Using and Installing Python

Overview

Different versions of python are available on HPC both as system modules as well as system software on each compute node. Python 2 is available but is no longer supported by the Python Foundation, so we recommend you use Python 3. Python version 3 requires the *python3* command or *pip3 list* to differentiate. It is very different from Python version 2, so do not assume that Python 3 will work for you or that all older modules will work with version 3.

We recommend you use the virtual environment supported by Python, with more details in the sections below. One of the biggest reasons is that you can preserve your compute environment. The packages provided in the system Python are subject to change which may introduce incompatibilities at runtime.

Installation & Package Policy

We maintain a two tiered approach to Python packages

- **Tier 1:** We install the basic Python packages that are required by most users (these are mostly libraries rather than packages, such as numpy and scipy). This is done for the versions of Python that we install as modules. Adding some packages might force an upgrade of numpy for example, which might break a user's environment that was dependent on the prior version.

- **Tier 2:** For packages that we do not provide we STRONGLY recommend the use of *virtualenv,* which is detailed below and provides a custom and easy to use person Python environment.

Available Python Versions

Python 2 is no longer officially supported by the Python Software Foundation.

Four versions of Python are available on HPC. They are only available on compute nodes and are accessible either using a batch submission or interactive session.

Version	Accessibility	Notes
Python 2.7.5	system version (no module)	Accessible as *python*

Python 3.6.8	system version (no module)	Accessible as python3 (unless python module is loaded)
Python 3.6.5	*module load python/3.6/3.6.5*	Includes many packages
Python 3.8.2	*module load python/3.8/3.8.2*	Includes more packages

---

Installing Python Packages Using virtualenv

Useful overview of virtualenv and venv: InfoWorld Article: Python virtualenv and venv do's and don'ts

One of the best things about Python is the number of packages provided by the user community. On a personal machine, the most popular method today for managing these packages is the use of a package manager, like pip. Unfortunately, these require root access and are not a viable solution on the clusters.

There is an easy solution, however. You can use virtualenv to create a personal python environment that will persist each time you log in. There is no risk of packages being updated under you for another user.

To find packages you might want to start with python.org.

In the following instructions any module commands have to be run from an interactive session on a compute node

Virtual Environment Instructions

1. Set up your virtual environment in your account. This step is done one time only and will be good for all future uses of your Python environment. You will need to be in an interactive session to follow along.

   Note: In the commands below, **/path/to/virtual/env** is the path to the directory where all of your environment's executables and packages will be saved. For example, if you use the path ~/mypyenv, this will create a directory in your home called mypyenv. Inside will be directories `bin`, `lib`, `lib64`, and `include`.

   Commands
   Python Version < 3.8

   ```
   module load python/<version>
   virtualenv --system-site-packages /path/to/virtual/env
   ```

   Python Version 3.8

   ```
   module load python/<version>
   python3 -m venv --system-site-packages /path/to/virtual/env
   ```

2. To use your new environment, you'll need to activate it. Inside your virtual environment, there's a directory called `bin` that has a file called `activate`. Sourcing this will add all of the paths needed to your working environment. To activate, run the following, replacing /path/to/virtual/env with the path specific to your account:

   ```
   source /path/to/virtual/env/bin/activate
   ```

3. Once your environment is active, you can pip install your package. For example:

   ```
   pip install pycurl
   ```

4. If you would like your virtual environment to always be active, you can add the activate command to your ~/.bashrc. This is a hidden file in your home directory that sets up your environment each time you log in. To edit it, open the file using your favorite text editor. Then, add the following to a blank line:

```
module load python/<version>
source /path/to/virtual/env/bin/activate
```

Using and Installing Python Packages with Conda

Users have access to `conda` to install packages locally in their account. For a cheat sheet on `conda` commands, see: https://docs.conda.io/projects/conda/en/latest/user-guide/cheatsheet.html

Example for setting up a local conda environment:

```
module load anaconda/2020
conda init bash                    # only needs to be run one time in your account
source ~/.bashrc                   # Makes the init changes live. Only needs to be run after the one-time
initialization
conda create --name py37 python=3.7 # Build a local environment with a specific version of python
conda activate py37                # activate your environment.
```

Once your environment is activated, you will be able to download and use custom packages with `conda`.

It should be noted that the `conda init bash` step will modify your ~/.bashrc file so that Anaconda is automatically activated every time you log in. This behavior is known to cause some issues when using HPC resources such as OOD Desktop sessions (For information, see: FAQ -- resolving Anaconda issues).

