Worksheet 4: Version Control (Python)

Updated: 15th February, 2022

There are two versions of this worksheet. This is the Python version.

> **Note:** The choice of language makes no difference to how version control works, but we have to pick a language to demonstrate it.
>
> Regardless, don't worry if you don't yet completely understand everything in the code. If you make a mistake and the code stops working, that's not a problem for Git. Git just treats your code as plain text. It doesn't care what language you're using, or whether your code even makes sense!

Before doing anything else, configure Git so it knows your name and email address:

```
[user@pc]$ git config --global user.name "My Name"
[user@pc]$ git config --global user.email "my.name@student.curtin.edu.au"
```

This isn't signing you up to anything – it's just so that your commits can be identified.

Now, obtain a copy of `Maze.py`.

1. Commits

We'll take a look at basic staging and committing with Git:

(a) Create and initialise a local Git repository. To do this, create a directory (your "project directory"), change into to it, and run `"git init"`. (See the lecture notes!)

Inside your project directory, you should now have a ".`git/`" hidden directory (to store the repository data). You can see it if you type:

```
[user@pc]$ ls -a
```

> **Warning:** Leave the ".`git/`" directory alone, except to verify that it exists inside your project directory.

(b) Copy `Maze.py` into the project directory, and make your first commit:

```
[user@pc]$ git status
[user@pc]$ git add Maze.py
[user@pc]$ git status
[user@pc]$ git commit -m "Initial commit."
[user@pc]$ git status
```

The "`git status`" commands are optional, but you should observe what they tell you at different points in the process.

Question: what does "`git add`" actually do?

(c) Say we want to modify `Maze.py` by splitting it into two separate files.

> **Side note:** This kind of change is called "refactoring", and does not affect what the program actually does. We do it because it makes the code easier to work with. We'll talk more about it when we cover "modularity", later in the semester.

(i) Create a file called `GridViewer.py`, and copy the following code into it:

```
import Maze

def view(grid):
    # TBD
```

(ii) Notice that the original file `Maze.py` contains *two large, identical* sections:

```
for i in range(len(grid)):
    for j in range(len(grid[i])):
        ...
    print()
```

Let's call this the "viewing code". There's more code in place of "…", of course. Make sure you can tell exactly where each copy of the viewing code starts *and ends*.

Copy and paste the viewing code (one copy of it only) into `GridViewer.py`, in place of "`# TBD`" (being careful to keep the correct indentation).

(iii) In `GridViewer.py`, replace "`EMPTY`" with "`Maze.EMPTY`", since this value is still defined inside `Maze.py`. Do the same for the other constants: `WALL`, `START`, `END` and `VISITED`.

(iv) In `Maze.py`, delete both copies of the viewing code, and insert the following line in place of each one:

```
GridViewer.view(grid)
```

Also insert the following at the very top of `Maze.py`:

```
import GridViewer
```

(v) Examine the changes you just made:

```
[user@pc]$ git diff --color
```

Git will only show you the changes made to `Maze.py` at this point, because it doesn't yet know about `GridViewer.py`. The "`--color`" argument is optional, but nice.

(vi) Stage *both* files and make a second commit. Choose an appropriate commit message.

(vii) Ensure that "`git status`" *does not* say anything about the following:

- "`Changes not staged for commit`", which basically means you forgot to `add` something; or

- "Changes to be committed", which means you haven't committed since the last time you staged something.

If you ran the code, the Python interpreter may have created a "_pycache_" directory and compiled .pyc files. If so, Git will mention these as "Untracked files", but you can ignore this.

> **Note:** Git quietly complains if the working directory contains files that aren't in the local repo. But you *don't want* compiled files in the repo, because there's no point – they can be automatically recreated.
>
> You can make Git shut up about untracked files by setting up another file called ".gitignore". (See the Internet for advice on that one!)

(viii) See your two commits so far:

```
[user@pc]$ git log
```

Question: why do we have to stage both files, if only GridViewer.py is new?

