Installing your own package

setup.py

- Is used to install the package
- Contains metadata on the package

Package directory structure

Directory tree for package with subpackages

```
mysklearn/      <-- outer directory
|-- mysklearn   <--- inner source code directory
|   |-- __init__.py
|   |-- preprocessing
|   |   |-- __init__.py
|   |   |-- normalize.py
|   |   |-- standardize.py
|   |-- regression
|   |   |-- __init__.py
|   |   |-- regression.py
|   |-- utils.py
|-- setup.py   <-- setup script in outer
```

3. Why should you install your own package?
However, if you move the script and the package apart, you will no longer be able to import your package. This is because the script can search for packages in its parent directory, but it won't search outside that. But if you install it, you can import the package no matter where the script is located. Just like you can always import NumPy, or any other package, when it is installed.

Inside setup.py

```
# Import required functions
from setuptools import setup

# Call setup function
setup(
    author="James Fulton",
    description="A complete package for linear regression.",
    name="mysklearn",
    version="0.1.0",
)
```

4. setup.py
To make the package installable, you are going to add a new file to the package - the setup-dot-py script. This script also contains metadata for the package, and will be important later on if you want to publish your package.

version number = (major number) . (minor number) . (patch number)

Inside setup.py

```
# Import required functions
from setuptools import setup, find_packages

# Call setup function
setup(
    author="James Fulton",
    description="A complete package for linear regression.",
    name="mysklearn",
    version="0.1.0",
    packages=find_packages(include=["mysklearn", "mysklearn.*"]),
)
```

5. Package directory structure
To add the setup script, we need to restructure our directory slightly. The setup script is part of the package, but not part of the source code.

Editable installation

```
pip install -e .
```

- `.` = package in current directory
- `-e` = editable

Directory tree for package with subpackages

```
mysklearn/  <-- navigate to here
|-- mysklearn
|   |-- __init__.py
|   |-- preprocessing
|   |   |-- __init__.py
|   |   |-- normalize.py
|   |   |-- standardize.py
|   |-- regression
|   |   |-- __init__.py
|   |   |-- regression.py
|   |-- utils.py
|-- setup.py
```

6. Package directory structure
Therefore we create a new folder inside the package directory to keep the source code. It's very common to name the inner and outer folders the same thing. Here the outer package directory is called my-sklearn, and inside this directory there is the directory of source code, also called my-sklearn.

7. Package directory structure
The outer directory also contains the setup script. Doing this keeps the source code separate from extra files the package needs. We'll be adding more of these files later in this chapter.

Dealing with dependencies

What are dependencies?

- Other packages you import inside your package
- Inside `mymodule.py`:

```
# These imported packages are dependencies
import numpy as np
import pandas as pd
...
```

Adding dependencies to setup.py

```
from setuptools import setup, find_packages

setup(
    ...
    install_requires=['pandas', 'scipy', 'matplotlib'],
)
```

Controlling dependency version

```
from setuptools import setup, find_packages

setup(
    ...
    install_requires=[
        'pandas>=1.0',          # good
        'scipy==1.1',           # bad
        'matplotlib>=2.2.1,<3' # good
    ],
)
```

- Allow as many package versions as possible
- Get rid of unused dependencies

Python versions

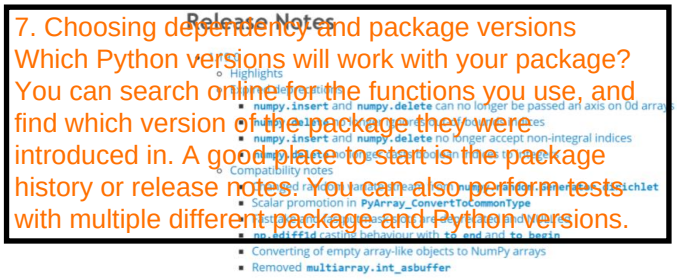
```
from setuptools import setup, find_packages

setup(
    ...
    python_requires='>=2.7, !=3.0.*, !=3.1.*',
)
```

6. Python versions
... Here we say that the version of Python installed must be >=2.7, but cannot be three-point-zero or three-point-one. The star must be used when we are excluding version numbers.

3. Adding dependencies to setup.py
To ensure that your users have the right packages installed, you can set the install-requires parameter inside the setup function of the setup script. Here you list the packages which your package

Choosing dependency and package versions

- Check the package history or release notes
  - e.g. the **NumPy release notes**
- Test different versions



7. Choosing dependency and package versions
Which Python versions will work with your package? You can search online for the functions you use, and find which version of the package they were introduced in. A good place to start is the package history or release notes. You can also perform tests with multiple different package and Python versions.

Making an environment for developers

```
pip freeze
```

```
alabaster==0.7.12
appdirs==1.4.4
argh==0.26.2
...
wrapt==1.11.2
yapf==0.29.0
zipp==3.1.0
```

4. Controlling dependency version
but sometimes your code will depend on functions which only exist in specific versions of another package. Let's say that a function used in our package was only introduced in pandas version one-point-zero. Here you say the user must have this version of pandas or later. You can also specify an exact version of a package. Here the user must have scipy version one-point-one. No other version will do. And here you say that the version of matplotlib must be at least two-point-two-point-one but cannot be version three or more.

