Day I - Part III - Introduction to git

December 3, 2020

1 Introduction to git

Version control is a system which:

- records all files that make up a project (down to the line) over time
- tracks their development
- provides the ability to recall previous versions of files.

This type of system is essential for ensuring reproducibility of scientific research

There are a number of popular tools for version control, the particular tool we will use is **git**.

2 Setting up git

Initially we need to set up git.

git keeps track of the entire history of a project. This does not only mean keeping track of what was done but also who did it. So we start by telling git who we are by running the following two commands:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "Your Email"
```

Note this is not data that is being collected by any cloud service or similar. It just stays with your project.

Windows Note that all these commands work on the anaconda prompt but if you want to use tab completion you can use the git bash command line specifically for git.

3 Initialising a git repository

In order to demonstrate how version control with git works we are going to use the `rsd-workshop` folder we created before.

We need tell git to start keeping an eye on this repository (folder/project). While in the `rsd-workshop` directory type:

```
$ git init
```

You should then see a message saying that you have successfully initialized a git repository.

4 Staging and committing changes

To see the status of the repository we just initialized type:

```
$ git status
```

We should see something like:

There are various pieces of useful information here, first of all that `addition.py`, `if-statement.py` and `while-loops.py` are not currently tracked files.

We are now going to track the `addition.py` file:

```
$ git add addition.py
```

If we run git status again we see:

So the `addition.py` file is now ready to be "committed".

```
$ git commit
```

When doing this, a text editor should open up prompting you to write what is called a commit message. Your machine will probably have one of the following command line editors set up as a default:

- Nano
- Vim

For the purposes of using git these are more than sufficient, all you need to know how to do is:

- Write (in Nano: just type, in Vim: press i and type);
- Save (in Nano: Ctrl + O, in Vim: press esc, then :, then w + Enter);
- Quit (in Nano: Ctrl + X, in Vim: press esc, then :, then q + Enter).

Note it is possible to set up a different default editor but instructions for this can be machine specific.

For unix you can use the following command:

```
$ git config --global core.editor "nano"
```

Type the following as the first commit message:

```
Add addition script

Addition script contains a function which adds two numbers.
```

save and exit.

git should confirm that you have successfully made your first commit.

Note A commit message is made up of 2 main components:

```
<Title of the commit>

<Description of what was done>
```

- The title should be a description in the form of "if this commit is applied `<title of the commit>` will happen". The convention is for this to be rather short and to the point.

- The description can be as long as needed and should be a helpful explanation of what is happening.

A commit is a snapshot that git makes of your project, you should use this at meaningful steps of the progress of a project.

5 Ignoring files

There are still two files in the repository that are currently not being tracted. These are `if-statement.py` and `while-loops.py`.

We do not want to keep tract of those files as they are not related to our project.

To tell git to ignore these files we will add them to a blank file entitled `.gitignore`.

Open your editor and open a new file (`File > New file`) and type:

```
if-statement.py
while-loops.py
```

Save that file as `.gitignore` and then run:

```
$ git status
```

We see now that `git` is ignoring those 2 files but is aware of the `.gitignore` file.

Let us add and commit that file:

```
$ git add .gigignore
$ git commit
```

Use `Add .gitignore` as the commit message.

Now if we run `git status`, we see a message saying that everything in our repository is tracked and up to date.

6 Tracking changes to files

Let's assume that we want to change the function `add_two_numbers` to `add_two_even_numbers`. So that the function can adds two numbers that are even.

Change the file `addition.py` to look like this:

```python
def add_two_even_numbers(a, b):
    if a % 2 == 0 and b % 2 == 0:
        return a + b
    else:
        print("Please use even numbers.")

print(add_two_even_numbers(4, 6))
```

Save your file and then run:

```
$ git status
```

We now see that git is aware of a change to our file:

To see what has been modified you need to type:

```
$ git diff addition
```

and press `q` to exit.

To “stage” the file for a commit we use git add again:

```
$ git add addition
```

Now let us commit:

```
$ git commit
```

With the following commit message:

```
Change add two numbers function to add two even numbers
```

Finally, we can check the status: git status to confirm that everything has been done correctly.

7 Exploring history

`git` allows us to see the history of a project and in some cases even change it. To view the history of a repository type:

```
$ git log
```

This displays the full log of the project:

We see that there are 3 commits there, each with a seemingly random set of numbers and characters. This set of characters is called a “hash”.

The first commit with title `adds addition script` has hash: aab73629642568b9be5ca5faa5e091ea9a629d67.

Note that on your machines this hash will be different, in fact every hash is mathematically guaranteed to be unique, thus it is uniquely assigned to the changes made.

Hashes can be very useful but we are not going to cover their many uses in this workshop.

8 Creating branches

The final part that we will cover in Day I is creating branches. Branches allow us to work in parallel which is very important when developing software, and also when we do research.

When typing `git status` we have seen that one piece of information regularly given was:

```
On branch master
```

This is telling us which branch of “history” we are currently on. We can view all branches with the command:

```
$ git branch
```

This shows:

```
* master
```

So currently there is only one branch called master. Let us create a new branch called `implement-add-odd-numbers`:

```
$ git branch implement-add-odd-numbers
```

When we now type git branch we see that 2 branches exist but the active branch is indicated by *:

```
implement-add-odd-numbers
* master
```

To move to this new branch the command is:

```
$ git implement-add-odd-numbers
```

Run `git branch` and then `git status` to see how this has worked.

While we are on this branch we are going to add a new function in the `addition.py` and that is a function that adds two odd numbers.

Add the following code to `addition.py`:

```python
def add_two_odd_numbers(a, b):
    if a % 2 != 0 and b % 2 != 0:
        return a + b
    print("Please use odd numbers.")

print(add_two_odd_numbers(1, 3))
```

Let us add and commit this:

```
$ git add addition
$ git commit
```

Commit message:

```
Implement function for adding odd numbers
```

Let us now return to the master branch.

```
$ git checkout master
```

If you open the file `addition.py` you will see that the latest change is not there. That is because this change was done in a different branch.

Branches allow us to bring work from different people done in parallel and merged them to the main branch of the repository. This can be done locally using `git` and also on the cloud using GitHub. This is demonstrated in the material of Day II in the GitHub example.