Installing Packages

Before we look at how we can go about making our own packages installable, let's cover installing a package from someone else. The primary place we'll be installing packages from will be from the ["Python Package Index"](https://pypi.org/) or ["PyPi"](https://pypi.org/) for short.

To install packages, we'll use [pip](https://pip.pypa.io/en/stable/quickstart/). Let's install one of the most popular Python packages, the [requests](https://pypi.org/project/requests/) package.

$ pip3.7 install requests

Collecting requests

Downloading https://files.pythonhosted.org/packages/51/bd/23c926cd341ea6b7dd0b2a00aba99ae0f828be89d72b2190f27c11d4b7fb/requests-2.22.0-py2.py3-none-any.whl (57kB)

|????????????????????????????????| 61kB 2.4MB/s

Collecting certifi>=2017.4.17 (from requests)

Downloading https://files.pythonhosted.org/packages/b9/63/df50cac98ea0d5b006c55a399c3bf1db9da7b5a24de7890bc9cfd5dd9e99/certifi-2019.11.28-py2.py3-none-any.whl (156kB)

|????????????????????????????????| 163kB 8.0MB/s

Collecting idna<2.9,>=2.5 (from requests)

Downloading https://files.pythonhosted.org/packages/14/2c/cd551d81dbe15200be1cf41cd03869a46fe7226e7450af7a6545bfc474c9/idna-2.8-py2.py3-none-any.whl (58kB)

|????????????????????????????????| 61kB 10.8MB/s

Collecting urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 (from requests)

Downloading https://files.pythonhosted.org/packages/e8/74/6e4f91745020f967d09332bb2b8b9b10090957334692eb88ea4afe91b77f/urllib3-1.25.8-py2.py3-none-any.whl (125kB)

|????????????????????????????????| 133kB 10.8MB/s

Collecting chardet<3.1.0,>=3.0.2 (from requests)

Downloading https://files.pythonhosted.org/packages/bc/a9/01ffebfb562e4274b6487b4bb1ddec7ca55ec7510b22e4c51f14098443b8/chardet-3.0.4-py2.py3-none-any.whl (133kB)

|????????????????????????????????| 143kB 12.8MB/s

Installing collected packages: certifi, idna, urllib3, chardet, requests

Successfully installed certifi-2019.11.28 chardet-3.0.4 idna-2.8 requests-2.22.0 urllib3-1.25.8

$

The requests package has some dependencies on other packages so pip will go ahead and download those dependencies. For the purposes of the PCAP exam, we just need to know how to install packages, but it is definitely worth viewing the other commands provided by pip by running pip --help.

Making a Package Installable

To make a package installable, it needs to have a file in the root of the package called setup.py. The structure of installable packages can vary, but the presence of a setup.py is constant. Let's make our helpers package installable by adding a setup.py and configuring it using the [setup function](https://packaging.python.org/guides/distributing-packages-using-setuptools/#setup-args). The "Python Packaging Authority" is the working group that maintains the core projects use for Python packaging, and they provide an [example project](https://github.com/pypa/sampleproject). We're going to take the setup.py from that project as a starting point and modify it for our purposes. To begin, we do need to change our helpers directory to be the container for our installable package (different than a "python package"). Let's move things around before creating our setup.py.

$ cd ~/using\_modules

$ mkdir -p helpers/src/helpers

$ mv helpers/\*.py helpers/src/helpers/

Using tree on our directory structure for helpers will provide us a better way to view our directories. Note that you may have to install tree using sudo yum install tree.

$ tree helpers

helpers/

|---> src

|---> helpers

|---> \_\_init\_\_.py

|---> strings.py

|---> variables.py

2 directories, 3 files

The outer helpers directory is there just to hold onto our code and isn't actually a Python package. The inner helpers will provide the package that can be imported after the distribution of this code is installed. For our code to be installable, we still need a setup.py file, which will go in the outer helpers directory. Feel free to download it directly using the curl command or copy and paste the contents below.

