next page:



to create a PythonAnywhere-hosted web app,

Create new web app

Quickstart new Flask project

Enter a path for a Python file you wish to use to hold your Flask app. If this file already exists, its contents will be overwritten with the new app.

Path

/home/flasksimpletutorial/mysite/flask_app.py

Cancel « Back Next »

This page is asking you where you want to put your code. Code on PythonAnywhere is stored in your home directory, `/home/yourusername`, and in its subdirectories. Flask is a particularly lightweight framework, and you can write a simple Flask app in a single file. PythonAnywhere is asking you where it should create a directory and put a single file with a really really simple website. The default should be fine; it will create a subdirectory of your home directory called `mysite` and then will put the Flask code into a file called `flask_app.py` inside that directory.

(It will overwrite any other file with the same name, so if you're not using a new PythonAnywhere account, make sure that the file that it's got in the "Path" input box isn't one of your existing files.)

Once you're sure you're OK with the filename, click "Next". There will be a brief pause while PythonAnywhere sets up the website, and then you'll be taken to the configuration page for the site:

pythonanywhere

Dashboard Consoles Files Web Tasks Databases

All done! Your web app is now set up. Details below.

flaskimpletutorial.pythonanywhere.com

Add a new web app

Configuration for flaskimpletutorial.pythonanywhere.com

Reload:

Reload flaskimpletutorial.pythonanywhere.com

Best before date:

We're happy to host your free website -- and keep it free -- for as long as you want to keep it running, but you'll need to log in at least once every three months and click the "Run until 3 months from today" button below. We'll send you an email a week before the site is disabled so that you don't forget to do that. [See here for more details.](#)

This site will be disabled on **Thursday 08 March 2018**

Run until 3 months from today

Paying users' sites stay up forever without any need to log in to keep them running.

Traffic:

How busy is your site?

This month (previous month)	0 (0)
Today (yesterday)	0 (0)
Hour (previous hour)	0 (0)

Want some more data? [Paying accounts](#) get pretty charts :)

Code:

What your site is running

You can see that the host name for the site is on the left-hand side, along with the "Add a new web app" button. If you had multiple websites in your PythonAnywhere account, they would appear there too. But the one that's currently selected is the one you just created, and if you scroll down a bit you can see all of its settings. We'll ignore most of these for the moment, but one that is worth noting is the "Best before date" section.

If you have a paid account, you won't see that -- it only applies to free accounts. But if you have a free account, you'll see something saying that your site will be disabled on a date in three months' time. Don't worry! You can keep a free site up and running on PythonAnywhere for as long as you want, without having to pay us a penny. But we do ask you to log in every now and then and click the "Run until 3 months from today" button, just so that we know you're still interested in keeping it running.

Before we do any coding, let's check out the site that PythonAnywhere has generated for us by default. Right-click the host name, just after the words "Configuration for", and select the "Open in new tab" option; this will (of course) open your site in a new tab, which is useful when you're developing -- you can keep the site open in one tab and the code and other stuff in another, so it's easier to check out the effects of the changes you make.

Here's what it should look like.


flaskimpletutorial.pythonanywhere.com

Hello from Flask!


OK, it's pretty simple, but it's a start. Let's take a look at the code! Go back to the tab showing the website configuration (keeping the one showing your site open), and click on the "Go to directory" link next to the "Source code" bit in the "Code" section:

Code:

What your site is running.

Source code:	/home/flasksimpletutorial/mysite	Go to directory
Working directory:	/home/flasksimpletutorial/	Go to directory
WSGI configuration file:	/var/www/flasksimpletutorial.pythonanywhere.com/wsgi.py	
Python version:	3.6 	

You'll be taken to a different page, showing the contents of the subdirectory of your home directory where your website's code lives:

 pythonanywhere

Dashboard Consoles **Files** Web Tasks Databases

[Open Bash console here](#) 0% full ~ 72.0 KB of your 512.0 MB quota


Directories

Enter new directory name [New directory](#)

[...pycache...](#)


Files

Enter new file name, eg hello.py [New file](#)

[flask_app.py](#)  2017-12-08 14:52 186 bytes

[Upload a file](#)

Click on the `flask_app.py` file, and you'll see the (really really simple) code that defines your Flask app. It looks like this:

 /home/flasksimpletutorial/mysite/flask_app.py

Keyboard shortcuts: Normal [Share](#) [Save](#) [Save as...](#) [Run](#) [Refresh](#) [Menu](#)

```
1 # A very simple Flask hello world app for you to get started with...
2
3
4 from flask import Flask
5
6 app = Flask(__name__)
7
8 @app.route('/')
9 def hello_world():
10     return 'Hello from Flask!'
11
12
```

[Run this file](#) [Bash console here](#)

It's worth working through this line-by-line:

```
from flask import Flask
```


As you'd expect, this loads the Flask framework so that you can use it.

```
app = Flask(__name__)
```

This creates a Flask application to run your code.

```
@app.route('/')
```

This decorator specifies that the following function defines what happens when someone goes to the location "/" on your site -- eg. if they go to `http://yourusername.pythonanywhere.com/`. If you wanted to define what happens when they go to `http://yourusername.pythonanywhere.com/foo` then you'd use `@app.route('/foo')` instead.

```
def hello_world():  
    return 'Hello from Flask!'
```

This simple function just says that when someone goes to the location, they get back the (unformatted) text "Hello from Flask".

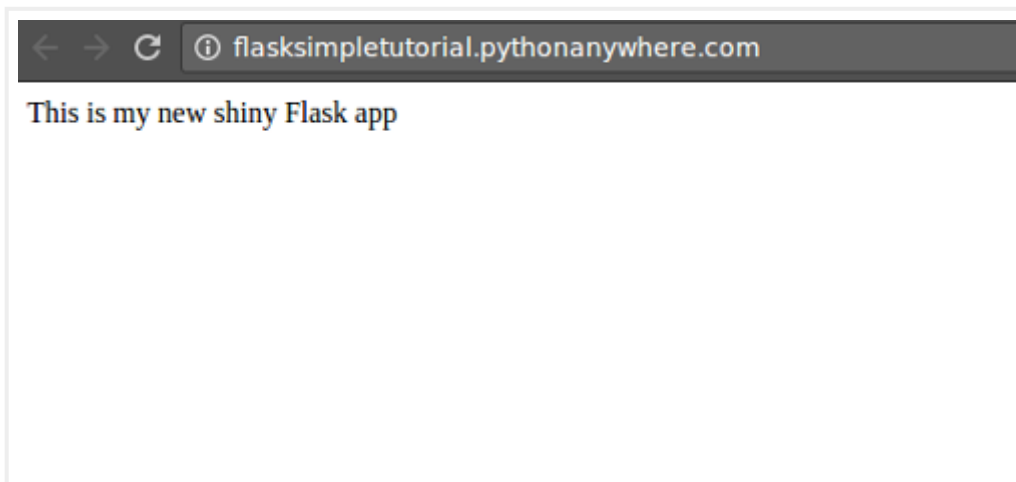
Try changing it -- for example, to "This is my new shiny Flask app". Once you've made the change, click the "Save" button at the top to save the file to PythonAnywhere:



...then the reload button (to the far right, looking like two curved arrows making a circle), which stops your website and then starts it again with the fresh code.



