2-Getting-Started

September 1, 2016

1 Git, GitHub and Jupyter Notebook

1.1 What you will need for this course

This course focuses on developing practical skills in working with data and providing students with a hands-on understanding of classical data analysis techniques.

This will be a coding-intensive course.

As discussed in Lecture 1, we are using Python, since it allows for fast prototyping and is supported by a great variety of scientific (and, specifically, data related) libraries.

The materials of this course can be found under this GitHub account.

Both the lectures and the homeworks of this course are in the format of Jupyter notebooks.

1.2 Installing Python

There are many ways to install Python. We will provide instructions for the preferred method.

Note that we are using Python 3 (get the latest version, which should be 3.5 or later).

1.3 Git

One of the goals of this course is make you familiar with the workflow of code-versioning and collaboration.

We will be using GitHub to host all the materials of the course, and we will expect you to use it also when submitting your homeworks.

If you don't have it already, you should download **git** from here.

If you don't already have one, you should also create an account on GitHub.

You can find extensive documentation on how to use **git** on the Help Pages of Github, on Atlassian, on GitRef and many other sites.

1.4 Working with Git

1.4.1 Configuration

The first time we use *git* on a new machine, we need to configure our name and email

```
$ git config --global user.name "Katherine Zhao"
$ git config --global user.mail "kzhao@bu.edu"
```

Use the email that you used for your GitHub account.

1.4.2 Creating a Repository

After installing Git, we can configure our first repository. First, let's create a new directory.

```
$ mkdir thoughts
$ cd thoughts
```

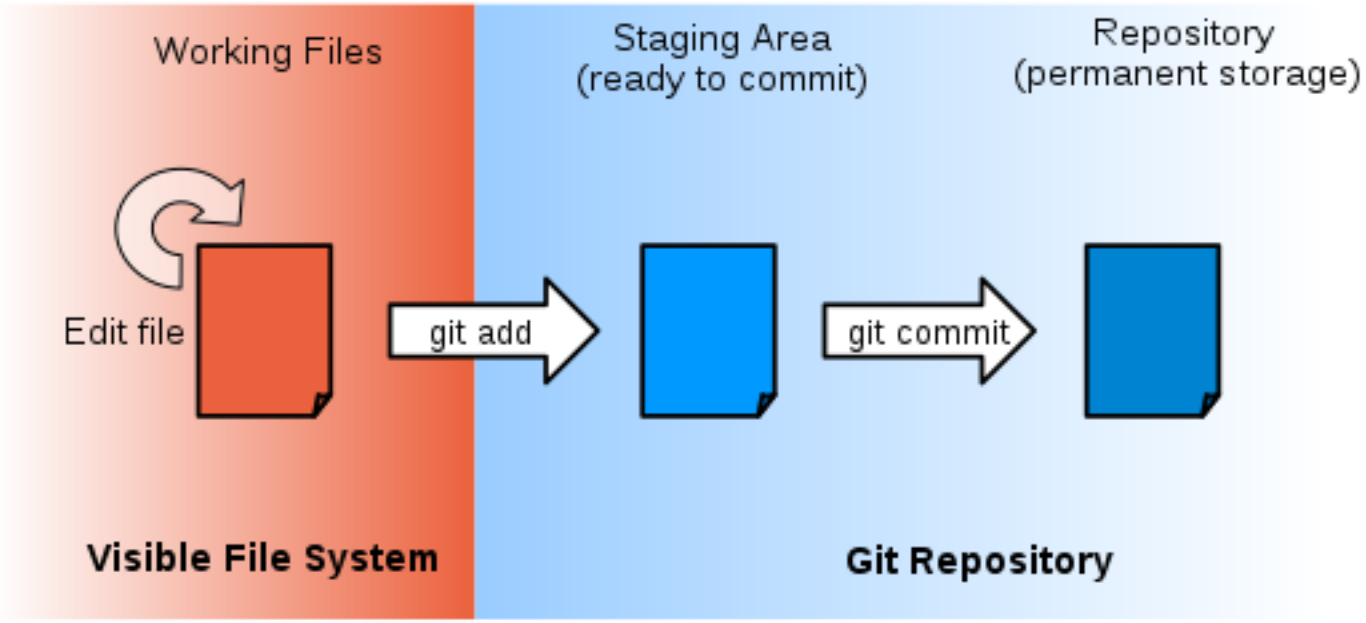

Figure 1: Alt text

Now, we can create a *git* repository in this directory.

```
$ git init
```

We can check that everything is set up correctly by asking *git* to tell us the status of our project.

```
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

Now, create a file named `science.txt`, edit it with your favorite text editor and add the following lines

```
Starting to think about data
```

If we check the status of our repository again, *git* tells us that there is a new file:

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        science.txt

nothing added to commit but untracked files present (use "git add" to track)
```

The "untracked files" message means that there's a file in the directory that *git* isn't keeping track of. We can tell *git* that it should do so using `git add`:

```
$ git add science.txt
```

and then check that the file is now being tracked:

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   science.txt
```

git now knows that it's supposed to keep track of `science.txt`, but it hasn't yet recorded any changes for posterity as a commit.

To get it to do that, we need to run one more command:

```
$ git commit -m "Preparing for science"
[master (root-commit) f516d22] Preparing for science
 1 file changed, 1 insertion(+)
 create mode 100644 science.txt
```

When we run `git commit`, *git* takes everything we have told it to save by using `git add` and stores a copy permanently inside the special `.git` directory.

This permanent copy is called a **revision** and its short identifier is *f516d22*. (Your revision will have another identifier.)

We use the -m flag (for "message") to record a comment that will help us remember later on what we did and why.

If we just run `git commit` without the `-m` option, *git* will launch an editor such as `vim` (or whatever other editor we configured at the start) so that we can write a longer message.

If you are using Windows and you are not familiar with `vim`, try installing GitPad.

If we run git status now:

```
$ git status
On branch master
nothing to commit, working directory clean
```

it tells us everything is up to date. If we want to know what we've done recently, we can ask *git* to show us the project's history using `git log`:

```
$ git log
Author: Katherine Zhao <kzhao@bu.edu>
Date:   Sun Jan 25 12:48:44 2015 -0500

    Preparing for science
```

1.4.3 Changing a file

Now, suppose that we edit the file:

```
Starting to think about data
I need to attend CS591
```

Now if we run `git status`, *git* will tell us that a file that it is tracking has been modified:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   science.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

The last line is the key phrase: *"no changes added to commit"*.

We have changed this file, but we haven't told *git* we will want to save those changes (which we do with `git add`) much less actually saved them.

Let's double-check our work using `git diff`, which shows us the differences between the current state of the file and the most recently saved version:

```
$ git diff
diff --git a/science.txt b/science.txt
index 0ac4b7b..c5b1b05 100644
--- a/science.txt
+++ b/science.txt
@@ -1 +1,2 @@
 Starting to think about data
+I need to attend CS591
```

OK, we are happy with that, so let's commit our change:

```
$ git commit -m "Related course"
On branch master
Changes not staged for commit:
        modified:   science.txt

no changes added to commit
```

Whoops! Git won't commit the file because we didn't use `git add` first. Let's fix that:

```
$ git add science.txt
$ git commit -m "Related course"
[master 1bd7277] Related course
 1 file changed, 1 insertion(+)
```

Git insists that we add files to the set we want to commit before actually committing anything because we may not want to commit everything at once.

For example, suppose we're adding a few citations to our project. We might want to commit those additions, and the corresponding addition to the bibliography, but not commit the work we're doing on the analysis (which we haven't finished yet).

To allow for this, *git* has a special staging area where it keeps track of things that have been added to the current change set but not yet committed. `git add` puts things in this area, and `git commit` then copies them to long-term storage (as a commit):

1.4.4 Recovering old versions

We can save changes to files and see what we have changed. How can we restore older versions however? Let's suppose we accidentally overwrite the file:

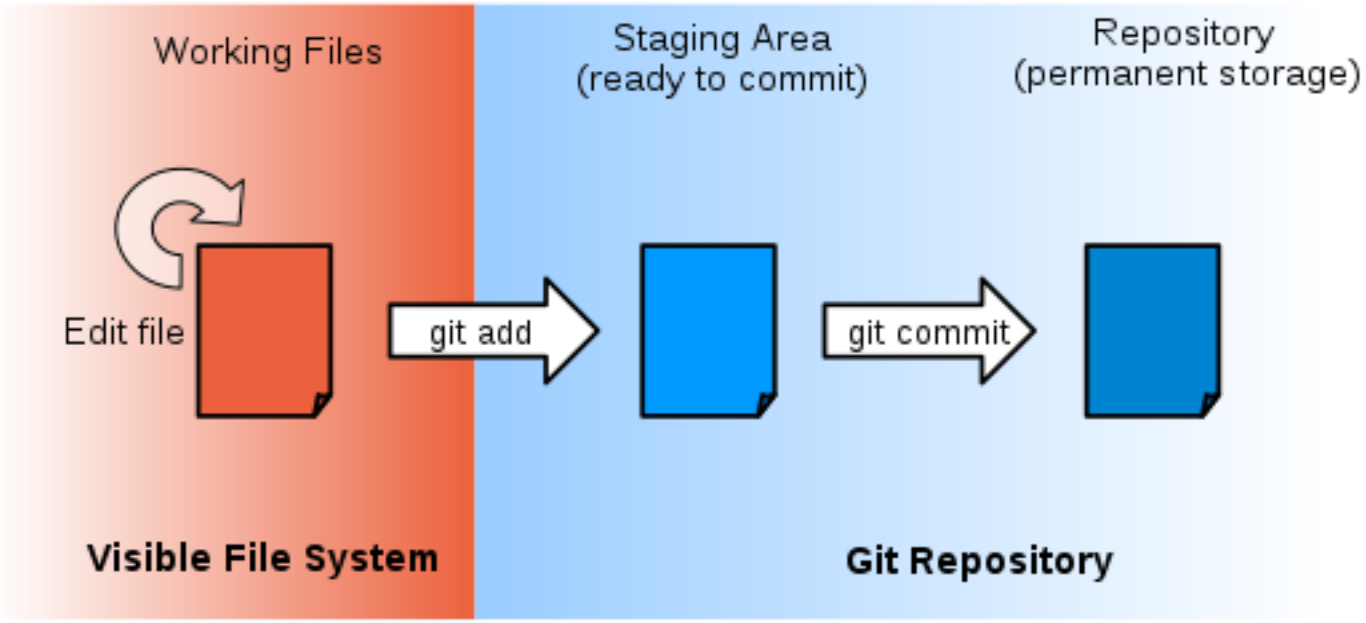

Figure 2: Alt text