$ cd helpers/

$ curl -O https://raw.githubusercontent.com/pypa/sampleproject/master/setup.py

Here's what it will look like:

~/using\_modules/helpers/setup.py

from setuptools import setup, find\_packages

from os import path

here = path.abspath(path.dirname(\_\_file\_\_))

# Get the long description from the README file

with open(path.join(here, 'README.md'), encoding='utf-8') as f:

long\_description = f.read()

setup(

name='helpers', # Required

version='1.0.0', # Required

description='Our custom collection of helper functions and variables.', # Optional

# long\_description=long\_description, # Optional

# long\_description\_content\_type='text/markdown', # Optional (the README is markdown so we want to set this)

# url='https://github.com/pypa/sampleproject', # Optional

author='Keith Thompson', # Optional

author\_email='keith@linuxacademy.com', # Optional

# Classifiers help users find your project by categorizing it.

#

# For a list of valid classifiers, see https://pypi.org/classifiers/

classifiers=[ # Optional

# How mature is this project? Common values are

# 3 - Alpha

# 4 - Beta

# 5 - Production/Stable

'Development Status :: 3 - Alpha',

# Indicate who your project is intended for

'Intended Audience :: Developers',

'Topic :: Software Development :: Build Tools',

# Pick your license as you wish

'License :: OSI Approved :: MIT License',

# Specify the Python versions you support here. In particular, ensure

# that you indicate whether you support Python 2, Python 3 or both.

# These classifiers are \*not\* checked by 'pip install'. See instead

# 'python\_requires' below.

'Programming Language :: Python :: 3',

'Programming Language :: Python :: 3.5',

'Programming Language :: Python :: 3.6',

'Programming Language :: Python :: 3.7',

'Programming Language :: Python :: 3.8',

],

keywords='helpers', # Optional

# When your source code is in a subdirectory under the project root, e.g.

# `src/`, it is necessary to specify the `package\_dir` argument.

package\_dir={'': 'src'}, # Optional

# You can just specify package directories manually here if your project is

# simple. Or you can use find\_packages().

#

# Alternatively, if you just want to distribute a single Python file, use

# the `py\_modules` argument instead as follows, which will expect a file

# called `my\_module.py` to exist:

#

# py\_modules=["my\_module"],

#

packages=find\_packages(where='src'), # Required

# Specify which Python versions you support. In contrast to the

# 'Programming Language' classifiers above, 'pip install' will check this

# and refuse to install the project if the version does not match. If you

# do not support Python 2, you can simplify this to '>=3.5' or similar, see

# https://packaging.python.org/guides/distributing-packages-using-setuptools/#python-requires

python\_requires='!=3.0.\*, !=3.1.\*, !=3.2.\*, !=3.3.\*, !=3.4.\*, <4',

# This field lists other packages that your project depends on to run.

# Any package you put here will be installed by pip when your project is

# installed, so they must be valid existing projects.

#

# For an analysis of "install\_requires" vs pip's requirements files see:

# https://packaging.python.org/en/latest/requirements.html

# install\_requires=['peppercorn'], # Optional

# List additional groups of dependencies here (e.g. development

# dependencies). Users will be able to install these using the "extras"

# syntax, for example:

#

# $ pip install sampleproject[dev]

#

# Similar to `install\_requires` above, these must be valid existing

# projects.

# extras\_require={ # Optional

# 'dev': ['check-manifest'],

# 'test': ['coverage'],

# },

# If there are data files included in your packages that need to be

# installed, specify them here.

#

# If using Python 2.6 or earlier, then these have to be included in

# MANIFEST.in as well.

# package\_data={ # Optional

# 'sample': ['package\_data.dat'],

# },

# Although 'package\_data' is the preferred approach, in some case you may

# need to place data files outside of your packages. See:

# http://docs.python.org/3.4/distutils/setupscript.html#installing-additional-files

#

# In this case, 'data\_file' will be installed into '<sys.prefix>/my\_data'

# data\_files=[('my\_data', ['data/data\_file'])], # Optional

# To provide executable scripts, use entry points in preference to the

# "scripts" keyword. Entry points provide cross-platform support and allow

# `pip` to create the appropriate form of executable for the target

# platform.

#

# For example, the following would provide a command called `sample` which

# executes the function `main` from this package when invoked:

# entry\_points={ # Optional

# 'console\_scripts': [

# 'sample=sample:main',

# ],

# },

# List additional URLs that are relevant to your project as a dict.

#

# This field corresponds to the "Project-URL" metadata fields:

# https://packaging.python.org/specifications/core-metadata/#project-url-multiple-use

#

# Examples listed include a pattern for specifying where the package tracks

# issues, where the source is hosted, where to say thanks to the package

# maintainers, and where to support the project financially. The key is

# what's used to render the link text on PyPI.

# project\_urls={ # Optional

# 'Bug Reports': 'https://github.com/pypa/sampleproject/issues',

# 'Funding': 'https://donate.pypi.org',

# 'Say Thanks!': 'http://saythanks.io/to/example',

# 'Source': 'https://github.com/pypa/sampleproject/',

# },

)

