One of the key features of Python is that the actual core language is fairly small. This is an intentional design feature to maintain simplicity. Much of the powerful functionality comes through external modules and packages.

The main work of installation so far has been to supplement the core Python with useful modules for science analysis.

Module

A module is simply a file containing Python definitions, functions, and statements. Putting code into modules is useful because of the ability to import the module functionality into your script or IPython session, for instance:

```
import astropy
import astropy.table
data = astropy.table.Table.read('my_table.fits')
```

You'll see `import` in virtually every Python script and soon it will be second nature.

Question:

Importing modules and putting the module name in front is such a bother, why do I need to do this?

Answer:

It keeps everything modular and separate. For instance many modules have a `read()` function since this is a common thing to do. Without using the `<module>.<function>(...)` syntax there would be no way to know which one to call.

Tip

Sometimes it is convenient to make an end-run around the `<module>.` prefixing. For instance when you run `ipython --pylab` the interpreter does some startup processing so that a number of functions from the numpy and matplotlib modules are available *without* using the prefix.

Python allows this with this syntax:

```
from <module> import *
```

That means to import every function and definition from the module into the current namespace (in other words make them available without prefixing). For instance you could do:

```
from astropy.table import *
data = Table('my_table.fits')
```

A general rule of thumb is that `from <module> import *` is OK for interactive analysis within IPython but you should avoid using it within scripts.

Package

A package is just a way of collecting related modules together within a single tree-like hierarchy. Very complex packages like NumPy or SciPy have hundreds of individual modules so putting them into a directory-like structure keeps things organized and avoids name collisions. For example here is a partial list of sub-packages available within SciPy

scipy.fftpack	Discrete Fourier Transform algorithms
scipy.stats	Statistical Functions
scipy.lib	Python wrappers to external libraries
scipy.lib.blas	Wrappers to BLAS library

scipy.lib.lapack	Wrappers to LAPACK library
scipy.integrate	Integration routines
scipy.linalg	Linear algebra routines
scipy.sparse.linalg	Sparse Linear Algebra
scipy.sparse.linalg.eigen	Sparse Eigenvalue Solvers
scipy.sparse.linalg.eigen.arpack	Eigenvalue solver using iterative methods.

Exercise: Import a package module and learn about it

Import the Linear algebra module from the SciPy package and find out what functions it provides.

Finding and installing other packages

If you've gotten this far you have a working scientific Python environment that has *most* of what you will ever need. Nevertheless it is almost certain that you will eventually find a need that is not met within your current installation. Here we learn **where** to find other useful packages and **how** to install them.

Package resources

Google

Google "python blah blah" or "python astronomy blah blah"

Resource lists

There are a number of sites specifically devoted to Python for astronomy with organized lists of useful resources and packages.

- Astropython.org resources
- Comfort at 1 AU
- Astronomical Python

Good vs. bad resources

When you find some package on the web, look for a few things:

- Good modern-looking documentation with examples
- Installs easily without lots of dependencies (or has detailed installation instructions)
- Actively developed

PyPI

The Python Package Index is the main repository for 3rd party Python packages (about 14000 packages and growing). An increasing number of astronomy related packages are available on PyPI, but this list misses a lot of available options.

The advantage of being on PyPI is the ease of installation using `pip install <package_name>`.

Exercise: Find packages for coordinate manipulations

Find one or more Python packages that will transform coordinates from Galactic to FK5 ecliptic.

Hint: tags are helpful at astropython.org and don't forget the "next" button at the bottom.

Package installation

There are two standard methods for installing a package.

pip install

The `pip install` script is available within our scientific Python installation and is very easy to use (when it works). During the installation process you already saw many examples of `pip install` in action. Features include:

- If supplied with a package name then it will query the PyPI site to find out about that package. Assuming the package is there then `pip install` will automatically download and install the package.

