Why build a package anyway?

- To make your code easier to reuse.
- To avoid lots copying and pasting.
- To keep your functions up to date.
- To give your code to others.

Course content

You will build a full package, and cover:

- File layout
- Import structure
- Making your package installable
- Adding licenses and READMEs
- Style and unit tests for a high quality package
- Registering and publishing your package to PyPI
- Using package templates

Scripts, modules, and packages

- Script - A Python file which is run like `python myscript.py` .
- Package - A directory full of Python code to be imported
  - e.g. `numpy` .
- Subpackage - A smaller package inside a package
  - e.g. `numpy.random` and `numpy.linalg` .
- Module - A Python file inside a package which stores the package code.
  - e.g. example coming in next 2 slide.
- Library - Either a package, or a collection of packages.
  - e.g., the Python standard library (`math` , `os` , `datetime` ,...)

Directory tree of a package

Directory tree for simple package

```
mysimplepackage/
|-- simplemodule.py
|-- __init__.py
```

- This directory, called `mysimplepackage` , is a Python Package
- `simplemodule.py` contains all the package code
- `__init__.py` marks this directory as a Python package

Contents of simple package

`__init__.py`

```

```

Empty file

`simplemodule.py`

```
def cool_function():
    ...
    return cool_result


...


def another_cool_function():
    ...
    return another_cool_result
```

File with generalized functions and code.

Subpackages

Directory tree for package with subpackages

```
mysklearn/
|-- __init__.py
|-- preprocessing
|   |-- __init__.py
|   |-- normalize.py
|   |-- standardize.py
|-- regression
|   |-- __init__.py
|   |-- regression.py
|-- utils.py
```

Why include documentation?

- Helps your users use your code
- Document each
  - Function
  - Class
  - Class method

```python
import numpy as np
help(np.array)
```

```
...
    array(object, dtype=None, copy=True)

    Create an array.

    Parameters
    ----------
    object : array_like
        An array, any object exposing the array
        interface ...
    dtype : data-type, optional
        The desired data-type for the array.
    copy : bool, optional
        If true (default), then the object is copied.
...
```

Why include documentation?

- Helps your users use your code
- Document each
  - Function
  - Class
  - Class method

```python
import numpy as np
x = np.array([1,2,3,4])
help(x.mean)
```

```
...
mean(...) method of numpy.ndarray instance
    a.mean(axis=None, dtype=None, out=None)

    Returns the average of the array elements
    along given axis.

    Refer to `numpy.mean` for full documentation.
...
```

Function documentation

```python
def count_words(filepath, words_list):
    """Count the total number of times these words appear.

    The count is performed on a text file at the given location.

    [explain what filepath and words_list are]

    [what is returned]
    """
```

Documentation style

Google documentation style

```
"""Summary line.

Extended description of function.

Args:
    arg1 (int): Description of arg1
    arg2 (str): Description of arg2
```

NumPy style

```
    """Summary line.

    Extended description of function.

    Parameters
    ----------
    arg1 : int
        Description of arg1 ...
```

reStructured text style

```
    """Summary line.

    Extended description of function.

    :param arg1: Description of arg1
    :type arg1: int
    :param arg2: Description of arg2
    :type arg2: str
```

Epytext style

```
"""Summary line.

  Extended description of function.

  @type arg1: int
  @param arg1:  Description of arg1
  @type arg2: str
  @param arg2: Description of arg2
```

NumPy documentation style

Popular in scientific Python packages like

- `numpy`
- `scipy`
- `pandas`
- `sklearn`
- `matplotlib`
- `dask`
- etc.

NumPy documentation style

```python
import scipy
help(scipy.percentile)
```

```
percentile(a, q, axis=None, out=None, overwrite_input=False, interpolation='linear')
    Compute the q-th percentile of the data along the specified axis.

    Returns the q-th percentile(s) of the array elements.

    Parameters
    ----------
    a : array_like
        Input array or object that can be converted to an array.
```

Other types include - `int` , `float` , `bool` , `str` , `dict` , `numpy.array` , etc.

NumPy documentation style

```python
import scipy
help(scipy.percentile)
```

```
percentile(a, q, axis=None, out=None, overwrite_input=False, interpolation='linear')
    ...
    Parameters
    ----------
    ...
    axis : {int, tuple of int, None}
    ...
    interpolation : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}
```

- List multiple types for parameter if appropriate
- List accepted values if only a few valid options

NumPy documentation style

```python
import scipy
help(scipy.percentile)
```

```
percentile(a, q, axis=None, out=None, overwrite_input=False, interpolation='linear')
    ...
    Returns
    -------
    percentile : scalar or ndarray
        If `q` is a single percentile and `axis=None`, then the result
        is a scalar. If multiple percentiles are given, first axis of
        the result corresponds to the percentiles...
    ...
```

Documentation templates and style translation

- `pyment` can be used to generate docstrings
- Run from terminal
- Any documentation style from
  - Google
  - Numpydoc
  - reST (i.e. reStructured-text)
  - Javadoc (i.e. epytext)
- Modify documentation from one style to another

