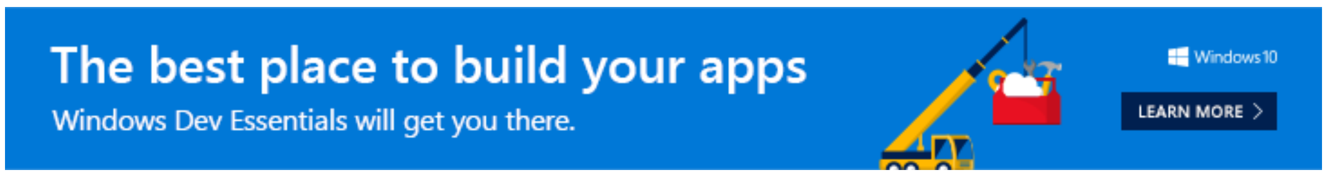
Execution of Python code with -m option or not


The best place to build your apps
Windows Dev Essentials will get you there.
Windows 10
LEARN MORE

The python interpreter has `-m` *module* option that "Runs library module *module* as a script".

With this python code a.py:

```
if __name__ == "__main__":
    print __package__
    print __name__
```

I tested `python -m a` to get

```
"" <-- Empty String
__main__
```

whereas `python a.py` returns

```
None <-- None
__main__
```

To me, those two invocation seems to be the same except __package__ is not None when invoked with -m option.

Interestingly, with `python -m runpy a`, I get the same as `python -m a` with python module compiled to get a.pyc.

What's the (practical) difference between these invocations? Any pros and cons between them?

Also, David Beazley's Python Essential Reference explains it as "The -m option runs a library module as a script which executes inside the __main__ module prior to the execution of the main script". What does it mean?

python module package

edited Sep 7 '16 at 14:54
Mad Physicist
13.7k 8 30 54

asked Mar 7 '14 at 4:25
prosseek
47.1k 118 377 669

1 This should be a good start, I believe – thefourtheye Mar 7 '14 at 12:25

3 Answers

When you use the `-m` command-line flag, Python will import a module *or package* for you, then run it as a script. When you don't use the `-m` flag, the file you named is run as *just a script*.

The distinction is important when you try to run a package. There is a big difference between:

```
python foo/bar/baz.py
```

and

```
python -m foo.bar.baz
```

as in the latter case, `foo.bar` is imported and relative imports will work correctly with `foo.bar` as the starting point.

Demo:

```
$ mkdir -p test/foo/bar
$ touch test/foo/__init__.py
$ touch test/foo/bar/__init__.py
$ cat << EOF > test/foo/bar/baz.py
> if __name__ == "__main__":
>     print __package__
>     print __name__
>
> EOF
$ PYTHONPATH=test python test/foo/bar/baz.py
None
__main__
$ PYTHONPATH=test bin/python -m foo.bar.baz
foo.bar
__main__
```

As a result, Python has to actually care about packages when using the `-m` switch. A normal script can never *be* a package, so `__package__` is set to `None`.

But run a package or module *inside* a package with `-m` and now there is at least the *possibility* of a package, so the `__package__` variable is set to a string value; in the above demonstration it is set to `foo.bar`, for plain modules not inside a package, it is set to an empty string.

As for the `__main__` *module*; Python imports scripts being run as it would a regular module. A new module object is created to hold the global namespace, stored in `sys.modules['__main__']`. This is what the `__name__` variable refers to, it is a key in that structure.

For packages, you can create a `__main__.py` module and have that run when running `python -m package_name`; in fact that's the only way you *can* run a package as a script:

```
$ PYTHONPATH=test python -m foo.bar
python: No module named foo.bar.__main__; 'foo.bar' is a package and cannot be
directly executed
$ cp test/foo/bar/baz.py test/foo/bar/__main__.py
$ PYTHONPATH=test python -m foo.bar
foo.bar
__main__
```

So, when naming a package for running with `-m`, Python looks for a `__main__` module contained in that package and executes that as a script. It's name is then still set to `__main__`, and the module object is still stored in `sys.modules['__main__']`.

edited Mar 7 '14 at 12:35 answered Mar 7 '14 at 12:30
Martijn Pieters ♦
549k 84 1593 1669

When a module inside a package that does not import other modules also has **package** == None: stackoverflow.com/questions/4437394/... – prosseek Mar 7 '14 at 16:20

Execution of Python code with -m option or not

Use the `-m` flag.

