Software Development for Environmental Scientists
Level 2 Exercises

DAY 1

Git revision & team working - slide 6

For these exercises, you will be working with three repositories on your own machine, and you will assume two identities – those of a developer(**D**) and a tester(**T**). This mimics what might happen with two people working on the same codebase – they could just as easily be two developers. **D** and **T** will be used to denote your different roles, and you will find it easier to have two terminal sessions open, one for each role. Open a file manager or two (or use cd and ls in the terminal sessions) to explore the contents of the repositories at any stage in the proceedings.

I would strongly advise you to record the actions on a diagram as encouraged at step 3: you will find that it's easier to see what's happening. Draw a line to represent the lifetime of a repository, mark the commands, and link lines together as you clone, branch and merge.

Set up the developer's repository:

1. **D:** Initialise a repository in any suitable location in your home directory /home/developer/mainrepo. Do this by creating the directory, then initialising git. Set up the 'user.name', 'user.email', 'core.editor' and 'merge.tool' configuration properties. You'll need:
   ```
   git init
   git config --global <property> <value>
   ```
   You can check settings at any time with:
   ```
   git config -l
   ```

2. **D:** Create some dummy files in the mainrepo/ folder and populate the repository with them using:
   ```
   git add .
   git commit -m "<your message here>"
   ```
   At any time you can use:
   ```
   git status
   git log [--oneline]
   ```
   to see what's going on.

Clone the developer's repository and track it:

3. **From this point, diagram the relationship between the repositories and what the commands do so that you can pictorially keep track of things.**
   D: Create a remote repository… for now, this is just a local one which you will treat as remote for the purposes of getting to grips with the principles. We will create it from your existing repository using:

```
git clone --bare . ../dummyRepo.git
```
This assumes that you are creating `dummyRepo.git` in a sibling folder: the first dot means 'here', then the dotdot means 'parent'.
What's the result? What changes if you don't use the `'--bare'` argument? Try it again without the argument and a second clone `'dummyRepo2'`. You can delete any repository at any time and start again! We won't need `dummyRepo2` from now on.

4. **D:** Track the bare remote repository from your initial one with a nickname `'origin'`:
```
git remote add origin ../dummyRepo.git
```
You can view information on tracked remote repositories with:
```
git remote show
```
and details using:
```
git remote -v
```

Tester gets the developer's code and adds their own files, pushes them to the remote repository.

5. **T:** Now in the tester role, you want to get hold of the repository. We're going to use 'clone' again. Change to a suitable parent folder (e.g. /home/tester) then execute:
```
git clone dummyRepo.git testerclone
```
where `'dummyRepo'` is the source and `'testerclone'` is the target. The tester now uses the latter as his/her repository.
Check that your diagram agrees with everyone else!

6. **T:** Simulate writing new code by adding a file in a test/ subdirectory. Check that you are connected to the correct remote repository (`'dummyRepo.git'`) using:
```
git remote -v
```
You want to get the file onto the remote repository and the command for this is `'push'`, but what happens if you do the following straight away?
```
git push origin master
```

7. **T:** You need to add and commit your new file! Seeing that nothing happens in the above step should warn you that you've forgotten.
Try again using:
```
git push origin master
```
What do the `'origin'` and `'master'` arguments refer to?

8. **D:** Now put your developer hat back on, and make a branch called `'devel'` in the original repository which you created at the start of this exercise; switch to it. Make some changes to files and/or add new ones.

Developer updates branch (more advanced for those people finding this easy!)

9. **D:** While in the `'devel'` branch, if you do:
```
git push origin master
```
what happens and why?

10. **D:** If you do:
```
git pull origin master
```

what happens? Hint: check the contents of your repository folder.

11. **D:** If you do:
    ```
    git push origin master
    ```
    again, what happens and why?

Developer gets the tester's changes

12. **D:** Switch back to your master branch, and get the tester's changes using:
    ```
    git pull origin master
    ```
    which gets the tester's files that they put on their master branch, and pushed to `'dummyRepo.git'` which is your shared remote.

Developer works on a branch then merges it back into the master branch when it's ready (there are no conflicts in this case) and the tester picks them up

13. **D:** Switch back to your `'devel'` branch, make a couple of new files, add and commit them.

14. **D:** With work completed on your `'devel'` branch, you want to get the changes back into the main codebase so execute:
    ```
    git checkout master
    git merge devel
    ```
    Can you explain what's happening?

15. **D:** And you also need to push the changes up to the remote repository for everyone else to access. What's the command? Execute it.

The tester gets the latest work

16. **T:** The tester wants all the latest changes which the developer has just merged into the master branch, so what's the command to do this? Execute it.

You can share branches too (more advanced)

17. **D:** Push your `'devel'` branch to `'dummyRepo.git'`. What's the command? Execute it.

18. **T:** Get the `'devel'` branch, check which branches are available to you, switch to `'devel'`, check again and check the contents of your folder to make sure you're happy it's worked. What are the commands? Execute them.