A "spinner" will appear next to the button to tell you that PythonAnywhere is working. Once it has disappeared, go to the tab showing the website again, hit the page refresh button, and you'll see that it has changed as you'd expect.

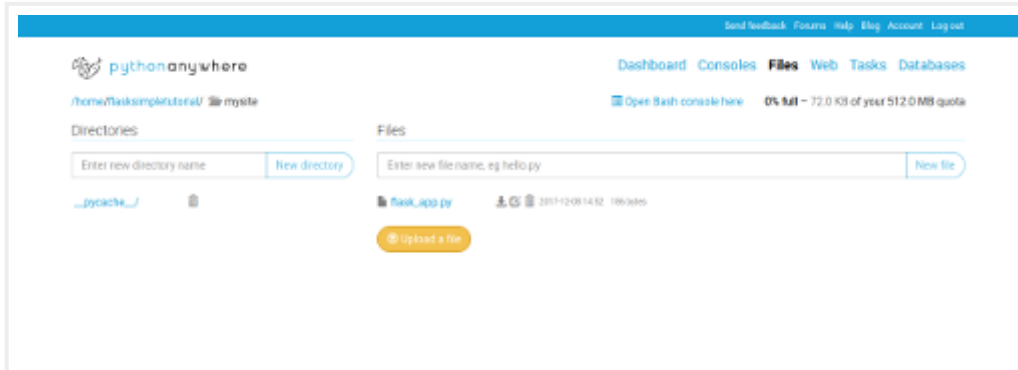


Step 3: make the processing code available to the web app

Now, we want our Flask app to be able to run our code. We've already extracted it into a function of its own. It's generally a good idea to keep the web app code -- the basic stuff to display pages -- separate from the more

complicated processing code (after all, if we were doing the stock analysis example rather than this simple add-two-numbers script, the processing could be thousands of lines long).

So, we'll create a new file for our processing code. Go back to the browser tab that's showing your editor page; up at the top, you'll see "breadcrumb" links showing you where the file is stored. They'll be a series of directory names separated by "/" characters, each one apart from the last being a link. The last one, just before the name of the file containing your Flask code, will probably be `mysite`. Right-click on that, and open it in a new browser tab -- the new tab will show the directory listing you had before:



In the input near the top right, where it says "Enter new file name, eg. hello.py", enter the name of the file that will contain the processing code. Let's (uninventively) call it `processing.py`. Click the "New file" button, and you'll have another editor window open, showing an empty file. Copy/paste your processing function into there; that means that the file should simply contain this code:

```
def do_calculation(number1, number2):
    return number1 + number2
```

Save that file, then go back to the tab you kept open that contains the Flask code. At the top, add a new line just after the line that imports Flask, to import your processing code:

```
from processing import do_calculation
```

While we're at it, let's also add a line to make debugging easier if you have a typo or other error in the code; just after the line that says

```
app = Flask(__name__)
```

...add this:

```
app.config["DEBUG"] = True
```

Save the file; you'll see that you get a warning icon next to the new `import` line. If you move your mouse pointer over the icon, you'll see the details:

```
1 from flask import Flask
2 from processing import do_calculation
3
4 'processing.do_calculation' imported but unused
5 app.config["DEBUG"] = True
6
7 @app.route('/')
8 def hello_world():
9     return 'This is my new shiny Flask app'
10
11
```

It says that the function was imported but is not being used, which is completely true! That moves us on to the next step.

Step 4: Accepting input

What we want our site to do is display a page that allows the user to enter two numbers. To do that, we'll change the existing function that is run to display the page. Right now we have this:

```
@app.route('/')
def hello_world():
    return 'This is my new shiny Flask app'
```

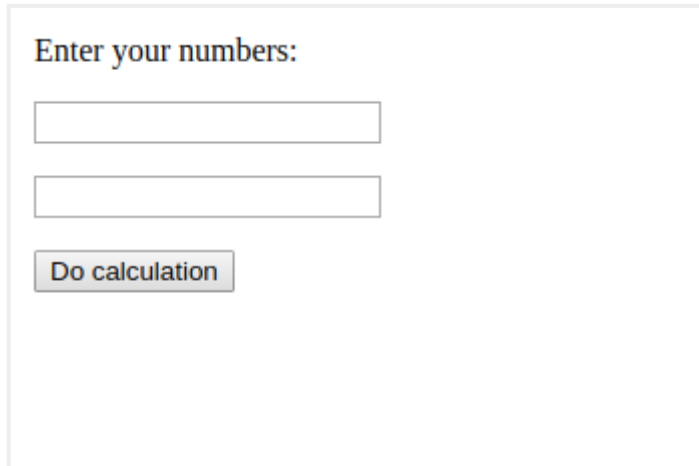
We want to display more than text, we want to display some HTML. Now, the *best* way to do HTML in Flask apps is to use templates (which allow you to keep the Python code that Flask needs in separate files from the HTML), but we have [other tutorials](#) that go into the details of that. In this case we'll just put the HTML right there inside our Flask code -- and while we're at it, we'll rename the function:

```
@app.route('/')
def adder_page():
    return '''
        <html>
        <body>
            <p>Enter your numbers:</p>
            <form>
                <p><input name="number1" /></p>
                <p><input name="number2" /></p>
                <p><input type="submit" value="Do calculation" /></p>
            </form>
        </body>
        </html>
    '''
```

We won't go into the details of how HTML works here, there are lots of excellent tutorials online and one that suits the way you learn is just a Google search away. For now, all we need to know is that where we were previously returning a single-line string, we're now returning a multi-line one (that's what the three quotes in a line mean, in case you're not familiar with them -- one string split over multiple lines). The multi-line string contains HTML code, which just displays a page that asks the user to enter two numbers, and a button that says "Do calculation". Click on the editor's "reload website" button:



...and then check out your website again in the tab that you (hopefully) kept open, and you'll see something like this:

A web form with a light gray border. It contains the text "Enter your numbers:" in a bold, dark blue font. Below this text are two empty rectangular input fields stacked vertically. At the bottom of the form is a button with a gray background and the text "Do calculation" in a bold, dark blue font.

