Tutorial - Git

0. Welcome

Welcome to the first tutorial for SOFT2412/COMP9412. Each tutorial will include practical activities based on the previous lecture, which will feed into the assessment for this unit.

1. Meet & Greet
2. What is Git?
3. Installing Git into your machine
4. Creating your first Git repository
5. Adding files to the staging area
6. Commit your first change
7. Tracking status, changes and commit history
8. Checking commit logs
9. Resetting your commits
10. Stashing your changes
11. Creating a branch
12. Merging git branches

1. Meet and Greet

The assessments in this unit are completed by group work, except the quizzes where you will work individually. Because of this, you will need to meet your lab mates that you will be working with for your assignments. You will have to form a group within your tutorial schedule later when the assignments are released. The tutorial tasks will also often include activities between different people, to allow you to practice agile practices.

Finally, take this time to meet your tutor who will help you during the whole semester.

2. What is Git?

Git is a free and open source distributed version control system that is used to track changes in source code. It is an actively maintained and mature project. There are many compelling features that makes Git a convenient and a “must-have” tool in the software development community, including:

1. Git is an open source and free tool
2. Git is easy to use and very fast
3. Features like branching: create a branch to try out a new idea, switch back to a previous state in your code easily without losing any changes. Create a branch to experiment it, realise it’s not going to work and just delete it – nobody will ever see it.
4. Cryptographic integrity of every bit of your project is ensured.
5. Git is decentralised and distributed, which means that collaborating code among many developers couldn’t get better!

And these are just a few advantages of using Git, amongst a big list of advantages, which makes Git a necessary tool during the process of any software development phase.

Some terms you will often be coming across and eventually be using throughout your experience with Git are as follows:

1. Commit: A single point in the code history. Intuitively, but not formally, it is a snapshot that is comprised of a particular change or collection of changes in the code base and can be retrieved any time.
2. Branch: An active line of development.
3. Repository: A collection of commits accompanied by meta data.

Don't worry if you are a bit confused, you will understand more about commits in the coming tasks.

3. Installing Git into your machine

Usually with Mac OS and Linux systems, Git is present by default. You check this by typing `git` and you should see a usage message. If not, then follow the instructions in **https://git-scm.com/book/en/v2/Getting-Started-Installing-Git (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git)** .

As for windows, it is recommended that you use **WSL (Windows Subsystem for Linux) (https://www.microsoft.com/en-au/p/ubuntu-1804-lts/9n9tngvndl3q)** which is a command line of linux

in windows. It will help you be familiar with linux CLI commands.
You may otherwise download, if you are confident and familiar with windows CLI, the Git installer from **https://git-scm.com/downloads (https://git-scm.com/downloads)** .

Throughout the tutorial exercises, it's better to download Atlassian's Git Cheat Sheet to understand more about the common or frequently used commands when using Git. The link to the cheat sheet is **https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet (https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet)** .

4. Creating your first git repository

We will begin the exercises by initialising an empty repository in your local machine. Start by opening a **Git Bash terminal window** (Windows) or **Terminal** (Mac/Linux). The command that follows in this tutorial will be slightly different if you open a Git CMD terminal window (on Windows) instead. Once you have opened the Git Bash terminal (or terminal for Mac/Linux users), create a new directory by running command:

```
mkdir SOFT2412
```


Then, change the current directory to your working directory by running:

```
cd SOFT2412
```

Next, run the following commands to create another new directory and go to the new directory (this directory will be the root working or project directory):

```
mkdir tutorial1
cd tutorial1
```

Lastly, we can create a new text file with content “soft2412” using these commands:

```
echo soft2412 > unit.txt
```

Then we can **create a new git repository** using the following command:

```
git init
```

The echo command writes the content “soft2412” to a text file named “unit.txt”.
The “git init” command initialises an empty local Git repository inside the tutorial1 directory. After executing the commands, you should be able to see a hidden “.git” directory using the commands:

```
ls -a
```

The “.git” repository stores all the metadata and actual project data in sub-directories. The structure of the directories should be the same for all Git repositories. Try entering this command:

```
ls .git
```


You will see the sub-directories of the “.git” directory. It should contain directories like: HEAD, objects, refs, etc.

