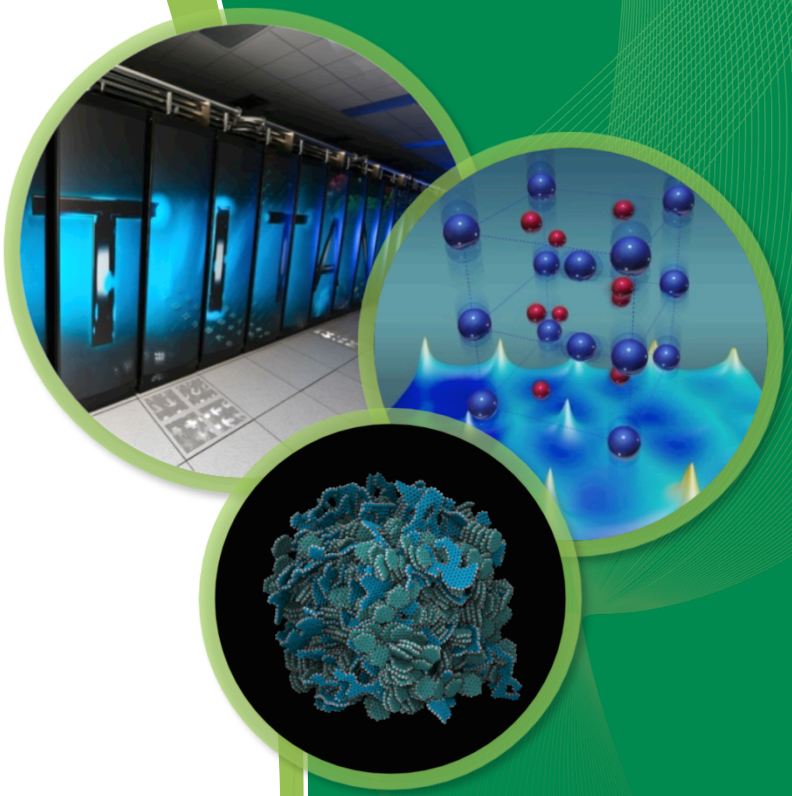
Managing Python on OLCF Resources

Tips for managing your own third-party Python packages, distributions, and environments on OLCF resources.



Using Python the standard way

- Load the center provided Tcl environment modules*:
  - `module avail “python_”`
    `module load python python_numpy/1.9.2`
- This works great…
  - We do the work
  - Packages optimized when possible for the target environment
- …except when it doesn’t
  - Innumerable reasons this can’t work for everyone

*Nomenclature Disambiguation	
Envmod	Tcl Environment Module
Pymod	Python Module (i.e. any *.py file)
Python Package	A collection of pymods denoted by a file ‘package/__init__.py’

Ways to manage your own Python stack

- Install packages to your user site-packages directory
  - Easy; Useful when package is: pure-python; not shared among project; not used on Cray compute nodes
- Install packages to an arbitrary alternate prefix
- Install custom Python stack snapshots with virtualenv
  - Flexible and relatively easy
- Deploy custom python root install
  - Need Python 2 and 3 simultaneously? Install from source.
  - Or Anaconda; promises to be easier but isn't always so

General Recommendations

- Use a consistent PrgEnv for all packages that will be used together
  - Many utility packages (pure Python) have no compiler concerns
- Use system compiler when possible on Crays so binaries compatible with both front- and back-end node architectures
- Precautions must be taken to avoid envmod conflicts
  - Always load dependency environment modules before altering `PATH`, `PYTHONPATH`, `LD_LIBRARY_PATH`, etc. yourself
  - Environment modules can interfere with manual changes you make
  - Alternative: work WITH environment modules and write your own modulefile (see backup slide)

Filesystem considerations: Where should custom packages be installed?

- Project NFS space `/ccs/proj/{PROJECT_ID}`
  - Generally recommended
  - Project-shared by default
  - Not purged; available on Cray compute nodes
- Home NFS space (`$HOME`, `site.USER_SITE`)
  - Convenient for personal packages; not purged
  - Not mounted on Cray compute nodes, OK on Rhea
  - USER_SITE packages may conflict between resources/interpreters
  - Not shared with collaborators without effort
- Paths on Atlas/Lustre are not recommended
  - Generally avoid `$PROJWORK`, `$MEMBERWORK`, `$WORLDWORK` for project executables.