- Will accept a local tar file (assuming it contains an installable Python package) or a URL pointing to a tar file.
- Can install in the user package area via `pip install <package or URL> --user` (but see discussion further down)

python setup.py install

Some packages may fail to install via `pip install`. Most often there will be some obvious (or not) error message about compilation or missing dependency. In this case the likely next step is to download the installation tar file and untar it. Go into the package directory and look for files like:

```
INSTALL
README
setup.py
setup.cfg
```

If there is an INSTALL or README file then hopefully you will find useful installation instructions. Most well-behaved python packages do the installation via a standard `setup.py` script. This is used as follows:

```
python setup.py --help  # get options
python setup.py install # install in the python area (root / admin req'd)
python setup.py install --user # install to user's package area
```

More information is available in the Installing Python Modules page.

Where do packages get installed?

An important option in the installation process is where to put the package files. We've seen the `--user` option in `pip install` and `python setup.py install`. What's up with that? In the section below we document how this works. See the discussion in Multiple Pythons on your computer for a reason you might want to do this, but first please read this warning:

Warning

We strongly recommend against installing packages with `--user` unless you are an expert and really understand what you are doing, . This is because the local user version will always take precedence and can thus potentially disrupt other Python installations and cause hard-to-understand problems. Big analysis packages like CIAO, STSci_Python or CASA are carefully tested assuming the integrated environment they provide. If you start mucking this up then all bets are off.

WITH `--user`

Packages get installed in a local user-owned directory when you do something like either of the following:

```
pip install --user aplpy
python setup.py install --user
```

This puts the packages into:

Mac	~/Library/Python/2.x/lib/python/site-packages
Linux	~/.local/lib/python-2.x/site-packages
Windows	%APPDATA%/Python/Python2x/site-packages

Note

On Mac if you did not use Anaconda or the EPD Python Framework then you may see user packages within `~/.local/lib` as for linux. This depends on whether Python is installed as a MacOS Framework or not.

WITHOUT `--user`

If you use Anaconda or a non-root Python installation then there is no issue with permissions on any platform since the entire installation is local to a directory you own.

However, installing to a system-wide Python installation will require root or admin privilege. Installing this way has the benefit of making the package available for all users of the Python installation, but has the downside that it is more difficult to back out changes if required. In general we recommend using *only* the system package manager (e.g. `yum`) to install packages to the system Python. This will ensure integrity of your system Python, which is important even if you are the only user.

How do I find a package once installed?

Finding the file associated with a package or module is simple, just use the `help` command in IPython:

```
import scipy
help scipy
```

This gives something like:

```
NAME
    scipy

FILE
    /usr/local/lib/python2.6/site-packages/scipy/__init__.py

DESCRIPTION
    SciPy: A scientific computing package for Python
    ================================================

    Documentation is available in the docstrings and
    online at http://docs.scipy.org.
    ...
```

Uninstalling packages

There is no simple and fully consistent way to do this unless you use solutions like Anaconda or Canopy. The Python community is working on this one. In most simple cases, however, you can just delete the module file or directory that is revealed by the technique shown above.

Getting help on package installation

If you attempt to install a package but it does not work, your basic options are:

- Dig in your heels and start reading the error messages to see why it is unhappy. Often when you find a specific message it's time to start googling by pasting in the relevant parts of the message.
- Send an email to the AstroPy mailing list astropy@scipy.org. Include:
    - Package you are trying to install
    - URL for downloading the package tar file
    - Your platform (machine architecture and exact OS version)
    - Exactly what you typed
    - Entire output from the `python setup.py install` process

Do NOT just write and say “I tried to install BLAH and it failed, can someone help?”

Where does Python look for modules?