5. Adding files to the staging area

Firstly, let's understand what **staging** is. Before any commit it made to the repository, the desired files including changes/additions must be placed in what’s known as a staging area. In simple words, a staging area is a file contained in your directory that stores information about what will go to the next commit. In essence, when you stage a file, the repository makes a note of this stage, and knows that whenever the user makes a commit using the `git commit` command, this file with its changes should be part of the new commit. Files can be added to the staging area using `git add`

Before committing the files, you will need to add files to your repository. The git add command below will add unit.txt to the staging area. Enter the following commands at your project root directory.

```
git add unit.txt
find .git/objects -type f
```

```
You should be able to see something similar to this:
.git/objects/de/c4cb0dba536b6baaaa0d487c0d34655d46da84
```

The command `git add unit.txt` adds the text file into the repository. It creates a blob object in the `.git/objects` directory. The file is named using the SHA1 hash of the contents of the

file. The first 2 characters of the SHA1 hash refers to the directory name and the rest is the object's file name. The SHA1 hash of the above example is dec4cb0dba536b6baaaa0d487c0d34655d46da84 (need to add the prefix `.git/objects/de/c4cb0dba536b6baaaa0d487c0d34655d46da84 -> dec4cb0dba536b6baaaa0d487c0d34655d46da84`) where `de` refers to the directory name.

To view the content of the object dec4cb0dba536b6baaaa0d487c0d34655d46da84, you need to execute the following command at the "`.git/objects`" directory:

```
cd .git/objects
git cat-file -p dec4cb0dba536b6baaaa0d487c0d34655d46da84
```

You should be able to see the content of the object as "soft2412". When you have 2 files with the same content, e.g. created by copying the other, there will only be 1 copy in the Git's object database. This is because the content shares the SHA1 value. Change the directory back to where your unit.txt was. You can test this out executing the following commands.


```
cp unit.txt unit.bak
git add unit.bak
```

What you have done is to create a copy using the "`cp`" command of "`unit.txt`" and named the file as "`unit.bak`". Finally, you add the new file into the Git repository. Check if there are any changes made to the repository.

The "`git add`" command can also be used to add directories. This is helpful when you have multiple files in a directory. Refer to the cheat sheet again and see how you can execute the git add command for a directory.

6. Committing your first change

Let's understand what a **commit** is in Git. As explained in previous parts, when a file or a set of files is added to the staging area, these set of changes hasn't been recorded. We have only yet kept a note of **what must be recorded,** but haven't actually recorded anything. To make sure the content of the staged files is recorded, we perform what is known as a commit. This involves creating a snapshot of the staged files, and associating a message to it thereby creating a blob object. A SHA1 hash is also computed for every commit made into the repository, thus ensuring integrity. Basically, a git commit "records the changes made to the repository".

Before you can start running any git commit, you are required to set up your name and email address of the committer. This way, the commit will have an author. Run the following commands and replace the information with your actual name and email address.

```
git config --local user.name "Your Name"
git config --local user.email "Your Email Address"
```

If you wish to keep the same username and email across any repository you create (unless otherwise made local explicitly), then use the `--global` flag instead.

Assuming you still have followed the previous steps (staging your files), you are ready for your first commit. Run the following command to create a commit for all files in staging area:

```
git commit -m "My First Commit"
find .git/objects -type f
```


(Note that simply typing "git commit" without the -m flag is also alright, it would prompt you into the vim editor. To insert a message into vim, hit "i" key and type your message. Once you are done, hit "esc" and then ":wq" to ensure the changes are written. The w stands for a write and the q stands for quitting the vim editor)

You will notice that there are 3 objects found. The commit creates additional 2 object types, a tree object and a commit object. The tree object records the mapping between the blob and the file names. The commit object records the content of this commit as well as the reference to its parent commit. To inspect the tree object, use the command to print the object's content:


```
git cat-file -p <object name>
```

You should be able to see a similar output as the following:

```
100644 blob dec4cb0dba536b6baaaa0d487c0d34655d46da84 unit.bak
100644 blob dec4cb0dba536b6baaaa0d487c0d34655d46da84 unit.txt
```

The content of the object contains 2 rows, with each row representing a file. The format of the rows displays the file type, the object type, the hash and the file name. For example, the file type "10644" refers to a normal file. Next, you can inspect the commit object. The output will be similar to the following:

```
tree 0748b6dad6574ce9ccf7a2c56b87f9cf994f334d
author Basem <basem.suleiman@sydney.edu.au> 1532580259 +1000
committer Basem <basem.suleiman@sydney.edu.au> 1532580259 +1000
```

```
My First Commit
```

It contains information about the file structure, metadata, author, committer and the commit message.

