Section B1 Git

Git

A basic Git work-flow

1. Initialise a new local **repository**, or **clone** an existing remote one
   If you're starting a new project or don't yet have a repository set up, navigate your shell to the directory
   ```
   git init
   ```
2. **Stage** files
   If you have existing files, or as you create new ones, you can stage the files by running:
   ```
   git add file1.py file2.py
   ```
   where `file1.py` and `file2.py` are the files you want staged.

3. **Commit** changes
   Once you've staged all your files, you can now save a revision:
   ```
   git commit -m "A message about what's changed in this revision"
   ```
   This makes a new revision containing all of the staged files.

4. (optional) **Push** changes
   If your repository is also stored on a remote server, you can upload your committed revisions:
   ```
   git push
   ```
5. Get the **Status** of your repository
   ```
   git status
   ```

Staging Changes

- We can use wildcards to add multiple files:
  ```
  git add file*.py
  ```
  which would add all files that begin with file and end in .py
  or
  ```
  git add *
  ```
  which would add **all** files in the current folder. Be careful with this one.
- We can also add whole folders in the same way:
  ```
  git add my_directory/
  ```
  which would add my_directory and all the files contained therein. *Note: empty folders cannot be added until they contain at least 1 file.*
- We can use the -a flag when committing to automatically add all modified files that have been previously added (see *Committing* below)

```
# Delete a file and stage the deletion
git rm file1.py

# Move/rename a file, then stage the move
git mv file1.py file5.py
```

Committing Changes Locally

Once you've staged all your files, you can now save a revision:

```
git commit -m "A message about what's changed in this revision"
```

Files that were added in previous revisions can also be automatically included if they were changed with the **a** flag:

```
git commit -am "A message about what's changed in this revision"
```

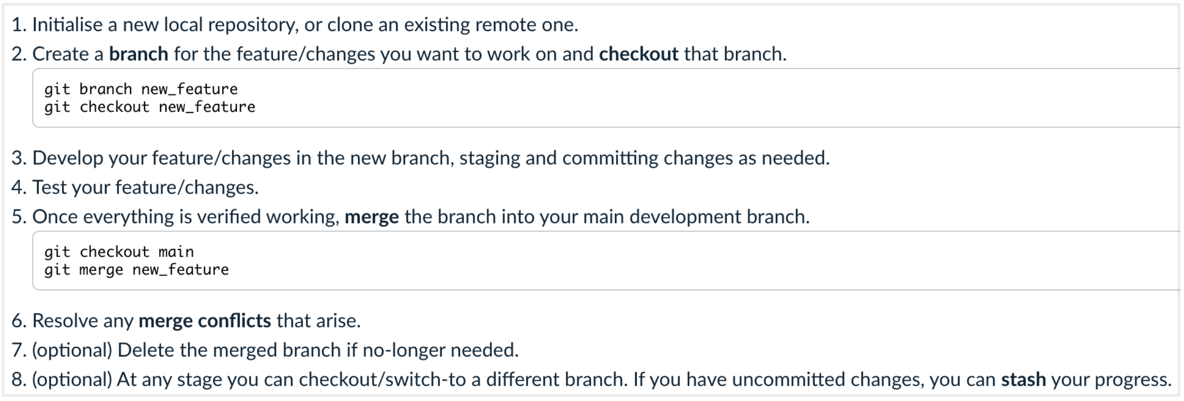
Saving Changes Remotely

If your repository is also stored on a remote server, you can upload your committed revisions:

```
git push
```

Branch

A Branching Git Workflow



1. Initialise a new local repository, or clone an existing remote one.
2. Create a **branch** for the feature/changes you want to work on and **checkout** that branch.

```
git branch new_feature
git checkout new_feature
```

3. Develop your feature/changes in the new branch, staging and committing changes as needed.
4. Test your feature/changes.
5. Once everything is verified working, **merge** the branch into your main development branch.

```
git checkout main
git merge new_feature
```

6. Resolve any **merge conflicts** that arise.
7. (optional) Delete the merged branch if no-longer needed.
8. (optional) At any stage you can checkout/switch-to a different branch. If you have uncommitted changes, you can **stash** your progress.

Merging Branches

Once you've made your changes or created your new feature, you're probably ready to merge it back into your main development branch.
First, checkout your main/master branch:

```
git checkout main
```

This sets the working copy to the HEAD of the main branch but does not overwrite changes that you have made - these will be merged as follows:

```
git merge new_feature
```

Git will attempt to merge your changes from the new_feature branch to your main/master branch.

