Introduction to Python Environments

Encapsulating packages and their dependencies

How does Python structure code?

- Namespace
  - A container used at runtime to hold Python symbols (and their values)
  - A symbol could be:
    - A variable
    - A class definition
    - A function definition
    - A module (containing its own namespace of functions, variables, classes, etc.)

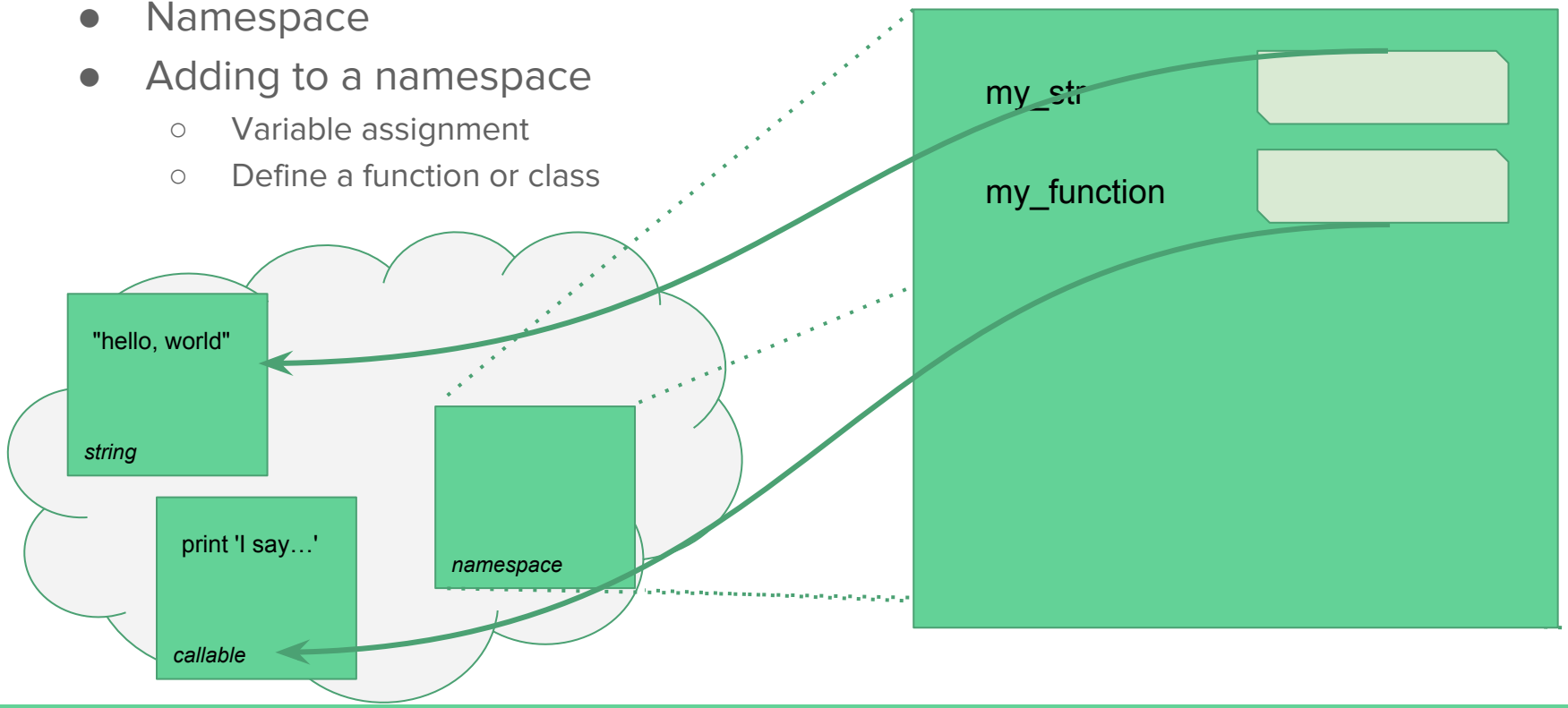
How does Python structure code?

