Edinburgh Napier University

SET09103 - Advanced Web Technologies

Lab 01 - Hello World

Dr Simon Wells

1 Overview

GOAL: Our aim in this practical is to get a simple Flask web-app up and running. This means the following:

1. Acquire and understand how to set up our Python environment and install Flask into it so that we can create Web Services Gateway Interface (WSGI) web-apps.
2. Create and edit a source file containing the code for a simple "Hello World" WSGI Flask web-app.
3. Run the web-app and view the output in our web browser.

This web-app will be the foundation for the majority of the work that we do in the rest of the trimester so it is important to get comfortable with it. Don't just go through the practical once and then move on. Instead, try the following approach:

1. Go through the practical once exactly as presented.
2. Go through the lab at least twice more so that you are used to everything involved. In subsequent practicals, I'll just ask you to create a new Flask app, which means essentially to go through the process involved in this practical. Understanding what each command does and why is quite integral to this process and will make life *much* easier in the rest of the module.

ASIDE #1: If you are already familiar with Python and tools like pip and virtual environments, then you are free to use those tools instead on your own laptop. They are equivalent to the env tool that we use in the labs and which I'll refer to in the practicals. However env works nicely to enable Python and other important tools to be downloaded, run, and managed really flexibly on lab machines.

Corollary to ASIDE #1: If you have your own laptop, or desktop machine, then you can use that for practical work and your assessments instead of the lab machines. It's probably easiest to install uv so that you environment matches the lab environment. If you are installing to your own machine then just follow the installation instructions for your operating system[1]. For lab machines we will follow a slightly different process however which is detailed in the sections below. Either way the result is the same; an environment in which we can create isolated python projects so that we can try out lots of different things with Python and Flask.

ASIDE #2: In case of either aside #1 or it's corollary, you might need to be careful with commands because things might be slightly different on your system. Remember that the aim here is to learn things and not to blindly type in commands until you get to the end. Read the instructions, consider them, and if you understand what they should be doing, then do them.

> **NOTICE:** We will be working with the command line. There are three important things to know if you aren't used to working with the command line. **First**, if a command completes successfully then there won't necessarily be output to tell you that, instead you will just get your prompt back. Some commands will give you output and sometimes that output will indicate if there was an error. So you need to read and understand any output before executing the next command. Otherwise you will just be compounding error upon error. **Second**, if you make changes to a command then you might get different output so think it through before you do it. If you put things in different places then you need to take account of that fact subsequently. **Third**, if you are unsure whether something has worked correctly, or you get unexpected output that is different to what I've shown, then don't just move on to the next command and hope it will be OK because it probably won't be.

[1] `https://docs.astral.sh/uv/getting-started/installation/`

2 Activities

2.1 Getting Started

Log into your lab machine. For this practical you will need a web browser, a text editor, the windows terminal (cmd.exe), and the file manager open as you will probably use a mix of all of them at various points:

Web Browser - to see our web app once it's running.

Text Editor - to edit the source code for our web-apps and, later, other files such as configuration, HTML, CSS, or JS files. Your choice of editor is completely up to you but a good place to start is with Notepad++[2].

Terminal - to execute our commands. There isn't really a good GUI environment for the work we'll be doing so working at the command line is actually the best way to get going.

File Manager - just to keep things a bit easier. If you're not used to using the command line then the file manager can be quicker for manipulating files and folders.

2.2 Installation

The first thing we need to do is to download the uv tool. There are lots of tools for installing differnt versions of the Python interpreter, installing libraries, and managing Python projects and uv is just one more, quite nice version that happens to be less hassle to work with. It basically bundles up a whole bunch of existing popular tools and presents them as a single executable[3]. In your browser, navigate to: `https://github.com/astral-sh/uv` then click on the latest release (0.8.17 at time of writing) which will take you to another page. From this page you should select the following file **uv-x86_64-pc-windows-msvc.zip** and allow it do download.[4].