We left a lot of comments in there because they are good to read and understand, but they're for optional fields. Some of the important and potentially confusing lines to look at are the package\_dir and packages arguments. We've put our code into the src directory. We've set these two arguments and used the find\_packages function from setuptools to automatically find the packages that we're providing when someone installs this.

Building a Distribution

Making code installable in Python means that we need to create a distribution. There are two primary types of distributions: eggs and wheels. Wheels are the modern way to create a distribution and they're a single file that can be installed by pip. They will install any dependencies and place or unpack the source code into the site-packages directory for our Python installation. For us to build a wheel distribution, we need to install the wheel package and run a command using Python and our setup.py file. Let's install wheel first.

$ pip3.7 install --upgrade wheel

...

Setuptools provides us with multiple different subcommands if we process our setup.py through the Python interpreter. Let's take a look at those commands.

$ python3.7 setup.py --help

Traceback (most recent call last):

File "setup.py", line 7, in <module>

with open(path.join(here, 'README.md'), encoding='utf-8') as f:

FileNotFoundError: [Errno 2] No such file or directory: '/home/cloud\_user/using\_modules/helpers/README.md'

Our setup.py specifies that we'll provide documentation in a README.md file, but that file doesn't exist, so we can't read it. We'll cover file IO later in the course, but for now, we just need to make sure that that file exists.

$ touch README.md

Now, let's try again.

$ python3.7 setup.py --help

Common commands: (see '--help-commands' for more)

setup.py build will build the package underneath 'build/'

setup.py install will install the package

Global options:

--verbose (-v) run verbosely (default)

--quiet (-q) run quietly (turns verbosity off)

--dry-run (-n) don't actually do anything

--help (-h) show detailed help message

--no-user-cfg ignore pydistutils.cfg in your home directory

--command-packages list of packages that provide distutils commands

Information display options (just display information, ignore any commands)

--help-commands list all available commands

--name print package name

--version (-V) print package version

--fullname print <package name>-<version>

--author print the author's name

--author-email print the author's email address

--maintainer print the maintainer's name

--maintainer-email print the maintainer's email address

--contact print the maintainer's name if known, else the author's

--contact-email print the maintainer's email address if known, else the

author's

--url print the URL for this package

--license print the license of the package

--licence alias for --license

--description print the package description

--long-description print the long package description

--platforms print the list of platforms

--classifiers print the list of classifiers

--keywords print the list of keywords

--provides print the list of packages/modules provided

--requires print the list of packages/modules required

--obsoletes print the list of packages/modules made obsolete

usage: setup.py [global\_opts] cmd1 [cmd1\_opts] [cmd2 [cmd2\_opts] ...]

or: setup.py --help [cmd1 cmd2 ...]

or: setup.py --help-commands

or: setup.py cmd --help

This gives us a lot of output, but only the common commands are provided to us. Reading the first line of the output, we can see that the rest of the commands can be shown by using --help-commands instead of --help. Let's do that.

$ python3.7 setup.py --help-commands

Standard commands:

build build everything needed to install

build\_py "build" pure Python modules (copy to build directory)

build\_ext build C/C++ extensions (compile/link to build directory)

build\_clib build C/C++ libraries used by Python extensions

build\_scripts "build" scripts (copy and fixup #! line)

clean clean up temporary files from 'build' command

install install everything from build directory

install\_lib install all Python modules (extensions and pure Python)

install\_headers install C/C++ header files

install\_scripts install scripts (Python or otherwise)

install\_data install data files

sdist create a source distribution (tarball, zip file, etc.)