- Namespace
- Adding to a namespace
  - Variable assignment
  - Define a function or class

```
my_str = 'hello, world'

def my_function(s = ''):
    print 'I say:  {0:s}'.format(s)
```

How does Python structure code?

- Namespace
- Adding to a namespace
  - Variable assignment
  - Define a function or class



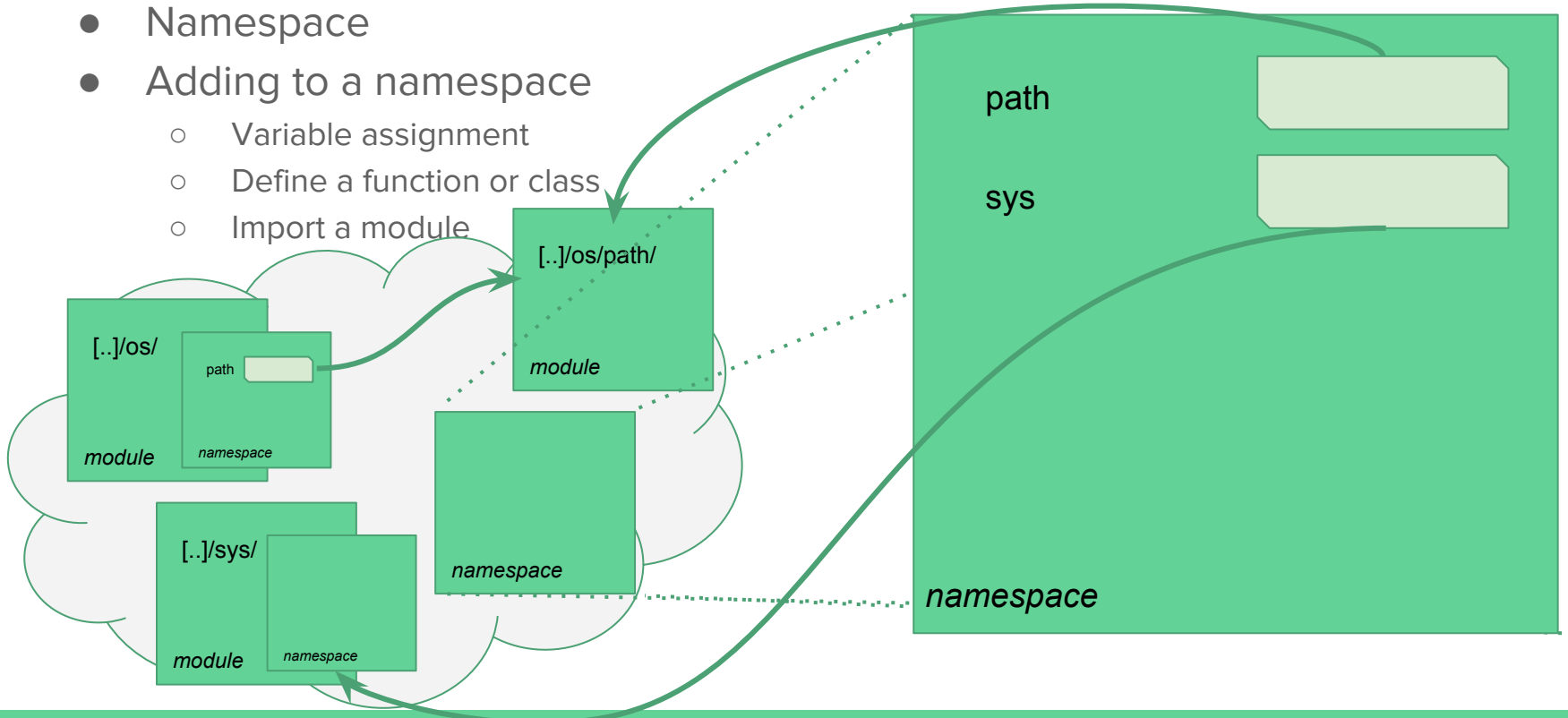
How does Python structure code?