Documentation templates and style translation

```
pyment -w -o numpydoc textanalysis.py
```

```
def count_words(filepath, words_list):
    # Open the text file
    ...
    return n
```

- `-w` - overwrite file
- `-o numpydoc` - output in NumPy style

Documentation templates and style translation

```
pyment -w -o numpydoc textanalysis.py
```

```
def count_words(filepath, words_list):
    """

    Parameters
    ----------
    filepath :

    words_list :


    Returns
    -------
    type
    """
```

Translate to Google style

```
def count_words(filepath, words_list):
    """Count the total number of times these words appear.

    The count is performed on a text file at the given location.

    Parameters
    ----------
    filepath : str
        Path to text file.
    words_list : list of str
        Count the total number of appearances of these words.

    Returns
    -------
```

Translate to Google style

```
pyment -w -o google textanalysis.py
```

```
def count_words(filepath, words_list):
    """Count the total number of times these words appear.

    The count is performed on a text file at the given location.

    Parameters
    ----------
    filepath : str
        Path to text file.
    words_list : list of str
        Count the total number of appearances of these words.

    Returns
    -------
```

Translate to Google style

```
pyment -w -o google textanalysis.py
```

```
def count_words(filepath, words_list):
    """Count the total number of times these words appear.

    The count is performed on a text file at the given location.

    Args:
      filepath(str): Path to text file.
      words_list(list of str): Count the total number of appearances of these words.

    Returns:


    """
```

Package, subpackage and module documentation

`mysklearn/__init__.py`

```
"""
Linear regression for Python
============================

mysklearn is a complete package for implmenting
linear regression in python.
```

`mysklearn/preprocessing/__init__.py`

```
"""
A subpackage for standard preprocessing operations.
"""
```

`mysklearn/preprocessing/normalize.py`

```
"""
A module for normalizing data.
"""
```

20. Package, subpackage and module documentation
There is also module, package, and subpackage documentation. This helps your users navigate your package and understand what it can do. Package documentation is placed in a string at the top of the package init-dot-py file. This should summarize the package. Similarly, subpackage documentation is placed at the top of the subpackage init-dot-py. Module documentation is written at the top of the module file.

STRUCTURING IMPORTS

Without package imports

```python
import mysklearn
```

```python
help(mysklearn.preprocessing)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'mysklearn' has no
  attribute 'preprocessing'
```

Directory tree for package with subpackages

```
mysklearn/
|-- __init__.py
|-- preprocessing
|   |-- __init__.py
|   |-- normalize.py
|   |-- standardize.py
|-- regression
|   |-- __init__.py
|   |-- regression.py
|-- utils.py
```

Without package imports

```python
import mysklearn.preprocessing
```

```python
help(mysklearn.preprocessing)
```

```
Help on package mysklearn.preprocessing in
mysklearn:

NAME
    mysklearn.preprocessing - A subpackage
      for standard preprocessing operations.
```

Directory tree for package with subpackages

```
mysklearn/
|-- __init__.py
|-- preprocessing
|   |-- __init__.py
|   |-- normalize.py
|   |-- standardize.py
|-- regression
|   |-- __init__.py
|   |-- regression.py
|-- utils.py
```

Without package imports

```python
import mysklearn.preprocessing
```

```python
help(mysklearn.preprocessing.normalize)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module
  'mysklearn.preprocessing' has no attribute
  'normalize'
```

Directory tree for package with subpackages

```
mysklearn/
|-- __init__.py
|-- preprocessing
|   |-- __init__.py
|   |-- normalize.py
|   |-- standardize.py
|-- regression
|   |-- __init__.py
|   |-- regression.py
|-- utils.py
```

5. Without package imports
So each module has to be imported too. We can stop this by using internal imports, to import the modules into the subpackages, and the subpackages into the package. This gives our package structure, and allows us connect its different parts.

Without package imports

```python
import mysklearn.preprocessing.normalize
```

```python
help(mysklearn.preprocessing.normalize)
```

```
Help on module mysklearn.preprocessing.normalize
in mysklearn.preprocessing:

NAME
    mysklearn.preprocessing.normalize - A module
      for normalizing data.
```

Directory tree for package with subpackages

```
mysklearn/
|-- __init__.py
|-- preprocessing
|   |-- __init__.py
|   |-- normalize.py
|   |-- standardize.py
|-- regression
|   |-- __init__.py
|   |-- regression.py
|-- utils.py
```

Importing subpackages into packages

`mysklearn/__init__.py`

Absolute import

```python
from mysklearn import preprocessing
```

- Used most - more explicit

Relative import

```python
from . import preprocessing
```

- Used sometimes - shorter and sometimes simpler

Directory tree for package with subpackages

```
mysklearn/
|-- __init__.py        <--
|-- preprocessing
|   |-- __init__.py
|   |-- normalize.py
|   |-- standardize.py
|-- regression
|   |-- __init__.py
|   |-- regression.py
|-- utils.py
```

6. Importing subpackages into packages
In the package's init-dot-py file, we import the subpackages. We can do this using absolute or relative imports. The path of absolute imports starts at the top of the package. Alternatively, we can use a relative import, and start the path from the current file. The dot here means the current file's parent directory.

