PEP 338 -- Executing modules as scripts

PEP:	338
Title:	Executing modules as scripts
Author:	Nick Coghlan <ncoghlan at gmail.com>
Status:	Final
Type:	Standards Track
Created:	16-Oct-2004
Python-Version:	2.5
Post-History:	8-Nov-2004, 11-Feb-2006, 12-Feb-2006, 18-Feb-2006

Contents

Abstract (#id11)

This PEP defines semantics for executing any Python module as a script, either with the -m command line switch, or by invoking it via runpy.run_module(modulename) .

The -m switch implemented in Python 2.4 is quite limited. This PEP proposes making use of the PEP 302 (/dev/peps/pep-0302) [4] (#id9) import hooks to allow any module which provides access to its code object to be executed.

Rationale (#id12)

Python 2.4 adds the command line switch -m to allow modules to be located using the Python module namespace for execution as scripts. The motivating examples were standard library modules such as pdb and profile , and the Python 2.4 implementation is fine for this limited purpose.

A number of users and developers have requested extension of the feature to also support running modules located inside packages. One example provided is pychecker's pychecker.checker module. This capability was left out of the Python 2.4 implementation because the implementation of this was significantly more complicated, and the most appropriate strategy was not at all clear.

The opinion on python-dev was that it was better to postpone the extension to Python 2.5, and go through the PEP process to help make sure we got it right.

Since that time, it has also been pointed out that the current version of -m does not support zipimport or any other kind of alternative import behaviour (such as frozen modules).

Providing this functionality as a Python module is significantly easier than writing it in C, and makes the functionality readily available to all Python programs, rather than being specific to the CPython interpreter. CPython's command line switch can then be rewritten to make use of the new module.

Scripts which execute other scripts (e.g. profile , pdb) also have the option to use the new module to provide -m style support for identifying the script to be executed.

Scope of this proposal (#id13)

In Python 2.4, a module located using -m is executed just as if its filename had been provided on the command line. The goal of this PEP is to get as close as possible to making that statement also hold true for modules inside packages, or accessed via alternative import mechanisms (such as zipimport).

Prior discussions suggest it should be noted that this PEP is **not** about changing the idiom for making Python modules also useful as scripts (see PEP 299 (/dev/peps/pep-0299) [1] (#id6)). That issue is considered orthogonal to the specific feature addressed by this PEP.

Current Behaviour (#id14)

Before describing the new semantics, it's worth covering the existing semantics for Python 2.4 (as they are currently defined only by the source code and the command line help).

When -m is used on the command line, it immediately terminates the option list (like -c). The argument is interpreted as the name of a top-level Python module (i.e. one which can be found on sys.path).

If the module is found, and is of type PY_SOURCE or PY_COMPILED , then the command line is effectively reinterpreted from python <options> -m <module> <args> to python <options> <filename> <args> . This includes setting sys.argv[0] correctly (some scripts rely on this - Python's own regrtest.py is one example).

If the module is not found, or is not of the correct type, an error is printed.

Proposed Semantics (#id15)

The semantics proposed are fairly simple: if -m is used to execute a module the PEP 302 (/dev/peps/pep-0302) import mechanisms are used to locate the module and retrieve its compiled code, before executing the module in accordance with the semantics for a top-level module. The interpreter does this by invoking a new standard library function runpy.run_module .

This is necessary due to the way Python's import machinery locates modules inside packages. A package may modify its own __path__ variable during initialisation. In addition, paths may be affected by *.pth files, and some packages will install custom loaders on sys.metapath . Accordingly, the only way for Python to reliably locate the module is by importing the containing package and using the PEP 302 (/dev/peps/pep-0302) import hooks to gain access to the Python code.

Note that the process of locating the module to be executed may require importing the containing package. The effects of such a package import that will be visible to the executed module are:

- the containing package will be in sys.modules
- any external effects of the package initialisation (e.g. installed import hooks, loggers, atexit handlers, etc.)