- Namespace
- Adding to a namespace
  - Variable assignment
  - Define a function or class
  - Import a module

```
my_str = 'hello, world'

def my_function(s = ''):
    print 'I say:  {0:s}'.format(s)

import sys

from os import path
```

How does Python structure code?

- Namespace
- Adding to a namespace
  - Variable assignment
  - Define a function or class
  - Import a module



What is a module?

- A Python module is a directory containing Python scripts
  - Most often, the scripts represent a reusable code library
  - __init__.py script initializes the namespace when the module is loaded
    - set variables
    - define functions
    - import entities from other scripts in the directory
    - check for presence of dependencies (requisite version, etc.)
  - the module can, in turn, have subdirectories that define additional modules within its own namespace: e.g. "xml" is a module; "xml.dom," "xml.parsers," "xml.sax," "xml.etree" are all modules defined within the "xml" module

What is a module?

- A Python module is a directory containing Python scripts
- An installable package for a module contains:
  - The module directory with all scripts and subdirectories
  - A setup.py script to drive the installation process
    - Name of the module
    - Version of the module
    - Other modules required by this module
      - OPTIONAL: Minimum/maximum versions of those dependencies
  - The setup.py script is used for all aspects of the distribution process:
    - Building distributable packages (source, binary)
    - Installing/updating the package

How is a module located?

- Similar to how Unix finds the executable for a command
  - A series of directories are searched for a file with:
    - the specified name
    - accessible or exec by the current user (e.g. "x" bit set)
  - First one found matching the criterion is executed
- The user can influence this behavior with the PATH environment variable

```
$ ls
bin sw workgroup

$ which ls
/usr/bin/ls

$ export PATH="/home/1001/bin:$PATH"

$ ls
Bet you thought you'd see files, eh?

$ which ls
/home/1001/bin/ls
```

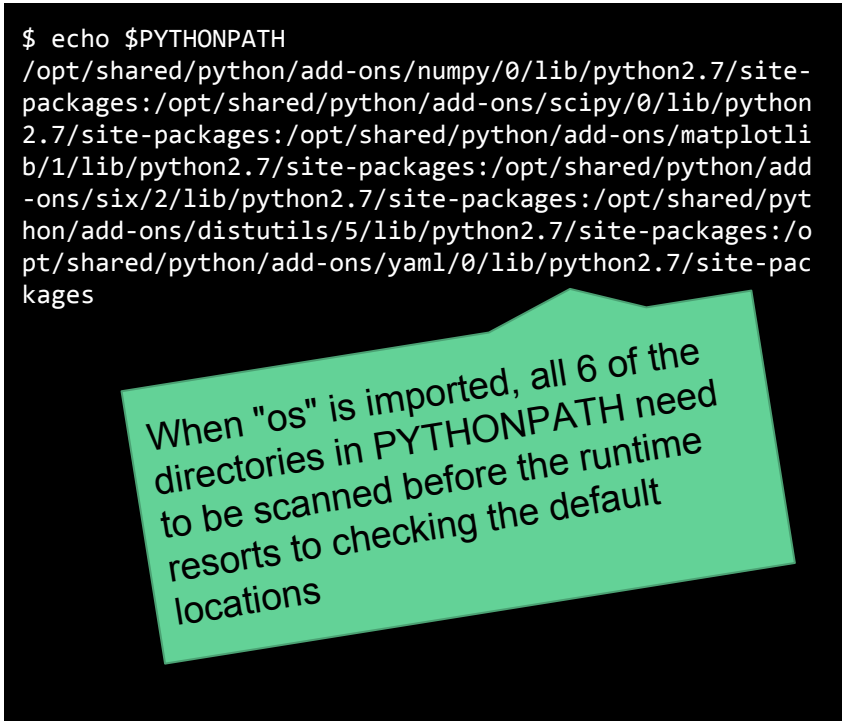
How is a module located?