User Site-Packages

- Located at `$HOME/.local/lib/pythonX.Y/site-packages`
  - Python searches this path for packages automatically
  - Scripts and binaries installed to `$HOME/.local/bin`
    - Append to your $PATH early in your Shell's initialization (~/.profile || ~/.bash_profile || ~/.zshenv):
      `export PATH=$PATH:$HOME/.local/bin # or equivalent`
  - Reserve for simple packages; may conflict with other extension methods
- Use with center-provided Python distribution
  - `module load python/x.y.z python_pip`
    `pip install --user $PKGNAME`
  - `module load python python_setuptools`
    `python setup.py install --user`

Installing to an arbitrary prefix

- Prefer architecture-specific prefixes
  - Clearly separates machine-specific packages
  - Install all packages for each system in each prefix
- Install with pip:
  -
```
pip install --no-binary :all: \
    --prefix=/ccs/proj/{PROJID}/local/titan \
    PKGNAME
```
- Optimize for target
  - `` `--no-binary :all:` `` compiles all binaries when able.
  - Not necessarily as optimized as center-provided envmods
  - Load all desired dependency envmods *prior to install and at runtime*.

Virtualenv: Isolated python deployments

- Virtualenv allows many independent user-managed python environments to co-exist on a single system.
- All your eggs in one basket:
  - a single site-packages
  - All package binaries, scripts, libs installed under one tree
- Bootstrap with:
  - ```
    $ module load python/x.y.z python_virtualenv
    $ virtualenv /ccs/proj/{PROJECT_ID}/venvs/titan-app
    $ module unload python_virtualenv
    ```
- `python_virtualenv` envmod only needed to create the virtualenv
    ```

Virtualenv: Isolated python deployments

- Private python binary links against parent `libpython`
  - Must always have original interpreter envmod loaded
- Activated and deactivated dynamically
  - via shell scripts/functions to modify a shell environment

```
$ . /ccs/proj/{PROJECT_ID}/venvs/app/bin/activate
(app)$ which python; which pip; pip --version
(app)$ pip --trusted-host pypi.python.org install -U pip
(app)$ pip install --no-binary :all: numpy nose ipython
(app)$ python -c "import numpy as np; np.lib.test('full');"
$ deactivate
```

Virtualenv + Environment Modules

- Original interpreter version and all non-python dependencies *need be loaded prior to activation*
  - `$ module load python hdf5...` # pip and virtualenv not needed!
    `$ . /path/to/venv/bin/activate`

- Environment modules and virtualenv both alter `PATH` with *utter disregard* for each other
  - Running `` `module load` ``, `` `swap` `` or `` `unload` `` while a virtualenv is active will *always* leave the environment in a broken state
  - Always deactivate venvs before changing environment modules

Virtualenv + provided packages

- Install all the packages you need into your own venv
- For most simple packages, installing from the Python Package index with pip works fine.

Not recommended
to mix, copy, or symlink
packages provided by envmods into a venv!

Build your own root install

- Build from source
  - Allows concurrent install of Python 2 and 3
  - Example build script: `https://code.ornl.gov/snippets/11`
  - Best left to the fearless
    - Best performance when using complex packages but challenging to manage dependencies compared to Anaconda
    - Less convenient than virtualenvs if using simple packages
- Anaconda
  - Easier than above; complete scientific python distribution
  - Base installation uses pre-compiled binaries
    - may cause problems between frontend-backend Cray nodes
    - Use anaconda's pip to force upgrade and re-build binaries
    - `.../anaconda/bin/pip install -U --force $PKG`

Anaconda

- Best to have separate install for each resource
  - `wget https://repo.continuum.io/archive/Anaconda2-4.1.1-Linux-x86_64.sh`
    `bash ./Anaconda2-4.1.1-Linux-x86_64.sh -p /ccs/proj/.../anaconda/titan`
  - Do not let it modify *PATHs in your login scripts!
    - Your init scripts must work on multiple resources.
    - Activating your install at login is fine, but should be managed carefully
- Some packages must be (re)built specially for Crays
  - Use Anaconda-installed pip binary as shown in a moment
  - Wheels work well on Rhea.

Activating personal extensions

- Always load dependency envmods first (have I said this enough?)
- `usersite` is always searched when reachable.
- Alternate Prefixes (each):
  - export PYTHONPATH="$PREFIX/lib/pythonX.Y/site-packages:$PYTHONPATH"
    # and if needed:
    export PATH="$PREFIX/bin:$PATH"
    export LD_LIBRARY_PATH="$PREFIX/lib:$LD_LIBRARY_PATH"
- Virtualenvs:
  - `. /path/to/venv/bin/activate`
- Full install from source:
  - `export PATH="/path/to/titan/root/bin:$PATH"`
    `export LD_LIBRARY_PATH="/path/to/titan/root/lib:$LD_LIBRARY_PATH"`
- Anaconda:
  - `export PATH="/path/to/titan/root/bin:$PATH"`
- Good practice to script these tasks along with any dependency envmods or write custom modulefiles (see backup slide)

Batch Job and `aprun` Considerations

- Activating extensions from shell init scripts may not apply
  - Good idea to verify loaded modules and *PATHs are correct:
    ```
    module -t list
    (echo "Loaded PATH is\n$PATH:--" | tr ':' '\n' 1>&2)
    ```
- Cray ALPS transfer of binaries to compute nodes can interfere with paths needed by some pymods
  - Use `aprun -b` to bypass transfer
    - slower execution
    - all binaries and libs must be on Lustre or project NFS filesystem
  - Or explicitly set environment variables on the compute nodes with `aprun -e PYTHONPATH=$PYTHONPATH ...`

Complex Packages for Cray Compute Nodes

- Some builds require Cray cross-compile wrappers:
  - ```
    $ module swap PrgEnv-pgi PrgEnv-gnu
    $ . /path/to/venv/bin/activate
    (venv)$ env CRAYPE_LINK_TYPE=dynamic MPICC=cc MPICXX=CC \
            pip -v install --no-binary :all: mpi4py
    ```
  - Binaries built with Cray wrappers unlikely to execute on login nodes!
    ```

