Lecture 18 - Sharing Code

Aidan Hunt, University of Washington

Learning Objectives

After this lesson, students will be able to:

- Import modules to use scripts that are in other directories on their computer
- Create and import simple packages
- Use git for version control and sharing code

Check-in

- Homework 6 pseudocode graded
- Homework 6 full code due Friday
- Weds office hours moved to Friday (virtual)

Framing

- We've written a lot of good code this quarter for solving engineering problems.
- In your careers, you may start to build-up a codebase of scripts that you'd like to use and re-use.
- You may also wish to share this codebase with others.
- Today, we're going to talk about:
  - Importing code we've written for us in other scripts
  - Using git for sharing and version control of code

Modules and Packages

A **module** is another name for a simple Python script. Every `.py` file we've created this quarter is a module. As we've seen throughout the quarter, we can import modules to use them in our code.

Some examples of modules that we've used:

- `string` module
- `glob` module
- `os` module

Importing our own modules

How can we import a module? It's sort of like reading files. If the module is in the same directory as our script, it's easy. Consider `testModule1.py`, a script in the same folder as this notebook.

In [ ]:
```
# contents of testModule1.py
"""
This is a module that is in the same folder as our main script
"""

# Define some variables
var1 = 1
var2 = 2
var3 = 3

# Define a function
def printWordLength(word):
    print(word, 'has', len(word), 'characters in it.')

# Print a message
print('Hello from test module #1!')
```

In [1]:
```
import testModule1
```

```
Hello from test module #1!
```

Notice that when we import this module:

- the code inside is automatically executed (hence the message is printed). Note that this only happens the first time we import the module.
- We also have access to any variables and/or functions that are defined via the dot notation we have used with other modules and packages we have imported

In [2]:
```
# Access var1 attribute of exampleModule
testModule1.var1
```

Out[2]:
```
1
```

In [3]:
```
# Access var2 attribute of exampleModule
testModule1.var2
```

Out[3]:
```
2
```

In [4]:
```
# Use the printWordLength function of exampleModule
testModule1.printWordLength('perfunctory')
```

```
perfunctory has 11 characters in it.
```

As with any other module or package, we can also import our own modules under custom names (e.g., `import numpy as np`):

In [5]:
```
import testModule1 as tm

tm.printWordLength('perfunctory')
```

```
perfunctory has 11 characters in it.
```

Importing modules in other directories

What about importing modules that are in other directories on our computer? For example, consider another module, `testModule2.py`, that is located in the Test Modules subdirectory, one level below the current directory.

In [ ]:
```
# File structure:
```

```
# .\
    # (This notebook)

    # testModule1.py

    # Test Modules\
        # testModule2.py
```

In [ ]:

```
# contents of testModule2.py
def printMessage():
    print('Hello! This is a message from the test module!')

def addNumbers(num1, num2):
    return num1 + num2

print('You just imported the test module #2! Nice!')
```

In [6]:

```
import testModule2
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_10788\2673808499.py in <module>
----> 1 import testModule2

ModuleNotFoundError: No module named 'testModule2'
```

As we found before with files, we need to tell Python where to look for modules and packages we'd like to import. Python looks in directories that are "on the path".

In [7]:

```
# Import the sys module to get the system path on Aidan's computer
import sys
sys.path
```

Out[7]:

```
['C:\\Users\\Aidan Hunt\\MREL Dropbox\\Aidan Hunt\\STEP-UP - ME 498\\Lectures\\Lecture 1
8 - Sharing Code',
 'C:\\Users\\Aidan Hunt\\anaconda3\\python39.zip',
 'C:\\Users\\Aidan Hunt\\anaconda3\\DLLs',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib',
 'C:\\Users\\Aidan Hunt\\anaconda3',
 '',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib\\site-packages',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib\\site-packages\\win32',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib\\site-packages\\win32\\lib',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib\\site-packages\\Pythonwin',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib\\site-packages\\IPython\\extensions',
 'C:\\Users\\Aidan Hunt\\.ipython']
```

Note that the current directory is on the path, in addition to a bunch of other directories where packages like numpy are stored.

We can add a directory of interest to the path as using `sys.path.append()`:

In [8]:

```
# Create the directory path we want to add using the os package
# Let's make a relative path for ease of teaching, but note that an absolute path would
import os
newPath = os.path.join('.', 'Test Modules')

# Add the path
sys.path.append(newPath)

sys.path
```

Out[8]:

```
['C:\\Users\\Aidan Hunt\\MREL Dropbox\\Aidan Hunt\\STEP-UP - ME 498\\Lectures\\Lecture 1
8 - Sharing Code',
 'C:\\Users\\Aidan Hunt\\anaconda3\\python39.zip',
 'C:\\Users\\Aidan Hunt\\anaconda3\\DLLs',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib',
 'C:\\Users\\Aidan Hunt\\anaconda3',
 '',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib\\site-packages',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib\\site-packages\\win32',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib\\site-packages\\win32\\lib',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib\\site-packages\\Pythonwin',
 'C:\\Users\\Aidan Hunt\\anaconda3\\lib\\site-packages\\IPython\\extensions',
 'C:\\Users\\Aidan Hunt\\.ipython',
 '.\\Test Modules']
```