- Python has the PYTHONPATH environment variable to serve the same purpose
  - The directories are searched in order for a directory with the module name
  - If no match found, the default locations are searched
    - e.g. /usr/lib64/python2.7
    - typically where modules like "os" or "sys" will be found

```
import os
import sys
import re

import my_cool_module
```

How is a module located?

- Python has the PYTHONPATH environment variable to serve the same purpose
- One way to add modules to Python: install each separately and add to PYTHONPATH
  - This is OK, but for a large number of modules the PYTHONPATH will grow toward the inherent length limit and could slow down module import in general



How is a module located?

- Primary problem with installing modules individually is dependencies
  - "I need the pandas module"
  - No, you need:
    - "pandas"
    - "pyyaml"
    - "numpy"
    - "scipy"
    - "sqlite"
    - "blas"
    - et al.

```
$ ... install pandas ...

The following NEW packages will be INSTALLED:

    blas:            1.0-mkl
    ca-certificates: 2018.03.07-0
    certifi:         2018.8.24-py37_1
    intel-openmp:    2019.0-118
    libedit:         3.1.20170329-h6b74fdf_2
    libffi:          3.2.1-hd88cf55_4
    libgcc-ng:       8.2.0-hdf63c60_1
    libgfortran-ng:  7.3.0-hdf63c60_0
    libstdcxx-ng:    8.2.0-hdf63c60_1
    mkl:             2019.0-118
    mkl_fft:         1.0.6-py37h7dd41cf_0
    mkl_random:      1.0.1-py37h4414c95_1
    ncurses:         6.1-hf484d3e_0
    numpy:           1.15.2-py37h1d66e8a_1
    numpy-base:      1.15.2-py37h81de0dd_1
    openssl:         1.0.2p-h14c3975_0
       :
```

Solution 1: Store all modules into a common directory

- Only one path to add to PYTHONPATH (thus, one path to be checked)
- The common directory holds all dependencies for your modules, too

```
$ ls my_python_env
drwxr-xr-x  2 frey everyone    bin
drwxr-xr-x  2 frey everyone    lib

$ ls my_python_env/lib
drwxr-xr-x  2 frey everyone    python2.7

$ ls my_python_env/lib/python2.7
drwxr-xr-x  2 frey everyone    site-packages

$ ls my_python_env/lib/python2.7/site-packages
drwxr-xr-x 35 frey everyone    scipy
drwxr-xr-x 35 frey everyone    numpy
drwxr-xr-x 17 frey everyone    matplotlib
drwxr-xr-x 17 frey everyone    pandas
   :
```

Solution 1: Store all modules into a common directory

- Only one path to add to PYTHONPATH (thus, one path to be checked)
- The common directory holds all dependencies for your modules, too
- Caveat: you must download, build, and install each module — and all its dependencies — by hand!

```
$ ls my_python_env
drwxr-xr-x  2 frey everyone    bin
drwxr-xr-x  2 frey everyone    lib

$ ls my_python_env/lib
drwxr-xr-x  2 frey everyone    python2.7

$ ls my_python_env/lib/python2.7
drwxr-xr-x  2 frey everyone    site-packages

$ ls my_python_env/lib/python2.7/site-packages
drwxr-xr-x 35 frey everyone    scipy
drwxr-xr-x 35 frey everyone    numpy
drwxr-xr-x 17 frey everyone    matplotlib
drwxr-xr-x 17 frey everyone    pandas
   :
```

