

# AMS 595: Fundamentals of Computing (Python Unit)

## Lecture 10: Advanced Topics: Packaging; Extending Python with C

Xiangmin Jiao

Stony Brook University

# Outline

- 1 Modules and Packaging
- 2 Extending Python with C

# More about Modules

- Use “import file\_base\_name” to access a module
  - ▶ Module's name is available as `__name__` in the module
  - ▶ Separate namespace by default
  - ▶ If run as “python modulename.py args” then `__name__` is set “to `__main__`”
- Just like UNIX has a path that it searches, in order, for an executable, python uses this to find modules. Search order:
  - ▶ Current directory
  - ▶ PYTHONPATH environment variable
  - ▶ System-wide python installation default path
- `sys.path` will show the path

# Useful Facts about Modules

- “Private” variables in a module
  - ▶ “from modulename import `*`” imports all variables and functions in the module, except for any names starting with “`_`”
  - ▶ If you change a module, you can re-import it as: “`reload(modulename)`”
- Python is interpreted, but it creates byte-code files (`.pyc` instead of `.py`) when a module is first imported
  - ▶ This speeds up the loading of the module—it does not change the speed of execution
  - ▶ `.pyc` is automatically recreated based on the file modification times
  - ▶ Note that `.pyc` files are not, in general, portable

# Packaging of Python Modules

- Often you separate a project into multiple files / directories
- Example from python docs:

sound/	Top-level package
__init__.py	Initialize the sound package
formats/	Subpackage for file format conversions
__init__.py	
wavread.py	
wavwrite.py	
aiffread.py	
aiffwrite.py	
auread.py	
auwrite.py	
...	
effects/	Subpackage for sound effects
__init__.py	
echo.py	
surround.py	
reverse.py	
...	
filters/	Subpackage for filters
__init__.py	
equalizer.py	
vocoder.py	
karaoke.py	
...	

# Packaging of Python Modules

- You can do:
  - ▶ `import sound.effects.echo` (just get that single module)
    - ★ Access as `sound.effects.echo.echofilter(...)`
  - ▶ `from sound.effects import echo`
    - ★ Access as `echo.echofilter(...)`
  - ▶ `from sound.effects.echo import echofilter` (make that specific function available)
    - ★ Access as `echofilter(...)`
- In order to do `from sound.effects import *`, we need to define what we mean by all
  - ▶ The package should have an `__init__.py` file—this tells python that a directory contains packages
    - ★ Can be empty, but needs to be present (example with our previous module)
    - ★ Set `__all__` to the list of modules that should be imported by default

# Packaging of Python Modules

- The main purpose of packaging is to put your python module in a system-wide location for you can import it from other python modules
- If you are just writing a simple script that will not be used by other python modules, then you don't need to worry about packaging— just distribute the source
  - ▶ Same for a more complex, but still standalone python program.
  - ▶ As seen in the path, there is a system-wide location and user-specific location  $\times$  the number of python versions
- Some example cases:
  - ▶ You have a single .py (mymodule.py) file that you want people to be able to import.
    - ★ This will be placed in the main library directory, like:  
/usr/lib/python2.7/site-packages/
    - ★ You can then just import mymodule
  - ▶ You have extension modules for your application, and you want to make it easy on the user

# Packaging of Python Modules

- There are several different options for packaging your python code for other users
  - ▶ Unfortunately, these appear to be in a state of flux at the moment
  - ▶ `distutils` and `setup.py` makes writing extensions easy
- Main methods (at the moment):
  - ▶ `distutils`: this is part of python (2 and 3)
  - ▶ `setuptools`: newer than `distutils`, offers more functionality. Introduced `easy_install`, and `setuptools` module to import into `setup.py`



# distutils

- Simple example (from their docs):

- ▶ We want to distribute foo.py
- ▶ setup.py:

```
from distutils.core import setup
```

```
setup(name='foo ',  
      version='1.0 ',  
      py_modules=['foo '],)
```

- ▶ Keyword arguments to setup() are metadata and the list of modules that comprise the package
- ▶ Create source distribution: `python setup.py sdist`
  - ★ Creates a .tar.gz (or .zip)
- ▶ Install the package: `python setup.py install`
- ▶ Commands exist for creating Windows executable installers, RPMs, ...
  - ★ See `python setup.py bdist --help-formats`

# Outline

- 1 Modules and Packaging
- 2 Extending Python with C

# Interoperability with C

- In contrast to Python, which is interpreted, C is a compiled language
- There are primarily two reasons to use C code from Python
  - ▶ Speed up the code
  - ▶ Reuse existing codes
- Extension Methods
  - ▶ [C-API](#) for writing extension modules in Python or embed Python in C
  - ▶ [ctypes](#): a module that allows you to call functions in shared libraries
  - ▶ [SWIG](#): simplified wrappers and interface generator
  - ▶ [Cython](#) is an optimizing static compiler for both Python and extended Cython and can convert python into compiled C code
- Cython is often where you should start

Readings: [Interface with C](#) by Haenel as part of SciPy lecture notes

# Speeding Up Python

- Let's look at performance of these different extensions.
- Consider an example of solving Laplace's equation using Gauss-Seidel iteration

pure python: 10.9347379208 s

NumPy: 0.0756549835205 s

Cython: 0.0472810268402 s

ctypes: 0.149159908295 s

C-API: 0.0446910858154 s

# Using Cython

- Following [examples here](#)