2. Deletion and Renaming

We'll make a third commit to demonstrate another aspect of Git and staging.

Let's say we don't like the name "GridViewer", and we want to simplify it to just "Viewer". To make this change, you need to rename the file GridViewer.py to Viewer.py, and also perform a search-and-replace inside the code (both files).

Now, run "git status". Observe that Git can see that we've deleted GridViewer.py, and also that Viewer.py is "untracked".

We can stage Viewer.py and Maze.py easily enough:

```
[user@pc]$ git add Viewer.py Maze.py
```

But we must also *stage the removal* of GridViewer.py. If we don't do this, it will still exist in the repo – not just in past commits (which are unchangeable) but in future ones too.

```
[user@pc]$ git add -A GridViewer.py
```

> **Note:** The "-A" flag, roughly speaking, tells Git to stage deletions as well as file creations and modifications.
>
> Git also provides a couple of shorthand commands: "git rm" and "git mv", which perform an ordinary "rm" or "mv" followed by "git add".

Now run "git status" again. Git should now tell you what it's figured out: that you've renamed the file. And so you have actually *staged a rename*.

Finally we can make the commit:

```
[user@pc]$ git commit -m "Updated module name."
```

3. Branching and Merging

Now, we'll try out some branching and merging.

> **Warning:** *Close all your files* and *commit all changes* before running branch/checkout/merge commands. Otherwise, you will encounter severe confusion over which version of your files you're actually dealing with!
>
> At least, this applies to using Git at the command-line, as shown in this worksheet. In practice, most people use Git (or another VCS) *via* their IDE, and IDEs know about branches and thus when to save and reload your files.

(a) Create two new branches called "`updating_maze`" and "`updating_messages`". (See the lecture notes.) The names don't really matter, but as with all names, they should be meaningful.

Check to see that this worked by running `git branch` with no further arguments:

```
[user@pc]$ git branch
```

(You should see "master" and the two new branches.)

> **Note:** In the real world, we probably wouldn't bother creating *two* branches for only a small set of changes, if all the work is being done by one person. Rather, we'd create *one* branch and (perhaps) two commits in that branch.
>
> But we do want to get some practice with branches.

(b) Checkout "`updating_maze`":

```
[user@pc]$ git checkout updating_maze
```

> **Warning:** Did you remember to close all your files first?

Open up `Maze.py`, and find the following code:

```
grid = [
    [ WALL,  WALL,  WALL,  WALL,  WALL,  WALL,  WALL,  WALL,  WALL, WALL],
    [START, EMPTY,  WALL,  WALL, EMPTY, EMPTY, EMPTY, EMPTY,  WALL, WALL],
    ...
]
```

Imagine that you're developer A, who has been asked to modify the structure of the maze grid. Simply change a few of the values; e.g. change some `WALL`s to `EMPTY`s, or vice versa, or change the `START` or `END` locations. Even changing a single value is fine for our purposes here.

Stage and commit your change(s).

Now make *another, second* change, and stage and commit that too. Making multiple commits to a single branch is what typically happens, and it will also give us a more impressive visualisation later on.

Finally, *close* `Maze.py`.

(c) Checkout the other new branch "`updating_messages`".

Open up `Maze.py` again (it's important that you closed it first!). You should observe that your changes are gone, and the code is back to its original form. Don't worry – they're still there in the other branch.

Now, scroll down further in the file until you find the following line, and others like it:

```
print("MESSAGE 1")
```

This is a terrible message, very bad for usability. So, imagine that you're now developer B, and your task is to improve this message, as well as the other similar messages:

- `"MESSAGE 1"` → `"You have no idea where you're going."`
- `"MESSAGE 2"` → `"You fall into the chasm of doom."`
- `"MESSAGE 3"` → `"You stumble blindly into a solid concrete wall."`
- `"MESSAGE 4"` → `"SOLVED!"`
- `"MESSAGE 5"` → `"You have failed to escape. Future archeologists gaze upon your remains in bafflement."`

Make these changes in two separate commits. For instance, fix messages 1–3 first, then commit, then fix messages 4–5, then commit. (This is just to show you that you generally have multiple commits in a branch.)