Solution 2: Use PIP and a common directory

- PIP ("PIP Installs Packages") references online repositories of installable Python modules
  - Dependencies can be resolved recursively — and automatically — by PIP
  - Installs into the default locations for modules (e.g. /usr/lib64/python2.7)
    - ...but a --prefix option specifies an alternative directory
    - --ignore-installed forces default modules to be ignored
- https://pypi.org/

```
$ pip install --prefix="$(pwd)/my_python_env" \
> --ignore-installed \
> matplotlib==2.2.3
Collecting matplotlib==2.2.3
 Downloading https://files.pythonhosted.org/packages/a2/c
    100% |########################| 133kB 5.9MB/s
Collecting six>=1.10 (from matplotlib==2.2.3)
     :
Building wheels for collected packages: matplotlib
  Running setup.py bdist_wheel for matplotlib ... done
  Stored in directory: /home/1001/.cache/pip/wheels/f8/9e
Successfully built matplotlib
Installing collected packages: six, python-dateutil, pytz
Successfully installed backports.functools-lru-cache-1.5

$ ls -l my_python_env/lib/python2.7/site-packages
drwxr-xr-x  2 frey everyone    backports
    :
drwxr-xr-x 14 frey everyone    matplotlib
drwxr-xr-x  2 frey everyone    matplotlib-2.2.3.dist-info
    :
```

Solution 2: Use PIP and a common directory

- PIP ("PIP Installs Packages") references online repositories of installable Python modules
- Add the necessary paths to PATH and PYTHONPATH to use the common directory
- I've employed this method in the past for LARGE module collections (e.g. pandas)

```
$ pip install --prefix="$(pwd)/my_python_env" \
> --ignore-installed \
> matplotlib==2.2.3
Collecting matplotlib==2.2.3
 Downloading https://files.pythonhosted.org/packages/a2/c
    100% |########################| 133kB 5.9MB/s
Collecting six>=1.10 (from matplotlib==2.2.3)
     :
Building wheels for collected packages: matplotlib
  Running setup.py bdist_wheel for matplotlib ... done
  Stored in directory: /home/1001/.cache/pip/wheels/f8/9e
Successfully built matplotlib
Installing collected packages: six, python-dateutil, pytz
Successfully installed backports.functools-lru-cache-1.5

$ ls -l my_python_env/lib/python2.7/site-packages
drwxr-xr-x  2 frey everyone    backports
    :
drwxr-xr-x 14 frey everyone    matplotlib
drwxr-xr-x  2 frey everyone    matplotlib-2.2.3.dist-info
    :
```

Side note: other helpful PIP stuff

- You can use PIP to download module packages
- You can use PIP to install packages not present in the online repositories
  - E.g. your own packaged modules, like PyMuTT

```
$ pip download matplotlib==2.2.3
   :
Successfully downloaded matplotlib six python-dateutil

$ ls matplotlib*
matplotlib-2.2.3-cp27-cp27m-manylinux1_x86_64.whl

$ ls PyMuTT*
PyMuTT-1.0.0.tar.gz

$ pip install PyMuTT-1.0.0.tar.gz
Processing ./PyMuTT-1.0.0.tar.gz
Collecting ASE>=3.16.2 (from PyMuTT==1.0.0)
  :
Collecting matplotlib>=2.2.3 (from PyMuTT==1.0.0)
  :
Collecting numpy>=1.15.1 (from PyMuTT==1.0.0)
  :
Successfully built PyMuTT
Installing collected packages: numpy, kiwisolver, six, cy
Successfully installed ASE-3.16.2 Jinja2-2.10 MarkupSafe-
```

So what's the problem with Solution 2?

- PIP knows about Python code and its Python-oriented dependencies
  - Major issues when working with modules that contain compiled components

```
$ pip3 install --prefix="$(pwd)/tf" \
> --ignore-installed \
> tensorflow
Collecting tensorflow
  Downloading
https://files.pythonhosted.org/packages/ce/d5/38cd4543401
    :
Installing collected packages: six, numpy, h5py, keras-ap
Successfully installed absl-py-0.5.0 astor-0.7.1 gast-0.2

$
```

So what's the problem with Solution 2?

