Using Conda and Python

Updated 12/21/2022: This documentation has been revised to include the new update schedule for the NCAR Python Library as well as information for determining when to use managed environments vs. when to create your own environments.

NCAR system users access Python via Conda environments, which are self-contained installations of Python itself, Python packages, and the software dependencies those packages rely on. The Conda environment manager is accessible by loading an environment module as described below.

After loading the module, you can create your own environments or use NCAR Python Library (NPL) environments, which include Python 3.8 and a wide array of pre-installed geoscience, math, and computing packages.

Page contents

Python and Conda

Python is one of the most popular languages for scientific data analysis, visualization, and machine learning. Not only is it well-suited for such applications, one of its main strengths is the availability of supported and well-made third-party libraries for many different applications. These libraries, which include a collection of Python modules and software, can be installed into a Python environment.

While various methods and tools are available for managing Python environments, CISL recommends using Conda to manage them on NCAR systems. If Python packages are not managed carefully, the environments may suffer from incompatibilities between chosen packages and even cause problems for other applications. One of the key concerns is dependency management.

Many Python users are already familiar with Conda, a sophisticated package manager that eases the burden of setting Python up to do scientific analysis. While it is not difficult to install Conda yourself, we provide a Conda environment module on all of our systems to let you skip that step.

The version of Conda that the module provides is updated frequently and also includes Mamba. (Mamba is a re-implementation of Conda with a similar interface and it is typically much faster and more robust at creating environments from complex sets of requirements). Once loaded, the Conda module:

- provides access to **conda** and **mamba** commands.
- allows you to run **conda activate** without initializing Conda in your startup files.
- adds managed environments to your list of available Python environments.
- sets sensible default behaviors for Conda, such as making **conda-forge** the default package channel, locating your cached package downloads in your scratch space, and locating your personal Python environments in your work space.

If you already have a personal Miniconda installation initialized in your environment, the Conda module will take precedence only when it is loaded. To use your own install instead, simply unload the module. You can also override or augment our Conda configuration settings by defining your own in a **~/.condarc** file.

For more information on using Conda, consider exploring the official documentation. Additionally, the Conda Cheat Sheet is a helpful quick reference that covers the commands most users will need.

Accessing Python via Conda

Once you have the Conda module loaded, you can start using Python a number of ways. It is easy and often convenient to activate one of the environments CISL provides, such as the NCAR Python Library (NPL). However, you may wish to craft an environment that contains only the packages you care about, or use specific versions of different packages. Conda makes this process relatively straightforward. Both approaches are detailed in the following sections.

Using managed environments

You can access a number of pre-installed Python environments by loading the Conda module. To see available environments, run the **list** subcommand:

```
conda env list
```

The output will include both managed environments and any personal environments you have created, provided they are in Conda's search path.

Our primary managed environment is the NPL, which contains a large collection of packages related to geoscience and data processing. (It does not currently contain GPU or machine learning packages.) The NPL can be accessed as follows:

```
conda activate npl
```

Once the environment is activated, the **python** command in your shell will use the version provided by the NPL. Currently, it is version Python 3.8, which is supported by the majority of relevant packages. Once package compatibility improves in newer versions, the NPL will be updated.

To see the full list of packages in a managed environment, run the following command:

```
conda list --name npl-2022b
```

Update schedule for the NCAR Python Library

The NPL is updated twice a year and identified with environment names such as **npl-2022b** and **npl-2023a**. New NPL versions contain the same collection of packages as the previous environments (plus any requested and approved additions), but we let the Mamba resolver find and use the latest compatible versions of each package.

For convenience, we also provide an environment named "npl" that always points to the most recent version of the NPL. We recommend that you load a specific version instead if you want to ensure consistent behavior from each installed package.

