APT Session 3: Version control and testing

Laurence
Tratt

Software Development Team

2015-11-04

What to expect from this session

1. What is version control?
2. `git` essentials.

What to expect from this session

1. What is version control?
2. `git` essentials.
3. Testing.

Prerequisites

You should have:

1. Created a github account.
2. Downloaded and installed git 2.6.x from http://git-scm.com/downloads
3. Downloaded and installed Python 3 from https://www.python.org/download/releases/3.5.0/
4. Ensured your laptop can connect to one of the College's wireless networks.

Version control

- 'Version control' systems keep track of changes to source code.

Version control

- ‘Version control’ systems keep track of changes to source code.
- Allows multiple people to edit a single system – even a single file – in a predictable manner.
- Without version control, working in teams is torture.

An incomplete history of version control

- 1982: RCS.

An incomplete history of version control

- 1982: RCS.
- 1990: CVS.

An incomplete history of version control

- 1982: RCS.
- 1990: CVS.
- 2000: SVN.

An incomplete history of version control

- 1982: RCS.
- 1990: CVS.
- 2000: SVN.
- 2005: git / mercurial.

An incomplete history of version control

- 1982: RCS.
- 1990: CVS.
- 2000: SVN.
- 2005: git / mercurial.
- My advice: use git or (if you have to) Mercurial. Ignore the rest.

An incomplete history of version control

- 1982: RCS.
- 1990: CVS.
- 2000: SVN.
- 2005: git / mercurial.
- My advice: use git or (if you have to) Mercurial. Ignore the rest.
- Free hosting sites: github (Git), Bitbucket (Git, Mercurial, multiple free private repos).

Distributed version control

- Typical scenario: a ‘central’ repository:
  - from which everyone *pulls* other people’s changes.
  - to which everyone *pushes* changes they have made.

Distributed version control

- Typical scenario: a ‘central’ repository:
  - from which everyone *pulls* other people’s changes.
  - to which everyone *pushes* changes they have made.
- Best practice: regularly push and pull (at least daily, in general).
- **But don’t:**
  - push half-finished changes.
  - pull if you’re in the middle of something.

git

- We will be using git (and its terminology).
- Fearsome complexity if you go looking for it. Relatively simple, if you keep it that way.

git

- We will be using git (and its terminology).
- Fearsome complexity if you go looking for it. Relatively simple, if you keep it that way.
- All commands are of the form `git <cmd> [options]`

Cloning

- To work with someone else's repository, we first *clone* it, to get a local copy.
- Use: `git clone` <*repo*>
- Note: once cloned, you can edit the repository as much as you want. No changes make their way back to the 'central' repository until you explicitly do so.

Cloning

- To work with someone else's repository, we first *clone* it, to get a local copy.
- Use: `git clone <repo>`
- Note: once cloned, you can edit the repository as much as you want. No changes make their way back to the 'central' repository until you explicitly do so.

Exercises:

1. Clone the repository http://github.com/ltratt/apt/
2. Run `python3 old.py 150`; enter whole numbers (on the same line) and press return. Quit with Ctrl-D.

diff

- `diff -u <old file> <new file>` shows you what changes you would need to apply to `old file` to change it into `new file`.
- Lines beginning with:
  - `---` or `+++` tell you the old / new filenames.
  - `@@` tells you where within the file you're looking.
  - `␣` (i.e. a space) are lines that are unchanged.
  - `-` is a deleted line
  - `+` is a newly added line

diff

- `diff -u <old file> <new file>` shows you what changes you would need to apply to `old file` to change it into `new file`.
- Lines beginning with:
    - `---` or `+++` tell you the old / new filenames.
    - `@@` tells you where within the file you're looking.
    - ␣ (i.e. a space) are lines that are unchanged.
    - `-` is a deleted line
    - `+` is a newly added line

Exercises:

1. Can you spot the difference between `old.py` and `new.py`?
2. Run `python3 simplediff.py old.py new.py` to see output equivalent to running `diff -u` (or `git diff`).

Pulling

- To integrate all changes other people have made since you cloned/pulled, `git pull`.
- If you have made local changes you have to `git stash` before pulling, then `git stash pop` afterwards.
- You can see which files you've modified with `git status`.
- You can permanently remove your local changes by `git checkout <file>`.