Then, as before, *close* `Maze.py`.

(d) Now we genuinely have multiple parallel versions of the software. You should be able to see these as follows:

```
[user@pc]$ git log --branches --graph
```

(GUI tools will show this in a prettier fashion though.)

(e) Now it's merge time. First, switch back to the "master" branch. Note that, when we merge, we're really merging another branch *into* the current branch.

Go ahead and merge both new branches into master, one after the other. (See the lecture notes.) Git should be able to perform this merge automatically, as the branches have modified separate pieces of code.

Or at least *almost* automatically. On the second merge, it will probably open an editor and ask you to provide a message. You can accept the default and just quit the editor, or write in something more meaningful instead.

> **Note:** Merging often creates a new commit, which is why Git asks for a message. You can also add `-m "Some message"` to your `merge` command (just like for the `commit` command).
>
> However, merging doesn't *always* create a new commit. If one branch is unchanged (since the common ancestor), then merging is trivial, and Git just uses the last commit on the changed branch as the merge point.

When done, open `Maze.py` again. Confirm that it now contains *both* sets of fixes you made to the two branches.

You should also now be able to see the merge commit, bringing the two branches together, if you run:

```
[user@pc]$ git log --branches --graph
```

> **Note:** Some branches may not appear in the visualisation (as you might expect them to) if they don't actually contain any of their own unique commits.

(f) You should now delete the two new branches. Otherwise, as our project develops, we'll accumulate hundreds of old, unused branches, which could easily cause confusion.

However, it's also nice to preserve some record of our old branches, and we can do this by "tagging" the last commit in each brach:

```
[user@pc]$ git tag archive/updating_maze updating_maze
[user@pc]$ git tag archive/updating_messages updating_messages
```

> **Note:** As far as Git is concerned, "`archive/updating_maze`" is just a simple name, and "/" is just like another letter. However, it's good to have naming conventions, and this is a common one when using tags for this purpose. It indicates (to the rest of our team) that the tag represents an archived branch.
>
> Tags are used for other things too, in which case we'd name them differently.

Then you can perform the actual branch deletion:

```
[user@pc]$ git branch -d updating_maze
[user@pc]$ git branch -d updating_messages
```

4. Merge Conflicts

We'd also like to see what happens when Git *cannot* merge automatically.

(a) Create another two branches, say "`if_replace`" and "`rename_vars`".

(b) Switch to the "`if_replace`" branch, and open `Maze.py`.

Find the following snippet of code:

```
if grid[currentRow][currentCol] == EMPTY:
    grid[currentRow][currentCol] = VISITED

elif grid[currentRow][currentCol] == WALL:
    done = True
    print("You stumble blindly into a solid concrete wall.")

elif grid[currentRow][currentCol] == END:
    done = True
    solved = True
    print("SOLVED!")

else:
    pass # Do nothing
```

Refactor this into (i.e. replace with) a slightly different form, as follows:

```
cell = grid[currentRow][currentCol]
if cell == EMPTY:
    grid[currentRow][currentCol] = VISITED

elif cell == WALL:
    done = True
    print("You stumble blindly into a solid concrete wall.")

elif cell == END:
    done = True
    solved = True
    print("SOLVED!")

else:
    pass # Do nothing
```

Close `Maze.py` and commit your changes.

(c) Switch to the "`rename_vars`" branch, and open `Maze.py`. The original "if-else" should still be there, and we're now going to make a *conflicting* change.

We're going to rename the "`currentRow`" to simply "`row`", and "`currentCol`" to "`col`". You can do this with a simple find-and-replace in whatever editor you're using.

