4 Distribution

It's a safe bet you'll want to distribute your software at some point. As tempted as you might be to just zip up your code and upload it to the Internet, Python provides tools to help you make sure your end users will have no trouble getting your software to work. You should already be familiar with using `setup.py` to install Python applications and libraries, but you've probably never delved into how it actually works behind the scenes, or how to make a `setup.py` of your own.

4.1 A bit of history

distutils has been part of the standard Python library since 1998. It was originally developed by Greg Ward, who sought to create an easy way for developers to automate the installation process for their end users:

Example 4.1 `setup.py` using *distutils*

```
#!/usr/bin/python
from distutils.core import setup

setup(name="rebuildd",
      description="Debian packages rebuild tool",
      author="Julien Danjou",
      author_email="acid@debian.org",
      url="http://julien.danjou.info/software/rebuildd.html",
      packages=['rebuildd'])
```

And that's it. All users have to do to `build` or `install` your software is run `setup.py` with the appropriate command. If your distribution includes C modules in addition to native Python ones, it can even handle those automatically as well.

Development on *distutils* was abandoned in 2000; since then, other developers picked up where it left off, building their own tools based on it. One of the most notable successors to *distutils* is the packaging library known as *setuptools*, which offered more frequent updates and advanced features such as automatic dependency handling, the *Egg* distribution format, and the `easy_install` command. Since *distutils* was still the canonical means of packaging software included with the Python Standard Library, *setuptools* also provided a degree of backwards compatibility with it.

Example 4.2 `setup.py` using *setuptools*

```
#!/usr/bin/env python
import setuptools


setuptools.setup(
    name="pymunincli",
    version="0.2",
    author="Julien Danjou",
    author_email="julien@danjou.info",
    description="munin client library",
    license="GPL",
    url="http://julien.danjou.info/software/pymunincli/",
    packages=['munin'],
    classifiers=[
        "Development Status :: 2 - Pre-Alpha",
        "Intended Audience :: Developers",
        "Intended Audience :: Information Technology",
        "License :: OSI Approved :: GNU General Public License (GPL)",
        "Operating System :: OS Independent",
        "Programming Language :: Python"
    ],
)
```

Eventually, development on *setuptools* slowed down, and people began to consider it a dead project like the original *distutils*. It wasn't long before another group of developers forked it to create a new library called *distribute*, which offered several advantages over *setuptools*, including fewer bugs and Python 3 support. All the best stories have a twist ending, though, and this one's no different: in March 2013, the teams behind *setuptools* and *distribute* decided to merge their code bases under the aegis of the original *setuptools* project. So *distribute* is now deprecated, and *setuptools* is once more the canonical way to handle advanced Python installations.

While all this was happening, another project known as *distutils2* was developed with the intention of replacing *distutils* in the Python Standard Library outright. One of its most notable differences from both *distutils* and *setuptools* was that it stored package metadata in a plain text file, `setup.cfg`, which was both easier for developers to write and easier for external tools to read. However, it also retained some of the failings of *distutils*, such as its obtuse command-based design, and lacked support for things like entry points and native script execution on Windows - both features provided by *setuptools*. For these and other reasons, plans to include *distutils2* in the Python 3.3 Standard Library as *packaging* fell through, and the project was abandoned in 2012.

However, *packaging* still has a chance to rise from the ashes through *distlib*, an up-and-coming effort to replace *distutils* which - hopefully - will become part of the Standard Library in 3.4. It includes the best features from *packaging* and implements the basic groundwork described in the packaging-related PEPs.

So, to recap:

- *distutils* is part of the Python standard library and can handle simple package installations.
- *setuptools*, the standard for advanced package installations, was at first deprecated but is now back in active development.
- *distribute* has been merged back into *setuptools* as of version 0.7.
- *distutils2* (a.k.a. *packaging*) has been abandoned.
- *distlib might* replace *distutils* in the future.

There are other packaging libraries out there, though these five are the ones you'll encounter the most in practice. Be careful when looking up information about them on the Internet: there's plenty of documentation out there that's outdated due to the complicated history outlined above. The official documentation is, at least, up to date.

The short version of all this is, *setuptools* is the distribution library to use for the time being, but keep an eye out for *distlib* in the future.

