Development environment setup

Software engineering for scientists

Introduction (PLEASE READ)

People use the term ***development environment*** as a vague catchall to describe the set of tools, libraries, software, etc. that they use when writing code; it's the environment in which they develop code. For us, since we'll mostly be using the command line, shell scripting, Python, and git, every student will need access to a Unix-like command line and a text editor for writing code. There are many different text editors and people have their preferences. In my experience, most Python (and other language) software developers use PyCharm or Visual Studio Code. If you don't already have a preference or you do and it's not Visual Studio Code (VS Code), I **HIGHLY** suggest you use VS Code. It has many nice features some of which we'll cover in this class. AND VS Code is included in CU's reference environment (more about this below).

It can be tedious and time-consuming to get one's development environment set up. CU has a handy reference environment with many tools and software we'll use already installed. However, for your future sciencing and coding, you'll probably mostly be using your personal computer so it's probably worth spending some time to try to set up your personal computer to have a nice development environment. That's generally less of a headache on Mac and Linux computers than Windows.

For Mac and Linux users, I suggest using your personal computer rather than CU's reference environment.

For Windows users, I suggest spending some time trying to set up your personal computer and, if that's proving too painful, then use CU's reference environment.

Regardless of whether you use your personal computer or CU's reference environment or both, you need to do the following:

1. Make sure you have access to a Unix-like command line
    a. Macs and Linuxs come with a Unix-like command line. Just make sure you know how to access it.
    b. Windows users have a few options one of which is described in the first/top-voted answer in this thread (assumes you're using Visual Studio Code). Another option is here
2. Pick and install a text editor (again, I highly suggest using Visual Studio Code)
3. Make sure you have git installed and can access it via the command line
4. Install mamba and make sure you can create and activate a new environment

As always, please let me know if you're struggling. Although do you know that software installation and setup can be one of the most frustrating parts of coding. So Googling yourself out of some hiccups is good practice. Hopefully this class will help you write software that's NOT hard to install and use!

The rest of these notes describe

- The package manager we will be using (i.e., mamba)
- CU's reference environment and how to set it up (you can skip this if you're not going to use CU's reference environment)
- How to install mamba
- Some basic mamba commands

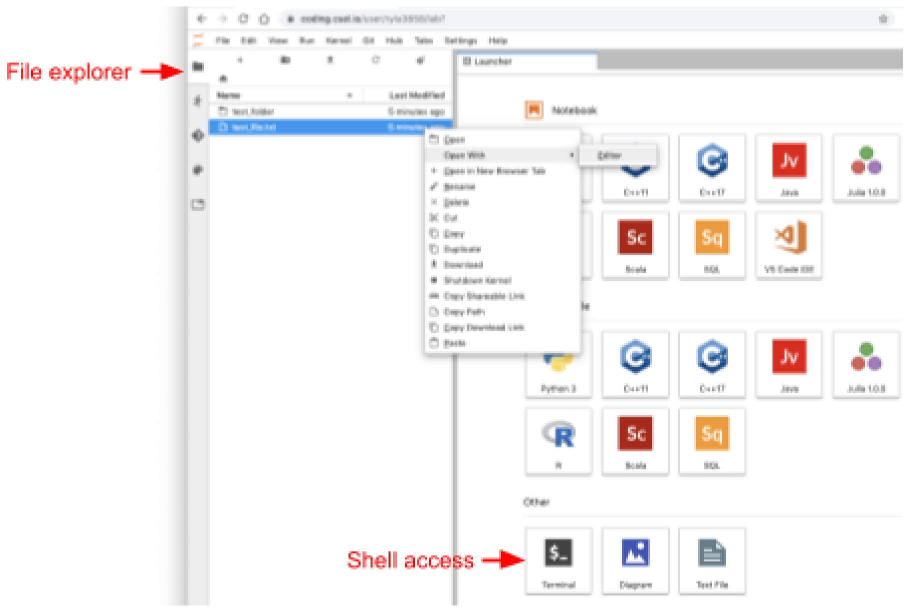
Conda/Mamba and pip

Ensuring that all software dependencies are met is a major obstacle to reproducible scientific software. There are many ways to help communicate these dependencies and simplify the installation process. In this class, we will use conda (https://conda.io/) which is an environment manager and a package manager. More specifically we will use a new, faster implementation of conda called mamba. We'll use mamba mostly for Python dependency and package management but conda/mamba can also be used with other languages such as Java, JavaScript, C/C++, and R

Pip is the basic Python Packaging Authority's recommended tool for installing packages from the Python Package Index, PyPI. You can use pip to install python packages on your computer, without being in a conda environment. Pip installs Python software packaged as wheels (pre-compiled) or source distributions (not pre-compiled). The latter may require that the system have compatible compilers, and possibly libraries, installed before invoking pip to succeed.

CU’s reference environment

The reference environment is the CU CS JupyterHub at https://coding.csel.io/. Choose “Default Coding Environment”. This interface includes a file explorer for creating and editing files and a command-line interface.

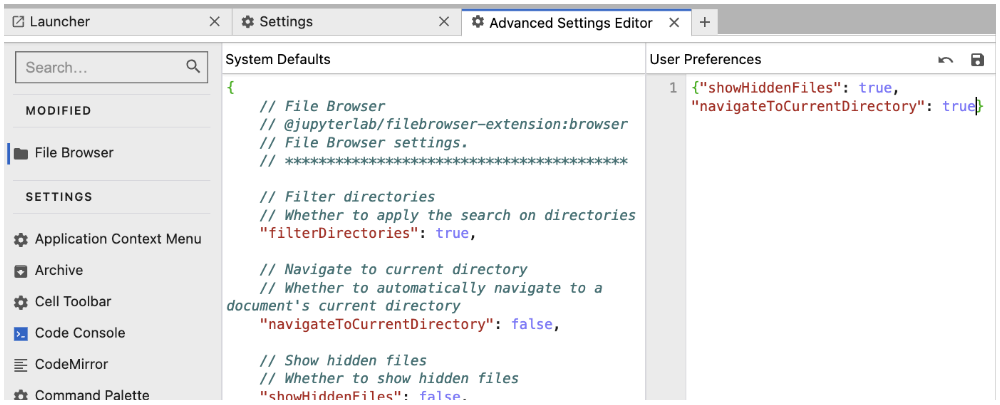


Initialize CU CS JupyterLab setup (only need to do this once!)