The official reference on Modifying Python’s Search Path gives all the details. In summary:

When the Python interpreter executes an import statement, it looks for modules on a search path. A default value for the path is configured into the Python binary when the interpreter is built. You can determine the path by importing the sys module and printing the value of `sys.path`:

```
$ python
Python 2.2 (#11, Oct  3 2002, 13:31:27)
[GCC 2.96 20000731 (Red Hat Linux 7.3 2.96-112)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.3', '/usr/local/lib/python2.3/plat-linux2',
 '/usr/local/lib/python2.3/lib-tk', '/usr/local/lib/python2.3/lib-dynload',
 '/usr/local/lib/python2.3/site-packages']
>>>
```

Within a script it is possible to adjust the search path by modify `sys.path` which is just a Python list. Generally speaking you will want to put your path at the front of the list using insert:

```
import sys
sys.path.insert(0, '/my/path/python/packages')
```

You can also add paths to the search path using the PYTHONPATH environment variable.

Multiple Pythons on your computer

This is a practical problem that you are likely to encounter. Straight away you probably have the system Python (/usr/bin) and the Anaconda Python. Then if you install PyRaf, CIAO, and CASA you will get one more Python installation for each analysis package (there are good reasons for this). **In general, different Python installations cannot reliably share packages or resources**. Each installation should be considered as its own local Python universe.

Installing within each Python

Now that you know about all the great packages within our Scientific Python installation, you might want to start using them in your PyRAF or CASA or CIAO analysis.

If you start digging into Python you will likely come across the technique of setting the `PYTHONPATH` environment variable to extend the list of search paths that Python uses to look for a module. Let’s say you are using CIAO Python and want to use SciPy functions. You might be tempted to set `PYTHONPATH` to point to the directory in EPD where the SciPy modules live. This will fail because the EPD Python modules were compiled and linked assuming they’ll be run with EPD Python. With effort you might find a way to make this work, but in general it’s not a workable solution.

What *will* often work is to follow the package installation procedure for each desired package within each Python installation. This assumes that you have write permission into the directories where the analysis package files live. Simply enter the appropriate analysis environment, then do then following:

- At the command line do `which python` to verify that `python` is the correct one from the analysis environment.
- Navigate to http://pypi.python.org/pypi/pip#downloads
- Download the latest version of pip (*pip-X.Y.tar.gz*)

- Untar that file, go in the tar directory, and do `python setup.py install`
- Do `rehash` (for csh) then `which pip` to make sure the new `pip` got installed into your analysis environment path.
- Now you can do `pip install <package>` or `python setup.py install` for each desired package within that analysis environment.

It's worth noting that the original example of SciPy will not install with `pip`. It requires a very tricky installation from source, so unless SciPy ships with your favorite analysis environment you are out of luck with that one.

