JupyterLab

P. S. Langeslag

October 31, 2022

What Is It, And Why Do I Need It?

For this course, you will be programming in Python. JupyterLab is a browser-based IDE (integrated development environment) you can use for all your homework. It saves you from having to install Python locally, but it also offers a convenient way for us to swap code in the form of so-called notebooks. If you prefer to run Python on your own machine instead, you can, but you'll have to set up a local Jupyter server as well; and keep in mind that the data in the course repository are not for distribution.

Accessing JupyterLab

Head to https://jupyter-cloud.gwdg.de and log in using your academic credentials. If you aren't returning to an active or saved workspace, JupyterLab should open to a launcher window (see Figure 1). From here, you can technically select Python 3 in the Notebook or Console categories to start coding. However, you'll probably want to create a new directory for each course or project you undertake; and furthermore, you'll want to clone the course's `git` repository, which creates a course directory and copies in all the required files. This step is explained under the next heading.

Cloning the Course Repository

You will access the repository using your GWDG (i.e. student) account, but if you haven't used https://gitlab.gwdg.de before, you'll have to log in at that address first to activate the user account. Use your normal academic credentials, but with your full email address (`user@stud.uni-goettingen.de`), not just the user name, as user. Once you've logged in here, you can close the window and return to JupyterLab. In the launcher window, from the category "Other" select "Terminal." This opens a command-line interface responding to many of the same commands you may know from UNIX-like systems such as Linux, or even macOS. Paste in the following command:

```
git clone https://gitlab.gwdg.de/langeslag-teaching/cmptr.git
```

(or use the SSH address if you have a key set up). Identify yourself using the same credentials as above. This should create a folder `cmptr` and populate it with our course files. The folder will appear in the file system in the left-hand pane. Once it's done cloning, you can close the terminal window (click the `x` or type `exit`) and double-click the `cmptr` folder on the left. Nothing changes except the current folder (or "working directory"). Now if you start Python 3, the files you've pulled in are available within your working directory.

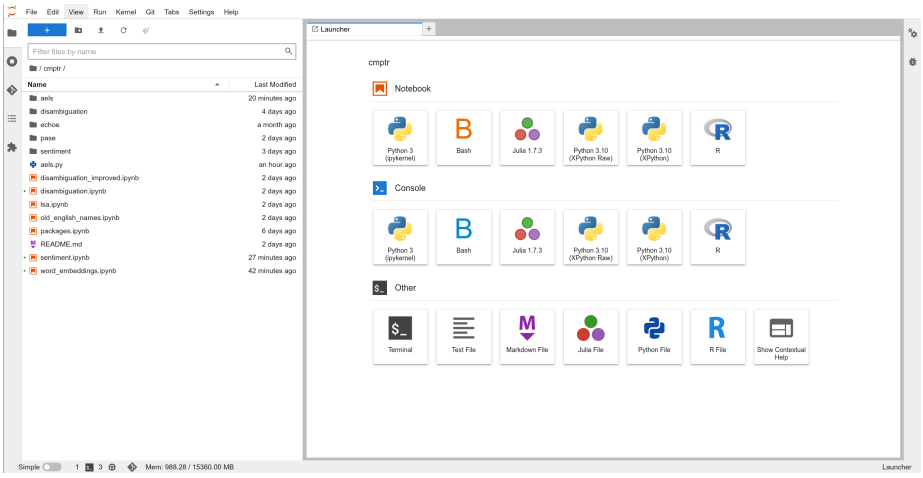

Figure 1: JupyterLab's interface: file system on the left, launcher on the right.

Updating your Working Copy

If the course materials should ever be updated in the course of the term, you'll have to update your working copy to take advantage of these changes. To do so, enter the `cmptr` folder in the file system, launch a terminal window, and enter `git pull`. Once the line `Writing objects` reaches 100%, your working copy is up to date and you can close the window. If you receive a warning that your working copy contains changes not in the master branch, that's where `git` gets more complicated; try this guide.

Notebook or Console?

JupyterLab offers two interfaces: Notebook and Console. You can technically do your homework in either one, but if you want to save or submit any of your work you may want to opt for the notebook. The console offers a traditional interactive interpreter experience; the notebook is often considered more useful in a learning setting because you can save and export your work, you can reorder and rerun your code blocks, and you have formatting options (Markdown, or even LaTeX) for any notes you wish to take. You can also open a console alongside a notebook that is aware of functions and variables set in the notebook (File → New Console for Notebook). Finally, to ease your workload I have done some of your coding for you and made it available in the repository in the form of notebooks.

So all things considered, you'll probably want to do your final work in a notebook but any experimental work and "exploration" in a console spawned within your notebook session. The textbooks we'll use assume a regular interactive interpreter, but their exercises will work just as well in a notebook.

Notebook Modes and Commands

Like the Vim editor popular among programmers, the Jupyter Notebook interface has a command mode and an edit mode. When you open a notebook, it is in command mode and will respond to keyboard commands like the following:

`Enter`	Enter edit mode
`Shift+Enter`	Run cell code
`k` or arrow	Navigate up
`j` or arrow	Navigate down
`a`	Add a cell above
`b`	Add a cell below
`y`	Change cell mode to code
`m`	Change cell mode to Markdown
`dd`	Delete current cell

Once you enter edit mode, you have access to commands like the following:

`Esc`	Enter command mode
`Shift+Enter`	Run cell code and command new cell below

For more keyboard shortcuts, see e.g. here.

Markdown

Markdown is an efficient structured text format which it is well worth learning about, as it is the most convenient and powerful format in which to write nearly all the documents you need in your professional and personal life (except spreadsheets). Fuller guides are all over the web (my own is here), but the short version is that it relies on formatting cues like the following:

```
# Top-level headings receive one hash/pound sign

## Lower-level headings receive two

Paragraphs are ended by double line breaks.

Like so.
```

```
[Links](https://are.rendered.thus) or they may be rendered <https://thus.ly>

```python
Code demonstrations ("listings") are set off within triple backtick boundaries.
```
```

The relevance of Markdown to Jupyter Notebooks is that it is the format for note-taking cells. You can toggle the format of a selected cell from code to Markdown with the `y` and `m` keybindings (press `Esc` first if in edit mode), or using the dropdown menu. A Markdown cell serves to document or annotate your code, or to create logical sections within a Python script.

Package Management

Our textbooks assume that you have access to a Python package manager, usually `pip`, on the command line. JupyterLab in fact incorporates `pip` directly into the interpreter, so to install a Python package like `nltk` you can type `pip install nltk` directly into your notebook or console, or you can take the traditional route and do your package management from within a terminal window.

JupyterLab doesn't automatically save your Python configuration: the next time you log in, your packages will be gone. To install all the required packages before you start a stint of work, you can open a terminal window and run `pip install -r requirements.txt` from within the `cmptr/` folder. Alternatively, you can open the notebook entitled `packages.ipynb` and select "Run All Cells." Both methods take time, though, and are overkill if you just need one or two packages. Instead, whenever you start out work with one or more non-stock packages, just install the ones you need (`pip install nltk gensim`). Likewise, if in your work you are ever prompted by an error along the lines of `ModuleNotFoundError: No module named 'gensim'`, simply install the missing package (`pip install gensim`) and get on with your work. `cltk` has a lot of package dependencies, so `pip install cltk` takes a little longer, but also covers `nltk`, `numpy`, `scipy`, and `scikit-learn` in one go; so simply starting any notebook with `pip install cltk` (ignore the warnings) will often get the job done, too.