However, as we haven't done anything to wire up the input to the processing, clicking the "Do calculation" button won't do anything but reload the page.

Step 5: validating input

We could at this stage go straight to adding on the code to do the calculations, and I was originally planning to do that here. But after thinking about it, I realised that doing that would basically be teaching you to shoot yourself in the foot... When you put a website up on the Internet, you have to allow for the fact that the people using it will make mistakes. If you created a site that allowed people to enter numbers and add them, sooner or later someone will type in "wombat" for one of the numbers, or something like that, and it would be embarrassing if your site responded with an internal server error.

So let's add on some basic validation -- that is, some code that makes sure that people aren't providing us with random marsupials instead of numbers.

A good website will, when you enter an invalid input, display the page again with an error message in it. A bad website will display a page saying "Invalid input, please click the back button and try again". Let's write a good website.

The first step is to change our HTML so that the person viewing the page can click the "Do calculation" button and get a response. Just change the line that says

```
<form>
```

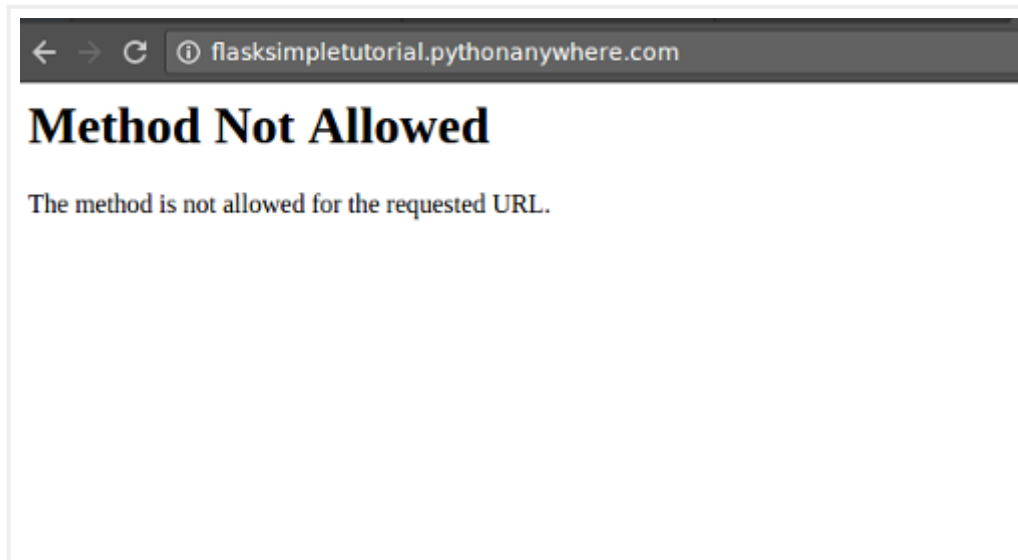
So that it says this:

```
<form method="post" action=".">
```

What that means is that previously we had a form, but now we have a form that has an "action" telling it that when the button that has the type "submit" is clicked, it should request the same page as it is already on, but this time it should use the "post" method.

(HTTP methods are extra bits of information that are tacked on to requests that are made by a browser to a server. The "get" method, as you might expect, means "I just want to get a page". The "post" method means "I want to provide the server with some information to store or process". There are vast reams of details that I'm skipping over here, but that's the most important stuff for now.)

So now we have a way for data to be sent back to the server. Reload the site using the button in the editor, and refresh the page in the tab where you're viewing your site. Try entering some numbers, and click the "Do calculation" button, and you'll get... an incomprehensible error message:



Well, perhaps not entirely incomprehensible. It says "method not allowed". Previously we were using the "get" method to get our page, but we just told the form that it should use the "post" method when the data was submitted. So Flask is telling us that it's not going to allow that page to be requested with the "post" method.

By default, Flask view functions only accept requests using the "get" method. It's easy to change that. Back in the code file, where we have this line:

```
@app.route('/')
```

...replace it with this:

```
@app.route("/", methods=["GET", "POST"])
```

Save the file, hit the reload button in the editor, then go to the tab showing your page; click back to get away from the error page if it's still showing, then enter some numbers and click the "Do calculation" button again.

You'll be taken back to the page with no error. Success! Kind of.

Now let's add the validation code. The numbers that were entered will be made available to us in our Flask code via the `form` attribute of a global variable called `request`. So we can add validation logic by using that. The first step is to make the `request` variable available by importing it; change the line that says

```
from flask import Flask
```

to say

```
from flask import Flask, request
```

Now, add this code to the view function, before the `return` statement:

```
errors = ""
if request.method == "POST":
    number1 = None
```

```

number2 = None
try:
    number1 = float(request.form["number1"])
except:
    errors += "<p>{!r} is not a number.</p>\n".format(request.form["number1"])
try:
    number2 = float(request.form["number2"])
except:
    errors += "<p>{!r} is not a number.</p>\n".format(request.form["number2"])

```

Basically, we're saying that if the method is "post", we do the validation.

Finally, add some code to put those errors into the page's HTML; replace the bit that returns the multi-line string with this:

```

return '''
    <html>
        <body>
            {errors}
            <p>Enter your numbers:</p>
            <form method="post" action=".">
                <p><input name="number1" /></p>
                <p><input name="number2" /></p>
                <p><input type="submit" value="Do calculation" /></p>
            </form>
        </body>
    </html>
    '''.format(errors=errors)

```

This is exactly the same page as before, we're just interpolating the string that contains any errors into it just above the "Enter your numbers" header.

Save the file; you'll see more warnings for the lines where we define variables called `number1` and `number2`, because we're not using those variables. We know we're going to fix that, so they can be ignored for now.

Reload the site, and head over to the page where we're viewing it, and try to add a koala to a wallaby -- you'll get an appropriate error:

Try adding 23 to 19, however, and you won't get 42 -- you'll just get the same input form again. So now, the final step that brings it all together.

Step 6: doing the calculation!

We're all set up to do the calculation. What we want to do is:

- If the request used a "get" method, just display the input form
- If the request used a "post" method, but one or both of the numbers are not valid, then display the input form with error messages.
- If the request used a "post" method, and both numbers are valid, then display the result.