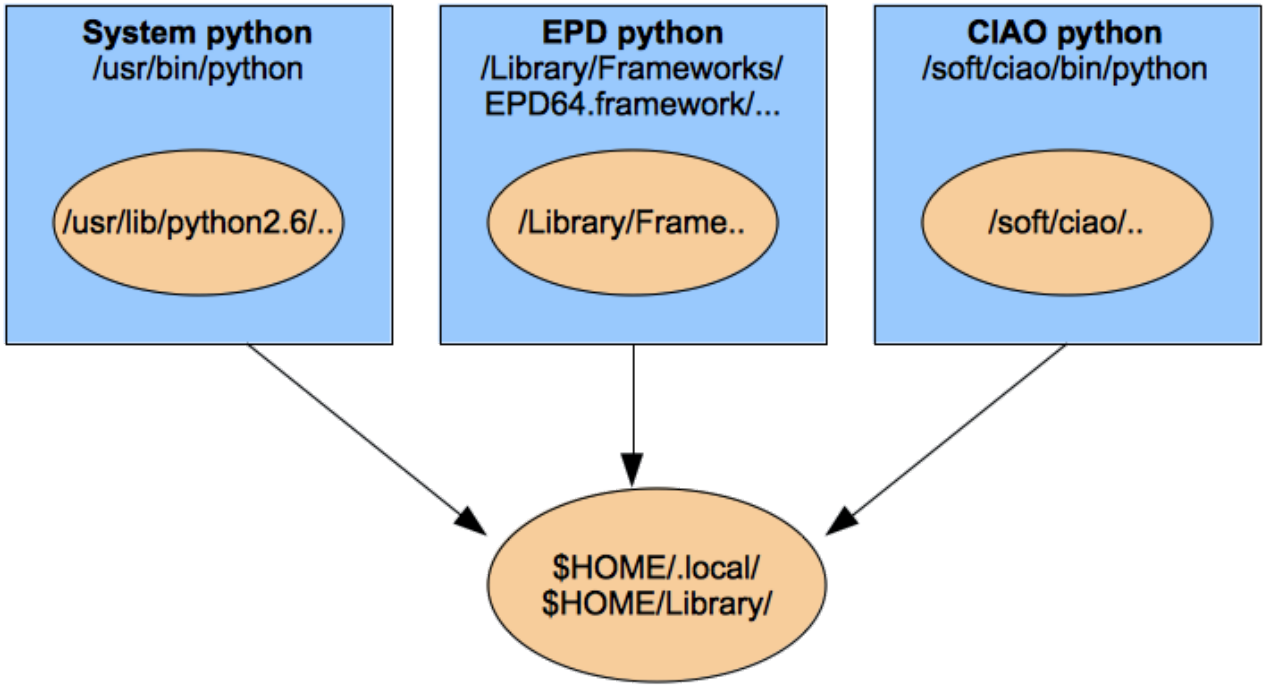
If you do *not* have write access to the analysis package directories, then you need to use the `--prefix` option in `pip` to install in a local area and then set a corresponding `PYTHONPATH`.

Can we share packages?

In some cases you can successfully share between Pythons. Although we don't recommend doing this it is nevertheless useful to illustrate how this works.

> **Warning**
>
> This technique is prone to breaking things in strange ways and we do not recommend it.

The first rule is that they need to be the same major version, i.e. all 2.6 or 2.7. This is because Python always includes a major version like `python2.6/` in the default search path so Python 2.7 will never find 2.6 packages. The second rule is to install packages using the `--user` option in `pip install` or `setup.py install`. This results in the situation shown below where each Python can find common packages in the local user area:



Be sure to test that the package you installed works within the other Python environments.

Virtualenv

Virtualenv is a very useful tool for creating isolated Python environments. As seen in the linux non-root install it provides a way to make a virtual clone of an existing Python environment. This clone can then be used as the package installation location.

One use case is wanting to install a new or experimental version of a package without overwriting the existing production version in your baseline environment.

For a good introductory tutorial see http://iamzed.com/2009/05/07/a-primer-on-virtualenv/.

Anaconda environments

The Anaconda Python distribution, in conjunction with the `conda` package manager, makes it easy maintain multiple Python environments under one tree. This is extremely useful if you need to install different versions of packages, perhaps for testing or for running a particular application which has certain package requirements. See Python Packages and Environments with conda for an introduction.

Final exercises

Exercise [intermediate]: Fully install APLpy

Go to the APLpy install page and read the instructions. Manually install all of the Python package dependencies with the `--user` option or try the auto-install script available there.

For extra credit install the Montage C library as discussed on the APLpy install page. Then try to run the example Making a publication quality plot that was shown in the introductory talk. The necessary input files are in the `install_examples.tar` file.

Exercise [intermediate]: Install HDF5 and PyTables

Install HDF5 and PyTables. This will let you read HDF5 tables in Python. HDF5 is a data file format which can store and manipulate extremely large or complex datasets in a scalable manner. It is the baseline for some data-heavy facilities such as LOFAR.

Exercise [expert]: Install SciPy and all dependencies from source

Attempt to follow the instructions for building from source in the Installing SciPy page. (No binary downloads!). This will be useful if you want to use the very latest development version of Python or else want to use the system-dependent build optimization so your numerical libraries are the fastest possible. For most people this is not needed.

If you can do this then consider yourself an expert on Python installation.