setup.py

All Versions

Examples

Purpose of setup.py

The setup script is the centre of all activity in building, distributing, and installing modules using the Distutils. It's purpose is the correct installation of the software.

If all you want to do is distribute a module called foo, contained in a file foo.py, then your setup script can be as simple as this:

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

To create a source distribution for this module, you would create a setup script, setup.py, containing the above code, and run this command from a terminal:

```
python setup.py sdist
```

sdist will create an archive file (e.g., tarball on Unix, ZIP file on Windows) containing your setup script setup.py, and your module foo.py. The archive file will be named foo-1.0.tar.gz (or .zip), and will unpack into a directory foo-1.0.

If an end-user wishes to install your foo module, all she has to do is download foo-1.0.tar.gz (or .zip), unpack it, and—from the foo-1.0 directory—run

```
python setup.py install
```

Using source control metadata in setup.py

setuptools_scm is an officially-blessed package that can use Git or Mercurial metadata to determine the version number of your package, and find Python packages and package data to include in it.

```
from setuptools import setup, find_packages

setup(
    setup_requires=['setuptools_scm'],
    use_scm_version=True,
    packages=find_packages(),
    include_package_data=True,
)
```

This example uses both features; to only use SCM metadata for the version, replace the call to find_packages() with your manual package list, or to only use the package finder, remove use_scm_version=True .

Adding command line scripts to your python package

Command line scripts inside python packages are common. You can organise your package in such a way that when a user installs the package, the script will be available on their path.

If you had the greetings package which had the command line script hello_world.py .

```
greetings/
   greetings/
      __init__.py
      hello_world.py
```

You could run that script by running:

```
python greetings/greetings/hello_world.py
```

However if you would like to run it like so:

```
hello_world.py
```

You can achieve this by adding `scripts` to your `setup()` in `setup.py` like this:

```
from setuptools import setup
setup(
  name='greetings',
  scripts=['hello_world.py']
)
```

When you install the greetings package now, `hello_world.py` will be added to your path.

Another possibility would be to add an entry point:

```
entry_points={'console_scripts': ['greetings=greetings.hello_world:main']}
```

This way you just have to run it like:

```
greetings
```

Adding installation options

As seen in previous examples, basic use of this script is:

```
python setup.py install
```

But there is even more options, like installing the package and have the possibility to change the code and test it without having to re-install it. This is done using:

```
python setup.py develop
```

If you want to perform specific actions like compiling a *Sphinx* documentation or building *fortran* code, you can create your own option like this:

```
cmdclasses = dict()

class BuildSphinx(Command):

    """Build Sphinx documentation."""

    description = 'Build Sphinx documentation'
    user_options = []

    def initialize_options(self):
        pass

    def finalize_options(self):
        pass

    def run(self):
        import sphinx
        sphinx.build_main(['setup.py', '-b', 'html', './doc', './doc/_build/html'])
        sphinx.build_main(['setup.py', '-b', 'man', './doc', './doc/_build/man'])

cmdclasses['build_sphinx'] = BuildSphinx

setup(
...
cmdclass=cmdclasses,
)
```

`initialize_options` and `finalize_options` will be executed before and after the `run` function as their names suggests it.

After that, you will be able to call your option:

```
python setup.py build_sphinx
```

Syntax

Parameters

Parameter	Usage
name	Name of your distribution.
version	Version string of your distribution.
packages	List of Python packages (that is, directories containing modules) to include. This can be specified manually, but a call to setuptools.find_packages() is typically used instead.
py_modules	List of top-level Python modules (that is, single .py files) to include.

Remarks

For further information on python packaging see:

Introduction

For writing official packages there is a packaging user guide .