register register the distribution with the Python package index

bdist create a built (binary) distribution

bdist\_dumb create a "dumb" built distribution

bdist\_rpm create an RPM distribution

bdist\_wininst create an executable installer for MS Windows

check perform some checks on the package

upload upload binary package to PyPI

Extra commands:

bdist\_wheel create a wheel distribution

alias define a shortcut to invoke one or more commands

bdist\_egg create an "egg" distribution

develop install package in 'development mode'

dist\_info create a .dist-info directory

easy\_install Find/get/install Python packages

egg\_info create a distribution's .egg-info directory

install\_egg\_info Install an .egg-info directory for the package

rotate delete older distributions, keeping N newest files

saveopts save supplied options to setup.cfg or other config file

setopt set an option in setup.cfg or another config file

test run unit tests after in-place build

upload\_docs Upload documentation to PyPI

usage: setup.py [global\_opts] cmd1 [cmd1\_opts] [cmd2 [cmd2\_opts] ...]

or: setup.py --help [cmd1 cmd2 ...]

or: setup.py --help-commands

or: setup.py cmd --help

There are plenty of commands in here to play with, but the one that we care about is the extra command bdist\_wheel. This will build a wheel distribution that will work perfectly with pip. Let's run that now.

$ python3.7 setup.py bdist\_wheel

running bdist\_wheel

running build

running build\_py

creating build

creating build/lib

creating build/lib/helpers

copying src/helpers/\_\_init\_\_.py -> build/lib/helpers

copying src/helpers/strings.py -> build/lib/helpers

copying src/helpers/variables.py -> build/lib/helpers

installing to build/bdist.linux-x86\_64/wheel

running install

running install\_lib

creating build/bdist.linux-x86\_64

creating build/bdist.linux-x86\_64/wheel

creating build/bdist.linux-x86\_64/wheel/helpers

copying build/lib/helpers/\_\_init\_\_.py -> build/bdist.linux-x86\_64/wheel/helpers

copying build/lib/helpers/strings.py -> build/bdist.linux-x86\_64/wheel/helpers

copying build/lib/helpers/variables.py -> build/bdist.linux-x86\_64/wheel/helpers

running install\_egg\_info

running egg\_info

writing src/helpers.egg-info/PKG-INFO

writing dependency\_links to src/helpers.egg-info/dependency\_links.txt

writing top-level names to src/helpers.egg-info/top\_level.txt

reading manifest file 'src/helpers.egg-info/SOURCES.txt'

writing manifest file 'src/helpers.egg-info/SOURCES.txt'

Copying src/helpers.egg-info to build/bdist.linux-x86\_64/wheel/helpers-1.0.0-py3.7.egg-info

running install\_scripts

creating build/bdist.linux-x86\_64/wheel/helpers-1.0.0.dist-info/WHEEL

creating 'dist/helpers-1.0.0-py3-none-any.whl' and adding 'build/bdist.linux-x86\_64/wheel' to it

adding 'helpers/\_\_init\_\_.py'

adding 'helpers/strings.py'

adding 'helpers/variables.py'

adding 'helpers-1.0.0.dist-info/METADATA'

adding 'helpers-1.0.0.dist-info/WHEEL'

adding 'helpers-1.0.0.dist-info/top\_level.txt'

adding 'helpers-1.0.0.dist-info/RECORD'

removing build/bdist.linux-x86\_64/wheel

We now have a build and dist directory inside of the upper helpers directory. The artifact that we created will be within the dist directory and end with a .whl extension.

Going back to ~/using\_modules, we'll actually run into issues if we try to run main.py right now because there is no helpers package local to the file anymore. Here's what we'll see when we run that script:

$ cd ~/using\_modules

$ python3.7 main.py

Traceback (most recent call last):

File "main.py", line 1, in <module>

from helpers.strings import extract\_lower

ModuleNotFoundError: No module named 'helpers.strings'

To get around this, we'll install our package using pip and the wheel we built.

$ pip3.7 install helpers/dist/helpers-1.0.0-py3-none-any.whl

Processing ./helpers/dist/helpers-1.0.0-py3-none-any.whl

Installing collected packages: helpers

Successfully installed helpers-1.0.0

When we run a script or load the REPL, we can load the helpers package and its internal modules.

$ python3.7 main.py

Lowercase letters (from strings): ['e', 'i', 't', 'h', 'h', 'o', 'm', 'p', 's', 'o', 'n']

Uppercase letters (from package): ['K', 'T']

Off of helpers: ['e', 'i', 't', 'h', 'h', 'o', 'm', 'p', 's', 'o', 'n']

Our package is installed and our script runs again without using a module local to the script. We're not going to cover publishing a package to PyPi in this course, but the PyPA documentation also details how to do that.