‹ Introduction to Conda for (Data) Scientists (/introduction-to-conda-for-data-scientists/) (/introduction-to-conda-for-data-scientists/03-using-packages-and-channels/index.html) (/intro to- conda- for- data- scienti manag cuda- depen ›

Sharing Environments

Overview

Teaching: 30 min
Exercises: 15 min
Questions

- Why should I share my Conda environment with others?
- How do I share my Conda environment with others?
- How do I create a custom kernel for my Conda environments inside JupyterLab?

Objectives

- Create an environment from a YAML file that can be read by Windows, Mac OS, or Linux.
- Create an environment based on exact package versions.
- Create a custom kernel for a Conda environment for use inside JupyterLab and Jupyter notebooks.

Working with environment files

When working on a collaborative research project it is often the case that your operating system might differ from the operating systems used by your collaborators. Similarly, the operating system used on a remote cluster to which you have access will likely differ from the operating system that you use on your local machine. In these cases it is useful to create an operating system agnostic environment file which you can share with collaborators or use to re-create an environment on a remote cluster.

Creating an environment file

In order to make sure that your environment is truly shareable, you need to make sure that that the contents of your environment are described in such a way that the resulting environment file can be used to re-create your environment on Linux, Mac OS, and Windows. Conda uses YAML ("YAML Ain't Markup Language") for writing its environment files. YAML is a human-readable data-

serialization language that is commonly used for configuration files and that uses Python-style indentation to indicate nesting.

Creating your project's Conda environment from a single environment file is a Conda "best practice". Not only do you have a file to share with collaborators but you also have a file that can be placed under version control which further enhancing the reproducibility of your research project and workflow.

📌 Default `environment.yml` file

Note that by convention Conda environment files are called `environment.yml`. As such if you use the `conda env create` sub-command without passing the `--file` option, then `conda` will expect to find a file called `environment.yml` in the current working directory and will throw an error if a file with that name can not be found.

Let's take a look at a few example `environment.yml` files to give you an idea of how to write your own environment files.

Code

```
name: machine-learning-env

dependencies:
  - ipython
  - matplotlib
  - pandas
  - pip
  - python
  - scikit-learn
```

This `environment.yml` file would create an environment called `machine-learning-env` with the most current and mutually compatible versions of the listed packages (including all required dependencies). The newly created environment would be installed inside the `~/miniconda3/envs/` directory, unless we specified a different path using `--prefix`.

Since explicit versions numbers for all packages should be preferred a better environment file would be the following.

Code

```
name: machine-learning-env

dependencies:
  - ipython=7.13
  - matplotlib=3.1
  - pandas=1.0
  - pip=20.0
  - python=3.6
  - scikit-learn=0.22
```

Note that we are only specifying the major and minor version numbers and not the patch or build numbers. Defining the version number by fixing only the major and minor version numbers while allowing the patch version number to vary allows us to use our environment file to update our environment to get any bug fixes whilst still maintaining significant consistency of our Conda environment across updates.

📌 *Always* version control your `environment.yml` files!

While you should *never* version control the contents of your `env/` environment sub-directory, you should *always* version control your `environment.yml` files. Version controlling your `environment.yml` files together with your project's source code means that you always know which versions of which packages were used to generate your results at any particular point in time.

Let's suppose that you want to use the `environment.yml` file defined above to create a Conda environment in a sub-directory of some project directory. Here is how you would accomplish this task.

Code

```
$ cd ~/Desktop/introduction-to-conda-for-data-scientists
$ mkdir project-dir
$ cd project-dir
```

Once your project folder is created, create `environment.yml` using your favourite editor, for instance `nano`. Finally create a new conda environment:

If you teach this lesson, please tell the authors and provide feedback by opening an issue in the source repository (https://github.com/carpentries-incubator/introduction-to-conda-for-data-scientists/issues)

Bash

```
$ conda env create --prefix ./env --file environment.yml
$ conda activate ./env
```

Note that the above sequence of commands assumes that the `environment.yml` file is stored within your `project-dir` directory.

Automatically generate an `environment.yml`

To export the packages installed into the previously created `machine-learning-env` you can run the following command:

Bash

```
$ conda env export --name machine-learning-env
```

When you run this command, you will see the resulting YAML formatted representation of your Conda environment streamed to the terminal. Recall that we only listed five packages when we originally created `machine-learning-env` yet from the output of the `conda env export` command we see that these five packages result in an environment with roughly 80 dependencies!