We can do that by adding something inside the `if request.method == "POST":` block, just after we've checked that `number2` is valid:

```
if number1 is not None and number2 is not None:
    result = do_calculation(number1, number2)
    return '''
        <html>
            <body>
                <p>The result is {result}</p>
                <p><a href="/">Click here to calculate again</a>
            </body>
        </html>
    '''.format(result=result)
```

Adding that code should clear out all of the warnings in the editor page, and if you reload your site and then try using it again, it should all work fine!

The result is 42.0

[Click here to calculate again](#)

Pause for breath...

So if all has gone well, you've now converted a simple script that could add two numbers into a simple website that lets other people add numbers. If you're getting error messages, it's well worth trying to debug them yourself to find out where any typos came in. An excellent resource is the website's error log; there's a link on the "Web" page:

Log files:

The first place to look if something goes wrong.

Access log: [flasksimpletutorial.pythonanywhere.com.access.log](https://flasksimpletutorial.pythonanywhere.com/access.log)

Error log: [flasksimpletutorial.pythonanywhere.com.error.log](https://flasksimpletutorial.pythonanywhere.com/error.log)

Server log: [flasksimpletutorial.pythonanywhere.com.server.log](https://flasksimpletutorial.pythonanywhere.com/server.log)

Log files are periodically rotated. You can find old logs here: </var/log>

...and the most recent error will be at the bottom:

```

2018-09-28 15:21:27,397: Error running WSGI application
2018-09-28 15:21:27,401: ModuleNotFoundError: No module named 'falsk'
2018-09-28 15:21:27,401:   File "/var/www/flasksimpletutorial/pythonanywhere_com_wsgi.py", line 16, in <module>
2018-09-28 15:21:27,402:     from flask_app import app as application
2018-09-28 15:21:27,402:
2018-09-28 15:21:27,402:   File "/home/flasksimpletutorial/mysite/flask_app.py", line 1, in <module>
2018-09-28 15:21:27,402:     from falsk import Flask, request
2018-09-28 15:21:27,402: *****
2018-09-28 15:21:27,402: If you're seeing an import error and don't know why,
2018-09-28 15:21:27,402: we have a dedicated help page to help you debug:
2018-09-28 15:21:27,402: https://help.pythonanywhere.com/pages/DebuggingImportError/
2018-09-28 15:21:27,403: *****

```

That error message is telling me that I mistyped "flask" as "falsk", and the traceback tells me exactly which line the typo is on.

However, if you get completely stuck, here's the code you should currently have:

```

from flask import Flask, request

from processing import do_calculation

app = Flask(__name__)
app.config["DEBUG"] = True

@app.route("/", methods=["GET", "POST"])
def adder_page():
    errors = ""
    if request.method == "POST":
        number1 = None
        number2 = None
        try:
            number1 = float(request.form["number1"])
        except:
            errors += "<p>{!r} is not a number.</p>\n".format(request.form["number1"])
        try:
            number2 = float(request.form["number2"])
        except:
            errors += "<p>{!r} is not a number.</p>\n".format(request.form["number2"])
    if number1 is not None and number2 is not None:
        result = do_calculation(number1, number2)
        return '''
            <html>
            <body>
                <p>The result is {result}</p>
                <p><a href="/">Click here to calculate again</a>
            </body>
            </html>
            '''.format(result=result)

    return '''
    <html>
    <body>
        {errors}
        <p>Enter your numbers:</p>
        <form method="post" action=".">
            <p><input name="number1" /></p>
            <p><input name="number2" /></p>
            <p><input type="submit" value="Do calculation" /></p>
        </form>
    '''

```



```
        </body>
    </html>
    ''' .format(errors=errors)
```

The next step -- multi-phase scripts

So now that we've managed to turn a script that had the simple three-phase input-process-output structure into a website, how about handling the more complicated case where you have more phases? A common case is where you have an indefinite number of inputs, and the output depends on all of them. For example, here's a simple script that will allow you to enter a list of numbers, one after another, and then will display the statistical mode (the most common number) in the list, with an appropriate error message if there is no most common number (for example in the list [1, 2, 3, 4]).

```
import statistics

def calculate_mode(number_list):
    try:
        return "The mode of the numbers is {}".format(statistics.mode(number_list))
    except statistics.StatisticsError as exc:
        return "Error calculating mode: {}".format(exc)

inputs = []
while True:
    if len(inputs) != 0:
        print("Numbers so far:")
        for input_value in inputs:
            print(input_value)
    value = input("Enter a number, or just hit return to calculate: ")
    if value == "":
        break
    try:
        inputs.append(float(value))
    except:
        print("{} is not a number")

print(calculate_mode(inputs))
```

How can we turn that into a website? We could display, say, 100 input fields and let the user leave the ones they don't want blank, but (a) that would look hideous, and (b) it would leave people who wanted to get the mode of 150 numbers stuck.

(Let's put aside for the moment the fact that entering lots of numbers into a website would be deathly dull -- there's a solution coming for that :-)

What we need is a page that can accumulate numbers; the user enters the first, then clicks a button to send it to the server, which puts it in a list somewhere. Then they enter the next, and the server adds that one to the list. Then the next, and so on, until they're finished, at which point they click a button to get the result.

Here's a naive implementation. By "naive", I mean that it sort of works in some cases, but doesn't in general; it's the kind of thing that one might write, only to discover that when other people start using it, it breaks in really weird and confusing ways. It's worth going through, though, because the way in which it is wrong is instructive.