```
$ cat science.txt
Despair! Nothing works
```

Now, `git status` tells us that the file has been changed, but those changes haven't been staged:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   science.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

We can put things back the way they were by using `git checkout`:

```
$ git checkout HEAD science.txt
$ cat science.txt
Starting to think about data
I need to attend CS591
```

1.4.5 More information

You can find more information on *git* here: * Software Carpentry * GitHub Help * Atlassian Help * GitRef * Git Ready

** IMPORTANT! **

Never `git add` sensitive files, e.g. *passwords*, *keys*, etc., unless you are really sure you need this.

1.5 Jupyter Notebook

You should get used to developing your code *incrementally.*

If you typically are writing files that you then run from the command line, e.g., `python code.py`, then you are not getting one of the main benefits of working in pythyon.

Instead you should interact with the interpreter.

A simple way to do this is to write your code in a file, but then cut and paste pieces of it into the interpreter to test.

To support this mode of work, use `IPython`. It has become the standard for interactive computing in Python.

There is no reason you shouldn't be using `ipython` instead of the regular python interpreter.

An even better alternative is to use `Jupyter notebook`. This puts a browser front-end on `ipython`.

These slides are actually just a Jupyter notebook.

To run the Jupyter Notebook server from the command line, type `jupyter notebook`. Your web browser will open and load the environment.

In the notebook, you can type and run code:

```
In [14]: print "hi!"
```

```
hi!
```

You can use auto-complete (with the TAB key) and see the documentation (by adding '¿):

```
In [ ]: import os
        os.listdir
        os.listdir?
```

The errors are nicely formatted:

```
In [15]: 1/0
```

```
        ---------------------------------------------------------------------------

        ZeroDivisionError                         Traceback (most recent call last)

        <ipython-input-15-05c9758a9c21> in <module>()
    ----> 1 1/0


        ZeroDivisionError: integer division or modulo by zero
```

1.5.1 Essential Shortcuts

- *Esc / Enter*: Switch between edit and command mode
- Execute cells
  - *Shift-Enter*: Run and move to the next cell
  - *Alt-Enter*: Run and make new cell
  - *Ctrl-Enter*: Run in place
- *a / b* : Insert cell below / above
- *d*: Delete cell

1.5.2 Working from the Undergraduate lab

Anaconda has been installed in the Linux machines of the Undergraduate lab as well. If you want to work from there, you need to follow the next steps:

1. Add the `conda` executable to your PATH

   ```
   $ export PATH=/usr/local/anaconda/bin:$PATH
   ```

2. Create a new environment (only do this once)

   ```
   $ conda create -p ~/envs/test numpy scipy networkx pandas  scikit-learn matplotlib beautiful-soup i
   ```

   You can change its name to something other than `test`.

3. Activate the environment

   ```
   $ source activate ~/envs/test
   ```

4. Run IPython or IPython notebook

   ```
   $ ipython2 notebook --ip=127.0.0.1
   ```

5. Deactivate the environment

   ```
   $ source deactivate
   ```

1.6 GitHub

Systems like *git* allow us to move work between any two repositories.

In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web rather than on someone's laptop. Most programmers use hosting services like GitHub or BitBucket to hold those master copies.

For the purpose of our course, we will be using GitHub to host the course material.

You will also submit your homeworks through this platform.

Next, we will cover how you can fork and clone the course's repository and how to submit your solutions to the homework.

For more information on how to create your own repository on GitHub and upload code to it, please see the tutorial by Software Carpentry.

1.6.1 Course repositories

The material of the course is hosted on GitHub, under this account.

1.6.2 Clone the lecture repository

In order to download a copy of the lectures and run them locally on your computer, you need to clone the lecture repository. To do that:

1. Create a new folder for the course.

   ```
   $ mkdir cs591
   $ cd cs591
   ```

2. Copy the clone url from the repository's website.

3. Clone the repository from *git*.

```
$ git clone https://github.com/mcrovella/CS505-Computational-Tools-for-Data-Science.git
```

You should now have a directory with the course material.
To update the repository and download the **new material**, type

```
$ git pull
```

1.6.3 Create a private homework repository

Go to GitHub Education and click on "Request a discount" to get free private repos. You will put all your homework submissions in this repo.

Under *Settings* of your private repo, add **datascience16** as one of your *Collaborators*.

1.7 Practice

Now, practice what we have seen today by solving and submitting Homework 0.