One way to more effectively control your environment is to turn off conda's auto-activation feature. This can be done by running the command:

```
conda config --set auto_activate_base false
```

This will prevent Anaconda from being loaded into your environment until you manually activate it using:

```
conda activate
```

If you have turned off auto-activation, you can still use Anaconda in a batch script using:

```
source ~/.bashrc && conda activate
```

Jupyter Notebooks on OOD

Prior to maintenance on 3/27/2022, OnDemand Jupyter Notebooks used Python 3.6.5. Because Python 3.6 has reached end of life, Jupyter now uses Python 3.8.2.

HPC provides access to Jupyter notebooks on all three clusters through our Open OnDemand interface. For more information on using this service, see our page on Open On Demand.

Custom Kernels

To use locally-installed packages in your Jupyter session, you can create a virtual environment and install your own kernel.

The default version of Python available in Jupyter is 3.8.2. If you would like to create a virtual environment using a standard python module, you will need to use the default version that Jupyter uses. If you want to use your own version of python, you can use an Anaconda environment. Steps for both options are provided below:

Using a Python Module

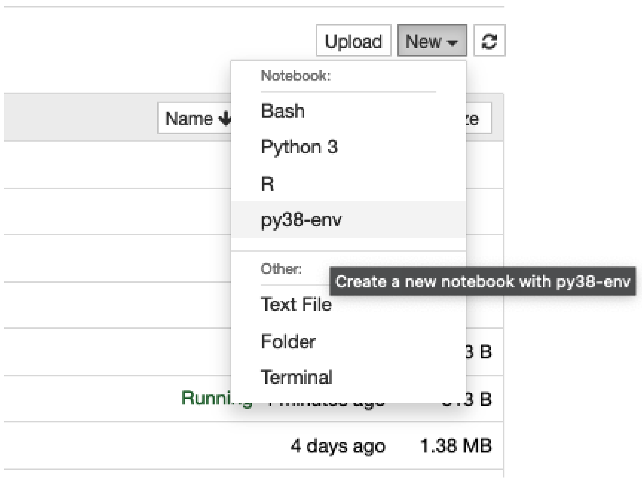
In a terminal session, start by logging into an interactive session since modules are not available on the login nodes. Once you're ready to go, load the python version 3.8.2 and create your virtual environment:

```
module load python/3.8/3.8.2                      # Customize these commands to fit your needs
python3 -m venv --system-site-packages ~/py38-env # See the section on virtual environments above for more info
source ~/py38-env/bin/activate                    # activate your environment
```

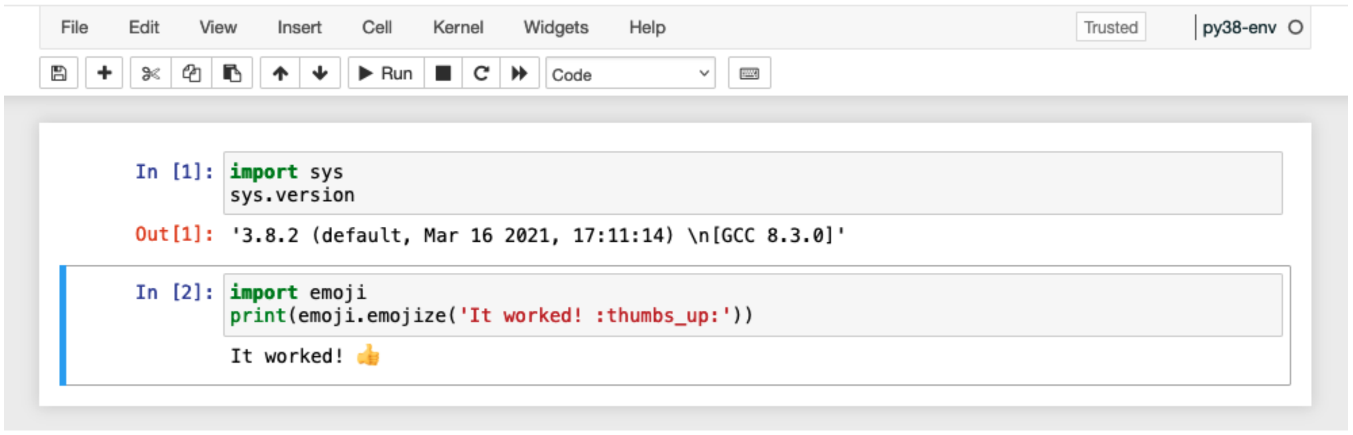
Once your environment is ready to go, pip-install `jupyter` and create your own custom kernel. The `--force-reinstall` flag will allow you to install the jupyter package in your local environment and will not affect the system version. This will create a directory in `~/.local/share/jupyter/kernels/` in your account:

```
pip install jupyter --force-reinstall
ipython kernel install --name py38-env --user
pip install emoji # example python package for demonstration purposes
```