Complex Packages for Cray Compute Nodes

- Others will configure linked-dependencies themselves:
  - ```
    $ module swap PrgEnv-pgi PrgEnv-gnu
    $ module load cray-hdf5
    $ . /path/to/venv/bin/activate
    (venv)$ pip -v install --no-binary :all: h5py
    ```
- Packages build with system gcc normally
  - Will run on login nodes if linked libraries use reduced instruction set
  - Linked dependencies envmods must be loaded prior to `import`
- Use the `pip -v` flag to see cause of build failures.
    ```

Final thoughts

- Managing your own Python stacks and extensions is a practical way to tailor Python to your needs be it through
  - User site-packages,
  - Alternate prefix installs,
  - Virtualenvs,
  - An interpreter source install or Anaconda
- We are working to improve the Python user experience
  - We welcome your thoughts, concerns, and feedback and would appreciate if you shared these with us through our User Survey.

And now for something completely different (Backup)

Provided Environment Modules: Why so complicated?

- Library dependencies provided by other envmods
  - HDF5, MPI, BLAS/LAPACK, PrgEnv-gnu, other pymods, etc
  - ABI compatibility issues for compiled components
    - Only pure python works everywhere under one installation
    - Compiled binaries typically require separate build for each arch/compiler
- Must support specific versions site-packages
  - Each project has different package/version needs
  - `pkg_resources` approach infeasible for user projects
- Provided packages are limited or versions are outdated
  - Complex dependency graph; pollution of PYTHONPATH
  - Python 2 and 3 do not currently co-exist

It is impractical to provide+maintain envmods for all packages/versions/targets

How Python modules are imported

- $PYTHONPATH (sys.path) searched in-order for pymods/ packages:
  - current working directory
  - site.prefix/lib/pythonX.Y
  - site.prefix/usr/lib/pythonX.Y/site-packages
  - site.prefix/usr/lib/pythonX.Y/site-packages/*.pth
  - site.USER_SITE (if site.ENABLED_USER_SITE): $HOME/.local/lib/pythonX.Y/site-packages
- Pure Python packages are trivially re-locatable
- Encountered `.pth` files do complex things to `sys` and `site`.
- Namespaces are searched upwards until first module match

The problem with PYTHONPATH complexity

- Complex `PYTHONPATH`s can lead to runtime module conflicts
- Suppose `PYTHONPATH="$FOO:$BAR"`:
  - Path $BAR contains a *poorly-packaged* module `` `bar` `` which imports module `` `foo` ``, needing version 3 or greater. `` `foo` `` version 3.2 is co-installed in $BAR
  - Path $FOO contains a crufty `` `foo` `` version 0.1a1
  - On first import of `foo`, the search in sys.path yields crufty `foo`
  - `imports` happen only once and the `global` namespace is now polluted with a `` `foo` `` that will cause runtime errors in `` `bar` ``.

Pip SSL Errors

Older versions of pip may not be able to verify newer SSL/TLS certificates at https://pypi.python.org.

If you cannot install a package using the latest available version of pip, a host can be explicitly trusted using:

```
pip --trusted-host pypi.python.org \
    install --user $PKGNAME
```

Don't get man-in-the-middled! Only bypass this check if you verify the remote host is valid!

Otherwise, manually install the package from source.

Your very own pip

- The following should be exercised with extreme caution.
- Install latest version of pip to your `userbase`:
  - Securely obtain `https://bootstrap.pypa.io/get-pip.py`
  - Run:
  - ```
    module load python/x.y.z
    python get-pip.py -v -U --user
    ```
- This work-around for the impatient is unsupported, but relatively safe for non-ccsstaff users.
- Users with write access to `/sw/*/python` installations must absolutely not do this: mistakes can leave the python installation in an inconsistent state.
    ```

Alternate Virtualenv Activation

- Venvs can also be activated within a python script run by the parent interpreter
  - ```
    venv_path = '/path/to/venv/bin/activate_this.py'
    execfile(venv_path, dict(__file__=venv_path))
    ```
    ```

Cray Libsci vs Lapack Lite vs Anaconada?

- Cray libsci install:

```
module swap PrgEnv-pgi PrgEng-gnu
module load fftw
. /path/to/venv/bin/activate
mkdir -p /tmp/$USER; cd $_
wget https://github.com/numpy/numpy/archive/v1.11.1.tar.gz
tar xf $_
cd numpy-1.11.1
# edit site.cfg - see backup slide
python setup.py build
python setup.py install
cd $HOME;
python -c "import numpy as np; np.test()"
```

Ran 6151 tests in 111.088s
OK (KNOWNFAIL=6, SKIP=3)

Cray Libsci vs Lapack Lite vs Anaconada?

- Anaconda

```
module swap PrgEnv-pgi PrgEnv-gnu
export PATH=$PREFIX/bin:$PATH"
python -c "import numpy as np; np.test()"
```

Ran 6151 tests in 123.582s
OK (KNOWNFAIL=6, SKIP=3)

- Lapack Lite install:

```
module swap PrgEnv-pgi PrgEng-gnu
. /path/to/venv/bin/activate
pip install --no-binaries :all: numpy
python -c "import numpy as np; np.test()"
```

Ran 6151 tests in 109.440s
OK (KNOWNFAIL=6, SKIP=3)

similar performance for locally compiled numpy:
better optimized than pre-compiled binaries.
Is the effort building against libsci justified?

Numpy site.cfg using PrgEnv-gnu, cray-libsci

```
[blas]
blas_libs = sci_gnu,sci_gnu_mp
library_dirs = $CRAY_LIBSCI_PREFIX_DIR/lib
rpath = $CRAY_LIBSCI_PREFIX_DIR/lib

[lapack]
language = f77
lapack_libs = sci_gnu,sci_gnu_mp
library_dirs = $CRAY_LIBSCI_PREFIX_DIR/lib
include_dirs = $CRAY_LIBSCI_PREFIX_DIR/include
rpath = $CRAY_LIBSCI_PREFIX_DIR/lib


[fftw]
libraries = fftw3
library_dirs = $FFTW_LIB
include_dirs = $FFTW_INC
```

$ENVARS must be fully expanded in practice

Matplotlib

- GTk rendering backends are not usually available
- Work-around by configuring the use of TkAgg
- In file `$HOME/.matplotlib/matplotlibrc` set:
  - backend: TkAgg

Play Nice with Environment Modules

Example: Personal anaconda

- Install Anaconda to `/ccs/proj/abc123/anaconda/titan`
- `export MODULEPATH="$HOME/.modulefiles:$MODULEPATH"`
- In `$HOME/.modulefiles/my-anaconda/titan`:
  - #%Module
    ```
    module-whatis "My Anaconda on Titan"
    prereq PrgEnv-gnu
    conflict python python_anaconda my-anaconda
    set PREFIX /ccs/proj/abc123/anaconda/titan
    prepend-path PATH $PREFIX/bin
    ```
- Now you can `module load my-anaconda/titan`
- See `man modulefile` for full set of directives

Activating Extensions at Login

```
-------------
$HOME/.bashrc
-------------
function _source_env () {
    [ -f "$HOME/.envs/$1" -a -z "$_NOSOURCEMYENV" ] && . "$HOME/.envs/$1";
}

case "$HOSTNAME" in
    titan*) _source_env(titan.sh) ;;
    rhea*)  _source_env(rhea.sh) ;;
    eos*)   _source_env(eos.sh) ;;
    *) ;;
esac
export _NOSOURCEMYENV=1

--------------------
$HOME/.envs/titan.sh
--------------------
# Make envmod changes from clean state to desired state with all python
# package dependencies. Only source in a clean environment!
module swap PrgEnv-pgi PrgEnv-gnu
module load cray-hdf5
export PATH="/path/to/my/anaconda/bin:$PATH"
# Or add alternate prefix PYTHONPATHS, etc...
```