Packaging and Applications

Modules

- See https://docs.python.org/2/tutorial/modules.html
  - We follow this explicitly here
- Simplest form: single file
  - `import file_base_name` to access the module's contents
    - Module's name is available as `__name__` in the module
  - Separate namespace by default
  - Can have a block that is executed if run from the commandline
    - If run as "`python modulename.py args`" then `__name__` is set to "`__main__`"
- "private variables"
  - If you do: "`from modulename import *`", then everything is imported into the current namespace *except* for any names that start with "`_`"
- If you change a file, you can re-import it as: `reload(modulename)`

Search Path

- 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

Compilation

- 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

- 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

- 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

- 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
  - We already saw how `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`.
  - See: http://stackoverflow.com/questions/6344076/differences-between-distribute-distutils-setuptools-and-distutils2 and https://python-packaging-user-guide.readthedocs.org/en/latest/tutorial.html

Commandline Arguments

- Python provides the usual argc/v variables through the sys module
- Several modules exist that help you parse these variables
  - `getopt`: the original module—need to do a lot manually
    - Based on the C `getopt()`
  - `optparse`: an alternative, but this is supposed to be deprecated
  - `argparse` automates a lot of things for you (including help and usage)
  - See: http://legacy.python.org/dev/peps/pep-0389/#why-aren-t-getopt-and-optparse-enough