Note that Python will remember this new path until we restart it. Now let's try to import:

In [9]:
```
import testModule2
```

```
You just imported the test module #2! Nice!
```

In [10]:
```
testModule2.printMessage()
```

```
Hello! This is a message from the test module!
```

Packages

Packages are simply collections of modules! Packages can also contain "subpackages", which in turn can contain modules and subpackages.

We've worked with various packages throughout the quarter:

- `numpy`
- `pandas`
- `matplotlib`

The syntax for importing a package is the same as that of importing a module. **But how can we create and import our own packages?**

Consider 3 modules, `subModule1.py`, `subModule2.py`, and `subModule3.py`, which we would like to organize into a package.

In [ ]:
```
# Contents of subModule1.py
"""
This module defines several constants (attributes).
"""

attribute1 = 5
attribute2 = 'hello'
attribute3 = False
```

In [ ]:
```
# Contents of subModule2.py
"""
This example module defines several simple functions for performing
mathematical operations on two numbers.
"""

def addNums(num1, num2):
    return num1 + num2
```

```python
def subtractNums(num1, num2):
    return num1 - num2

def multiplyNums(num1, num2):
    return num1 * num2

def divideNums(num1, num2):
    return num1 * num2
```

In [ ]:
```python
# Contents of subModule3.py
"""
This module defines several simple functions for performing string operations
"""

import string

def removePunctuation(word):
    for char in string.punctuation:
        word = word.replace(char, '')

    return word

def removeDigits(word):
    for num in string.digits:
        word = word.replace(num, '')

    return word
```

The first step is to put them all in a single directory:

In [ ]:
```python
# File structure:

# .\
    # (This notebook)

    # testModule1.py

    # Test Modules\
        # testModule2.py

        # testPackage\
            # subModule1.py

            # subModule2.py

            # subModule3.py
```

To indicate to Python that this directory is a package, we add a special file, `__init__.py`. This file can be empty, or it can include code that will be executed when the package is imported.

In [ ]:
```python
# Contents of __init__.py
"""
The __init__file tells Python that this directory is a package. It can be an
empty file, or include code that is executed upon importing the package.
"""

print('You just imported the test package!')
```

In [ ]:
```python
# File structure:

# .\
    # (This notebook)
    # testModule1.py

    # Test Modules\
        # testModule2.py

        # testPackage\
            # __init__.py

            # subModule1.py

            # subModule2.py

            # subModule3.py
```

Now we can import this package as normal (**note**: we already added the directory that `testPackage` is in to `sys.path`, so we don't need to re-add):

In [11]:
```
import testPackage
```

```
You just imported the test package!
```

However, by default, the submodules of `testPackage` are not imported:

In [12]:
```
# Try to access attribute2 in subModule1 of testPackage
testPackage.subModule1.attribute2
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_10788\2267735736.py in <module>
      1 # Try to access attribute2 in subModule1 of testPackage
----> 2 testPackage.subModule1.attribute2

AttributeError: module 'testPackage' has no attribute 'subModule1'
```

The easiest way to do this is state the subModules that you would like to import:

In [13]:
```
from testPackage import subModule1 as sm

sm.attribute2
```

Out[13]:
```
'hello'
```

Alternatively, if there are certain submodules we would always like to be imported, we can import those in our **init**.py file. Then they will be available simply from importing the overall package.

In [ ]:
```
# Updated __init__.py
"""
The __init__file tells Python that this directory is a package. It can be an
empty file, or include code that is executed upon importing the package.
"""

# Automatically subModule1 whenever testPackage is imported
from testPackage import subModule1

print('You just imported the test package!')
```

In [14]:
```
import testPackage

testPackage.subModule1.attribute2
```

Out[14]:
```
'hello'
```

Some packages automatically import all submodules (think `numpy`) whereas others don't import any by default (e.g., in `matplotlib`, we manually import submodules like `pyplot` or `colors`).

Sharing your code

So you've created a package and you want to share it. What's the best way?

- Typically, you should use Git. This is ideal for working with small teams, or keeping your code synchronized between several computers.
- If you intend for large scale use (e.g., anyone in the world), you could upload it to the **Python Package Index**.

Git

Git is a version control software that is used to create and manage repositories of files. To demonstrate Git, we must leave Jupyter Notebook. See the Git quick guide document on the course website.

In this quick guide document we cover:

- Creating remote repositories using Github
- Creating local repositories using the command line/Git Bash
- Adding files to a repository
- Commiting and pushing changes
- Looking at commit history in Github