Making an environment for developers

Save package requirements to a file

```
pip freeze > requirements.txt
```

Install requirements from file

```
pip install -r requirements.txt
```

8. Making an environment for developers
A really important part of developing any software is reproducibility. You and your package co-authors need to have the exact same versions of all the dependent packages so you can track down bugs. This is different to the install-requires dependencies where you try to allow as many different versions as possible, here you want to

```
mysklearn/
|-- mysklearn
|   |-- __init__.py
|   |-- preprocessing
|   |   |-- __init__.py
|   |   |-- normalize.py
|   |   |-- standardize.py
|   |-- regression
|   |   |-- __init__.py
|   |   |-- regression.py
|   |-- utils.py
|-- setup.py
|-- requirements.txt   <-- developer environment
```

9. Making an environment for developers
You should export this information into a text file which you include with your package. Then anyone can install all of the packages in this file later using this pip install command. Having the exact same set of packages makes it easier to hunt down any bugs.

Including licenses and writing READMEs

Open source licenses

- Find more information here
- Allow users to
  - use your package
  - modify your package
  - distribute versions of your package

What to include in a README

README sections

- Title
- Description and Features
- Installation
- Usage examples
- Contributing
- License

README format

Markdown (commonmark)

- Contained in `README.md` file
- Simpler
- Used in this course and in the wild

reStructuredText

- Contained in `README.rst` file
- More complex
- Also common in the wild

Commonmark

Contents of README.md

```
# mysklearn
mysklearn is a package for complete
**linear regression** in Python.

You can find out more about this package
on [DataCamp](https://datacamp.com)

## Installation
You can install this package using

```
pip install mysklearn
```
```

What it looks like when rendered

mysklearn

mysklearn is a package for complete **linear regression** in python.

You can find out more about this package on DataCamp

Installation

You can install this package using

```
pip install mysklearn
```

Adding these files to your package

Directory tree for package with subpackages

```
mysklearn/
|-- mysklearn
|   |-- __init__.py
|   |-- preprocessing
|   |   |-- ...
|   |-- regression
|   |   |-- ...
|   |-- utils.py
|-- setup.py
|-- requirements.txt
|-- LICENSE      <--- new files
|-- README.md    <--- added to top directory
```

MANIFEST.in

Lists all the extra files to include in your package distribution.

MANIFEST.in

Contents of `MANIFEST.in`

```
include LICENSE
include README.md
```

```
mysklearn/
|-- mysklearn
|   |-- __init__.py
|   |-- preprocessing
|   |   |-- ...
|   |-- regression
|   |   |-- ...
|   |-- utils.py
|-- setup.py
|-- requirements.txt
|-- LICENSE
|-- README.md
|-- MANIFEST.in   <---
```

Publishing your package

PyPI

Python Package Index

- `pip` installs packages from here
- Anyone can upload packages
- You should upload your package as soon as it might be useful

Distributions

- **Distribution package** - a bundled version of your package which is ready to install.
- **Source distribution** - a distribution package which is mostly your source code.
- **Wheel distribution** - a distribution package which has been processed to make it faster to install.

How to build distributions

```
python setup.py sdist bdist_wheel
```

- `sdist` = source distribution
- `bdist_wheel` = wheel distribution

4. How to build distributions
You can build source and wheel distributions from the terminal using this command. You run the setup script and pass sdist to make the source distribution and bdist-wheel to make the wheel distribution. This will create a dist directory and add wheel and source distributions inside. It will also create build and egg-info directories, but you can ignore these.

```
mysklearn/
|-- mysklearn
|-- setup.py
|-- requirements.txt
|-- LICENSE
|-- README.md
|-- dist                <---
|   |-- mysklearn-0.1.0-py3-none-any.whl
|   |-- mysklearn-0.1.0.tar.gz
|-- build
|-- mysklearn.egg-info
```

Getting your package out there

Upload your distributions to **PyPI**

```
twine upload dist/*
```

Upload your distributions to **TestPyPI**

```
twine upload -r testpypi dist/*
```

5. Getting your package out there
Now that you have built your distributions the only thing left to do is upload them. You can do this from the terminal using twine. Twine is a tool specifically made for uploading packages to PyPI. Here we upload all the distributions in the dist directory. You can also upload your distributions to the Test-PyPI repository, which is a version of the PyPI repository made for testing. In order to upload, you'll first have to go to either PyPI or Test-PyPI to register for an account.

```
mysklearn/
|-- mysklearn
|-- setup.py
|-- requirements.txt
|-- LICENSE
|-- README.md
|-- dist
|   |-- mysklearn-0.1.0-py3-none-any.whl
|   |-- mysklearn-0.1.0.tar.gz
|-- build
|-- mysklearn.egg-info
```

How other people can install your package

Install package from **PyPI**

```
pip install mysklearn
```

Install package from **TestPyPI**

```
pip install --index-url         https://test.pypi.org/simple
            --extra-index-url   https://pypi.org/simple
            mysklearn
```

6. How other people can install your package
Once you've done this, your package is live and anyone can install it using PIP. It is also possible to install your package from Test-PyPI using a longer command. You specify the index-url which is where the package is downloaded from, and the extra-index-url which is where PIP can search for your dependency packages.