Creating and Distributing Packages

The Python Package Index (PyPI, `https://pypi.org`) is the main repository for Python packages. PyPI hosts important packages, such as Numpy, Scipy, Tensorflow, Pandas and many others. PyPI hosts large and well developed packages, but it can also host packages developed by anyone, including you. These packages can then be installed in any system with `pip`.

In this lecture we will learn how to create a package that can be distributed by PyPI and installed by other Python users.

Clarifying Concepts: Package, Module, Script

A Python `script` is a text file ending in `.py` containing Python code that can be interpreted by the Python interpreter. We can run a script, say `script.py`, by calling:

```
python3 script.py
```

The contents of the file are then interpreted and executed. We use scripts to write short programs. However, when our programs start getting longer, keeping everything in a single file is just not recommended. The idea is to separate pieces of code that can be reused by our main script into another file (or files).

A `module` is a script file used to organize Python code and which can be reused by other scripts. Modules are written with the intent of being imported by other scripts or modules, and ideally contain code that can be reused. For example, you decide your original `script.py` file is too big and move some of its code to another file, `my_module.py`. This file can now be imported inside `script.py` with a statement like `import my_module`, which makes whatever was defined in `my_module.py` available for use.

Lastly, a `package` is a collection of modules, created via a folder hierarchy. For example, say you have three files `stocks.py`, `futures.py` and `options.py`. The common theme here is finance. So we could package these modules into a finance package. To do so, we create a folder `finance/` and move all of the module files inside that folder. We can then import these modules via any script via:

```
from finance import stocks, futures, options
```

Or:

```
import finance.stocks as Stocks
```

PyPI hosts packages. Each project hosted in PyPI is a package, which organizes one or more modules. Therefore, each project in PyPI is just a collection of python scripts, organized under a single name (namespace).

Before we move on to how to host a package on the PyPI repositories and make it available to other Python users, we need to discuss two things: the use of `__name__` in scripts and modules, and where your local Python searches for modules and packages.

Module and Scripts: The Global Variable `__name__`

Whenever Python executes a script, it creates some global variables with various information and then runs the code in the script. You can check what these variables are with the built-in function `globals`:

```
print(globals())
```

```
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '
```

One of these global variables is `__name__`. This variable is used to indicate whether a script file is being run as the main module or is being imported. Therefore, `__name__` will contain the string `'__main__'` if the file is being run as a script, as in `python my_script.py`. But, if the file is being imported as a module, as in `import my_script`, then `__name__` will contain the name of the module as a string.

For example, create the file:

```
print(__name__)
```

Save it and run it with `python my_script.py`. You will get back the string `'__main__'`.

Now, open a Python REPL and import `my_script.py`. To import the file as a module, Python will execute the contents of `my_script.py` in a separate environment where `__name__` will be `'my_script'`.

The most common use of the global variable `__name__` is to create a module that can be imported by other scripts, but that can also be used as a standalone script. To do so, we check whether `__name__` corresponds to `'__main__'` or to the module's name. For example:

```
# My module
def fibo(n: int) -> int:
    """Computes the nth number of the Fibonacci series."""
    phi = (1 + 5 ** 0.5) / 2
    return int((phi**n-((-phi)**(-n)))/(5**0.5))


if __name__ == '__main__':
    import sys
    print(fibo(int(sys.argv[1])))
```

If the file above is imported, `import my_module`, then Python will only execute the definition of the function `fibo`, since the `if` part will fail, and then we could access the function via `my_module.fibo`. However, if we run the file as a script, then Python will execute the definition of the function, but it will also evaluate the function with whatever arguments were given.

In summary, the use of `if __name__ ...` allows you to have a module that can be imported by other code, but that can also be run independently. The separation of those parts is done by checking whether the code is being imported as a module or run as a script.

Search Path

When importing a module (or a package) with `import module_name`, Python needs to search for files containing the code. Python first searches for a built-in module with

module name. If the module is not a built-in, then Python starts looking for the module in the folders contained in `sys.path`. The first item in the list `sys.path` will be the folder that contains the script Python is running, or the current directory if Python was launched interactively (REPL). The list will also contain a folder with a name similar to `lib/python3.6/site-packages`. This folder, `site-packages`, is where `pip` installs the packages.

This means, when creating a package, you should be careful about naming, so that it doesn't clash with an existing built-in package. If your package name is the same as some other package on PyPI then you will be asked to change its name.

