Python modules

Organization

- Your code should be organized in some way
- Code should often be split across multiple files for ease of maintenance and reuse
- For large projects you will probably have multiple directories each with multiple files

Modules

- Code in Python can be organized and accessed as modules
- We've already used some modules that are part of Python (`math`, `time`, etc.)
- These modules were accessed using the import statement

Import

Here is an example of importing and using a function from the `time` module:

```
import time
time.time()
```

```
time()
```

```
print(type(time))
print(type(time.time))
```

Keep in mind that the module name/object is different then the function that exists inside of the module.

Reference to a function

Functions are also objects and may be assigned to a variable:

```
t = time.time
type(t)
```

```
t is time.time
```

```
t()
```

Import a single function

We can import a single function from a module:

```
from time import time
print(type(time))
print(time())
```

```
import time
print(type(time))
print(time.time())
```

Another example is `from math import sqrt`.

Import and rename

We can rename a function in the import statement:

```
from time import time as timer
print(type(timer))
print(timer())
```

Wild card import

We can import everything from a module into the global namespace with:

```
from time import *
print(type(time))
print(time())
```

This is normally not a good idea, because you may unknowingly overwrite some symbols that have been defined elsewhere.

Modules and namespaces

- Not only do modules allow you to separate code into multiple files, but they also provide distinct namespaces
- Namespaces are particularly important in larger projects where reuse of common terms could be confusing at best
- Attribute renaming and/or wild card imports can make code less readable and more difficult to debug

Example Here we know where `time()` is coming from:

```
import time
import mymodule
# ...
t = time.time()
```

Does `time()` come from `time` or `mymodule`?

```
from time import *
from mymodule import *
# ...
t = time()
```

Recommendation: be explicit when using module functions!

Writing your first module

See file `mymodule1.py`:

```
def summation(a,b):
    total = 0
    for n in range(a,b+1):
        total += n
    return total

primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Using your first module

From the command line working in the `lecture-07` directory:

```
$ python3
>>> import mymodule1
>>> mymodule1.summation(1,100)
5050
>>> mymodule1.primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

From Jupyter notebook:

```
import mymodule1
print(mymodule1.summation(1,100))
print(mymodule1.primes)
```

Improving your module

Add test code in file `mymodule2.py`:

```
def summation(a,b):
    total = 0
    for n in range(a,b+1):
        total += n
    return total

primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

print('Testing function summation():...'),
total = summation(1,100)
if (total == 5050):
    print('OK')
else:
    print('FAILED')
```

Testing your new module

```
import mymodule2
print(mymodule2.summation(1,100))
print(mymodule2.primes)
```

Import process

When you do `import mymodule2` several things happen

1. Python interpreter looks for a `.py` file with the same name as the module, starting with your current directory followed by looking in system wide locations
2. Code is byte compiled from the `.py` file to a `.pyc` file
3. File is processed from top to bottom

Locating modules

- Searches for a module are based on directories in the `sys.path` list
- First item in the `sys.path` list is an empty string, `''`, which is used to denote the current directory

```
%ls
```

```
import sys
print(sys.path)
```

Let's remove this directory from `sys.path` and try to load a module (that we have not yet loaded).

```
import sys
sys.path.remove('')
import mymodule3
```

If we add it back, everything will be ok:

```
sys.path.insert(0,'')
import mymodule3
```

`.pyc` files

```
$ ls *.py*
mymodule1.py  mymodule1.pyc  mymodule2.py  mymodule2.pyc
```

- When you import a file Python byte compiles the file
- `.pyc` files are faster to load, but the runtime performance once you have them loaded is exactly the same

`__name__` and `__main__`

- Special variable `__name__` is equal to `__main__` if the file is being executed as the main program
- `__name__` will not be equal to `__main__` if the file is being imported

"Hiding" code during import

See `mymodule3.py`

```
def summation(a,b):
    total = 0
    for n in range(a,b+1):
        total += n
    return total

primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

if __name__ == '__main__':
    print('Testing function summation():...', end='')
    total = summation(1,100)
    if (total == 5050):
        print('OK')
    else:
        print('FAILED')
```

Another try at importing

```
import mymodule3
print(mymodule3.summation(1,100))
print(mymodule3.primes)
```

Running the test code

From the command line:

```
$ python3 mymodule3.py
Testing function summation()... OK
```

Documenting the module

See `mymodule4.py`:

```
"""
My module of misc code.
"""

def summation(a,b):
    """
    Returns the sum of numbers between, and including, a and b.
    """

    total = 0
    for n in range(a,b+1):
        total += n
    return total

primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

if __name__ == '__main__':
    print('Testing function summation():...'),
    total = summation(1,100)
    if (total == 5050):
        print('OK')
    else:
        print('FAILED')
```

Accessing your documentation

```
import mymodule4
help(mymodule4)
```

Recommended Reading

- Chapter 22: Modules: The Big Picture
- Chapter 23: Module Coding Basics