Pulling

- To integrate all changes other people have made since you cloned/pulled, `git pull`.
- If you have made local changes you have to `git stash` before pulling, then `git stash pop` afterwards.
- You can see which files you've modified with `git status`.
- You can permanently remove your local changes by `git checkout <file>`.

Exercises:

1. Pull my changes to old.py into your repository.

Pulling

- To integrate all changes other people have made since you cloned/pulled, `git pull`.
- If you have made local changes you have to `git stash` before pulling, then `git stash pop` afterwards.
- You can see which files you've modified with `git status`.
- You can permanently remove your local changes by `git checkout <file>`.

Exercises:

1. Pull my changes to old.py into your repository.
2. Add a print statement at line 4 (i.e. immediately after the `import` statement) `print("Enter numbers")`. Then pull, and read the instructions carefully. [Hint: you might need to `stash` and then `stash pop`.]
3. Remove your local changes to `old.py`.

Pushing

- `git add x` makes git track the file `x`.
- `git commit .` (notice the '.') records all changes into a *commit*.
- `git push` pushes all new commits to the central repository.

Pushing

- `git add x` makes git track the file `x`.
- `git commit .` (notice the '.') records all changes into a *commit*.
- `git push` pushes all new commits to the central repository.

Exercises:

1. Join up into pairs. Choose one person to create a new repository.
2. Login to your github account, click on the 'repositories' tab and click 'new'. Name it 'apttest'. Make sure 'Initialize this repository with a README' is checked then press 'create repository. Go to 'Settings > Collaborators' and add the other person's username.
3. Both clone the repository (over ssh).
4. One person should copy the file old.py into their cloned repository, add the file, commit it, and push.
5. One at a time: edit the file, commit, push; and have the other person pull. Swap roles.

Merges and conflicts

- If two people both modify the same file, the first to push 'wins'. The second person will have to pull and merge before pushing.
- Changes in different parts of a file are automatically merged.
- Changes in the same part of a file cause conflicts (between $<<<$ $===$ $>>>$) and require the user to manually resolve them. Can select either HEAD (your changes) or remote, or a mix of the two.
- Two merging cases: have / haven't committed.

Merges and conflicts

- If two people both modify the same file, the first to push ‘wins’. The second person will have to pull and merge before pushing.
- Changes in different parts of a file are automatically merged.
- Changes in the same part of a file cause conflicts (between `<<< === >>>`) and require the user to manually resolve them. Can select either HEAD (your changes) or remote, or a mix of the two.
- Two merging cases: have / haven’t committed.

Exercises:

1. Edit `old.py` so that line 1 is `#! /usr/bin/python3`
2. Try `git pull`: follow its suggestions carefully until you have successfully pulled and integrated your changes.

Merges and conflicts

- If two people both modify the same file, the first to push ‘wins’. The second person will have to pull and merge before pushing.
- Changes in different parts of a file are automatically merged.
- Changes in the same part of a file cause conflicts (between `<<< === >>>`) and require the user to manually resolve them. Can select either HEAD (your changes) or remote, or a mix of the two.
- Two merging cases: have / haven’t committed.

Exercises:

1 Edit `old.py` so that line 1 is `#! /usr/bin/python3`

2 Try `git pull`: follow its suggestions carefully until you have successfully pulled and integrated your changes.

3 Commit your changes. Pull my changes. Merge appropriately.

Commits

- Merge commits record where parallel development unified.

Commits

- Merge commits record where parallel development unified.
- How does git keep track of things when parallel development happens?
- Every commit has an ID (it's *hash*), which is a 40 character SHA-1 hash based on the commit's content. Not guaranteed to be unique; but it probably is.

Commits

- Merge commits record where parallel development unified.
- How does git keep track of things when parallel development happens?
- Every commit has an ID (it's *hash*), which is a 40 character SHA-1 hash based on the commit's content. Not guaranteed to be unique; but it probably is.

Exercises:

1. Run `gitk` (if you don't have that, try `git log --graph`).
2. Use `git diff` to see the differences between your latest version of `old.py` and the first version in the repository. [You will need to lookup the documentation for `git-diff`; you may want to use commit hashs.]