> **Warning:** In general, be careful with find-and-replace, because the editor does not understand your code. If you use this feature, it's best to replace occurrences one-by-one, so that you can check each one. Avoid using the "Replace All" button (or equivalent), unless you're *really, really* certain it won't replace the wrong things.

Close `Maze.py` and commit your changes.

(d) Switch back to "master", and attempt to merge the changes.

merge wih whcih branch? rename to replace or master to rename , master to replace?

The first merge should succeed (because the unmodified master branch can't create a conflict). The second merge should report a merge conflict.

(e) Run "`git status`" – it should say that `Maze.py` has been modified by both branches.

(f) Open `Maze.py` again. Scroll down, and you should find something like this:

```
<<<<<<< HEAD
                cell = grid[currentRow][currentCol]
                if cell == EMPTY:
                    grid[currentRow][currentCol] = VISITED

                elif cell == WALL:
                    done = True
                    print("You stumble blindly into a solid concrete wall.")

                elif cell == END:
=======
                if grid[row][col] == EMPTY:
                    grid[row][col] = VISITED

                elif grid[row][col] == WALL:
                    done = True
                    print("You stumble blindly into a solid concrete wall.")

                elif grid[row][col] == END:
>>>>>>> rename_vars
```

Git is providing you with both sets of conflicting changes, with some symbols to indicate where they start and end.

In this case, the easiest way to resolve this particular conflict is to:

(i) Delete the second part – everything from "`=======`" to "`>>>>>>> rename_vars`", inclusive.

(ii) In the section that remains, replace the two occurrences of "`currentRow`" with "`row`" and "`currentCol`" with "`col`".

(iii) Also delete the opening line "`<<<<<<< HEAD`".

Hopefully you can see that the result achieves what both conflicting branches were aiming for.

Once again, close `Maze.py`, and stage and commit your changes. This particular commit will complete the merge operation.

> **Note:** What does the text "`HEAD`" and "`rename_vars`" mean?
>
> Well, `rename_vars` is the branch that triggered the merge conflict. `HEAD` is a special alternative name for (basically) the current branch; where the next commit will end up. You might have noticed it appearing in "`git log`".

5. Remotes

In this part, we'll explore multiple repositories, so as to simulate a (very small) team of developers.

(a) Let's set up a "central" repository. Choose *EITHER* of the following approaches:

(i) Sign up for an account with one of the following services: bitbucket.org, github.com, gitlab.com, etc.

You want to find a service that offers free *private* repositories. Other restrictions, such as the maximum number of team member or collaborators, don't really matter for our purposes. (Depending on the service, you may need to look at "educational" plans/accounts.)

> **Warning:** Be careful how you use online repositories for university assignment and practical work (for any unit), or else you may invite allegations of collusion (academic misconduct).
>
> It is reasonable to use online repositories for storing assignment and practical work, and it is reasonable to showcase any work you believe to be of good quality.
>
> However, if you make your work publicly available *prior to OR too soon after** the due date, others may be able to copy it, and thus gain an unfair advantage. This is at least partly your responsibility. Therefore, make use of *private* repositories as needed.
>
> * Wait until the end of the semester.

You will need to create a new repository. Make sure it is a private Git repository. You can pick any reasonable name. Other details shouldn't matter for our purposes.

The service should provide you with a URL that Git can use to access the repository remotely; e.g.:
"`https://username@bitbucket.org/username/myproject.git`".

> **Note:** The URL Git uses to access a remote repo is *not* the same as the website URL.

(ii) If you have a in-principle objection (or there's some other hurdle) to accessing or registering your details with an external organisation, you can instead set up an extra local repository. You can use this to mimic a remote repo, but there is a subtle trick to it:

```
[user@pc]$ cd
[user@pc]$ mkdir mock_remote_repo
[user@pc]$ cd mock_remote_repo
[user@pc]$ git init --bare
```

From here, you can simply use "`/home/username/mock_remote_repo`" (or whatever the actual path is) in place of a URL for the remote repo.