4.2 Packaging with *pbr*

Now that I've spent some pages making your head confused with a lot of distribution tools, let's talk, about another tool and alternative, called *pbr*.

You probably already have written some package and tried to write a `setup.py`, either by copying one from some other project, or by skimming through the documentation. It isn't an obvious task, as the various problem we discussed earlier about which tool to use are usually a first obstacle. In this section I want to introduce you to *pbr*, a tool you should use to write your next *setup.py* so you'll never have to lose your time on that part again.

pbr stands for *Python Build Reasonableness*. The project has been started inside OpenStack as a set of tools around *setuptools* to facilitate installation and deployment of packages. It takes inspiration from *distutils2*, using a `setup.cfg` file to describe the packager's intents.

This is how a `setup.py` using *pbr* looks like:

```
import setuptools


setuptools.setup(setup_requires=['pbr'], pbr=True)
```

Two lines of code – it's that simple. The actual metadata that the setup requires is stored in `setup.cfg`:

```
[metadata]
name = foobar
author = Dave Null
author-email = foobar@example.org
summary = Package doing nifty stuff
license = MIT
description-file =
    README.rst
home-page = http://pypi.python.org/pypi/foobar
requires-python = >=2.6
classifier =
    Development Status :: 4 - Beta
    Environment :: Console
    Intended Audience :: Developers
    Intended Audience :: Information Technology
    License :: OSI Approved :: Apache Software License
    Operating System :: OS Independent
    Programming Language :: Python
```

```
[files]
packages =
    foobar
```

Sound familiar? That's right – this particular way of doing things was directly inspired by *distutils2*.

pbr also offers other features such as:

- automatic dependency installation based on `requirements.txt`
- automatic documentation using Sphinx
- automatic generation of `AUTHORS` and `ChangeLog` files based on *git* history
- automatic creation of file lists for *git*
- version management based on *git* tags

And all this with little to no effort on your part. *pbr* is well-maintained and in very active development, so if you have any plans to distribute your software, you should seriously consider including *pbr* in those plans.

4.3 The *Wheel* format

For most of Python's existence, there's been no official standard distribution format. While different distribution tools still generally use some kind of common archive format – even the *Egg* format introduced by *setuptools* is just a zip file with a different extension – their metadata and package structures are incompatible with each other. This problem was compounded when an official installation standard was finally defined in PEP 376, which was also incompatible with existing formats.

To solve these problems, PEP 427 was written to define a new standard for Python distribution packages called *Wheel.* The reference implementation of this format is available as a tool, also called *wheel*.

Wheel is supported by *pip* starting with version 1.4. If you're using **setuptools** and have the `wheel` package installed, it is automatically integrated as a command:

```
python setup.py bdist_wheel
```

This will create a `.whl` file in the `dist` directory. Like with the *Egg* format, a *Wheel* archive is just a zip file with a different extension, except *Wheel* archives don't require installation – you can load and run your code just by adding a slash followed by the name of your module:

```
$ python wheel-0.21.0-py2.py3-none-any.whl/wheel -h
usage: wheel [-h]

             {keygen,sign,unsign,verify,unpack,install,install-scripts, ↩
                convert,help}
             ...

positional arguments:
[...]
```

You might be surprised to learn this isn't a feature introduced by the *Wheel* format. Python can also run regular zip files as well, just like with Java's `.jar` files:

```
python foobar.zip
```

This is equivalent to:

```
PYTHONPATH=foobar.zip python -m __main__
```

In other words, the `__main__` module for your program will automatically be imported from `__main__.py`. It's also possible to import `__main__` from a module you specify by appending a slash followed by its name, just like with `Wheel`:

```
python foobar.zip/mymod
```

This is equivalent to:

```
PYTHONPATH=foobar.zip python -m mymod.__main__
```

One of the advantages of *Wheel* is that its naming conventions allow you to specify whether your distribution is intended for a specific architecture and/or Python implementation (CPython, PyPy, Jython, etc.). This is particularly useful if you need to distribute modules written in C.

4.4 Package installation

setuptools introduced the first useful command for installing packages, *easy_install*. It allows you to install Python modules from *Egg* archives with a single command; unfortunately, *easy_install* has suffered a bad reputation from the beginning due to some of its more questionable behaviors, such as ignoring best practices for system administration and its lack of uninstall functionality.