- PIP knows about Python code and its Python-oriented dependencies
- E.g. person who packaged-up TensorFlow did so on an Ubuntu system
    - All Python dependencies are satisfied by PIP...
    - ...but the pre-built shared libraries were linked against glibc 2.17...
    - ...so on our CentOS 6 system with glibc 2.12, the compiled component crashes and burns

```
$ PATH="$(pwd)/tf/bin:$PATH" \
> PYTHONPATH="$(pwd)/tf/lib/python3.6/site-packages" \
> python3 test.py
Traceback (most recent call last):
   :
ImportError: /lib64/libc.so.6: version `GLIBC_2.17' not
found (required by
/tmp/tf/lib/python3.6/site-packages/tensorflow/python/_py
wrap_tensorflow_internal.so)

During handling of the above exception, another exception

Traceback (most recent call last):
  :
ImportError: /lib64/libc.so.6: version `GLIBC_2.17' not
found (required by
/tmp/tf/lib/python3.6/site-packages/tensorflow/python/_py
wrap_tensorflow_internal.so)

Failed to load the native TensorFlow runtime.
```

Preface to Solution 3: Game the system

- Every "python" interpreter finds its Python script library by:
  - assume "python" => "/home/1001/myenv/bin/python"
  - check for "lib/pythonX.Y/os.py" at a sequence of paths:
    - "/home/1001/myenv/bin/lib/pythonX.Y/os.py"
    - "/home/1001/myenv/lib/pythonX.Y/os.py"
    - "/home/1001/lib/pythonX.Y/os.py"
    - "/home/lib/pythonX.Y/os.py"
  - if not found there, check PYTHONPATH, compiled-in library path, etc.
    - e.g. "/usr/lib64/pythonX.Y/os.py"
- Someone figured out that any directory setup in this specific way will be treated like a standalone Python installation
- Thus were born Python *virtual environments*

Solution 3: Virtual Environments

- With the "virtualenv" module installed, any Python installation becomes the basis for standalone containers
  - no PYTHONPATH necessary
  - pip automatically installs into the container
  - modules in container *override* those in the base installation...
  - ...but base installation will still be checked for any module NOT in the container

```
$ vpkg_require python/3.6.5
Adding package `python/3.6.5` to your environment

$ virtualenv myenv
Using base prefix '/opt/shared/python/3.6.5'
New python executable in /home/1001/myenv/bin/python3
Also creating executable in /home/1001/myenv/bin/python
Installing setuptools, pip, wheel...done.

$ source myenv/bin/activate

(myenv) $ file myenv/lib/python3.6/os.py
myenv/lib/python3.6/os.py: symbolic link to
`/opt/shared/python/3.6.5/lib/python3.6/os.py'

(myenv) $ du -sk myenv
21203   myenv
3540840 /opt/shared/python/3.6.5
```

Solution 3: Virtual Environments

- Activate virtual environment, then use pip to install modules
  - The virtualenv setup added setuptools-40.4.3 to the container…
  - …and tensorflow wants an older version (hence the uninstall)
  - but this did NOT alter the base Python installation at all

```
(myenv) $ pip install tensorflow
Collecting tensorflow
  :
  Found existing installation: setuptools 40.4.3
    Uninstalling setuptools-40.4.3:
      Successfully uninstalled setuptools-40.4.3
Successfully installed absl-py-0.5.0 astor-0.7.1 gast-0.2

(myenv) $ python3
Python 3.6.5 (default, Jun 13 2018, 10:30:54)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import tensorflow as tf
>>> tf.__version__
'1.11.0'
>>> ^D

(myenv) $ deactivate
$
```

Solution 3: Virtual Environments

- Inherits the same problem as solution 2
  - If the PyPI package was built against libraries not present on my system, pip will happily install it...
  - ...and it will happily crash when I try to use it.
  - This virtual environment was created on Caviness, where glibc 2.17 ***is*** present, so it actually works (versus Farber)

```
(myenv) $ pip install tensorflow
Collecting tensorflow
  :
  Found existing installation: setuptools 40.4.3
    Uninstalling setuptools-40.4.3:
      Successfully uninstalled setuptools-40.4.3
Successfully installed absl-py-0.5.0 astor-0.7.1 gast-0.2

(myenv) $ python3
Python 3.6.5 (default, Jun 13 2018, 10:30:54)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import tensorflow as tf
>>> tf.__version__
'1.11.0'
>>> ^D

(myenv) $ deactivate
$
```

Preface to Solution 4: Fix that problem!