Branches

- A repository (local and remote) can have explicit branches.
- The default branch is called `master`.
- Create branches with `git branch` <*name*>; switch branches with `git checkout` <*branch name*>.
- To merge branch *X* into *Y*, checkout *Y* and run `git merge` *X* (i.e. you say "I want to merge another branch into me").
- Branches are used extensively (e.g. some like feature branches).

Branches

- A repository (local and remote) can have explicit branches.
- The default branch is called `master`.
- Create branches with `git branch <name>`; switch branches with `git checkout <branch name>`.
- To merge branch *X* into *Y*, checkout *Y* and run `git merge X` (i.e. you say "I want to merge another branch into me").
- Branches are used extensively (e.g. some like [feature branches]).

Exercises:

1. Create a new branch in the `apt` repository called `floats`.
2. In the new branch, edit `old.py` so that `int(num)` becomes `float(num)`. Commit your change.
3. Switch back to `master`, and merge in the `floats` branch.

Code review and pull requests

- One of the best techniques for improving code quality is code review: changes are checked by someone other than their author before being merged into `master`.
- This workflow is naturally captured by *pull requests*.

Code review and pull requests

- One of the best techniques for improving code quality is code review: changes are checked by someone other than their author before being merged into `master`.
- This workflow is naturally captured by *pull requests*.

Exercises:

1. In your pairs, choose one person (*A*) to create a new branch in their clone of the other person's (*B*) repository. Edit the file `old.py` so that blank lines are deleted; commit; push (notice the message!).
2. Go to *B*'s github page for that repository, and open a pull request (there is a button for this – look carefully for it).
3. *B* will have received a pull request by email. Review the pull request and merge it in once you're happy with it.

Regression testing (1)

…as a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming. Theoretically, after each fix one must run the entire batch of test cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice, such regression testing must indeed approximate this theoretical idea, and it is very costly.

–Fred Brooks, The Mythical Man Month (1975), p. 122

Regression testing (2)

- Premise: program maintenance (re)introduces bugs.
- Keeping round a test suite prevents old bugs making it into production.

Regression testing (2)

- Premise: program maintenance (re)introduces bugs.
- Keeping round a test suite prevents old bugs making it into production.
- Every time a bug is found, a test *must* be added to the test suite to stop it ever reappearing.

Regression testing (2)

- Premise: program maintenance (re)introduces bugs.
- Keeping round a test suite prevents old bugs making it into production.
- Every time a bug is found, a test *must* be added to the test suite to stop it ever reappearing.
- Whenever a change is made to the system, the relevant parts of the regression suite are run again.

Regression testing (2)

- Premise: program maintenance (re)introduces bugs.
- Keeping round a test suite prevents old bugs making it into production.
- Every time a bug is found, a test *must* be added to the test suite to stop it ever reappearing.
- Whenever a change is made to the system, the relevant parts of the regression suite are run again.
- Regression testing builds confidence when modifying a system.
- Without regression testing, sensible people will tend to become nervous about modifying a large system for fear of breaking it.

Unit testing

- The *xunit* family of libraries are available for nearly every language.
- They allow us to easily write and run tests.
- Let’s try a few Python examples.

Travis CI

- Travis CI automates checking of unittests in pull requests.

Travis CI

- Travis CI automates checking of unittests in pull requests.

Exercises:

1. Copy the file `t.py` from my repository into your repository.
2. In you repository, go to ‘Settings > Webhooks & Services > Browse the Directory’, click on ‘Travis CI’, and then on ‘Add to GitHub’. This will take you to Travis CI. Click on the ‘+’ sign next to ‘My repositories’ then click ‘Sync’. Ensure your repository has a green tick next to it in the resulting list.
3. Create a file `.travis.yml` with the following content:
   ```
   language: python
   script: python -m unittest t
   ```
4. Add, commit, and push `t.py` and `.travis.yml`. View your build at https://travis-ci.org/
5. Make a new branch; make a test fail; then open a pull request against your partner’s repository.

Post-session exercises

Try these (no particular order):

- Experiment with committing parts of a file with `git gui`.
- Experiment with rebasing, particularly squashing small commits into bigger ones before merging (I set `rebase=true` in my `.gitconfig`). Be careful: you can go badly wrong if you rebase commits that have been pushed to another repository.
- Make your editor display which lines have been changed in a file relative to your last commit (e.g. in Vim, I use the vim-signify plugin).
- Download and use JUnit for Java.