The *pip* project offers a much better way to handle package installations. It's actively developed, well-maintained, and will be included with Python starting in 3.4 [1]. It can install or uninstall packages from PyPI, a tarball, or a *Wheel* (see Section 4.3) archive.

Its usage is simple:

```
$ pip install --user voluptuous
Downloading/unpacking voluptuous
  Downloading voluptuous-0.8.3.tar.gz
  Storing download in cache at ./.cache/pip/https%3A%2F%2Fpypi.python.org%2 ↩
     Fpackages%2Fsource%2Fv%2Fvoluptuous%2Fvoluptuous-0.8.3.tar.gz
```

[1]See PEP 453 and the `ensurepip` module

```
  Running setup.py egg_info for package voluptuous
    WARNING: Could not locate pandoc, using Markdown long_description.

Requirement already satisfied (use --upgrade to upgrade): distribute in / ↩
   usr/lib/python2.7/dist-packages (from voluptuous)
Installing collected packages: voluptuous
  Running setup.py install for voluptuous
    WARNING: Could not locate pandoc, using Markdown long_description.

Successfully installed voluptuous
Cleaning up...
```

You can also provide a `--user` option that makes *pip* install the package in your home directory. This avoids polluting your operating system directories with packages installed system-wide.

Tip

If you're using *pip* to install the same packages over and over, you can make it use a local cache instead of downloading the packages each time. Just set the environment variable `PIP_DOWNLOAD_CACHE` to a directory: *pip* will then use it to store downloaded tarballs and will check that location for packages before downloading them. This is very useful when using *tox* (see Section 6.7), which needs to download packages to build virtual environments. You can also add the `download-cache` option to your `~/.pip/pip.conf` file.

You can list the packages that are currently installed by using the `pip freeze` command:

```
$ pip freeze
Babel==1.3
Jinja2==2.7.1
```

```
commando=0.3.4
…
```

All other installation tools are being deprecated in favor of *pip*, so you shouldn't have any trouble if you treat it as your one-stop shop for all your package management needs.

4.5 Sharing your work with the world

Once you have a proper `setup.py` file, it's easy to build a source tarball that you can distribute. Just use the `sdist` command:

Example 4.3 Using `setup.py sdist`

```
$ python setup.py sdist
running sdist
[pbr] Writing ChangeLog
[pbr] Generating AUTHORS
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Processing SOURCES.txt
[pbr] In git context, generating filelist from git
warning: no previously-included files matching '*.pyc' found anywhere in ↩
   distribution
writing manifest file 'ceilometer.egg-info/SOURCES.txt'
running check
copying setup.cfg -> ceilometer-2014.1.a6.g772e1a7
```

```
Writing ceilometer-2014.1.a6.g772e1a7/setup.cfg

[…]

Creating tar archive
removing 'ceilometer-2014.1.a6.g772e1a7' (and everything under it)
```

This will create a tarball under the `dist` directory of your source tree that contains all your packages and can be used to install your software. As seen in Section 4.3, you can also build *Wheel* archives using the *bdist_wheel* command.

The final step is to make things easy on your end users by setting things up where your package can be installed using *pip*. This means publishing your project to PyPI.

Since you'll probably make mistakes if this is your first time, it pays to test out the publishing process in a safe sandbox rather than on the production server. You can use the PyPI staging server for this purpose: it replicates all the functionality of the main index, but it's used solely for testing purposes.

The first step is to register your project on the test server. Start by opening your `~/.pypirc` file and adding these lines:

```
[distutils]
index-servers =
    testpypi

[testpypi]
username = <your username>
password = <your password>
repository = https://testpypi.python.org/pypi
```

Now you can register your project in the index:

```
$ python setup.py register -r testpypi
```

```
running register
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Reusing existing SOURCES.txt
running check
Registering ceilometer to https://testpypi.python.org/pypi
Server response (200): OK
```