7.Tracking status, changes and commit history

Git is useful for tracking historical changes to a file. Append the following text content, "Agile Software Development Practices" to "`unit.txt`" and observe what has been changed since the last commit using the command git status.


```
    echo Agile Software Development Practices >> unit.txt
    git status
```

You should be getting an output similar to the following:

```
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:  unit.txt

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

Now, before staging “unit.txt”, we can see how different was it from the most recent commit. That is, we can see the specific changes made to “unit.txt”. Similar to how “diff” command works in linux, git diff can find differences between two git data sources (I.e. commits, branches, files and more). Performing a git diff here works as follows:

First do a git diff:

```
    git diff
```

You should get an output similar to the following:

```
    diff --git a/unit.txt b/unit.txt
    index dec4cb0..f4a4437 100644
```

```
--- a/unit.txt
+++ b/unit.txt
@@ -1 +1,2 @@
soft2412
+Agile Software Development Practices
```

Now, in order to commit these changes, do the following:

First add the file into staging area using commands: '

```
git add unit.txt
git status
```


You should be getting an output similar to the following:


```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   unit.txt
```


(Note that you can still see the changes in the staged files from the previous commit, you need to specify the “--staged” flag in order to do so. Try doing it on your own machines, is it different from the previous output?)

To commit this change, we can execute the following command:

```
git commit -m "add unit name to unit.txt"
```

By now, you should have 6 objects in the Git repository. 2 of them are the commit objects, each with its own tree object and 2 blob objects. There are 2 blob objects because the 2 files (unit.txt and unit.bak) have different contents now. In order to tell which is which, you can use the “git cat-file - p” command to inspect. However, any repositories would contain many commits and by default, Git only shows the latest one. To find the latest commit, we can inspect by using the cat command.

```
cat .git/refs/heads/master
```

The output of this command is the SHA1 hash file name which contains the latest commit. Let's inspect using the command:

```
git cat-file -p <file name outputted from the last command>
```

Your output should be similar to the following:

```
tree 78b63e73cb6a5dabdad47d1168dbbc0b7a405db4
parent c104d14008cdcef613ec0ef06bdd8af708db0d8b
author Basem <basem.suleiman@sydney.edu.au> 1532582657 +1000
committer Basem <basem.suleiman@sydney.edu.au> 1532582657 +1000

add unit name to unit.txt
```


If you were to compare this commit with the first commit you performed, you will notice an extra row of information, that links to the parent commit object. This is how Git tracks the history of changes to the content. All commits are linked together with the parent pointer as a linked list. The head of the list is pointed to by an external file. The repository's commit history looks like: head -> c2 -> c1

There will be a scenario where you want to commit all files without adding them to staging area first. For example, the terminal will show this output when you run “git status”:


```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   unit.txt
    modified:   unit.bak
```

You can run “git commit” with option “-a” to commit all these files (unit.txt and unit.bak) skipping “git add” command.

8. Checking commit logs

After committing all the changes, you might want to look back to what has happened. The history of commits made can be viewed as follows:

```
git log
```

You should see the history of commits as shown below:

```
Author: Basem <basem.suleiman@sydney.edu.au>
Date:   Sun Jul 21 20:18:24 2019 +1000

    add unit name to unit.txt

commit af55a4046cfc58804565fae37683fc902fb008ec
Author: Basem <basem.suleiman@sydney.edu.au>
Date:   Sun Jul 21 20:01:49 2019 +1000

    My First Commit
```

"git log", by default lists all the commits in reverse chronological order.

The log output can be limited. You can use limiting options as follows:

```
git log –n
```

Lists the last “n” commits only.