The results are pretty much the same when you have a script, but when you develop a package, without the `-m` flag, there's no way to get the imports to work correctly if you want to run a subpackage or module in the package as the main entry point to your program (and believe me, I've tried.)

The docs

Like the docs say:

> Search sys.path for the named module and execute its contents as the `__main__` module.

and

> As with the -c option, the current directory will be added to the start of sys.path.

so

```
python -m pdb
```

is roughly equivalent to

```
python /usr/lib/python3.5/pdb.py
```

(assuming you don't have a package or script in your current directory called pdb.py)

Explanation:

Behavior is made "deliberately similar to" scripts.

> Many standard library modules contain code that is invoked on their execution as a script. An example is the timeit module:

Some python code is intended to be run as a module: (I think this example is better than the commandline option doc example)

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
$ python -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 33.4 usec per loop
$ python -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 25.2 usec per loop
```

And from the release note highlights for Python 2.4:

> The -m command line option - python -m modulename will find a module in the standard library, and invoke it. For example, `python -m pdb` is equivalent to `python /usr/lib/python2.4/pdb.py`

Follow-up Question

> Also, David Beazley's Python Essential Reference explains it as "The -m option runs a library module as a script which executes inside the `__main__` module prior to the execution of the main script".

It means any module you can lookup with an import statement can be run as the entry point of the program - if it has a code block, usually near the end, with `if __name__ == '__main__':` .

`-m` without adding the current directory to the path:

A comment here elsewhere says:

> That the -m option also adds the current directory to sys.path, is obviously a security issue (see: preload attack). This behavior is similar to library search order in Windows (before it had been hardened recently). It's a pity that Python does not follow the trend and does not offer a simple way to disable adding . to sys.path

Well, this demonstrates the possible issue - (in windows remove the quotes):

```
echo "import sys; print(sys.version)" > pdb.py

python -m pdb
3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:13) [MSC v.1900 64
bit (AMD64)]
```

Use the `-I` flag to lock this down for production environments (new in version 3.4):

```
python -Im pdb
usage: pdb.py [-c command] ... pyfile [arg] ...
etc...
```

from the docs:

> **`-I`**
>
> Run Python in isolated mode. This also implies -E and -s. In isolated mode sys.path contains neither the script's directory nor the user's site-packages directory. All PYTHON* environment variables are ignored, too. Further restrictions may be imposed to prevent the user from injecting malicious code.

edited Sep 27 '16 at 15:38

answered Mar 7 '14 at 4:56
Aaron Hall ♦
79.1k 22 182 176

3 And how does this answer address the questions the OP raised? Why is `__package__` set to `None` or `''` , for example? – Martijn Pieters ♦ Mar 7 '14 at 11:16

@MadPhysicist I've updated my answer a bit here, there you go. Satisfactory? – Aaron Hall ♦ Sep 7 '16 at 22:35

Very much so. Flipped the vote. – Mad Physicist Sep 8 '16 at 13:52

The main reason to run a module (or package) as a script with -m is to simplify deployment, especially on Windows. You can install scripts in the same place in the Python library where modules normally go - instead of polluting PATH or global executable directories such as ~/.local (the per-user scripts directory is ridiculously hard to find in Windows).

Then you just type -m and Python finds the script automagically. For example, `python -m pip` will find the correct pip for the same instance of Python interpreter which executes it. Without -m, if user has several Python versions installed, which one would be the "global" pip?

If user prefers "classic" entry points for command-line scripts, these can be easily added as small scripts somewhere in PATH, or pip can create these at install time with entry_points parameter in setup.py.

So just check for `__name__ == '__main__'` and ignore other non-reliable implementation details.

edited Sep 8 '16 at 11:06

answered Sep 7 '16 at 22:40
ddbug
413 1 9

That the -m option also adds the current directory to sys.path, is obviously a security issue (see: **preload attack**). This behavior is similar to library search order in Windows (before it had been hardened recently). It's a pity that Python does not follow the trend and does not offer a simple way to disable adding . to sys.path. – ddbug Sep 8 '16 at 11:00