> **Note:** The addition of "`--bare`" means that this new repo won't have a "working directory". The directory will just contain all the data that would otherwise have been kept inside ".`git/`". Consequently, you won't be able to use it to directly *work* on your project, only for storage, viewing, and retrieval, using Git.
>
> This is important, because Git won't let you "push" branchX to another repo if that other repo already has branchX open (because that would be confusing and possibly destructive). It's cleaner if we prohibit the remote repo from having *any* branch open at all.

(b) We need to tell our local repo about the remote one:

```
[user@pc]$ git remote add origin https://...
```

Where:

- "`origin`" is how you will refer to the remote repo. (This can be any name you like, but "origin" is conventional.)
- "`https://...`" is the URL of the remote repo. You can alternatively give the full directory name of another (bare) local repo.

We can check what our existing remotes are like this:

```
[user@pc]$ git remote -v
```

(c) Let's try a push:

```
[user@pc]$ git push -u origin master
```

> **Note:** When working with BitBucket, GitHub, etc., expect to enter your username and password when pushing, etc., *unless* you configure the service to allow password-less, public-key-based SSH connections. However, that's outside the scope of this worksheet!

(d) Have a look at the web interface (if you're using BitBucket, GitHub, etc.), and see if you can identify the newly-uploaded repository contents.

> **Note:** You will probably find that the web-based interface is a lot nicer to use than the command-line. But being *able* to use the command-line is a very useful skill!

If you are using a second local repo instead, you can change into that directory and run "`git log`" as before.

> **Note:** A "bare" repository does not support certain git commands, such as "`git status`", because they're only useful in conjunction with a working directory.

(e) Let's simulate a second developer, by create another local repo. To do this, we will clone the remote. (For anyone who already has a second local repo, this will create a *third* one.)

First, change out of your project directory, and then run:

```
[user@pc]$ git clone https://... myproject2
```

Where:

- "`https://...`" is the URL of the remote repo, as before.
- "`myproject2`" is a directory that will be created for the new local repo. This is optional, but (in this particular case) we want to make sure the name doesn't conflict with the existing local repo(s). You can pick whatever name you like.

You should now have a directory called "`myproject2`" (or whatever name you chose). Change into it and have a look around.

> **Note:** When you run "`git clone`", the "origin" remote will be set up automatically for you.

Now we can pretend we have two developers!

(f) Go to the "Developer 1" local repo, make a new branch ("`change_symbols`"), and switch into it.

Open `Viewer.py`, and notice all the little two-character strings used to display the maze: `" "`, `"##"`, `"^^"`, `"$$"` and `".."`. To give developer 1 something to do, change some/all of these to different sets of characters (your choice).

Stage and commit your changes, and push your new branch to origin:

```
[user@pc]$ git push -u origin change_symbols
```

(g) Go to the "Developer 2" local repo, fetch the branch, and switch to it:

```
[user@pc]$ git fetch origin change_symbols
```

```
[user@pc]$ git checkout change_symbols
```

Check that it contains the fix you just wrote.

Note: There is a subtlety here. The fetched branch is not immediately treated as a real branch, but rather a *tracking branch* (Git's terminology), indicating that the local repo is just "tracking" what is happening on origin.

This avoids getting branches mixed up due to potential naming conflicts. If Developer 2 already had an equivalently-named branch, then origin's version of it is kept separate (under the name "`origin/change_symbols`" in this case).

We could then execute a merge:

```
[user@pc]$ git merge origin/change_symbols
```

In fact `fetch-merge` is such a common pair of operations that there is a shortcut: the "`git pull`" command. It executes a `fetch` and then a `merge`, in order to properly integrate downloaded changes into your local branch.

Assume "Developer 2" is happy with the fix. See if you (as "Developer 2") can merge the change into "master" and push it back to origin.

End of Worksheet