Firstly, in our `processing.py` file we have the processing code, just as before:

```
import statistics

def calculate_mode(number_list):
    try:
        return "The mode of the numbers is {}".format(statistics.mode(number_list))
    except statistics.StatisticsError as exc:
        return "Error calculating mode: {}".format(exc)
```

That should be pretty clear. Now, in `flask_app.py` we have the following code:

(A step-by-step explanation is coming later, but it's worth reading through now to see if you can see how at least some of it works.)

```
from flask import Flask, request

from processing import calculate_mode

app = Flask(__name__)
app.config["DEBUG"] = True

inputs = []

@app.route("/", methods=["GET", "POST"])
def mode_page():
    errors = ""
    if request.method == "POST":
        try:
            inputs.append(float(request.form["number"]))
        except:
            errors += "<p>{!r} is not a number.</p>\n".format(request.form["number"])

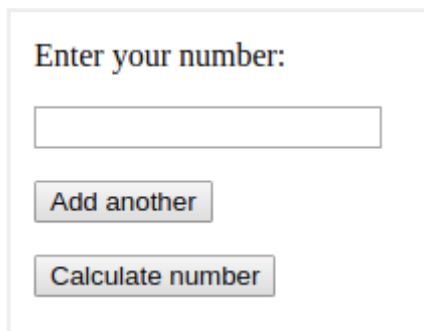
    if request.form["action"] == "Calculate number":
        result = calculate_mode(inputs)
        inputs.clear()
        return '''
            <html>
                <body>
                    <p>{result}</p>
                    <p><a href="/">Click here to calculate again</a>
                </body>
            </html>
            '''.format(result=result)

    if len(inputs) == 0:
        numbers_so_far = ""
    else:
        numbers_so_far = "<p>Numbers so far:</p>"
        for number in inputs:
            numbers_so_far += "<p>{}</p>".format(number)

    return '''
```

```
<html>
  <body>
    {numbers_so_far}
    {errors}
    <p>Enter your number:</p>
    <form method="post" action=".">
      <p><input name="number" /></p>
      <p><input type="submit" name="action" value="Add another" /></p>
      <p><input type="submit" name="action" value="Calculate number" /></p>
    </form>
  </body>
</html>
''' .format(numbers_so_far=numbers_so_far, errors=errors)
```

All clear? Maybe... It does work, though, sort of. Let's try it -- copy the code for the two files into your editor tabs, reload the site, and give it a go. If you have a free account, it will work!

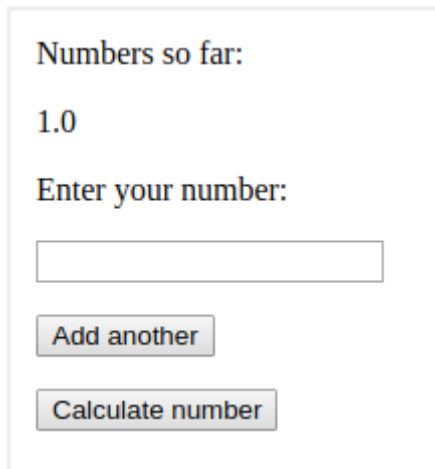


Enter your number:

Add another

Calculate number

Enter "1", and you get this:



Numbers so far:

1.0

Enter your number:

Add another

Calculate number

Enter some more numbers:

Numbers so far:

1.0

2.0

3.0

1.0

Enter your number:

Add another

Calculate number

...and calculate the result:

The mode of the numbers is 1.0

[Click here to calculate again](#)

But if you have a paid account, you'll see some weird behaviour. Exactly what you'll get will depend on various random factors, but it will be something like this:

Enter your number:

Add another

Calculate number

Enter 1, and you might get this:

Numbers so far:

1.0

Enter your number:

Add another

Calculate number

Enter 2, and you might get this:

Numbers so far:

2.0

Enter your number:

Add another

Calculate number

Huh? Where did the "1" go? Well, let's enter "3":

Numbers so far:

2.0

3.0

Enter your number:

Add another

Calculate number

Well, that seems to have worked. We'll add "4":

Numbers so far:

2.0

3.0

4.0

Enter your number:

Add another

Calculate number

And now we'll add "1" again:

Numbers so far:

1.0

1.0

Enter your number:

Add another

Calculate number

So now our original 1 has come back, but all of the other numbers have disappeared.

In general, it will seem to sometimes forget numbers, and then remember them again later, as if it has multiple lists of numbers -- which is exactly what it does.

Before we go into why it's actually wrong (and why, counterintuitively, it works *worse* on a paid account than on a free one), here's the promised step-by-step runthrough, with comments after each block of code. Starting off:

```
from flask import Flask, request

from processing import calculate_mode

app = Flask(__name__)
app.config["DEBUG"] = True
```

All that is just copied from the previous website.

```
inputs = []
```

We're initialising a list for our inputs, and putting it in the global scope, so that it will persist over time. This is because each view of our page will involve a call to the view function:

```
@app.route("/", methods=["GET", "POST"])
def mode_page():
```

...which is exactly the same kind of setup for a view function as we had before.

```
errors = ""
if request.method == "POST":
    try:
        inputs.append(float(request.form["number"]))
    except:
        errors += "<p>{!r} is not a number.</p>\n".format(request.form["number"])
```

We do very similar validation to the number as we did in our last website, and if the number is valid we add it to the global list.

```
if request.form["action"] == "Calculate number":
```

This bit is a little more tricky. On our page, we have two buttons -- one to add a number, and one to say "do the calculation" -- here's the bit of the HTML code from further down that specifies them:

```
<p><input type="submit" name="action" value="Add another" /></p>
<p><input type="submit" name="action" value="Calculate number" /></p>
```

This means that when we get a post request from a browser, the "action" value in the `form` object will contain the text of the submit button that was actually clicked.

So, if the "Calculate number" button was the one that the user clicked...

```
result = calculate_mode(inputs)
inputs.clear()
return '''
    <html>
        <body>
            <p>{result}</p>
            <p><a href="/">Click here to calculate again</a>
        </body>
    </html>
'''.format(result=result)
```

...we do the calculation and return the result (clearing the list of the inputs at the same time so that the user can try again with another list).

If, however, we get past that `if request.form["action"] == "Calculate number"` statement, it means either that:

- The request was using the post method, and we've just added a number to the list or set the error string to reflect the fact that the user entered an invalid number, or
- The request was using the get method

So:

```
if len(inputs) == 0:
    numbers_so_far = ""
else:
    numbers_so_far = "<p>Numbers so far:</p>"
    for number in inputs:
        numbers_so_far += "<p>{}</p>".format(number)
```

...we generate a list of the numbers so far, if there are any, and then:

```
return '''
    <html>
        <body>
            {numbers_so_far}
            {errors}
            <p>Enter your number:</p>
            <form method="post" action=".">
                <p><input name="number" /></p>
```

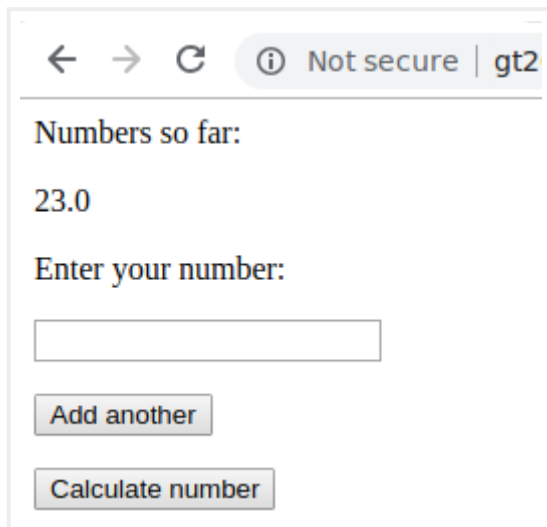
```
<p><input type="submit" name="action" value="Add another" /></p>
<p><input type="submit" name="action" value="Calculate number" /></p>
</form>
</body>
</html>
''.format(numbers_so_far=numbers_so_far, errors=errors)
```

We return our page asking for a number, with the list of numbers so far and errors if either is applicable.

