**Entry points in Python**

What's an entry point?

The most relevant bit of the description in [Wikipedia](https://en.wikipedia.org/wiki/Entry_point) is:

In computer programming, an entry point is where the first instructions of a program are executed, and where the program has access to command line arguments. To start a program's execution, the loader or operating system passes control to its entry point.

What we're going to cover in this blog post is the various methods that you can define entry points in your Python programs. Some with obvious downsides as we'll discover.

Keeping it simple: Python scripts

The simplest thing that you can do is treat your Python file as a self-contained script. Here's the old "Hello world!" example:

#!/usr/bin/env python

def say\_hi():

print('Hello world!')

say\_hi()

Running this file directly gives us an implicit entry point: Just by running the script we get right into execution:

$ ./hellow\_world.py

Hello world!

Easy, right? So what's the problem here?

The downside

What if your project became complex enough to warrant multiple modules? What if you want to import and say hello from another module? With the above, this is what would happen:

In [1]: import hello\_world

hello world!

As you can see in the above, the problem with the Python script approach is that it's overly simple and functionality is executed at **import time**. Not only is it functionally not what you're going to be after, but writing unit test for it will be difficult if not impossible, especially if there is any state change through side effects.

So how can we solve this? Read on!

Leveling up: What's my name?

So now we know that sticking all of our functionality in a Python script isn't the best approach if we want to reuse or write unit tests. So then, what's the next best thing? Using the if **name** == '**main**' pattern.

From the [Python docs](https://docs.python.org/3/library/__main__.html):

'\_\_main\_\_' is the name of the scope in which top-level code executes. A module’s name is set equal to '\_\_main\_\_' when read from standard input, a script, or from an interactive prompt. A module can discover whether or not it is running in the main scope by checking its own **name**, which allows a common idiom for conditionally executing code in a module when it is run as a script or with python -m but not when it is imported.

If we adopt this new learning to hello\_world.py, it would then look like this:

#!/usr/bin/env python

def say\_hi():

print('Hello world!')

if \_\_name\_\_ == '\_\_main\_\_':

say\_hi()

So, now running it from the command line will still yield the same result:

$ ./hellow\_world.py

Hello world!

The major difference though, is that now our import time problem has been solved. Rather than seeing "Hello world!" at module import, we have to explicitly call say\_hi():

In [1]: import hello\_world

# No output here anymore!

In [2]: hello\_world.say\_hi()

Hello world!

This will now also allow us to write and run unit tests against the hello\_world module without having to worry about import time side effects.

The observant reader will notice that the Python docs also refers to running Python modules directly with python -m. This would allow you to run hello\_world without making it executable and adding the shebang at the beginning of the file:

$ python -m hello\_world

Hello world!

Another example of this is the standard module's json.tool:

$ python -m json.tool <<< '{"hello": "world"}'

{

"hello": "world"

}

"This is awesome! All my problems are solved!" says you. "Nay" says I. "Not quite yet".

The downside

Now we have entry points in Python files that we can unit test against. Great. But what are we to do if our project **really** increases in size? How do we find which modules have entry points defined and which ones don't? There **must** be a better option than:

$ grep -r "if \_\_name\_\_\s\*==\s\*['\"]\_\_main\_\_['\"]"`

Indeed there is, dear reader. Indeed there is.

Come together: Script organization

Now that we know about the if \_\_name\_\_ == '\_\_main\_\_' pattern, how do we avoid the confusing aspect of figuring out which files are actually executable? Well, this one has two facets, both of which are fine on their own but shine when used together:

Add a scripts directory to your project

By adding a scripts directory, you can lump all of your entry points into a single directory, making it ridiculously simple for anyone looking at your project to find all of the various entry points for your project. After doing this with hello\_world.py, this is what our directory structure looks like:

$ tree .

.

├── scripts

│   └── hello\_world.py

└── setup.py

For a new user or developer just coming into the project, this makes it tremendously simple to figure out what entry points are available in your project.

How can we enhance this experience you ask? That part comes with the introduction of setuptools. In the above, you'll notice a setup.py file. It's within there that we want to define scripts to be installed when the user installs your package:

setuptools.setup(

name='hello\_world',

version='0.1.0,

scripts=['scripts/hello\_world']

)

For ease of use, I've also dropped the extension in hello\_world.

Now when a user installs the package, here's what they'll see:

$ python setup.py develop

running develop

running egg\_info

creating hello\_world.egg-info

writing hello\_world.egg-info/PKG-INFO

writing dependency\_links to hello\_world.egg-info/dependency\_links.txt

writing top-level names to hello\_world.egg-info/top\_level.txt

writing manifest file 'hello\_world.egg-info/SOURCES.txt'

reading manifest file 'hello\_world.egg-info/SOURCES.txt'

writing manifest file 'hello\_world.egg-info/SOURCES.txt'

running build\_ext

Creating /path/to/.venv/lib/python3.7/site-packages/hello\_world.egg-link (link to .)

