Setuptools Integration

When writing command line utilities, it's recommended to write them as modules that are distributed with setuptools instead of using Unix shebangs.

Why would you want to do that? There are a bunch of reasons:

1. One of the problems with the traditional approach is that the first module the Python interpreter loads has an incorrect name. This might sound like a small issue but it has quite significant implications.

   The first module is not called by its actual name, but the interpreter renames it to `__main__`. While that is a perfectly valid name it means that if another piece of code wants to import from that module it will trigger the import a second time under its real name and all of a sudden your code is imported twice.

2. Not on all platforms are things that easy to execute. On Linux and OS X you can add a comment to the beginning of the file (`#!/usr/bin/env python`) and your script works like an executable (assuming it has the executable bit set). This however does not work on Windows. While on Windows you can associate interpreters with file extensions (like having everything ending in `.py` execute through the Python interpreter) you will then run into issues if you want to use the script in a virtualenv.

   In fact running a script in a virtualenv is an issue with OS X and Linux as well. With the traditional approach you need to have the whole virtualenv activated so that the correct Python interpreter is used. Not very user friendly.

3. The main trick only works if the script is a Python module. If your application grows too large and you want to start using a package you will run into issues.

Introduction

To bundle your script with setuptools, all you need is the script in a Python package and a `setup.py` file.

Imagine this directory structure:

```
yourscript.py
setup.py
```

Contents of `yourscript.py`:

```
import click

@click.command()
def cli():
```

```
    """Example script."""
    click.echo('Hello World!')
```

Contents of setup.py:

```
from setuptools import setup

setup(
    name='yourscript',
    version='0.1.0',
    py_modules=['yourscript'],
    install_requires=[
        'Click',
    ],
    entry_points={
        'console_scripts': [
            'yourscript = yourscript:cli',
        ],
    },
)
```

The magic is in the `entry_points` parameter. Below `console_scripts`, each line identifies one console script. The first part before the equals sign (`=`) is the name of the script that should be generated, the second part is the import path followed by a colon (`:`) with the Click command.

That's it.

Testing The Script

To test the script, you can make a new virtualenv and then install your package:

```
$ python3 -m venv .venv
$ . .venv/bin/activate
$ pip install --editable .
```

Afterwards, your command should be available:

```
$ yourscript
Hello World!
```

Scripts in Packages

v: 8.1.x

If your script is growing and you want to switch over to your script being contained in a Python package the changes necessary are minimal. Let's assume your directory structure

changed to this:

```
project/
    yourpackage/
        __init__.py
        main.py
        utils.py
        scripts/
            __init__.py
            yourscript.py
    setup.py
```

In this case instead of using `py_modules` in your `setup.py` file you can use `packages` and the automatic package finding support of setuptools. In addition to that it's also recommended to include other package data.

These would be the modified contents of `setup.py`:

```
from setuptools import setup, find_packages

setup(
    name='yourpackage',
    version='0.1.0',
    packages=find_packages(),
    include_package_data=True,
    install_requires=[
        'Click',
    ],
    entry_points={
        'console_scripts': [
            'yourscript = yourpackage.scripts.yourscript:cli',
        ],
    },
)
```