To export this list into an environment.yml file, you can use `--file` option to directly save the resulting YAML environment into a file.

Bash

```
$ conda env export --name machine-learning-env --file environment.yml
```

Make sure you do not have any other `environment.yml` file from before in the same directory when running the above command.

This exported environment file will however not *consistently* produce environments that are reproducible across Mac OS, Windows, and Linux. The reason is, that it may include operating system specific low-level packages which cannot be used by other operating systems.

If you need an environment file that can produce environments that are reproducibile across Mac OS, Windows, and Linux, then you are better off just including those packages into the environment file that your have specifically installed.

Bash

```
$ conda env export --name machine-learning-env --from-history --file environment.yml
```

In short: to make sure others can reproduce your environment independent of the operating system they use, make sure to add the `--from-history` argument to the `conda env export` command.

Create a new environment from a YAML file.

Create a new project directory and then create a new `environment.yml` file inside your project directory with the following contents.

Code

```
name: scikit-learn-env

dependencies:
  - ipython=7.13
  - matplotlib=3.1
  - pandas=1.0
  - pip=20.0
  - python=3.6
  - scikit-learn=0.22
```

Now use this file to create a new Conda environment. Where is this new environment created? Using the same `environment.yml` file create a Conda environment as a sub-directory called `env/` inside a newly created project directory. Compare the contents of the two environments.

Solution To create a new environment from a YAML file use the `conda env create` sub-command as follows.

Bash

```
$ mkdir scikit-learn-project-dir
$ cd scikit-learn-project-dir
$ nano environment.yml
$ conda env create --file environment.yml
```

The above sequence of commands will create a new Conda environment inside the `~/miniconda3/envs` directory. In order to create the Conda environment inside a sub-directory of the project directory you need to pass the `--prefix` to the `conda env create` command as follows.

Bash

```
$ conda env create --file environment.yml --prefix ./env
```

You can now run the `conda env list` command and see that these two environments have been created in different locations but contain the same packages.

Specifying channels in the environment.yml

We learned in the previous episode, that some packages may need to be installed from other than the defaults channel. We can also specify the channels, that conda should look for the packages within the environment.yml file:

Code

```
name: pytorch-env

channels:
  - pytorch
  - defaults

dependencies:
  - pytorch=1.1
```

When the above file is used to create an environment, conda would first look in the `pytorch` channel for all packages mentioned under `dependencies`. If they exist in the `pytorch` channel, conda would install them from there, and not look for them in `defaults` at all.

Updating an environment

You are unlikely to know ahead of time which packages (and version numbers!) you will need to use for your research project. For example it may be the case that

- one of your core dependencies just released a new version (dependency version number update).
- you need an additional package for data analysis (add a new dependency).
- you have found a better visualization package and no longer need to old visualization package (add new dependency and remove old dependency).

If any of these occurs during the course of your research project, all you need to do is update the contents of your `environment.yml` file accordingly and then run the following command.

Bash

```
$ conda env update --prefix ./env --file environment.yml  --prune
```

Note that the `--prune` option tells Conda to remove any dependencies that are no longer required from the environment.

Rebuilding a Conda environment from scratch

When working with `environment.yml` files it is often just as easy to rebuild the Conda environment from scratch whenever you need to add or remove dependencies. To rebuild a Conda environment from scratch you can pass the `--force` option to the `conda env create` command which will remove any existing environment directory before rebuilding it using the provided environment file.

Bash

```
$ conda env create --prefix ./env --file environment.yml --force
```

Add Dask to the environment to scale up your analytics

Add to the `scikit-env` environment file and update the environment. Dask (https://dask.org/) provides advanced parallelism for data science workflows enabling performance at scale for the core Python data science tools such as Numpy Pandas, and Scikit-Learn.

Solution The `environment.yml` file should now look as follows.

