**How to easily import your own Python packages**

**Reference：<https://ajminich.com/2013/06/22/how-to-easily-import-your-own-python-packages/>**

[**AJ Minich**](https://ajminich.com/author/ajminich/) / [June 22, 2013](https://ajminich.com/2013/06/22/how-to-easily-import-your-own-python-packages/)

Python is my favorite language by far, but here’s one thing it’s absolutely terrible at: package management. Some decent tools ([easy\_install](http://pythonhosted.org/distribute/easy_install.html), [pip](https://pypi.python.org/pypi/pip)) exist, but what if you’ve created your own [module or package](http://docs.python.org/2/tutorial/modules.html)?  Ideally, you could start Python from anywhere and simply run:

>>> import mypackage

If you’re running Python from the same directory where the package is stored, this will be fine. But if you run from anywhere outside that directory, Python won’t be able to find the package.

So, here are three options for importing your own Python packages, in order of increasing hack-iness.

**Method 1: Distutils**

The professional way of creating a distributable package would be to create a setup.py file and install the package from the command-line:

$ python setup.py install

This industry-standard method uses [distutils](http://docs.python.org/2/distutils/), a package available in every Python version since 2.6. After running the setup, you can import your module from anywhere.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| * Reliable. Distutils takes care of the dirty work of figuring out where to install the package. * Easy for others to install. * Gets you one step closer to being able to open-source your package on Github or SourceForge! | * Creating the distutils package can be a little tedious, especially if you’ve never done it before. * Distutils actually creates a copy of your modules and installs them. Thus, if you edit your package and want to use the new changes, you will have to rerun the setup and installation process. |

**Method 2: Environment Variables**

Environment variables contain system information, so why not use them to store where your package is located? From command-line, run:

$ export package=""

where is your package’s directory. Thus, the package environment variable will be the location of your package. Typically, you’d have users add the line above to their .bashrc, or put it in a setup script that the user has to run in each session before using your package.

Then, to import your module in Python, run:

import os

path = os.environ["package"]

os.sys.path.append(path)

import mypackage

A bit wordy, but it gets the job done!

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| * Imports your module directly, so you don’t have to keep re-installing every time you change code. * Since this doesn’t actually “install” anything, it will work with any Python distribution on your system. | * Requires the user to either source the setup script in each terminal session, or follow a pretty arcane process to export the right environment variable. * Wordy. Importing a single package takes 3 lines instead of 1. * Makes scripts that import the module less portable, since users will need to set up environment variables correctly to use them. |

**Method 3: Setuptools Develop**

If you really get into the annals of Python, you’ll realize that “installing” a package only means copying it to a directory inside of Python called “site-packages”. This directory contains every package you’ve installed into Python; you can determine its location by running:

$ python -c "from distutils.sysconfig import get\_python\_lib; print(get\_python\_lib())"

So, when you run “import package” in Python, this is what it actually does:

1. look for the package in fundamental Python libraries (for example, the osand sys packages)
2. look for a local file/folder in the current directory with the same name as the package
3. look in site-packages for a file/folder with the same name as the package

So, assuming you’re on a Unix machine, you could make a symbolic link from the site-packages directory to your package. However, setuptools (a relative of distutils) allows you to do this easily by running the ‘develop’ target:

$ python setup.py develop

Then you can import your package the way you always wanted:

>>> import mypackage

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| * As similar to distutils as you can get without actually installing the package. * Easy – only requires a single command to set up. | * Like Distutils, the setup works only for the specific Python distribution. * The rationale behind the setup process is a bit confusing for users. |

**Wrapup**

Of these three methods, I find the ‘develop’ method ideal for development environments and distutils great for production. But every situation is different, so use the one that makes the most sense for your environment. Happy coding!