Phew!

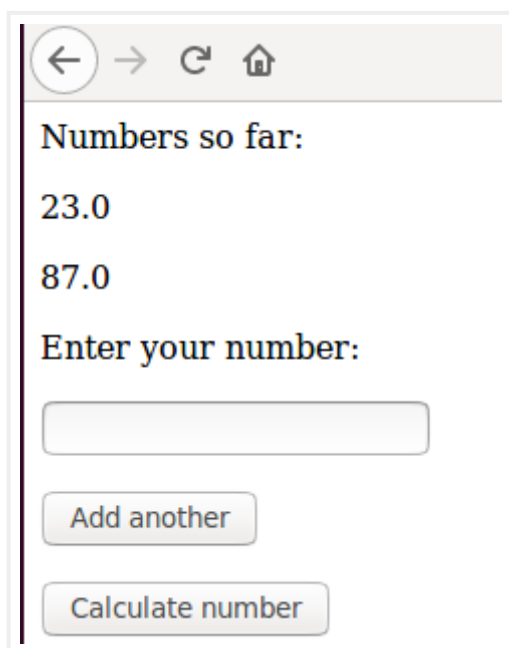
So why is it incorrect? If you have a paid account, you've already seen evidence that it doesn't work very well. If you have a free account, here's a thought experiment -- what if two people were viewing the site at the same time? In fact, you can see exactly what would happen if you use the "incognito" or "private tab" feature on your browser -- or, if you have multiple browsers installed, if you use two different browsers (say by visiting the site in Chrome and in Firefox at the same time).

What you'll see is that both users are sharing a list of numbers. The Chrome user starts off, and adds a number to the list:



A screenshot of a web browser window. The address bar shows 'Not secure | gt2'. The page content includes the text 'Numbers so far:' followed by the number '23.0'. Below this is a label 'Enter your number:' and an empty text input field. At the bottom are two buttons: 'Add another' and 'Calculate number'.

Now the Firefox user adds a number -- but they see not only the number they added, but also the Chrome user's number:



A screenshot of a web browser window. The address bar shows navigation icons. The page content includes the text 'Numbers so far:' followed by the numbers '23.0' and '87.0'. Below this is a label 'Enter your number:' and an empty text input field. At the bottom are two buttons: 'Add another' and 'Calculate number'.

It's pretty clear what's going on here. There's one server handling the requests from both users, so there's only one list of inputs -- so everyone shares the same list.

But what about the situation for websites running on paid accounts? If you'll remember, it looked like the opposite was going on there -- there were multiple lists, even within the same browser.

This is because paid accounts have multiple servers for the same website. This is a good thing, it means that if they get lots of requests coming in at the same time, then everything gets processed more quickly -- so they can have higher-traffic websites. But it also means that different requests, *even successive requests from the same browser*, can wind up going to different servers, and because each server has its own list, the browser will see one list for one request, but see a different list on the next request.

What this all means is that **global variables don't work for storing state in website code**. On each server that's running to control your site, everyone will see the same global variables. And if you have multiple servers, then each one will have a different set of global variables.

What to do?

Sessions to the rescue!

What we need is a way to keep a set of "global" variables that are specific to each person viewing the site, and are shared between all servers. If two people, Alice and Bob, are using the site, then Alice will have her own list of inputs, which all servers can see, and Bob will have a different list of inputs, separate from Alice's but likewise shared between servers.

The web dev mechanism for this is called sessions, and is built into Flask. Let's make a tiny set of modifications to the Flask app to make it work properly. Firstly, we'll import support for sessions by changing our Flask import line from this:

```
from flask import Flask, request
```

...to this:

```
from flask import Flask, request, session
```

In order to use sessions, we'll also need to configure Flask with a "secret key" -- sessions use cryptography, which requires a random number. Add a line like this just after the line where we configure Flask's debug setting to be True:

```
app.config["SECRET_KEY"] = "lkmaslkdsldsamldsdmasldsmkdd"
```

Use a different string to the one I put above; mashing the keyboard randomly is a good way to get a reasonably random string, though if you want to do things properly, find something truly random.

Next, we'll get rid of the global `inputs` list by deleting this line:

```
inputs = []
```

Now we'll use an `inputs` list that's stored inside the `session` object (which looks like a dictionary) instead of using our global variable. Firstly, let's make sure that whenever we're in our view function, we have a list of inputs associated with the current session if there isn't one already. Right at the start of the view function, add this:

```
if "inputs" not in session:
    session["inputs"] = []
```

Next, inside the bit of code where we're adding a number to the inputs list, replace this line:

```
inputs.append(float(request.form["number"]))
```

...with this one that uses the list on the session:

```
session["inputs"].append(float(request.form["number"]))
```

There's also a subtlety here; because we're changing a list inside a session (instead of adding a new thing to the session), we need to tell the session object that it has changed by putting this line immediately after the last one:

```
session.modified = True
```

Next, when we're calculating the mode, we need to look at our `session` again to get the list of inputs:

```
result = calculate_mode(inputs)
```

...becomes

```
result = calculate_mode(session["inputs"])
```

...and the line that clears the inputs so that the user can do another list likewise changes from

```
inputs.clear()
```

to:

```
session["inputs"].clear()  
session.modified = True
```

Finally, the code that generates the "numbers so far" list at the start of the page needs to change to use the session:

```
if len(inputs) == 0:  
    numbers_so_far = ""  
else:  
    numbers_so_far = "<p>Numbers so far:</p>"  
    for number in inputs:  
        numbers_so_far += "<p>{}</p>".format(number)
```

...becomes:

```
if len(session["inputs"]) == 0:  
    numbers_so_far = ""  
else:  
    numbers_so_far = "<p>Numbers so far:</p>"  
    for number in session["inputs"]:  
        numbers_so_far += "<p>{}</p>".format(number)
```