Reference Implementation (#id16)

A reference implementation is available on SourceForge ([2] (#id7)), along with documentation for the library reference ([5] (#id10)). There are two parts to this implementation. The first is a proposed standard library module runpy . The second is a modification to the code implementing the -m switch to always delegate to runpy.run_module instead of trying to run the module directly. The delegation has the form:

```
runpy.run_module(sys.argv[0], run_name="__main__", alter_sys=True)
```

run_module is the only function runpy exposes in its public API.

run_module(mod_name[, init_globals][, run_name][, alter_sys])

> *Execute the code of the specified module and return the resulting module globals dictionary. The module's code is first located using the standard import mechanism (refer to [PEP 302 (/dev/peps/pep-0302)](/dev/peps/pep-0302) for details) and then executed in a fresh module namespace.*
>
> *The optional dictionary argument* init_globals *may be used to pre-populate the globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by the run_module function.*
>
> *The special global variables* __name__ *,* __file__ *,* __loader__ *and* __builtins__ *are set in the globals dictionary before the module code is executed.*
>
> __name__ *is set to* run_name *if this optional argument is supplied, and the original* mod_name *argument otherwise.*
>
> __loader__ *is set to the [PEP 302 (/dev/peps/pep-0302)](/dev/peps/pep-0302) module loader used to retrieve the code for the module (This loader may be a wrapper around the standard import mechanism).*
>
> __file__ *is set to the name provided by the module loader. If the loader does not make filename information available, this argument is set to* None *.*
>
> __builtins__ *is automatically initialised with a reference to the top level namespace of the* __builtin__ *module.*
>
> *If the argument* alter_sys *is supplied and evaluates to* True *, then* sys.argv[0] *is updated with the value of* __file__ *and* sys.modules[__name__] *is updated with a temporary module object for the module being executed. Both* sys.argv[0] *and* sys.modules[__name__] *are restored to their original values before this function returns.*

When invoked as a script, the runpy module finds and executes the module supplied as the first argument. It adjusts sys.argv by deleting sys.argv[0] (which refers to the runpy module itself) and then invokes run_module(sys.argv[0], run_name="__main__", alter_sys=True) .

Import Statements and the Main Module (#id17)

The release of 2.5b1 showed a surprising (although obvious in retrospect) interaction between this PEP and [PEP 328 (/dev/peps/pep-0328)](/dev/peps/pep-0328) - explicit relative imports don't work from a main module. This is due to the fact that relative imports rely on __name__ to determine the current module's position in the package hierarchy. In a main module, the value of __name__ is always '__main__' , so explicit relative imports will always fail (as they only work for a module inside a package).

Investigation into why implicit relative imports *appear* to work when a main module is executed directly but fail when executed using -m showed that such imports are actually always treated as absolute imports. Because of the way direct execution works, the package containing the executed module is added to sys.path, so its sibling modules are actually imported as top level modules. This can easily lead to multiple copies of the sibling modules in the application if implicit relative imports are used in modules that may be directly executed (e.g. test modules or utility scripts).

For the 2.5 release, the recommendation is to always use absolute imports in any module that is intended to be used as a main module. The -m switch provides a benefit here, as it inserts the current directory into sys.path, instead of the directory contain the main module. This means that it is possible to run a module from inside a package using -m so long as the current directory contains the top level directory for the package. Absolute imports will work correctly even if the package isn't installed anywhere else on sys.path. If the module is executed directly and uses absolute imports to retrieve its sibling modules, then the top level package directory needs to be installed somewhere on sys.path (since the current directory won't be added automatically).

Here's an example file layout:

```
devel/
    pkg/
        __init__.py
        moduleA.py
        moduleB.py
        test/
            __init__.py
            test_A.py
            test_B.py
```

So long as the current directory is devel , or devel is already on sys.path and the test modules use absolute imports (such as import pkg moduleA to retrieve the module under test, [PEP 338 (/dev/peps/pep-0338)](/dev/peps/pep-0338) allows the tests to be run as:

```
python -m pkg.test.test_A
python -m pkg.test.test_B
```

The question of whether or not relative imports should be supported when a main module is executed with -m is something that will be revisited for Python 2.6. Permitting it would require changes to either Python's import semantics or the semantics used to indicate when a module is the main module, so it is not a decision to be made hastily.