Creating a package containing a single module

Let's first create a package that contains only a single module. To do so, we will create the following folders and files:

```
package_example/
|-- setup.py
|-- README.md
|-- LICENSE.txt
|-- project_name/
    |-- __init__.py
    |-- sequences.py
```

The file `setup.py` should contain the configurations necessary for PyPI to correctly classify your project and distribute it. The file `README.md` should contain the documentation of your project, with information as what APIs are available and how to use them. The file `LICENSE.txt` contains the license under which your project is distributed (examples). The following example contains most of the setup you will usually need:

```
from setuptools import setup, find_packages

# Load complete description of project from a markdown file
with open("README.md", "r") as fh:
    long_description = fh.read()

# Call setup to define project data
setup(
    # Name of the package: must be unique, later used with pip install
    name='project_name',
    # Project version number: MAJOR.MINOR[.MICRO]
    version='0.0.1',
    # One line description of project
    description='My project description',
    # Project complete documentation (loaded from README.md)
    long_description=long_description,
    long_description_content_type="text/markdown",
    # Project home repository
    url="http://github.com/username/project_name",
    # Author information
    author="First Name Last Name",
    author_email="email@email.com",
```

```
    # License information
    license="MIT",
    # Name the modules in your package
    packages=find_packages(),
    # Additional Metadata
    classifiers=[
        "Programming Language :: Python :: 3 :: Only",
        "Programming Language :: Python :: 3.6",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
    # Dependencies
    install_requires=[
        "numpy >= 1.15.0",
        "pandas >= 0.23.0",
        "matplotlib >= 3.0.0",
    ]
)
```

The package `setuptools` is a package that helps build and distribute other Python packages. It can be installed with `pip install -U setuptools`. We use two of its functions, `setup` and `find_packages`. The function `setup` is used to define the name and other important information about our project. The function `find_packages` is used to automatically find what modules were defined in our package, and it is useful for big projects with numerous files, but can be used for small projects as well.

The project version is a string following the format `MAJOR.MINOR[.MICRO]`. The `MAJOR` number designates a major change in the package, usually adding many new features, which may change the APIs and break backwards-compatibility. As an user of a package, we do not want to automatically upgrade this package when a major version is released, since it may break our code. The `MINOR` number is changed when there are changes that do not break the APIs or when there are bug fixes. The optional `MICRO` number can also be used to indicate bug fixes and very small changes.

The `classifiers` keyword takes a list of strings containing some pre-specified descriptors for projects to help sorting on PyPI's website. Follow this link for a list of all possible classifiers, and this link for PEP 301 which provides some more details on classifiers.

Now we can actually define the modules in our package. To do so, create a folder with the same name as the project, in this case the folder is called `project_name`. Inside this folder, create a script named `__init__.py`. The `__init__.py` script tells Python to treat the directory as containing packages. It is used to organize the `import` of multiple packages and for small projects is often left empty (empty file). For now, leave it empty.

Lastly, add the module files that contain the code you want to share in the `project_name` folder. In this case, we have the module `sequences.py`. Let's add some code to the module:

```
# sequences.py
def fibo(n: int) -> int:
    """Computes the nth number of the Fibonacci series."""
    phi = (1 + 5 ** 0.5) / 2
    return int((phi**n-((-phi)**(-n)))/(5**0.5))
```

Compilation

Before we can use the module, we need to compile it; and that is where `setup.py` is used. Install `setuptools` and `wheel` using `pip`:

```
python3 -m pip install --user --upgrade setuptools wheel
```

Then run:

```
# change the working directory to the package_example folder
cd ~/Desktop/package_example
# run setup to create install/distribution files
python3 setup.py sdist bdist_wheel
```

This will create a source distribution (`sdist`) and a wheel distribution (`bdist_wheel`). The source distribution has all of the files in the folder above, and is similar to zipping your entire project folder. Since you have the entire project, you can build it on any system.

The wheel distribution is also similar to zipping the entire project folder, but includes the compiled binaries from your code. These binaries are specific to your system and to a version of Python. The advantage of a wheel distribution is that it can be installed by simply unzipping its contents in the right location. This makes it fast to install packages on similar systems, since you skip the compilation step.

After running the command above, you will notice that a folder called `dist` was created. In that folder you will find the archives for your package. Notice the name of the archives depend on the version specified in `setup.py`.