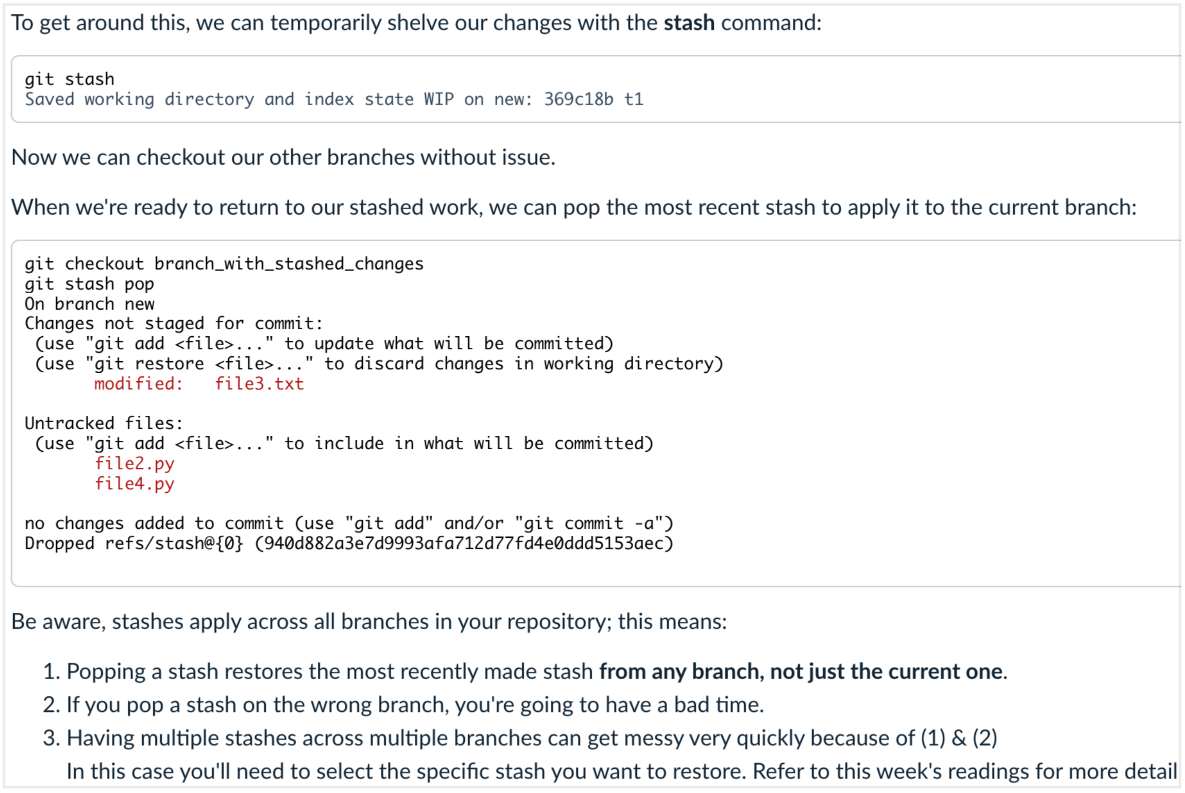
Deleting a Branch

```
git branch -d new_feature
```

- You cannot delete a branch that you're currently working in; checkout your main/master branch before deleting another branch.
- If a branch has commits that haven't been merged to the main/master

branch, Git will warn you. In this case, you'll need to use -D instead of -d to forcibly delete the branch if you still want to.

Switching Branches under Development



To get around this, we can temporarily shelve our changes with the **stash** command:

```
git stash
Saved working directory and index state WIP on new: 369c18b t1
```

Now we can checkout our other branches without issue.

When we're ready to return to our stashed work, we can pop the most recent stash to apply it to the current branch:

```
git checkout branch_with_stashed_changes
git stash pop
On branch new
Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git restore <file>..." to discard changes in working directory)
       modified:   file3.txt

Untracked files:
 (use "git add <file>..." to include in what will be committed)
       file2.py
       file4.py

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (940d882a3e7d9993afa712d77fd4e0ddd5153aec)
```

Be aware, stashes apply across all branches in your repository; this means:

1. Popping a stash restores the most recently made stash **from any branch, not just the current one**.
2. If you pop a stash on the wrong branch, you're going to have a bad time.
3. Having multiple stashes across multiple branches can get messy very quickly because of (1) & (2)
   In this case you'll need to select the specific stash you want to restore. Refer to this week's readings for more detail

Remote Repositories

A Git work-flow for a remote repository

1. Create a new remote repository, or find an existing one.
2. **Clone** the remote repository
3. Develop your feature/changes, branching, staging and committing changes as needed.
4. **Pull** from the remote repository to merge any updates.
5. **Push** your changes to the remote repository