Once all of those code changes have been done, you should have this:

```
from flask import Flask, request, session

from processing import calculate_mode

app = Flask(__name__)
app.config["DEBUG"] = True
app.config["SECRET_KEY"] = "lkmaslkdsldsamdlsdmasldsmkdd"

@app.route("/", methods=["GET", "POST"])
def mode_page():
    if "inputs" not in session:
        session["inputs"] = []

    errors = ""
    if request.method == "POST":
        try:
            session["inputs"].append(float(request.form["number"]))
            session.modified = True
        except:
            errors += "<p>{!r} is not a number.</p>\n".format(request.form["number"])

        if request.form["action"] == "Calculate number":
            result = calculate_mode(session["inputs"])
            session["inputs"].clear()
            session.modified = True
            return '''
                <html>
                <body>
                    <p>{result}</p>
                    <p><a href="/">Click here to calculate again</a>
                </body>
                </html>
            '''.format(result=result)

    if len(session["inputs"]) == 0:
        numbers_so_far = ""
    else:
        numbers_so_far = "<p>Numbers so far:</p>"
        for number in session["inputs"]:
            numbers_so_far += "<p>{</p>".format(number)

    return '''
        <html>
        <body>
            {numbers_so_far}
            {errors}
            <p>Enter your number:</p>
            <form method="post" action=".">
                <p><input name="number" /></p>
                <p><input type="submit" name="action" value="Add another" /></p>
                <p><input type="submit" name="action" value="Calculate number" /></p>
            </form>
    '''
```

```
</body>
</html>
''''.format(numbers_so_far=numbers_so_far, errors=errors)
```

Hit the reload button, and give it a try! If you have a paid account, you'll find that now it all works properly -- and if you have a free account, you'll see that separate browsers now have separate lists of numbers :-)

So now we have a multi-user website that keeps state around between page visits.

Processing files

Now, entering all of those numbers one-by-one would be tedious if there were a lot of them. A lot of Python scripts don't request the user to enter data a line at a time; they take a file as their input, process it, and produce a file as the output. Here's a simple script that asks for an input filename and an output filename. It expects the input file to contain a number of lines, each with a comma-separated list of numbers on it. It writes to the output file the same number of lines, each one containing the sum of the numbers from the equivalent line in the input file.

```
def process_data(input_data):
    result = ""
    for line in input_data.split("\n"):
        if line != "":
            numbers = [float(n) for n in line.split(",")]
            result += str(sum(numbers))
            result += "\n"
    return result

input_filename = input("Enter the input filename: ")
output_filename = input("Enter the output filename: ")

with open(input_filename, "r") as input_file:
    input_data = input_file.read()

with open(output_filename, "w") as output_file:
    output_file.write(process_data(input_data))
```

What we want is a Flask app that will allow the user to upload a file like the input file that that script requires, and will then provide the output file to download. This is actually pretty similar to the original app we did -- there's just three phases, input-process-output. So the Flask app looks very similar.

Firstly, we put our calculating routine into `processing.py`, as normal:

```
def process_data(input_data):
    result = ""
    for line in input_data.split("\n"):
        if line != "":
            numbers = [float(n) for n in line.split(",")]
            result += str(sum(numbers))
            result += "\n"
    return result
```

...and now we write a Flask app that looks like this:

```

from flask import Flask, make_response, request

from processing import process_data

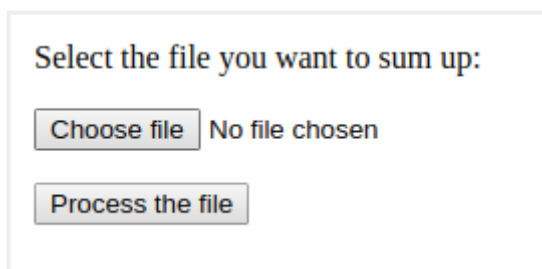
app = Flask(__name__)
app.config["DEBUG"] = True

@app.route("/", methods=["GET", "POST"])
def file_summer_page():
    if request.method == "POST":
        input_file = request.files["input_file"]
        input_data = input_file.stream.read().decode("utf-8")
        output_data = process_data(input_data)
        response = make_response(output_data)
        response.headers["Content-Disposition"] = "attachment; filename=result.csv"
        return response

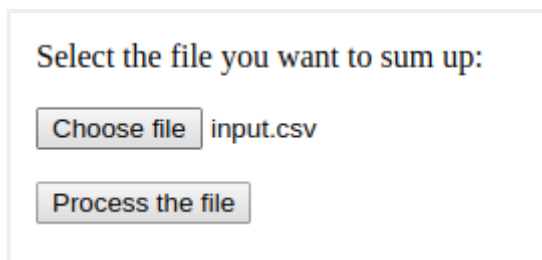
    return '''
        <html>
            <body>
                <p>Select the file you want to sum up:</p>
                <form method="post" action="." enctype="multipart/form-data">
                    <p><input type="file" name="input_file" /></p>
                    <p><input type="submit" value="Process the file" /></p>
                </form>
            </body>
        </html>
    '''

```

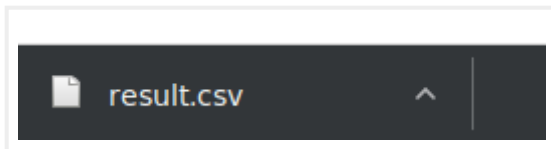
Again, we'll go through that bit-by-bit in a moment (though it's worth noting that although this feels like something that should be much harder than the first case, the Flask app is much shorter :-). But let's try it out first -- once you've saved the code on PythonAnywhere and reloaded the site, visit the page:



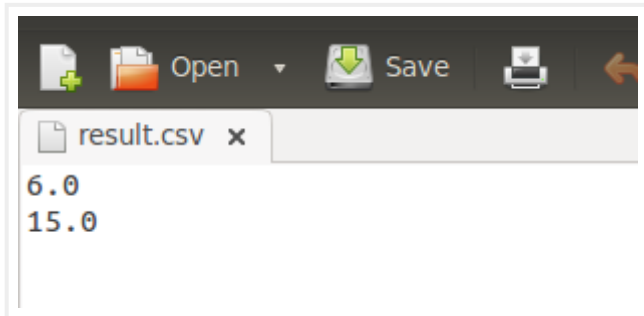
We specify a file with contents (mine just has "1, 2, 3" on the first line and "4, 5, 6" on the second):



...then we click the button. You'll have to watch for it, but a file download will almost immediately start. In Chrome, for example, this will appear at the bottom of the window:



Open the file in an appropriate application -- here's what it looks like in gedit:



We've got a website where we can upload a file, process it, and download the results :-)

Obviously the user interface could use a bit of work, but that's left as an exercise for the reader...