To test if our package works, we will install it locally (without sending it to PyPI). Local installation is crucial when developing the package and ironing out bugs, since you do not want to re-upload to PyPI every time you fix some small issue.

Using pip to install the package locally

To install a package locally, we call `pip install` with the `-e` option and give it the path to the project we want installed. For example:

```
pip install -e package_example
```

This will install the project in the usual place where Python installs projects, but instead of fetching the project from PyPI it uses the local source.

You can now import the package in Python using the name defined in the `setup.py` file. Remember the package has only one module named `sequences.py`.

```
from project_name import sequences
print(sequences.fibo(10))
```

Using `__init__.py`

To use the `sequences.py` module, we first had to import it from the package. We can use the `__init__.py` file to make the importing more straightforward. The `__init__.py` file is run by Python when importing a package, and it allows us to make the package itself behave as a module. Modify the `__init__.py` file to have the following:

```
# __init__.py
from .sequences import fibo
```

The `.` is to tell Python to find the `sequences` module in the local directory where `__init__.py` is. Now, when we `import project_name`, the `__init__.py` file is going to be run and its local variables are going to be available under the `project_name` name space.

Since we updated the package, we need to update the version in `setup.py` and then reinstall it. Update the version number to `0.0.2` and re-run the following:

```
cd ~/Desktop/package_example
python3 setup.py sdist bdist_wheel
cd ~/Desktop
pip install -e package_example
```

Now, we can import the package and the `fibo` function will be directly available:

```
import project_name
print(project_name.fibo(10))
```

The usefulness of `__init__.py` is to make some parts of the modules directly available under the package name, while keeping the rest of the modules separate.

Creating a package containing multiple modules

To add more modules to the package, simply add other `.py` files containing code to the `project_name` folder. Then update the version number in `setup.py` and reinstall.

Distributing your package with PyPI

To distribute the package we will use a package called `twine`. This package is a utility to assist with publishing packages on PyPI. Install it with `pip`:

```
pip install twine
```

We will first test uploading our package to a test version of PyPI so that we do not pollute the real PyPI with test projects. Start by registering an account on Test PyPI. Then we can just upload what we have already compiled with `twine`. To do so:

```
# Change working directory to project folder
cd ~/Desktop/package_example
# Upload the archive files to TestPyPI
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

Notice that `twine` is available as a standalone program, even though we installed it with `pip`. As an aside, this can be done with any Python program, it is just an extra step when distributing software, but one step we are not interested in right now (more details here).

The command above will upload the contents from the `package_example/dist` directory to the Test PyPI repository. Remember that the contents of the `dist` folder are archives of the source code. The `*` in `dist/*` is a special notation for `bash` which expands to all files in the `dist` folder.

When you run the command for the first time you may get an error about registering your email. Just follow the instructions in the error message and then try again.

Notice that we have created two versions of our project, namely versions `0.0.1` and `0.0.2`. When we compile the project for both versions, we created two sets of archives for each version. The command above will upload the archives for all versions to TestPyPI. It is important that all versions are available, so that if you need a specific version of the package you can get it from the repository.

Now, you can log in Test PyPI and you should see your project there. To install your project from Test PyPI run the following:

```
# Uninstall the version installed from the local files
pip uninstall project_name
# Install the version from Test PyPI
pip install -i https://test.pypi.org/simple/ project-name
```

Since we are using Test PyPI we need to use the option `-i https://test.pypi.org/simple/` to let `pip` know where to find the project. By default, `pip` would search on the official PyPI repository.

To distribute the package on the official PyPI repository we follow similar steps. Begin by registering an account on PyPI. Then upload with `twine`:

```
cd ~/Desktop/package_example
twine upload dist/*
```

If there are naming clashes or other issues you will get an error message. Just follow the steps to fix the issue. You can now install your package with `pip`:

```
pip install package_name
```

Summary

By the end of this lecture notes you should know:

- Difference between: script, module and package
- How to use `__name__` in a module
- The places and order where Python searches for packages
- How to create a package
- How to compile a package
- How to install a package from a local source archive
- How to use `__init__` to facilitate importing
- How to distribute your package with PyPI

If you find any typos or issues with these notes, please send me an email with your feedback.

References

- Packaging Python Projects Tutorial
- Building and Distributing Packages with setuptools
- Installing Packages
- What `__init__.py` is for?
- How to create a Python Package with `__init__.py`
- Using TestPyPI