```
git log –n -p
```

Lists the last “n” commits and for every commit it shows the differences from its previous commit.

There are some decorating features in the git log tool that might be of help. You might be familiar with the graphs on the lecture slides depicting the commit history. The git log tool provides features that can produce a graph in the command line:

```
git log --graph
```

This should output the following:

```
* commit e7432094641cfbf534295c95d4114df887f732df (HEAD -> master)
| Author: Basem <basem.suleiman@sydney.edu.au>
| Author: Basem <basem.suleiman@sydney.edu.au>
| Date:   Sun Jul 21 20:18:24 2019 +1000
|
|     add unit name to unit.txt
|
* commit af55a4046cfc58804565fae37683fc902fb008ec
  Author: Basem <basem.suleiman@sydney.edu.au>
```

```
    Date:   Sun Jul 21 20:01:49 2019 +1000

        My First Commit
```

Since this output is large, and may not seem untidy in this example, however for a repository with many commits, such an output will look messy. We can shorten the output of the commits by using the "–-oneline" and "–-decorate" flags:

```
git log --graph --oneline --decorate
```

Which should output the following:

(mailto:basem.suleiman@sydney.edu.au>)


```
* e743209 (HEAD -> master) add unit name to unit.txt
* af55a40 My First Commit
```


For a sample large project, this may look like (listed in reverse chronological order):

```
* 0e25143 (HEAD, master) Merge branch 'feature'
|\
| * 16b36c6 Fix a bug in the new feature
| * 23ad9ad Start a new feature
* | ad8621a Fix a critical security issue
|/
* 400e4b7 Fix typos in the documentation
* 160e224 Add the initial code base
```


Git also keeps track of updates to the tip of the branches using a mechanism called reference logs, or "reflogs". For every git ref, there is a reflog showing its updates in the repository. Use the command as follows:

```
git reflog
```

To see how the HEAD ref was updated throughout the course of all commits, type the following:

```
git reflog show HEAD
```

9. Resetting your commit

Let's say you committed and made a mistake in your commit log message. Running the following command lets you edit the previous commit message without altering the snapshot:

```
git commit --amend
```


Without being prompted to open an editor to change the log message, type the following:

```
git commit --amend -m "Updated log message"
```


A commit can also be modified using the amend flag, not just its commit message. Using the amend flag, the most recent commit can be modified. This is better than creating a new commit. It is worthwhile noting that the commit is replaced with an entirely new one with a new hash.

For more details on amend, refer to: **https://www.atlassian.com/git/tutorials/rewriting-history (https://www.atlassian.com/git/tutorials/rewriting-history)**

Now, you might be wondering how to undo any changes made in the repository. It is natural to make a mistake or to realise that some changes may have issues, and a previous commit may be a better version of the overall system. Although Atlassian has done a great job in explaining git reset and git revert and their differences here: **https://www.atlassian.com/git/tutorials/rewriting-history (https://www.atlassian.com/git/tutorials/rewriting-history)**, we will still discuss a little bit about how they work:

To undo into a previous commit, type the following commands:

```
git checkout <previous-commit>
git reset <previous-commit>
```

The default arguments of git reset is "--mixed". The arguments I.e. flags to git reset defines what happens to the staging area and the working directory once the reset is done. "--mixed" moves all the changes from the staging area to the working directory. There are two more options:

```
git reset —-hard <previous-commit>
```

“--hard” - The most dangerous, yet frequently used option. The reference pointers (i.e. HEAD) are changed, and the staging area and working directory are reset to match the state of the commit tree at that state, meaning any changes in the working directory and staging area is lost.

