Using modules

Modules allow for code-reuse and portabililty. Modules are sought in order along the `sys.path` directory list. The `PYTHONPATH` and `PYTHONHOME` environment variables are very influential.

Standard library

The standard library modules are documented here in the Global module index.

```
>>> import math          #Importing math module
>>> dir(math)            #Provides a list of module attributes
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'a
>>> help(math.sqrt)
>>> radius=14.2
>>> area=math.pi*(radius**2); area  #Using a module variable
633.47074266984589
>>> a=14.5; b=12.7
>>> c=math.sqrt(a**2+b**2); c   #Using a module function
19.275372888740698
```

Once a module is imported, it is included in the `sys.modules` list. The first step of `import` is to check this list for the desired module and not re-import it again. to Sometimes you just need a specific function from a given module.

The `from <module> import <name>` syntax handles that case,

```
>>> from math import sqrt # will overwrite previous definitions in the namespace
>>> sqrt(a) # shorter to use
3.8078865529319543
```

Note that you have to use the `reload` function to re-import modules into the workspace. Importantly, `reload` won't work with the `from` syntax used above. Thus, if you are developing code and constantly re-loading it, it is better to keep the top level module name so you can keep re-loading it with `reload`.

There are many places to hijack the import process. These make it possible to create virtual environments and to customize execution, but it's best to stick with the well-developed solutions for these cases (see `conda` below).)

Exercise

Exercise

1. Write a function that returns the number of calendar days in a given year and month. Hint: see the `calendar` module in the standard library.
2. Find the number of leap-years since 1900.
3. Find the day of the week of your birthday.

Writing and Using Your Own Modules

Put the following code in a separate file named "mystuff.py"

```
def sqrt(x):
   return x*x
```

and then return to the interative session

```
>>> from mystuff import sqrt
>>> print sqrt(3)
9
>>> import mystuff
>>> print dir( mystuff)
>>> mystuff.sqrt(3)
```

Now, add the following function to your mystuff.py file:

```
def poly_func(x):
   return 1+x+x*x
```

And change your previous function:

```
def sqrt(x):
   return x/2.
```

and then return to the interative session

```
>>> print mystuff.sqrt(3)
```

Did you get what you expected?

```
>>> print mystuff.poly_func(3)
```

Note

IPython provides some autoreloading but with lots of caveats. Better to use `reload`.

Using a directory as a module

The trick is to put a `__init__.py` file in the top level of the directory you want to import from. The file can be empty. For example,

```
package/
    __init__.py
    moduleA.py
```

So, if `package` is in your path, you can do `import package`. If `__init__.py` is empty, then this does nothing. To get any of the code in `moduleA.py`, you have to explicitly import it as `import package.moduleA` and then you can get the contents of that module. For example,

```
>>> package.moduleA.foo()
```

runs the `foo` function in the `moduleA.py` file. If you want to make `foo` available upon importing `package`, then you have to put `from moduleA import foo` in the `__init__.py` file. Then, you can do `import package` and then run the function as `package.foo()`. You can also do `from package import foo` to get foo directly. When developing your own modules, you can have fine-grained control of which packages are imported using relative imports.

Dynamic Importing

In case you don't know the names of the modules that you need to import ahead of time, the `__import__` function can load modules from a specified list of module names.

```
>>> sys = __import__('sys')  #  as a function that takes argument
>>> moduleNames = ['sys', 'os', 're', 'unittest']
```

```
>>> moduleNames
['sys', 'os', 're', 'unittest']
>>> modules = map(__import__, moduleNames) # access the modules from the list to use them
```

Using `__main__`

Namespaces are named in order to distinguish between importing and running a Python script.

```
if __name__ == '__main__':
   # these statements are not executed during import
   # do run statements here
```

Getting modules from the web

Python packaging has really improved over the last couple of years. This had always been a sore point, but now it is a *lot* easier to deploy and maintain Python codes that rely on linked libraries across multiple platforms.

Python packages in the main Python Package Index support `pip`.

```
% pip install name_of_module
```

This figures out all the dependencies and installs them, too. There are many flags that control how and where packages are installed. You don't need root access to use this. See the `--prefix` flag for non-root access. Strangely enough, Microsoft has been releasing free tools for packaging and distributing Python modules (especially for Python 3.x). Modern Python packaging relies on so-called *wheel* files that include fragments of compiled libraries that the module depends on. These have always been painful in Windows, but that is changing quickly due to the new support from Microsoft.

A fantastic source of scientific Python on Windows in the form of wheel files comes from Christoph Gohlke's lab at UCI.

Conda package management

You should really use `conda` whenever possible. It soothes so many package management headaches and you don't need administrator rights to use it effectively. The *anaconda* toolset is a curated list of scientific packages that is supported by Anaconda. This has just about all the scientific packages you want. Outside of this support, the community also supports a longer list of scientific packages as conda-forge. You can add `conda-forge` to your usual repository list with the following,

```
% conda config --add channels conda-forge
```

And then you can install new modules as in the following:

```
% conda install name_of_module
```

Additionally, `conda` also supports building self-contained sub-environments that are a great way to safely experiment with codes and even different Python versions. This can be key for automated provisioning of virtual machines in a cloud-computing environment. For example, the following will create an environment named `my_test_env` with the Python version 2.7.

```
% conda create -n my_test_env python=2.7
```

Package management resources

- PyPI – Python Package Index
- Conda documentation
- Advanced Features of Conda Part 1
- Using PyPi Packages with Conda
- Conda cheatsheet
- My Python Environment Workflow with Conda
- Explicit Conda Environment Specifications
- Using PyPi Packages with Conda
- Execute scripts in their own temporary environment
- Python Circular Imports
- python – How can I find out which module a name is imported from? – Stack Overflow