- The virtual environments are a nice way to put together (somewhat) lightweight collections of Python modules
- Address the issue of compiled components
  - Completely change pip/PyPI to track OS or library dependencies for compiled components
    - Not going to happen: pip/PyPI is very good at handling the Python stuff, why mess that up?
  - Create a separate package management infrastructure that DOES!

Solution 4: [Ana]conda

- The *conda* package management framework
  - In the spirit of many operating systems' package management
  - Software to access package metadata, download and install packages, keep track of what's installed
  - Various *distributions* containing the packages and metadata behind that software
- *anaconda* is one such distribution
  - principally targets scientific applications

```
$ vpkg_require anaconda/5.2.0:python3
Adding package `anaconda/5.2.0:python3` to your environme

$ conda create --prefix=$(pwd)/myenv
Solving environment: done

## Package Plan ##

  environment location: /home/1001/myenv


Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use:
# > source activate /home/1001/myenv
#
# To deactivate an active environment, use:
# > source deactivate
#

$
```

Solution 4: [Ana]conda

- Each conda container is a virtual environment
  - pip can still be used to manage pure Python modules
  - conda used to best-manage modules with compiled components

```
$ source activate /home/1001/myenv

(/home/1001/myenv) $ conda search tensorflow
Loading channels: done
# Name                  Version           Build  Channel
tensorflow            0.10.0rc0     np111py27_0  pkgs/free
tensorflow            0.10.0rc0     np111py34_0  pkgs/free
tensorflow            0.10.0rc0     np111py35_0  pkgs/free
tensorflow                1.0.1     np112py27_0  pkgs/free
tensorflow                1.0.1     np112py35_0  pkgs/free
  :
tensorflow               1.11.0 gpu_py36h4459f94_0  pkgs/main
tensorflow               1.11.0 gpu_py36h9c9050a_0  pkgs/main
tensorflow               1.11.0 mkl_py27h25e0b76_0  pkgs/main
tensorflow               1.11.0 mkl_py36ha6f0bda_0  pkgs/main


(/home/1001/myenv) $
```