Now, go to https://ood.hpc.arizona.edu/ and start a Jupyter notebook. Once the session starts, open it and click the "new" dropdown menu in the upper right. If everything is working correctly, you should see your custom name:



Once you've selected your environment, try loading a custom module to check that everything is working as expected:



```
In [1]: import sys
        sys.version
Out[1]: '3.8.2 (default, Mar 16 2021, 17:11:14) \n[GCC 8.3.0]'

In [2]: import emoji
        print(emoji.emojize('It worked! :thumbs_up:'))

        It worked! 👍
```

Using Anaconda

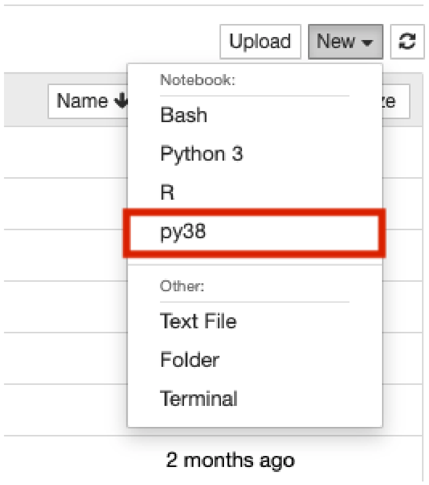
In a terminal session, start by logging into an interactive session, then load your preferred anaconda module, initialize and activate it in your account, and create your local environment:

```
module load anaconda/<version>         # replace <version> with the version you'd like to use
conda init bash                        # only needed if using conda for the first time
source ~/.bashrc && conda activate     # make conda initialization live
conda create --name py38 python=3.8    # create a local environment with your preferred python version. Name
and version are customizable.
conda activate py38                    # activate your new local environment
```

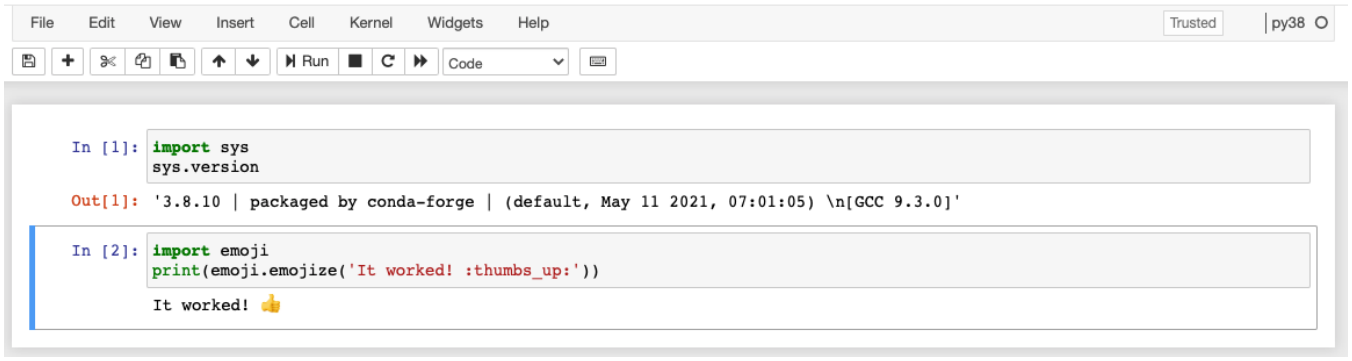
Next, you'll pip install `jupyter` and use that to create your own custom kernel. This will create a file saved to `~/.local/share/jupyter/kernels/` in your account:

```
pip install jupyter
ipython kernel install --name py38 --user
pip install emoji # example python package for demonstration purposes
```

Next, go to https://ood.hpc.arizona.edu/ and start a Jupyter notebook. Once the session starts, open it and click the "new" dropdown menu in the upper right. If everything is working correctly, you should see your custom name:



Once you've selected your environment, try checking the python version in your notebook using the `sys` module. Additionally, for demonstration purposes, we'll check that the package we installed can be imported and is working.



```
In [1]: import sys
        sys.version
Out[1]: '3.8.10 | packaged by conda-forge | (default, May 11 2021, 07:01:05) \n[GCC 9.3.0]'

In [2]: import emoji
        print(emoji.emojize('It worked! :thumbs_up:'))

        It worked! 👍
```