Open the downloaded zip and extract the three files it contains. These are the core tools that we'll need for the rest of the trimester, and we'll use them *a lot*, so we are going to need to put them somewhere where we can easily find and use them. I suggest creating a folder called *apps* on your H: drive, e.g. H:\apps and putting the three files there. If you place the files somewhere else that makes sense to you then you will need to adjust the commands that relate to H:\apps to reflect your choice.

Now you will be using the uv.exe program a lot so it is worth adding the apps folder to the PATH variable so that we don't have to keep typing the full path, (e.g. h:\apps\uv), every time we use it. The PATH is basically a way to tell the terminal where to find the executable we want it to use. We can do this using the following command in the terminal (cmd.exe) but note that it will **only** work if you've put them in h:\apps:

```
1    SET PATH=%PATH%;H:\apps\
```

What we've done here is use the SET command and pass it the argument PATH. There are many *environment variables* and PATH is just one of them[5]. What we've done is told SET to *set* PATH to the existing value of PATH, e.g. %PATH%, and then to add the location of the uv exectables, h:\apps after that. The semi-colon is just a separator between the two arguments. **Important:** If you placed the uv executables into a different folder then you will need to use your own path to that folder. This command is illustrated in Figure 1

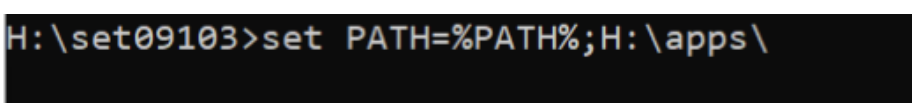

Figure 1

[2] `https://notepad-plus-plus.org/`

[3] Not really, it's actually a reimplementation of a bunch of popular tools that has been done really really well. It actually makes using Python on Windows less horrific which I think is a true achievement

[4] If the file fails to download on your lab machine then the likely cause is that your H: drive is full. Delete some files to make space then try again. If that still doesn't work then it means a trip to the help desk

[5] It's a good side exercise to find out what other environment variables are available as they can be useful. You can even set your own unique ones for use in your own apps.

When you use SET the changes will only affect your current session in the terminal. If you close the terminal, or open a different terminal window, then you will need to SET your path again. Because the SET PATH command will only work for the lifetime of the terminal, once it is closed or you log out then the changes will be lost. **OPTIONAL:** If you want to make things more permanent then you will need to change the *system's* environment path settings. Go to "My Computer" > "Properties" > "Advanced" > "Environment Variables" > "Path". Add the directory where you uv tools are located to the end of that string. Then open a new terminal (cmd.exe) for the new path to take effect.

Whether you SET your path temporarily or permanently, the best way to test if it has worked is to try the uv command. If you have successfully added uv to your path then when you type the uv command you will get output similar to that shown in Figure 2. If uv isn't found on your path then you will get a message saying that it couldn't be found. If uv is not working then don't proceed any further but instead go back over your commands in the terminal and see if you can identify where the error lies.

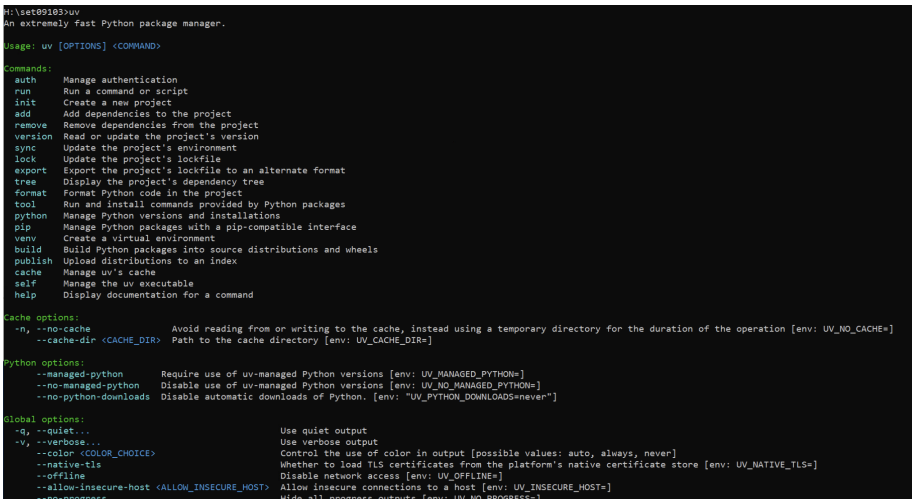

Figure 2

2.3 Setting up a new Flask project

We are going to:

1. set up some folders to keep our work in so that we are well organised
2. add a Python virtual environment to one of our folder so that we can store Python and the libraries we'll need in a useful place
3. learn to activate and deactivate our virtual environment
4. install the Flask library into our virtual environment

So, let's get started. First we need to create a folder on our H: drive for this module, e.g. set09103, a location to store all the code we develop in this module. As before, if you use a different folder location or folder name then you will have to change some of the commands. If you take the time to understand how the commands work then this should be straightforward. You can use the *mkdir* command in your terminal to create folders[6]. Assuming that you're in your H: drive, use:

```
1    mkdir set09103
```

[6]You can also use the file manager but you probably already know how to do that and by using the terminal then you are probably learning a new skill instead

Which should create a new folder called *set09103*. You can check that this is the case by using the *dir* command which lists the child folders that exist in your current location in the file systems. Once your set09103 folder is created then you can move into the newly created folder using the *cd* command, like so:

```
cd set09103
```

This has created a folder in which to store everything related to the module. We now need another folder for this specific activity, so let's create a folder for it. We'll call the new folder hello because we're going to put our "hello world" web-app into it:

```
mkdir hello
```

As before, move into the new hello folder:

```
cd hello
```

Notice how we're using the commands mkdir and cd to create new directories and move between them. You can also move from a folder into it's parent folder by using the '*cd ..*' command that is the cd command followed by a single space followed by two full-stops. There are many other terminal commands available and it is well worth your time to investigate them as they give you a superpower when it comes to controlling the machine. Basically, the more terminal commands you know, the more control you will have over the terminal environment.

Because we are going to use Python in this folder, we need to create a virtual environment. We do this using the *venv* argument of uv. We'll call the virtual environment "env"[7]. When we run the command it will create a new folder called 'env' in our current folder then put some scripts and other Python related things into some sub-folders. Let's try it out[8]:

We've just told uv to create a new virtual environment called 'env'. It's a good idea to explore the contents of env so that you know what's in there. Don't change anything, but it does no harm to take a look. A virtual environment is a way to isolate your code and any libraries it uses in one project from any other project you might be working on. This is because different projects might use different versions of a library so by isolating them you can keep things independent of each other. If things go wrong with your virtual environment then you can just delete it and create a new one.

Figure 3

A virtual environment needs to be activated and deactivated before and after use. The basic pattern is that we activate the virtual environment for a project when we start working on the project, then we do everything we need to do, editing files, running the project, and so forth, then when we are finished we deactivate the virtual environment. We activate a virtual environment using the activate script from the scripts folder within the environment. Similarly to deactivate the virtual environment we can use the deactive screipt from the scripts folder. If you have multiple terminals open then you will need to activate the virtual environment for each terminal in which you want to run a given Python project. Activate your virtual env like so:

```
env\Scripts\activate
```

and deactivate it as follows:

```
env\Scripts\deactivate
```

[7] I usually call my virtual environments env because then I can easily pick out the virtual environment in any given Python project

[8] If you get an error from uv at this point telling you that it failed to download things then you need to run uv just once as folows: *uv –native-tls –allow-insecure-host github.com* which tells uv that it is OK to retrieve software from Github

Figure 4

Notice that when a virtual environment is activated then the beginning of your terminal prompt will now show the name of your environment in parenthesis, e.g. (env). When you see this then you know that there is a Python interpreter available as well as whichever libraries are installed into that specific virtualenv. You can see what is installed in your new virtualenv by using the pip tool. Like venv, pip is an argument to the uv command which itself takes a further argument. Th pip tool is used to manage Python libraries and can install and uninstall them amongst other things[9]. To see what libraries pip is managing in the current virtual environment we use the *list* argument. Let's try that now:

```
1    uv pip list
```

Which should give you an empty list as we've only just created the environment and by default it doesn't contain any additional libraries beyond Python's *built-ins*.