Resolved Issues (#id18)

There were some key design decisions that influenced the development of the runpy module. These are listed below.

- The special variables __name__ , __file__ and __loader__ are set in a module's global namespace before the module is executed. As run_module alters these values, it does **not** mutate the supplied dictionary. If it did, then passing globals() to this function could have nasty side effects.
- Sometimes, the information needed to populate the special variables simply isn't available. Rather than trying to be too clever, these variables are simply set to None when the relevant information cannot be determined.
- There is no special protection on the alter_sys argument. This may result in sys.argv[0] being set to None if file name information is not available.
- The import lock is NOT used to avoid potential threading issues that arise when alter_sys is set to True. Instead, it is recommended that threaded code simply avoid using this flag.

Alternatives (#id19)

The first alternative implementation considered ignored packages' __path__ variables, and looked only in the main package directory. A Python script with this behaviour can be found in the discussion of the execmodule cookbook recipe [3] (#id8) .

The execmodule cookbook recipe itself was the proposed mechanism in an earlier version of this PEP (before the PEP's author read PEP 302 (/dev/peps/pep-0302)).

Both approaches were rejected as they do not meet the main goal of the -m switch -- to allow the full Python namespace to be used to locate modules for execution from the command line.

An earlier version of this PEP included some mistaken assumptions about the way exec handled locals dictionaries and code from function objects. These mistaken assumptions led to some unneeded design complexity which has now been removed - run_code shares all of the quirks of exec .

Earlier versions of the PEP also exposed a broader API that just the single run_module() function needed to implement the updates to the -m switch. In the interests of simplicity, those extra functions have been dropped from the proposed API.

After the original implementation in SVN, it became clear that holding the import lock when executing the initial application script was not correct (e.g. python -m test.regrtest test_threadedimport failed). So the run_module function only holds the import lock during the actual search for the module, and releases it before execution, even if alter_sys is set.

References (#id20)

[1] (#id2) Special __main__() function in modules (http://www.python.org/dev/peps/pep-0299/ (http://www.python.org/dev/peps/pep-0299/))

[2] (#id3) PEP 338 (/dev/peps/pep-0338) implementation (runpy module and -m update) (http://www.python.org/sf/1429601 (http://www.python.org/sf/1429601))

[3] (#id5) execmodule Python Cookbook Recipe (http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/307772 (http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/307772))

[4] (#id1) New import hooks (http://www.python.org/dev/peps/pep-0302/ (http://www.python.org/dev/peps/pep-0302/))

[5] (#id4) PEP 338 (/dev/peps/pep-0338) documentation (for runpy module) (http://www.python.org/sf/1429605 (http://www.python.org/sf/1429605))

Copyright (#id21)

This document has been placed in the public domain.

Source: https://github.com/python/peps/blob/master/pep-0338.txt (https://github.com/python/peps/blob/master/pep-0338.txt)

Tweets by @ThePSF

Python Software
@ThePSF

The Ego-less Developer: Community Service Award Recipient Ian Cordasco goo.gl/fb/Z4Z8H2

21 Apr

Python Software
@ThePSF

Pay What You Want for "The Humble Book Bundle: Python" and Benefit the PSF goo.gl/fb/yuzDlp

05 Apr

Python Software
@ThePSF

Python at Google Summer of Code: Apply by April 3 goo.gl/fb/izhpzd

29 Mar

Python Software
@ThePSF

Ernest takes the call - Community Service Award Recipient Ernest Durbin III goo.gl/fb/g1z0vg

10 Mar

Embed View on Twitter

The PSF

The Python Software Foundation is the organization behind Python. Become a member of the PSF and help advance the software and our mission.

Back to Top

Back to Top

Copyright ©2001-2017. Python Software Foundation Legal Statements Privacy Policy