Importing modules

We imported `preprocessing` into `mysklearn`

```python
import mysklearn
help(mysklearn.preprocessing)
```

```
Help on package mysklearn.preprocessing in
mysklearn:

NAME
    mysklearn.preprocessing - A subpackage
      for standard preprocessing operations.
```

But `preprocessing` has no link to `normalize`

```python
import mysklearn
help(mysklearn.preprocessing.normalize)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module
  'mysklearn.preprocessing' has no attribute
  'normalize'
```

8. Importing modules
Importing modules is the same as importing subpackages. Inside the subpackage's init-dot-py file, we add a statement to import the normalize module. We can use an absolute import, starting from the top of the package, or a relative import starting from this file.

Importing modules

`mysklearn/preprocessing/__init__.py`

Directory tree for package with subpackages

```
mysklearn/
|-- __init__.py
|-- preprocessing
|   |-- __init__.py    <--
|   |-- normalize.py
|   |-- standardize.py
|-- regression
|   |-- __init__.py
|   |-- regression.py
|-- utils.py
```

Absolute import

```python
from mysklearn.preprocessing import normalize
```

Relative import

```python
from . import normalize
```

Restructuring imports

```python
import mysklearn
```

```python
help(mysklearn.preprocessing.normalize.normalize_data)
```

```
Help on function normalize_data in module
mysklearn.preprocessing.normalize:

normalize_data(x)
    Normalize the data array.
```

9. Restructuring imports
Now when we import the package, we can access all the functions in this module. Sometimes this will leave you with a very long path to access a function.

Import function into subpackage

`mysklearn/preprocessing/__init__.py`

Directory tree for package with subpackages

```
mysklearn/
|-- __init__.py
|-- preprocessing
|   |-- __init__.py    <--
|   |-- normalize.py
|   |-- standardize.py
|-- regression
|   |-- __init__.py
|   |-- regression.py
|-- utils.py
```

Absolute import

```python
from mysklearn.preprocessing.normalize import \
    normalize_data
```

Relative import

```python
from .normalize import normalize_data
```

10. Import function into subpackage
You can set up shortcuts to get rid of this. Let's say in the normalize module, there is only one function you want users to access. The other code in the module is just helper functions for this. In the preprocessing init-dot-py file you can directly import the useful function.

Import function into subpackage

```
import mysklearn
```

```
help(mysklearn.preprocessing.normalize_data)
```

```
Help on function normalize_data in module
mysklearn_imp.preprocessing.normalize:

normalize_data(x)
    Normalize the data array.
```

Importing between sibling modules

In `normalize.py`

Absolute import

```
from mysklearn.preprocessing.funcs import (
    mymax, mymin
)
```

Relative import

```
from .funcs import mymax, mymin
```

Directory tree for package with subpackages

```
mysklearn/
|-- __init__.py
|-- preprocessing
|   |-- __init__.py
|   |-- normalize.py
|   |-- funcs.py
|   |-- standardize.py
|-- regression
|   |-- __init__.py
|   |-- regression.py
|-- utils.py
```

12. Importing between sibling modules
If a module file is getting too big, you might split it into multiple files. Here some of the functions from normalize have been moved into funcs-dot-py. However, we still need to use them inside the normalize module, so we need to import them. As usual, we can use absolute or relative imports to import the two functions.

Importing between modules far apart

A custom exception `MyException` is in `utils.py`

In `normalize.py`, `standardize.py` and `regression.py`

Absolute import

```
from mysklearn.utils import MyException
```

Relative import

```
from ..utils import MyException
```

Directory tree for package with subpackages

```
mysklearn/
|-- __init__.py
|-- preprocessing
|   |-- __init__.py
|   |-- normalize.py   <--
|   `-- standardize.py <--
|-- regression
|   |-- __init__.py
|   |-- regression.py  <--
`-- utils.py
```

13. Importing between modules far apart
There might be classes or functions you want to use in many of your modules. Here there is a custom exception class which is defined in utils-dot-py. To import this into these other modules, we can use this absolute import, or this relative import. Previously, we said that one dot in a path meant the parent directory of the file. Two dots means the parent of the parent directory. Starting from normalize-dot-py in the file tree, its parent is preprocessing, and its parent is my-sklearn. Then from the utils file in this directory we import the exception class.

Relative import cheat sheet

- `from . import module`
    - From current directory, import `module`
- `from .. import module`
    - From one directory up, import `module`
- `from .module import function`
    - From module in current directory, import `function`
- `from ..subpackage.module import function`
    - From subpackage one directory up, from `module` in that subpackage, import `function`