Solution 4: [Ana]conda

- Each conda container is a virtual environment
  - pip can still be used to manage pure Python modules
  - conda used to best-manage modules with compiled components
- In this example, I setup an environment with a GPU variant of TF 1.11.0

```
(/home/1001/myenv) $ conda install tensorflow=1.11.0=gpu_py36h9c9050a_0
Solving environment: done

## Package Plan ##

  environment location: /home/1001/myenv

  added / updated specs:
    - tensorflow==1.11.0=gpu_py36h9c9050a_0
         :
    wheel:               0.32.1-py37_0             --> 0.32.1-py36_0

The following packages will be DOWNGRADED:

    python:              3.7.0-h6e4f718_3          --> 3.6.6-h6e4f718_2

Proceed ([y]/n)? y

Downloading and Extracting Packages
tensorflow-1.11.0    | 3 KB      | ########################################
   :
python-3.6.6         | 28.9 MB   | ########################################
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

Solution 4: [Ana]conda

- What's in that virtual environment?
  - TensorFlow Python code
  - shared libraries needed by this variant of TensorFlow's compiled code
    - INCLUDING CUDA libraries for running on GPU
- Different *build* would have different pieces

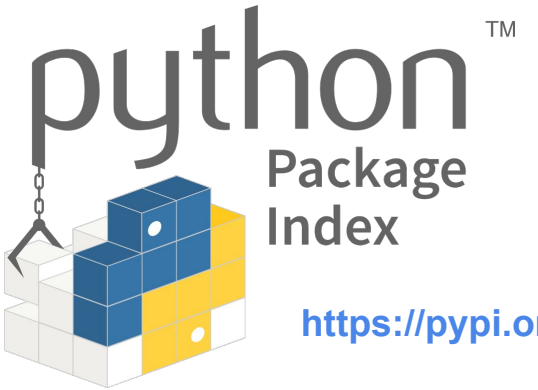
```
(/home/1001/myenv) $ ls -l myenv/lib/lib*cuda*
lrwxrwxrwx 1 frey everyone  myenv/lib/libcudart.so -> libcudart.so.9.2.148
lrwxrwxrwx 1 frey everyone  myenv/lib/libcudart.so.9.2 -> libcudart.so.9.2.148
-rwxrwxr-x 1 frey everyone  myenv/lib/libcudart.so.9.2.148
```

Summary

- Part of the draw of Python is the wealth of code libraries available
- The interdependencies as projects reuse more and more existing code become difficult to manage/satisfy
  - For standard (or simple) Python libraries, the PyPI repositories and pip work well
  - For large, compiled/optimized Python libraries, conda distributions are necessary
- Python "environments" can be a simple directory (PYTHONPATH) or a virtualenv and allow for:
  - isolation of one or more modules from the base Python installation
  - low overhead (no duplication of entire Python installation)
  - easy module maintenance with pip and conda

Questions?

https://www.python.org



https://pypi.org

https://anaconda.org

Appendix 1: Modules import once

1. test.py imports mymod
   a. mymod/__init__.py executed
   b. "mymod" namespace imports os, creates symbol "os" in itself pointing to that namespace
   c. adds a variable to the "os" namespace
2. test.py imports os
   a. namespace already imported
   b. creates symbol "os" pointing to the already-imported namespace
3. ∴ all namespaces' symbol "os" refer to the same namespace

```
$ cat mymod/__init__.py
import os

def add_something():
    os.also_set_by_mymod = 'Still the same'

os.set_by_mymod = 'See, I told you'
print 'inside mymod:  os.set_by_mymod = ' +
            os.set_by_mymod


$ cat test.py
import mymod
import os

print 'in test.py:  os.set_by_mymod = ' +
            os.set_by_mymod
mymod.add_something()
print os.also_set_by_my_mod
```

Appendix 1: Modules import once

- Test
- If test.py had cloned a copy of the "os" namespace augmented by mymod...
  - The add_something() function would not produce an alteration visible to test.py
  - The final print statement in test.py would produce an exception and stack dump

```
$ PYTHONPATH=$(pwd) python test.py
inside mymod:  os.set_by_my_mod = See, I told you
in test.py:  os.set_by_my_mod = See, I told you
Still the same
```

Appendix 2: Copying conda virtual environments

- For all modules installed using conda, export a description of the virtual environment
  - Single YAML file
- That YAML file can be used to recreate the conda environment
  - ...on any machine with Anaconda present
  - Also what gets uploaded to your Anaconda account when publishing environment descriptions

```
$ conda env export --prefix=$(pwd)/myenv \
> --file=myenv.yaml

$ conda env create --prefix=$(pwd)/based_on_myenv \
> --file=myenv.yaml
Using Anaconda API: https://api.anaconda.org
Solving environment: done
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate /Users/frey/env2
#
# To deactivate an active environment, use
#
#     $ conda deactivate

$
```

Appendix 2: Copying conda virtual environments

- Can also make direct copies
  - Clone one environment into a new environment
  - Eliminates the production of the YAML description of the environment

```
$ conda create --clone=$(pwd)/myenv \
> --prefix=$(pwd)/otherenv
Source:      /home/1001/myenv
Destination: /home/1001/otherenv
Packages: 35
Files: 0
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate /home/1001/otherenv
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```