Figure 5

As we're going to use the Flask library we'll should install it into our environment. Make sure your virtual environment is active[10]. Now let's install Flask into our new active virtual environment:

```
1    uv pip install flask
```

Basically we've just invoked the uv command and passed it an argument 'pip'. We've then passed an argument to pip telling it to install the Flask library. It will then download Flask and any dependencies that Flask has. By dependencies we just mean any other libraries that Flask uses to achieve it's functionality. Often you will find that a given library might make use of many other libraries that in turn depend upon others. Most of the time you needn't worry about this *dependency hierarchy* but you should know that it exists. Listing the installed Python packages should now show us all the Flask libraries and dependencies as shown in Figure 6.

```
1    uv pip list
```

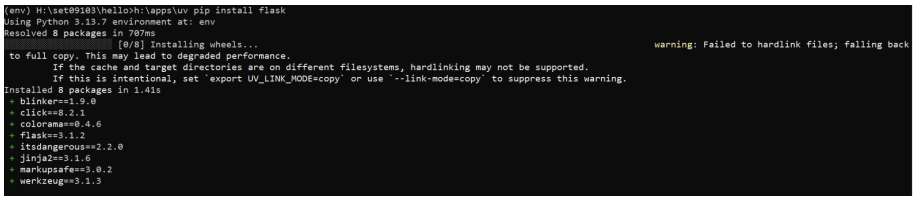

Figure 6

2.4 Creating a Flask Web-App

We are going to:

1. create a source file for a Flask web-app
2. run the web-app and view the results in our Web browser

[9]Like all of the tools we're discovering, pip does much more than this so dig into the documentation if you want to know more

[10]hint: check where the name of the environment is shown at the beginning of the prompt in parenthesis

So what we need to do now is to create an actual Flask app to run. Using your editor (e.g. notepad++) create a new file called hello.py in the hello folder. Add the following code to it:

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route("/")
6 def hello():
7     return "<p>Hello, World!</p>"
```

At this point, note that Python is white-space sensitive and so the indentation has meaning. Getting the indentation correct is important, every line apart from the one that begins with 'return' has no indentation and is left-aligned. The line beginning with return is indented once. If you change any aspect of the code then you might get very different results, or an error that fails to allow the code to run at all. Be aware that Python programming, like any kind of coding, is very sensitive to precision.

All being well, you can now run your new hello world flask web-app using the following:

