Packaging and Applications

Commandline Arguments

- Python provides the usual `argc/v` variables through the `sys` module
  - You don't want to parse these yourself
- 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 deprecated (as of python 3.2)
  - `argparse` automates a lot of things for you (including help and usage)
    - Use this, unless you need to support old versions of python
  - See: http://legacy.python.org/dev/peps/pep-0389/#why-aren-t-getopt-and-optparse-enough

Commandline Arguments

- Simple example: `argparse_example.py` in our class git repo
  - https://github.com/sbu-python-class/python-science/blob/master/examples/commandline/argparse_example.py
- Example: our course grading tool
  - https://github.com/zingale/slack_grader

Modules

- See https://docs.python.org/3/tutorial/modules.html
  - Good source of information

Modules

- Simplest form: single file examples/packaging/single_file/example.py) in class git repo
  - `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 "_"

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
- Note that there is a user-specific path too (typically in `~/.local/lib/`)
  - This is managed by site
  - Look at `python3 -m site --user-site`

Compilation

- Python is interpreted, but it creates byte-code files when module is first imported
  - Python 2: `.pyc` in same directory
  - Python 3: `.pyc` in a `__pycache__/` sub-directory
- 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
  - Simple example for our class (examples/packaging/example/ in the class git repo)

```
mytest/                          module name
├── demo/                        subpackage
│   ├── demo.py
│   └── __init__.py
├── __init__.py                  initialization file for our module
└── util/
    ├── __init__.py
    └── msg.py
```

- The `__init__.py` file is needed to make python see the directory as a package. This allows it to be imported
  - It can be completely empty
  - As we'll see, you can put python code in this to help make things convenient for the user

(that's the output from the commandline `tree` tool)

Packaging

- You can do:
  - import mytest.demo.demo
    - You access functions in demo.py through this (long) namespace, e.g., `mytest.demo.demo.show()`
  - `from mytest.demo import demo`
    - This will import `demo.py` and make the routines it contains available in the `demo` namespace, e.g., `demo.show()`
    - This is a preferred way to import submodules (this is not like using a `*` import)
  - `import mytest`
    - `help(mytest)` will show the doc string from the `__init__.py`
    - Actions taken depend on what's in the `__init__.py` (need to uncomment lines)
      - We have `from mytest.demo import demo`
      - this makes the `demo.py` module available directly in `mytest`, so we can do `mytest.demo.show()`

Packaging

- 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
  - For a simple script, usually don’t worry about packaging—just distribute the single .py file
  - For a complex, but standalone application, again, usually don’t need a centralized install
- As seen in the path, there is a system-wide location and user-specific location × the number of python versions

Packaging

- 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/python3.5/site-packages/`
    - You can then just `import mymodule`

Packaging

- You have an application split into submodules
  - Maybe some extensions that need to be compiled
  - Typical directory layout:

```
foo2/                          this is where version control is
├── foo2/                      this is what you import
│   ├── foo.py
│   └── __init__.py
└── setup.py                   this controls the packaging
```

  - The `setup.py` will coordinate the building and the entire (second) `foo2/` hierarchy will be put into `site-packages`
    - e.g., `python3 setup.py install --user`
  - A user can now `import foo2`

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
  - `distutils`/`setuptools` 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://packaging.python.org/

PyPI

- PyPI: the python package index
  - https://pypi.python.org/pypi
  - Provides a central place to install python packages from (using `pip`)
  - `pip` will install things from source or using a new format called wheels to do a binary install
  - `conda` is an alternative to `pip`, but not part of the main python distribution
- `setuptools` can be used to upload your package to PyPI

Current Recommendations

- See https://pāckaging.python.org/current/
- Installation:
  - `pip` to install packages from PyPI
  - `conda` for distribution cross-platform software stacks
- Packaging tools:
  - `setuptools` to create source distributions
  - `bdist_wheel` via `setuptools` to create binary distributions
  - `twine` to upload to PyPi

setuptools

- Setuptools works a lot like distutils
- Has a `develop` mode:

> *Setuptools allows you to deploy your projects for use in a common directory or staging area, but without copying any files. Thus, you can edit each project's code in its checkout directory, and only need to run build commands when you change a project's C extensions or similarly compiled files. You can even deploy a project into another project's checkout directory, if that's your preferred way of working (as opposed to using a common independent staging area or the site-packages directory).*
>
> *To do this, use the setup.py develop command. It works very similarly to setup.py install or the EasyInstall tool, except that it doesn't actually install anything. Instead, it creates a special .egg-link file in the deployment directory, that links to your project's source code. And, if your deployment directory is Python's site-packages directory, it will also update the easy-install.pth file to include your project's source code, thereby making it available on sys.path for all programs using that Python installation.*

https://pythonhosted.org/setuptools/setuptools.html#development-mode

setuptools

- See http://python-packaging.readthedocs.io/en/latest/minimal.html for a minimal example
- We write a setup.py

```
from setuptools import setup
setup(name="foo2",
      version="1.0",
      description="this is a setuptools test example",
      packages=["foo2"],
      author="Michael Zingale",
      author_email="michael.zingale@stonybrook.edu",
      license="BSD"
      )
```

- Available commands:
  - `python3 setup.py -h`
  - `python3 setup.py –help-commands`

setuptools

- Installing:
  - Systemwide install (as root): `python3 setup.py install`
  - User: `python3 setup.py install –user`
    - This creates an *egg* file in your `site-packages/` directory. This is like a zip file with all the necessary module info
    - Adds a line to the `easy-install.pth` file pointing to the egg
  - Test it out:
    - `import foo2`
    - `help(foo2)` will show that the module was imported from the egg in the `site-packages/` directory
  - Note: if you change the source afterwards, it will not be reflected in your installed version

setuptools

- Installing as a developer:
  - You want it in your system path, but you don't want to be able to continue to develop the sources
  - `python setup.py develop –user`
    - Creates an egg-link file that points to the source
    - Adds the path to your source into the `easy-install.pth` file
  - Now if you modify your source as you develop, it is reflected in your python environment
    - Note: if you have extension modules, you have to re-setup each time you change these

What We Haven't Covered...

- Visualization:
  - MayaVi: 3-d plotting using VTK (http://docs.enthought.com/mayavi/mayavi/)
  - Visual python: easy generation of 3-d figures (http://vpython.org/)
  - yt: visualizing grid-based and particle data (http://yt-project.org/)
- mpi4py
- h5py: HDF5
- pil: python image library (http://www.pythonware.com/products/pil/)
- GUIs
- pytest and unit testing
- Extension modules, Cython, numba, ...