Adding hello 0.1.0 to easy-install.pth file

Installing hello\_world script to /path/to/.venv/bin

Installed /path/to/hello\_world

Processing dependencies for hello\_world==0.1.0

Finished processing dependencies for hello\_world==0.1.0

The important thing to note in the above is Installing hello\_world script to /path/to/.venv/bin. This means that users now don't have to fiddle with running ./scripts/hello\_world or python -m hello\_world, they can simply run the script directly:

$ hello\_world

Hello world!

"What is this magic?!" you ask?

For each entry in the scripts entry, a file will be created in your environment's Python bin dir. The resulting file looks like this:

#!/path/to/.venv/bin/python3.7

# EASY-INSTALL-DEV-SCRIPT: 'hello\_world==0.1.0','hello\_world'

\_\_requires\_\_ = 'hello==0.1.0'

\_\_import\_\_('pkg\_resources').require('hello==0.1.0')

\_\_file\_\_ = '/path/to/hello/scripts/hello\_world'

exec(compile(open(\_\_file\_\_).read(), \_\_file\_\_, 'exec'))

So the executable script written to the bin directory effectively acts as a redirect to your source file, executing it.

The combination of the approaches to this point should be enough to satisfy even complex projects. However, if you want to know a completely different option (and my personal favorite), read on!

And now for something completely different: entry\_points

The combination of console\_scripts and entry\_points in a project's setup.py can help us achieve the same results as the previous sections, but without the need for the if \_\_name\_\_ == '\_\_main\_\_' pattern. This is achieved by specifying a callable entry points in the setuptools config:

Noting that we're back to the original directory structure:

$ tree .

.

├── hello\_world.py

└── setup.py

Our setup.py file might now look something like this:

import setuptools

setuptools.setup(

name='hello\_world',

version='0.1.0',

entry\_points={

'console\_scripts': [

'hello\_world=hello\_world:say\_hi',

]

}

)

The observant reader will notice that as we're configuring the entry point to be a callable, this is now something that can easily be unit tested directly.

The magic sauce that allows us to run the script directly looks a little different than the treatment it gets from scripts:

# EASY-INSTALL-ENTRY-SCRIPT: 'hello-world','console\_scripts','hello\_world'

\_\_requires\_\_ = 'hello-world'

import re

import sys

from pkg\_resources import load\_entry\_point

if \_\_name\_\_ == '\_\_main\_\_':

sys.argv[0] = re.sub(r'(-script\.pyw?|\.exe)?$', '', sys.argv[0])

sys.exit(

load\_entry\_point('hello-world', 'console\_scripts', 'hello\_world')()

)

Definitely has some functional differences, but at a high level the results are effectively the same:

$ hello\_world

Hello world!

Gettin' jiggy with it: Single entry point

"So I have a couple ways to define entry points" you say. "But what if I want to have multiple entry points? Does that mean that I need to have multiple executables for my single application?"

Great question! And of course, the answer is "no", but how you solve that problem is entirely up to you.

An approach that I've used in the past is to define a single entry point and to create a commands directory and a module for each sub-command I want available.

The project structure would now look like this:

$ tree .

.

├── hello\_world

│   ├── app.py

│   ├── commands

│   │   ├── \_\_init\_\_.py

│   │   └── say\_hi.py

│   └── \_\_init\_\_.py

└── setup.py

setup.py:

import setuptools

setuptools.setup(

name='hello\_world',

version='0.1.0',

entry\_points={

'console\_scripts': [

'hello\_world=hello\_world.app:main',

]

}

)

The app.py module is simplistic, but captures one potential approach (how it does what it does is an exercise left to the reader):

import argparse

import importlib

import sys

def main():

parser = argparse.ArgumentParser('Say hello to the world')

parser.add\_argument('command', type=str, help='Which command to run')

args = parser.parse\_args()

try:

module = importlib.import\_module(f'hello\_world.commands.{args.command}')

except ModuleNotFoundError:

print('Invalid command')

sys.exit(1)

getattr(module, 'Command')().run()

And the actual command (say\_hi.py) looks like this:

class Command():

def run(self):

print('Hello world!')

If you have multiple commands, which I would hope you are if you're using this approach, then you could either rely on duck typing as I am above, or define an abstract base class and ensure the command being imported in app.py is an instance of that interface. The latter is likely the better of the approaches if multiple people are contributing to the code base.

Then running it after installing becomes:

$ hello\_world say\_hi

Hello world!

Or... Y'know... You could use a sane library like [click](https://pypi.org/project/click/) to do this instead of rebuilding the wheel.

You're just a wee bit more dangerous now!

Hopefully you've learned some new tips and tricks here and your next project won't make your new hire want to pull their hair out trying to figure out how the entry points in your project work.