So, how does the code work? Here's the breakdown:

```
from flask import Flask, make_response, request

from processing import process_data

app = Flask(__name__)
app.config["DEBUG"] = True
```

This is our normal Flask setup code.

```
@app.route("/", methods=["GET", "POST"])
def file_summer_page():
```

As usual, we define a view.

```
if request.method == "POST":
```

If the request is use the "post" method...

```
input_file = request.files["input_file"]
input_data = input_file.stream.read().decode("utf-8")
```

...we ask Flask to extract the uploaded file from the `request` object, and then we read it into memory. The file it will provide us with will be in binary format, so we convert it into a string, assuming that it's in the UTF-8 character set.

```
output_data = process_data(input_data)
```

Now we process the data using our function. The next step is where it gets a little more complicated:

```
response = make_response(output_data)
response.headers["Content-Disposition"] = "attachment; filename=result.csv"
```

In the past, we just returned strings from our Flask view functions and let it sort out how that should be presented to the browser. But this time, we want to take a little more control over the kind of response that's going back. In particular, we don't want to dump all of the output into the browser window so that the user has to copy/paste the (potentially thousands of lines of) output into their spreadsheet or whatever. Instead, we want to tell the browser "the thing I'm sending you is a file called 'result.csv', so please download it appropriately". That's what these two lines do -- the first is just a way to tell Flask that we're going to need some detailed control over the response, and the second does that control. Next:

```
return response
```

...we just return the response.

Now that we're out of that first `if` statement, we know that the request we're handling isn't one with a "post" method, so it must be a "get". So we display the form:

```
return '''
    <html>
        <body>
            <p>Select the file you want to sum up:</p>
            <form method="post" action="." enctype="multipart/form-data">
                <p><input type="file" name="input_file" /></p>
                <p><input type="submit" value="Process the file" /></p>
            </form>
        </body>
    </html>
'''
```

In this case we just return a string of HTML like we did in the previous examples. There are only two new things in there:

```
<form method="post" action="." enctype="multipart/form-data">
```

The `enctype="multipart/form-data"` in there is just an extra flag that is needed to tell the browser how to format files when it uploads them as part of the "post" request that it's sending to the server, and:

```
<p><input type="file" name="input_file" /></p>
```

....is just how you specify an input where the user can select a file to upload

So that's it!

And we're done

In this blog post we've presented three different Flask apps, each of which shows how a specific kind of normal Python script can be converted into a website that other people can access to reap the benefits of the code you've written.

Hopefully they're all reasonably clear, and you can see how you could apply the same techniques to your own scripts. If you have any comments or questions, please post them in the comments below -- and if you have any thoughts about other kinds of patterns that we could consider adding to an updated version of this post, or to a follow-up, do let us know.

Thanks for reading!

Posted Oct. 2, 2018, 3:37 p.m. by giles

What do you think?

90 Responses



Upvote



Funny



Love



Surprised



Angry



Sad

9 Comments

PythonAnywhere News

1 Login ▾

♥ Recommend 3

🐦 Tweet

f Share

Sort by Oldest ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

**kyber** • 4 months ago

Excellent article, nicely done. I found it very straightforward to follow and have shared a link on the Facebook Python python.programmers group (over 100,000 members) I admin as a recommended guide to getting your first web app up and running. Hope you get a few of our members joining up.

I assume there is some need for session management in the third example, but it was left out for the sake of simplicity. This might be a bit confusing.

1 ^ | v • Reply • Share ›

**PythonAnywhere** Mod ➔ kyber • 4 months ago

Thanks! Glad you like the post :) All of the examples should work as-is -- the session management code is only present in the ones that need it.

1 ^ | v • Reply • Share ›

**Fred Chapman** • 3 months ago • edited

I'm working my way through your tutorial, and I noticed a minor HTML issue. The closing `</p>` tag is missing from this line:

`<p>Enter your numbers:</p>`

^ | v • Reply • Share ›

**PythonAnywhere** Mod ➔ Fred Chapman • 3 months ago

Thanks! We've fixed that.

1 ^ | v • Reply • Share ›

**Fred Chapman** ➔ PythonAnywhere • 3 months ago

Thank you for the speedy reply and for fixing it throughout!

This tutorial has been very helpful to me, so thank you for that too!

^ | v • Reply • Share ›



PythonAnywhere Mod → Fred Chapman • 3 months ago

No problem -- glad you like the tutorial!

1 ^ | v • Reply • Share ›



Dave J • 3 months ago

Great tutorial. Question though on the processing of the csv file. In your code you are using `from processing import process_data`. When trying to find this package through `pip_install` it looks like this isn't available after Python 2.4.

Do you happen to know of a library for Python 3.6 that can be used instead?

^ | v • Reply • Share ›



millenniumhand Mod → Dave J • 3 months ago

`processing` is not a `pip` package. It's a module that is created as part of the tutorial

^ | v • Reply • Share ›



kashif • a month ago

for just write 3 lines on webpage we have to read and learn this 3000 lines article ?

^ | v • Reply • Share ›

ALSO ON PYTHONANYWHERE NEWS

New feature: self-installation of SSL certificates!

2 comments • a year ago



willywongi — That's awesome! And I say that because I was one of those who forgot to update the cert and my site went blank :D

Back to school tips for teachers, from PythonAnywhere

5 comments • 3 years ago



PythonAnywhere — Ooh, interesting idea!

Dec blogpost - PythonAnywhere News

1 comment • 3 years ago



Alex Llerenas — Hahah. Awesome newsletter and thanks for the updates. Happy Holidays everyone!

Latest deploy: Some nice new features and a surprise

2 comments • 3 years ago



Greg — So, how long will this testing take? I'm very curious. :-)

✉ Subscribe ➕ Add Disqus to your siteAdd DisqusAdd 🔒 Disqus' Privacy PolicyPrivacy PolicyPrivacy Policy



PythonAnywhere is a Python development and hosting environment that displays in your web browser and runs on our servers. They're already set up with everything you need. It's easy to use, fast, and powerful. There's even a useful free plan.

[You can sign up here.](#)

Built with [Bootstrap](#). Blog source code available [on GitHub](#).
Copyright © 2011-2019 [PythonAnywhere LLP](#) — [Terms](#) — [Privacy](#)