Finally, you can upload a source distribution tarball:

```
% python setup.py sdist upload -r testpypi
running sdist
[pbr] Writing ChangeLog
[pbr] Generating AUTHORS
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Processing SOURCES.txt
[pbr] In git context, generating filelist from git
warning: no previously-included files matching '*.pyc' found anywhere in ↩
   distribution
writing manifest file 'ceilometer.egg-info/SOURCES.txt'
running check
creating ceilometer-2014.1.a6.g772e1a7
```

```
[…]

copying setup.cfg -> ceilometer-2014.1.a6.g772e1a7
Writing ceilometer-2014.1.a6.g772e1a7/setup.cfg
Creating tar archive
removing 'ceilometer-2014.1.a6.g772e1a7' (and everything under it)
running upload
Submitting dist/ceilometer-2014.1.a6.g772e1a7.tar.gz to https://testpypi. ↩
   python.org/pypi
Server response (200): OK
```

As well as a *Wheel* archive:

```
$ python setup.py bdist_wheel upload -r testpypi
running bdist_wheel
running build
running build_py
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Reusing existing SOURCES.txt
installing to build/bdist.linux-x86_64/wheel
running install
running install_lib
creating build/bdist.linux-x86_64/wheel

[…]
```

```
creating build/bdist.linux-x86_64/wheel/ceilometer-2014.1.a6.g772e1a7.dist- ↩
   info/WHEEL
running upload
Submitting /home/jd/Source/ceilometer/dist/ceilometer-2014.1.a6.g772e1a7- ↩
   py27-none-any.whl to https://testpypi.python.org/pypi
Server response (200): OK
```

You should now be able to search for your package on the PyPi staging server and see whether it uploaded properly. You can also try installing it using *pip*, specifying the test server using the `-i` option:

```
$ pip install -i https://testpypi.python.org/pypi ceilometer
```

If everything checks out, you can continue to the next step: uploading your project to the main PyPI server. Just add your credentials and the details for the server to your ~/.pypirc` file:

```
[distutils]
index-servers =
    pypi
    testpypi

[pypi]
username = <your username>
password = <your password>

[testpypi]
repository = https://testpypi.python.org/pypi
username = <your username>
password = <your password>
```

Running *register* and *upload* with the `-r pypi` switch will now upload your package to PyPI proper.

4.6 Interview with Nick Coghlan

Nick is a Python core developer working at Red Hat. He has written several PEP proposals, including PEP 426 (*Metadata for Python Software Packages 2.0*) for which he is acting as *BDFL* [2] delegate.

The number of packaging solutions (*distutils*, *setuptools*, *distutils2*, *distlib*, *bento*, *pbr*, etc.) for Python is quite impressive. In your opinion, what are the (possibly historical) reasons for such fragmentation and divergence?

The short answer is that software publication, distribution, and integration is a complex problem with plenty of room for multiple solutions tailored for different use cases. The long answer can be found in the Python Packaging User Guide. In my recent talks on this, I have noted that the problem is mainly one of age and the aforementioned tools being born in a somewhat different era of software distribution.

***setuptools* is the *de facto* standard for Python distributions nowadays. Is there anything you think users should be aware of when using it (or not)?**

setuptools is quite reasonable as a build system, especially for pure Python

[2] "Benevolent Dictator For Life," title given to Guido van Rossum, author of Python

projects, or those with only simple C extensions. It also offers a powerful system for plugin registration and good cross-platform script generation.

While effective, the multi-version support in *pkg_resources* is also a bit quirky and tricky to use properly. Unless there's a really compelling reason to have conflicting versions in the same environment, it's much easier to just use *virtualenv* or *zc.buildout*.

PEP 426, which defines a new metadata format for Python packages, is still fairly recent and not yet approved. Is it on good track? What motivated it in the first place, how do you think it'll tackle the current problems?

PEP 426 originally started as part of the *Wheel* format definition, but Daniel Holth eventually realized that *Wheel* could work with the existing metadata format defined by *setuptools*. PEP 426 is thus a consolidation of the existing *setuptools* metadata with some of the ideas from *distutils2* and other packaging systems (like *RPM* and *npm*), and also addresses some of the frustrations encountered with existing tools (like cleanly separating different kinds of dependencies).

If PEP 426 is accepted, what kinds of tools would you to see built to take advantage of what it offers?

The main gains will be a REST API on PyPI offering full metadata access, as well as (hopefully) the ability to automatically generate distribution policy-compliant packages from upstream metadata.

The *Wheel* format is fairly recent and not widely used yet, but it seems promising. Is there any reason it isn't part of the Standard Library, or are there already plans to include it?