Code

```
name: scikit-learn-env

dependencies:
  - dask=2.16
  - dask-ml=1.4
  - ipython=7.13
  - matplotlib=3.1
  - pandas=1.0
  - pip=20.0
  - python=3.6
  - scikit-learn=0.22
```

You could use the following command, that will rebuild the environment from scratch with the new Dask dependencies:

Bash

```
$ conda env create --prefix ./env --file environment.yml --force
```

Or, if you just want to update the environment in-place with the new Dask dependencies, you can use:

Bash

```
$ conda env update --prefix ./env --file environment.yml  --prune
```

📌 Installing via `pip` in `environment.yml` files

Since you write `environment.yml` files for all of your projects, you might be wondering how to specify that packages should be installed using `pip` in the `environment.yml` file. Here is an example `environment.yml` file that uses `pip` to install the `kaggle` and `yellowbrick` packages.

Code

```
name: example

dependencies:
 - jupyterlab=1.0
 - matplotlib=3.1
 - pandas=0.24
 - scikit-learn=0.21
 - pip=19.1
 - pip:
   - kaggle==1.5
   - yellowbrick==0.9
```

Note the double '==' instead of '=' for the pip installation and that you should include `pip` itself as a dependency and then a subsection denoting those packages to be installed via `pip` . Also in case you are wondering, The Yellowbrick (https://www.scikit-yb.org/en/latest/) package is a suite of visual diagnostic tools called "Visualizers" that extend the Scikit-Learn (https://scikit-learn.org/stable/) API to allow human steering of the model selection process. Recent version of yellowbrick can also be installed using `conda` from the `conda-forge` channel.

Code

```
$ conda install --channel conda-forge yellowbrick=1.2 --prefix ./env
```

An alternative way of installing dependencies via `pip` in your environment files is to store all the packages that you wish to install via `pip` in a `requirements.txt` file and then add the following to your `environment.yml file` .

Code

```
...
  - pip
  - pip:
    - -r file:requirements.txt
```

Conda will then install your `pip` dependencies using `python -m pip install -r requirements.txt` (after creating the Conda environment and installing all Conda installable dependencies).

Making Jupyter aware of your Conda environments

Both JupyterLab and Jupyter Notebooks automatically ensure that the standard IPython kernel is always available by default. However, if you want to use a kernel based on a particular Conda environment from inside Jupyter (and Juptyer is *not* installed inside your environment) then will need to create a kernel spec (https://jupyter-client.readthedocs.io/en/latest/kernels.html#kernelspecs) file for your Conda environments manually.

Before you can create a custom kernel for you Conda environment you need to make sure that the `ipykernel` (https://pypi.org/project/ipykernel/) package is installed in your Conda environment as you will need to use this package to create the kernel spec file. Here is the updated `xgboost-env environment.yml` file that includes the `ipykernel` package.

Code

```
name: ml-env

dependencies:
  - ipykernel=5.3
  - ipython=7.13
  - matplotlib=3.1
  - pandas=1.0
  - pip=20.0
  - python=3.6
  - scikit-learn=0.22
```

Next, rebuild the Conda environment using the following command.

Code

```
$ conda env create --prefix ./env --file environment.yml --force
```

Once the Conda environment has been re-built you can activate the environment and then create the custom kernel for the activated environment.

Code

```
$ conda activate ./env
$ python -m ipykernel install --user --name xgboost-env --display-name "XGBoost"
```

The last command installs a kernel spec file for the current environment. Kernel spec files are JSON files which can be viewed and changed with a normal text editor. The `--name` value is used by Jupyter internally; `--display-name` is what you see in the JupyterLab launcher menu as well as the Jupyter Notebook dropdown kernel menu. This command will overwrite any existing kernel with the same name.

Create a kernel for a Conda environment

Create a custom kernel for the `machine-learning-env` environment created in a previous challenge.

Solution In order to activate an existing environment by name you use the `conda activate` command as follows.

Bash

```
$ conda activate machine-learning-env
$ python -m ipykernel install --user --name machine-learning-env
```

Note that by leaving the `--display-name` unspecified, the display name will match the value provided to `--name` .

Key Points

- Sharing Conda environments with other researchers facilitates the reprodicibility of your research.
- Create an `environment.yml` file that describes your project's software environment.
- Creating custom kernels enables you to connect your Conda environments to an existing JupterLab install.

‹ (/introduction-to-conda-for-data-scientists/03-using-

(/intro to-conda- for- data- scienti manag cuda- depen ›

packages-and-channels/index.html)