```
git reset —-soft <previous-commit>
```

“--soft” - The ref pointers are updated, but the staging area and working directory are left untouched.

It is worthwhile noting that git reset never deletes a commit, it will make those commits “orphans”, i.e. there is no direct path to access them back. These orphaned commits can be found using git reflog. These orphaned commits will be deleted after git runs the internal garbage collector. By default, git is configured to run the garbage collector every 30 days.

A safer reset is known as a git revert, but works differently. A git revert will revert a specified commit to its previous commit, and this becomes a new commit. As a result, there are two identical copies of a commit. One might think it is useless to copy a particular commit, but it has its own advantages. A particular use case of a revert is when there is an error in some code of a particular commit, instead of going back to it and deleting the issue, one can inverse the changes back to that commit and append it to the front of the commit history. This is great for keeping the history intact, which is important for integrity of the revision history and for reliable collaboration.

For more information regarding revision, refer to this: **https://www.atlassian.com/git/tutorials/undoing-changes/git-revert (https://www.atlassian.com/git/tutorials/undoing-changes/git-revert)** **(https://www.atlassian.com/git/tutorials/undoing-changes/git-revert)**

10. Stashing your changes

git stash temporarily shelves (or *stashes*) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code

change and aren't quite ready to commit. This way you can save your “Work in progress” and retrieve it back.

For example, if you have made some changes to your code base and realized that there is a security issue in another branch that you must fix, you can stash your changes in order to not lose them, checkout to that commit to quickly fix that issue:

```
git stash
```

And after you’re done with fixing that issue, you can checkout back to the current working directory and pop off all your changes that you had made:

```
git stash pop
```


One can stash different changes by associating each stash with a message. This can be done by typing the following command:

```
git stash save "your message here"
```

You can list all the stashes as well:

```
git stash list
```

Note that along with a message, a stashID is stored for every different stash. The stashID is allocated in a chronological order starting from 1. You can view a summary of that stash by typing the following command:

```
git stash show <stashID>
```

For more tips and tricks involved in git stash, you may find this link useful:

https://www.freecodecamp.org/news/useful-tricks-you-might-not-know-about-git-stash-e8a9490f0a1a/ (https://www.freecodecamp.org/news/useful-tricks-you-might-not-know-about-git-stash-e8a9490f0a1a/)

11. Creating a branch

Branching allows you to add new content into the repository without the risk of breaking any codes or workflows. It allows you to isolate development work without affecting other branches in the repository. Each repository has one default branch and can have multiple branches. Branching will create a working repository that is separate from the main one. The working repository can be based on any commits made in the past. In Git, the first and default branch is always called parent.

To create a branch, you will need to decide which commit you want the branch to come out from. Therefore, you need to find out the name of the branch that contains the commit as well as the SHA1 file name of the commit object. For example, suppose you want to create a branch from the latest commit of the master branch. To find out what is the SHA1 file name of this commit object, Git provides a convenient command that lists the commits and associated object names. Execute the following command:

```
git log
```


You should be getting an output similar to:

```
commit f76db3c05ddc2b5944f0ff8c4d9d3da1cd5dd1ed (HEAD -> master)
Author: Basem <basem.suleiman@sydney.edu.au>
Date: Thu Jul 26 15:24:17 2018 +1000

    add unit name to unit.txt

commit c104d14008cdcef613ec0ef06bdd8af708db0d8b
Author: Basem <basem.suleiman@sydney.edu.au>
Date: Thu Jul 26 14:44:19 2018 +1000

    My First Commit
```

12. Merging branches

You can use the command “git branch” or “git status” to find out which branch is active at the moment. To switch to another branch, you can use the command “git checkout <branch name>”. To switch to another branch called "exp", you would be using the following command:

```
git checkout exp
```

Note that the above command won't work if an "exp" named branch did not exist.

Now, to create a new branch from this point onwards, type the following command:

```
git branch new_branch
```

And checkout to that branch with:

```
git checkout new_branch
```


The outcome of these two commands can be done in one command by:


```
git checkout -b new_branch
```


Combining the contents of two branches (usually into master but not always), is called a "git merge". Add another line to unit.txt file with the following content, "Unit Coordinator: Dr. Basem Suleiman". You can execute the following code to modify unit.txt file and commit your changes.


```
echo Unit Coordinator: Dr. Basem Suleiman >> unit.txt
```

You will notice that now you have 9 objects in your ".git/objects" directory.

Now, we will merge the exp branch to the master branch. Run the following commands:

```
git checkout master
git merge exp
```

Now, you should be getting an output similar to this:

```
Updating f76db3c..34d226c
Fast-forward
    unit.txt | 1 +
1 file changed, 1 insertion(+)
```

not remove this notice.

Live streamed classes in this unit may be recorded to enable students to review the content. If you have concerns about this, please visit our **student guide (https://canvas.sydney.edu.au/courses/4901/pages/zoom)** and contact the unit coordinator.