```
1     uv run flask --app hello run
```

Like the other commands we are passing an argument to uv, this time the *run* argument. We are telling uv to run the flask app and it will look for that app in the current folder in a file called "hello.py". We can see the output of this in Figure 7. As with all output from software, it is there for a reason, so take a moment to read it as it will tell you useful stuff. In particular it will tell you want Flask app is being served and whether the debug mode is activated, something we'll come to quite soon as it's really useful during development. What happens when we run our flask app like this is that Flask's built in WSGI compliant Web-server is run, then the web-app is loaded and *served* by the server to any clients that connect. Whilst your web-app is running, you will keep getting new output from the Flask server written into the terminal.

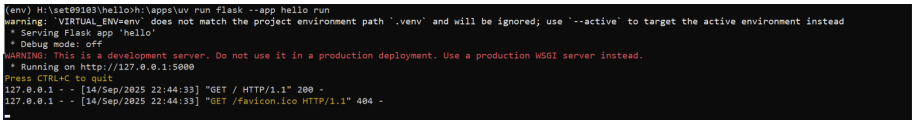

Figure 7

The other kind of output that you are likely to get here, instead of the Flask server output, is a Python stack trace. This is what happens when Python has found a problem in your code that needs to be fixed before it can be successfully run. If you get a stack trace then it usually contains clues as to what to fix. Read through the stack trace each time you get one because it is trying to communicate the problem. The more you read stack traces, the more you will develop skills in navigating the stack trace and discovering where the problem lies.

Assuming that you Flask server is running and your web-app was loaded correctly then you can now go to your browser and type `http://localhost:5000` into the address bar. You should be rewarded by something similar to the browser window shown in Figure 8.

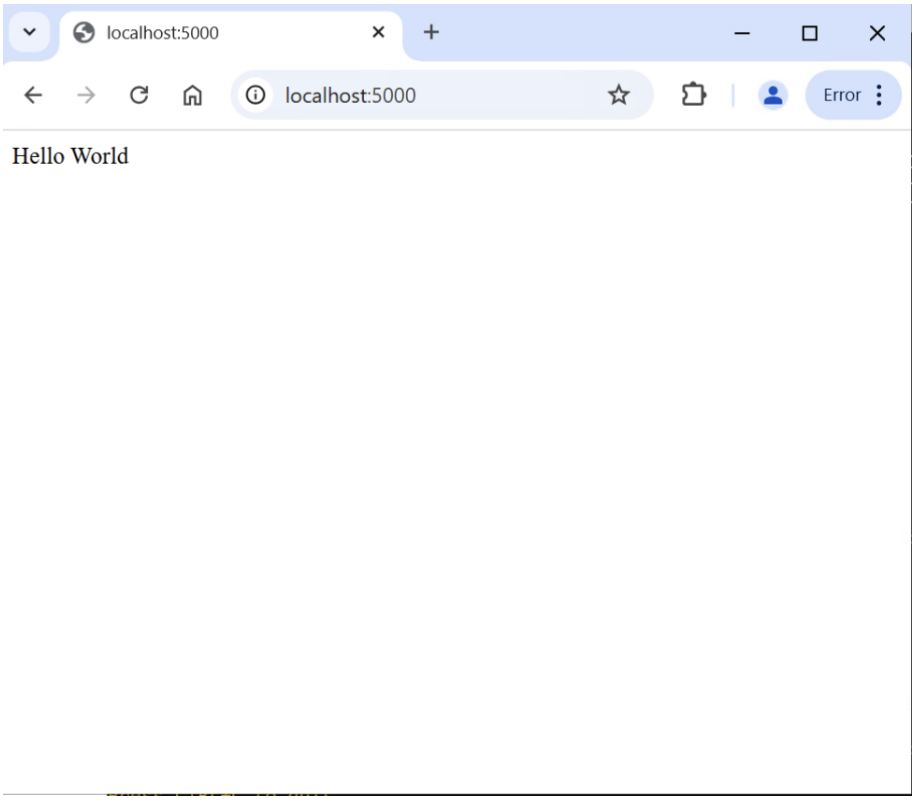

Figure 8

When we are done and want to turn off our Flask server then we can use the ctrl-c command in the terminal where it is running. This means hold down the ctrl button then type c. This will stop the server.

When we are completely finished for the day then we should deactivate our virtual environment as well. We can see that our environment is deactivated if the (env) is gone from the beginning of our terminal prompt.

```
1    env/Scripts/deactivate
```

2.5 Debug Mode

You will notice that you need to stop and start your Flask server each time you make changes to your python source code. The server can run in debug mode which will reload your new code each time you make an alteration to the source code and save it. By adding the –debug argument to the end of your invocation to the server you can start it running in debug mode as follows

```
1    uv run flask --app hello run --debug
```

3 Exercises

Taking the hello.py Flask app as a starting point:

1. Personalise the message so that it prints a personal message to yourself when run.
2. Notice that the hello.py app just returns a string containing HTML. Experiment with adding different HTML tags into it. Perhaps try adding some inline CSS or JS into the string and seeing what happens
3. Try to create a whole web page for yourself using the “return Hello, World! ” string in your hello.py flask app as a starting point. Make the webpage about yourself, a hobby, or something you are interested in. If you want to have the contents extend over multiple lines then you can use triple quotes, e.g. “““ and ””” around the lines.

For each of the exercises, create a new project folder in your set09103 folder. You will need to create a new virtual env each time and install pip. Each projec folder should have a unique name but don't call it app or flask as this will cause errors. Doing this will give you practise in setting up new Flask projects so that it becomes straightforward. We'll be doing that a lot over the coming weeks so making it less of a chore is a good idea.