Showing hidden files

In a Unix-like (e.g., Linux, MacOS) environment configurations are stored in *hidden* text files that start with a “.” and are not listed by default. This prevents them from cluttering your environment and makes accidental changes less likely, but we will be working with hidden files and need to change the environment so we can see them in the file explorer. To do this:

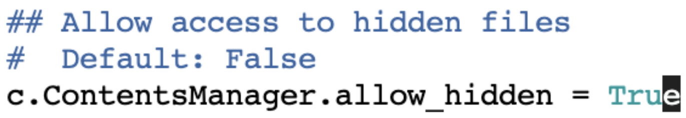
- Go to Settings->Advanced Settings Editor.
- Click "JSON Setting Editor" on top right.
- Click "File Browser" in the left pane. If you don’t see it, you can get it by entering "File Browser" in the search bar.
- Add `"showHiddenFiles": true` between the braces (`{}`) in the User preferences file on the right.
- You can also add `"navigateToCurrentDirectory": false` to that list. While not required I find it helpful. This is a list, so don
- Save by clicking the floppy disk button.



```
{
    // File Browser
    // @jupyterlab/filebrowser-extension:browser
    // File Browser settings.
    // ****************************************

    // Filter directories
    // Whether to apply the search on directories
    "filterDirectories": true,

    // Navigate to current directory
    // Whether to automatically navigate to a
document's current directory
    "navigateToCurrentDirectory": false,

    // Show hidden files
    // Whether to show hidden files
    "showHiddenFiles": false,
```

```
{"showHiddenFiles": true,
"navigateToCurrentDirectory": true}
```

- Launch the terminal and run `$ jupyter server --generate-config`
- The config file will be saved into `~/.jupyter/jupyter_server_config.py` where we need to change "c.ContentsManager.allow_hidden" to True in the file.
    - Open the file with `$ nano .jupyter/jupyter_server_config.py`
    - Jump to line 1041 with Ctrl + _
    - Remove the # and change `False` to `True`



```
## Allow access to hidden files
#  Default: False
c.ContentsManager.allow_hidden = True
```

    - Save and exit with Ctrl X
- Log-out (File->Log Out) and log-in is required to make this change effective

Managing dependencies

Mamba is already installed in this environment, but we need to do a few setup steps. **NOTE: These steps are specific to this environment.** Keep reading for instructions for getting mamba installed in other environments (i.e., your laptop).

```
$ mamba init
...
Added mamba to /home/jovyan/.bashrc

==> For changes to take effect, close and re-open your current shell. <==
```

This command added everything needed to initialize mamba to the `.bashrc` file in your home directory. The issue is that in this environment the `.bashrc` is ignored, which is not normal. To fix this run

```
$ echo ". $HOME/.bashrc" >> .bash_profile
```

Log out, then log back in. If it worked, then you should see `(base)` in the command line prompt.

```
(base) $
```

To ensure that your environments are persistent, we need to change the directory they are stored in.

- Right click in the file explore and create a new file
- Name it `.condarc`
- Open it (now that you can see it)
- Add the lines:

```
envs_dirs:
- /home/jovyan/.conda/envs
```

Installing conda in your local environment

If you don't have conda, running

```
$ conda
```

will give an error.

Anaconda is a python distribution that is meant to make data science easier. Anaconda provided **conda**. Conda became popular independent of anaconda, which led to **miniconda**, which cuts out all of the extra stuff in anaconda. Conda can be very slow, which led to the more efficient **mamba** (https://github.com/mamba-org/mamba). We will use mamba in this class.

Follow the instructions here for your operating system.

For example, on Macs with homebrew installed, you just need to run

```
$ brew install miniforge
```

On Linux, you'd run

```
$ cd $HOME
$ curl -L -O
"https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge3-$(uname)-$
(uname -m).sh"
$ bash Miniforge3-$(uname)-$(uname -m).sh
```

Once conda is installed, please update it and then install mamba

```
$ conda update --yes mamba
```

To activate environments, you'll likely need to run

```
$ mamba init
```

and then close and reopen your command line application. If it worked, then you should see `base)` in the command line prompt:

```
(base) $
```

Using mamba

Creating an environment

```
$ mamba create --yes -n swe4s
# This creates an environment called “swe4s”
#
# To activate the environment, use
#
#     $ conda activate swe4s
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

IMPORTANT – DO NOT INSTALL ANYTHING INTO THE BASE ENVIRONMENT

The mamba documentation in the “Note” section here says NOT to install anything into the default environment which is called “base”. So before installing anything always activate the environment you want to install it into like this:

```
$ mamba activate swe4s

(swe4s) $
# The activated environment’s name should be in parentheses to the left of the command
prompt
```

Installing Python packages

With Conda

```
(swe4s) $ mamba install -y pycodestyle
Looking for: ['pycodestyle']
...
Preparing transaction: done
Verifying transaction: done
```

```
Executing transaction: done
```

With pip

```
(swe4s) $ pip install pycodestyle
Collecting pycodestyle
...
```

If pip is not installed:

```
$ sudo easy_install pip
$ sudo pip install --upgrade pip
```