Creating your own Conda environment

There are many reasons you might prefer to install your own Conda environment rather than use one of the managed environments mentioned above. These may include:

- Preferring a small collection of relevant packages
- Existing workflows that create environments
- Faster or irregular package update cadence
- Desiring your own specified matrix of Python and package versions
- Maximizing stability in a package development environment

To install your own environment using Conda:

1. Load our Conda module (or use your own Miniconda install)
2. Use **mamba create** to create an environment from a set of requirements, as in the following example:

```
mamba create -n my-env python=3.9 numpy scipy matplotlib pandas geocat-comp wrf-python
```

That command will create a new environment called "my-env" with a recent version of Python 3.9.x. (you can choose a different Python version than the NPL when creating your own environments) and a small number of useful geoscience packages (plus any dependencies that Mamba determines are needed). The environment would be located in **/glade/work/$USER/conda-envs/my-env** if you are using the CISL Conda module.

> In general, any Conda command can be executed with Mamba instead. In practice, it is only beneficial to use Mamba when creating environments and installing packages. The Mamba developers currently *advise against using it* to activate environments.

Using Conda in batch jobs

The Conda module enables Python environment activation by setting a shell alias that calls initialization scripts. However, the alias does not carry over into batch jobs or other subshells. You can address this one of two ways:

1. Explicitly load the Conda module at the start of your batch job. This will restore the alias in the shell environment and allow you to activate Conda environments.
2. Run **conda init** (or **conda init tcsh** for tcsh users) after loading the module on a login node. That will add initialization commands to your **~/.bashrc** or **~/.tcshrc** file. Be aware that **~/.bashrc** is not always sourced at the start of batch jobs, so this approach may require some trial and error to cover your individual use case.

Reproducing Conda environments

Cloning an environment

It is possible to clone any environment in your *conda env list* using Mamba. For example, to clone the personal environment we created above, use the following command:

```
mamba create -n my-clone --clone my-env
```

Assuming the clone is created on the same file system (e.g., your conda-envs), it will use very little space as Mamba will attempt to use hard links to avoid duplication.

Using YAML files

Sometimes it is useful to create an environment from the environment file of another environment. This allows Mamba to resolve all package requirements at the same time, which can produce more coherent environments and avoid situations in which the dependency resolver fails to find a solution. If the cloning method described above fails or seems to be taking too long, try this approach instead. Other benefits include having a hard copy of the specifications of an environment that you can share with colleagues, install on other systems, and version track with software like Git.

First, use Conda to produce the YAML environment file from an existing environment. This example uses the NPL:

```
conda env export [--from-history] -n npl > npl-environment.yml
```

If you use the **--from-history** option, only packages that you explicitly request will be specified in the file, letting Conda choose newer dependencies for future environments. If you *do not* specify that option, all packages including dependencies will be documented explicitly in the YAML file. After creating the YAML file, you can use a text editor to make additions and changes to the package list; package additions and changes will be reflected in any new environments you create from the modified YAML file.

You can then create new environments from the YAML file using Mamba:

```
mamba env create -f npl-environment.yml -n my-npl
```

Large environments like the NPL can take a long time to clone or recreate using YAML files.

Using Conda environments in Jupyter

Managed environments can be accessed easily in a JupyterLab session using the NCAR JupyterHub service. Once you initiate a JupyterHub server, the managed kernels mentioned above (e.g., **NPL 2022b**) should be visible on the main launcher page. You can use them to run both Notebooks and Consoles.

Creating Jupyter kernels for your environments

There are two ways to make your environments accessible in a Jupyter interface like NCAR's JupyterHub. Both involve creating a kernel for the environment.

First, install the **ipykernel** package into your environment:

```
conda activate my-env
mamba install ipykernel
```

At this point, your environment should show up automatically as a kernel in any Jupyter server on Cheyenne or Casper. This method is convenient and robust, and any shell settings (e.g., environment variables) needed by packages will be set upon kernel launch.

You can also manually create a JSON-format kernel specification file in your home directory and give it a custom name using the **ipykernel** module. This approach allows you to share environments with colleagues, as they can simply activate your Conda environment on the command line and then create their own personal Jupyter kernel. With your environment activated, run the following:

```
python -m ipykernel install --user --name=my-kernel
```

Kernels created using this approach will not have the previously mentioned shell settings imposed by **conda activate**, which may result in some packages (e.g., pyproj) not functioning properly.