It turns out the Standard Library isn't really a suitable place for packaging standards: it evolves too slowly, and an addition to a later version of the

Standard Library can't be used with earlier versions of Python. So, at the Python language summit earlier this year, we tweaked the PEP process to allow *distutils-sig* to manage the full approval cycle for packaging-related PEPs. *python-dev* will only be involved for proposals that involve changing CPython directly (like *pip* bootstrapping).

What kind of future do you envision that would push developers to build and distribute *Wheel* packages?

pip is adopting it at as an alternative to the *Egg* format, allowing local caching of builds for fast virtual environment creation, and PyPI allows uploads of Wheel archives for Windows and Mac OS X. We still have some tweaks to make before it will be suitable for use on Linux.

4.7 Entry points

You may have already used *setuptools* entry points without knowing anything about them. If you haven't yet decided to use *setuptools* (or *pbr*, see Section 4.2) to provide a `setup.py` file with your software, here are a few features that might help you make up your mind.

Software distributed using *setuptools* includes important metadata describing things such as its required dependencies and – more relevantly to this topic – a list of "entry points." These entry points can be used by other Python programs to dynamically discover features that a package provides.

In the following sections, we will discuss how we can use entry points to add extensibility to our software.

4.7.1 Visualising entry points

The easiest way to visualize the entry points available in a package is to use a package called *entry_point_inspector*.

When installed, it provides a command called `epi` that you can run from your terminal to interactively discover the entry points provided by installed packages:

Example 4.4 Result of *epi group list*

```
+--------------------------+
| Name                     |
+--------------------------+
| console_scripts          |
| distutils.commands       |
| distutils.setup_keywords |
| egg_info.writers         |
| epi.commands             |
| flake8.extension         |
| setuptools.file_finders  |
| setuptools.installation  |
+--------------------------+
```

Example 4.4 shows that we have many different packages that provide entry points. You'll also notice this list includes *console_scripts*, which we'll discuss in Section 4.7.2.

Example 4.5 Result of *epi group show console_scripts*

```
+----------+----------+--------+--------------+-------+
| Name     | Module   | Member | Distribution | Error |
+----------+----------+--------+--------------+-------+
| coverage | coverage | main   | coverage 3.4 |       |
+----------+----------+--------+--------------+-------+
```

Example 4.5 shows us that an entry point named *coverage* refers to the member *main* of the module *coverage*. This entry point is provided by the package *coverage 3.4*. We can obtain more information by using *epi ep show*:

Example 4.6 Result of *epi ep show console_scripts coverage*

```
+--------------+-----------------------------------+
| Field        | Value                             |
+--------------+-----------------------------------+
| Module       | coverage                          |
| Member       | main                              |
| Distribution | coverage 3.4                      |
| Path         | /usr/lib/python2.7/dist-packages |
| Error        |                                   |
+--------------+-----------------------------------+
```

The tool we're using here is just a thin layer on top of a more complete Python library which can help us discover entry points for any Python library or program. Entry points are useful for various things, including console scripts and dynamic code discovery, as we're going to see in the next few sections.

4.7.2 Using console scripts

When writing a Python application, you almost always have to provide a launchable program – a Python script that the end user can actually run. This program needs to be installed inside a directory somewhere in the system path.

Most projects will have something along the lines of this:

```
#!/usr/bin/python
import sys
import mysoftware
```

```
mysoftware.SomeClass(sys.argv).run()
```

This is actually a best-case scenario: many projects have a much longer script installed in the system path. But using such scripts has some major issues:

- There's no way they can know where the Python interpreter is or which version it will be.
- They leak binary code that can't be imported by software or unit tests.
- There's no easy way to define where to install them.
- It's not obvious how to install this in a portable way (Unix vs Windows for example).

setuptools has a feature that helps us circumvent these problems: *console_scripts*. *console_scripts* is an entry point that can be used to make *setuptools* install a tiny program in the system path which then calls a specific function in one of your modules.

Let's imagine a *foobar* program that consists of a client and a server. Each part is written in its own module – *foobar.client* and *foobar.server*, respectively:

foobar/client.py

```
def main():
    print("Client started")
```

foobar/server.py

```
def main():
    print("Server started")
```

Of course, our program doesn't really do much of anything – our client and server don't even talk to each other. For the purposes of our example, though, all they need to do is print a message letting us know they've started successfully.