Working with conflicts

19. **D:** On your master branch, edit one of your files and commit the changes, push it up to the remote repository (you should know the command by now!).

20. **T:** On your master branch, edit the same file as you did in the previous step and commit the changes, push it up to the remote repository. **What happens?**

21. **T:** Evidently you've been beaten to the commit by somebody else! We could try to merge automatically… this sometimes works for simple changes but the algorithm is unable to deal with anything very complex, and you may end up with something not to your liking – you should always double check. So let's try anyway:

    ```
    git pull origin master
    ```

    sadly fails. A 'pull' tries to do two things at once, a 'fetch' and a 'merge', so in this case we need to do them one at a time and take over:

    ```
    git fetch origin master
    git mergetool <the pesky file!>
    ```

    This is the reason you set up the mergetool right at the start of these exercises. Now you can manually decide what to do, save and commit the file. A handy command to see what's causing the problems is:

    ```
    git diff origin master
    ```

    Now you can push the changes back to the remote and tell the developer.

    N.B. In the majority of cases when working with conflicts, one should 'fetch' and manually merge. Trying the automatic merge is probably not going to work most of the time.

22. **D:** Get the latest merged file from the remote.

For this next task, work in pairs or small groups:

23. One person volunteers to create a GitHub account and a fresh local repository with some files in it – you will need to tell your local repository about the GitHub remote. Push the files to GitHub. The others should then clone the GitHub files and carry out the exercises above again. It'll get messy! But take logical steps and try to understand what's happening at each stage.
    This is the perfect opportunity to try things out, get it wrong, and start again!
    If you have time, somebody else can do the GitHub initialisation and you can swap roles.

Version Control in an IDE

Every decent IDE has version control integration, PyCharm is no exception and can cope with many systems including git. In the following exercises you will have the opportunity to explore the features.

1. Open PyCharm and create a project using the source folder you used in the developer role in the previous exercises. Use File->Open, then navigate to mainrepo/. You will see that PyCharm detects that you have a repository… look at what's appeared at the bottom right of the window. Also, if you go to File->Settings… Version Control, you'll see that you have the option to control whether PyCharm is connected to the repository or not, also add new ones and so on.

2. PyCharm now offers a Git submenu on the mouse right-click on any file. Change things, check them in, compare them, play! Note that the file name's colour changes to tell you its git status.

3. PyCharm offers an inbuilt merging editor which is very handy, give it a go.

4. Use the Settings… Version Control dialogue to connect to your GitHub controlled code too. Now notice that you can push and pull from PyCharm. Try it.

We have covered the majority of the git commands which are most likely to use, but as you can see from the PyCharm options, there are many more. If you feel adventurous you could try some out!

Note that there are several ways to connect PyCharm to GitHub depending on your workflow:

- Create git repo. locally (either command line or using PyCharm), then push to GitHub which will create a clone – useful if you've already got code and want to control and share it with others.
- Create new empty repo. on GitHub, clone to PC, navigate to folder to create new PyCharm project – this will automatically hook them all together – useful if you're starting from scratch.
- Clone existing repo. from GitHub onto PC, create new PyCharm project from existing sources – useful if you want to work on something that's already controlled.

OO Design - object identification - slide 8

What are the methods and attributes of the following items? (a couple of examples of each will suffice). Can any of them be related, strongly or weakly?

Are they all concrete objects? What messages will they respond to?

my HB pencil
great white shark
blue cat collar
orbiting spacecraft
this course
Bob, my white cat
Bob's teeth
a satellite dish
my ball-point pen
an elephant
a fish
a mammal

Python Objects - slide 9

Use the dot notation to investigate the methods on some simple datatypes e.g. integer, string, float. Also try 'type()' on one or two of your variables – what do you get?

Create some more complex datatypes and investigate their methods: suggest looking at list, dictionary, Exception, numpy.array. How do Python types differ in how they may be created?

Try the Python function 'dir()' giving it a module or an object as its argument. For example:

```
import numpy
dir(numpy.pi)
```

What happens when you 'print()' an object? Do you get what you expect?

What happens when you check for equality of two objects?

OO Design - object relationships - slide 10

Using the list of random objects above, can you relate them better? Use inheritance, composition and aggregation.

HOMEWORK

The following paragraph is a user's requirements, identify the objects and the messages you'd need for the system, using the nouns and verb phrases:

> *This application will support the operations of a technical library for an R&D organization. This includes the searching for and lending of technical library materials, including books, DVDs, and technical journals. Users will enter their company IDs in order to use the system; and they will enter material ID numbers when checking out and returning items.*
>
> *Each borrower can be lent up to five items. Each type of library item can be lent for a different period of time (books 4 weeks, journals 2 weeks, DVDs 1 week). If returned after their due date, the library user's organization will be charged a fine, based on the type of item (books 50p/day, journals £1/day, DVDs £2/day).*
>
> *Materials will be lent to employees with no overdue lendables, fewer than five articles out, and total fines less than £50.*