We can now write the following `setup.py` file in the root directory:

setup.py

```
from setuptools import setup


setup(
    name="foobar",
    version="1",
    description="Foo!",
    author="Julien Danjou",
    author_email="julien@danjou.info",
    packages=["foobar"],
    entry_points={
        "console_scripts": [
            "foobard = foobar.server:main",
            "foobar = foobar.client:main",
        ],
     },
)
```

We define our entry points using the format `package.subpackage:function`.

When you run `python setup.py install`, *setuptools* will create a script that will look like this:

Example 4.7 A console script generated by *setuptools*

```
#!/usr/bin/python
# EASY-INSTALL-ENTRY-SCRIPT: 'foobar==1','console_scripts','foobar'
__requires__ = 'foobar==1'
import sys
from pkg_resources import load_entry_point


if __name__ == '__main__':
    sys.exit(
        load_entry_point('foobar==1', 'console_scripts', 'foobar')()
    )
```

This code scans the entry points of the `foobar` package and retrieves the `foobar` key from the `console_scripts` category, which is used to locate and run the corresponding function.

Using this technique will ensure that your code stays in your Python package and can be imported (and tested) by other programs.

Tip

If you're using *pbr* on top of *setuptools*, the generated script is simpler (and therefore faster) than the default one built by *setuptools* as it will call the function you wrote in the entry point without having to search the entry point list dynamically at runtime.

4.7.3 Using plugins and drivers

Entry points make it easy to discover and dynamically load code deployed by other packages. You can use **pkg_resources** to discover and load entry point files from within your Python programs. (You might notice that this is the same package used in the console script that *setuptools* creates, as seen in Example 4.7.)

In this section, we're going to create a *cron*-style daemon that will allow any Python program to register a command to be run once every few seconds by registering an entry point in the group `pytimed`. The attribute this entry point points to should be an object that returns `number_of_seconds, callable`.

Here's our implementation of *pycrond* using `pkg_resources` to discover entry points:

pytimed.py

```
import pkg_resources
```

```python
import time


def main():
    seconds_passed = 0
    while True:
        for entry_point in pkg_resources.iter_entry_points('pytimed'):
            try:
                seconds, callable = entry_point.load()()
            except:
                # Ignore failure
                pass
            else:
                if seconds_passed % seconds == 0:
                    callable()
        time.sleep(1)
        seconds_passed += 1
```

This is a very simple and naive implementation, but it's sufficient for our example. Now we can write another Python program that needs one of its functions called on a periodic basis:

hello.py

```python
def print_hello():
    print("Hello, world!")


def say_hello():
    return 2, print_hello
```

We register the function using the appropriate entry points:

setup.py

```python
from setuptools import setup
```

```
setup(
    name="hello",
    version="1",
    packages=["hello"],
    entry_points={
        "pytimed": [
            "hello = hello:say_hello",
        ],
     },)
```

And now if we run our *pytimed* script, we'll see "Hello, world!" printed on the screen every 2 seconds:

Example 4.8 Running pytimed

```
% python3
Python 3.3.2+ (default, Aug  4 2013, 15:50:24)
[GCC 4.8.1] on linux
Type "help", "copyright", "credits" or "license" for more ↩
   information.
>>> import pytimed
>>> pytimed.main()
Hello, world!
Hello, world!
Hello, world!
```

The possibilities this mechanism offers are huge: it allows you to build driver systems, hook systems, and extensions in an easy and generic way. Implementing this mechanism by hand in every program you make would be tedious, but fortunately, there's a Python library that can take care of the boring parts for us.

stevedore provides support for dynamic plugins based on the exact same mechanism demonstrated in our previous examples. Our use case in this example isn't very complicated, but we can still simplify it a bit using **stevedore**:

pytimed_stevedore.py

```python
from stevedore.extension import ExtensionManager
import time


def main():
    seconds_passed = 0
    while True:
        for extension in ExtensionManager('pytimed', invoke_on_load=True):
            try:
                seconds, callable = extension.obj
            except:
                # Ignore failure
                pass
            else:
                if seconds_passed % seconds == 0:
                    callable()
        time.sleep(1)
        seconds_passed += 1
```

Our example is still very simple, but if you look through the *stevedore* documentation, you'll see that *ExtensionManager* has a variety of subclasses that can handle different situations, such as loading specific extensions based on their names or the result of a function.