Tools and principles for software development

Work management

- https://www.atlassian.com/software/jira is industry standard but very likely overkill
- https://trello.com/ is possibly overkill
- Creating a github issue would allow easy tracking of who is meant to be working on what (but will this really be an issue?) and they can automatically close when pull requests are approved (https://docs.github.com/en/issues/tracking-your-work-with-issues/linking-a-pull-request-to-an-issue)

Type checking

- Because Python typing is gradual and checking is only performed by external tools this is a very low risk proposition and hopefully will give reasonable rewards (better IDE auto-complete if nothing else)
- https://mypy.readthedocs.io/en/stable/getting_started.html#installing-and-running-mypy
- https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html#cheat-sheet-py3

Linters

- The code itself is not a very important part of this particular coursework but it can be something to mention in the report
- Probably overkill but you can disable categories that you find overly nitpicky
- Security: https://bandit.readthedocs.io/en/latest/
    - Will need to disable the category for assert if running over tests
- Style: https://pylint.pycqa.org/en/latest/tutorial.html
- Documentation: https://pypi.org/project/pydocstring-coverage/

Testing

- https://coverage.readthedocs.io/en/7.4.0/ + https://docs.pytest.org/en/7.1.x/getting-started.html
- To run all tests with coverage tracking: `coverage run --branch --include=$PWD/*.py -m pytest; coverage html`
- Test driven development: Write tests for the functionality you are about to add, see them fail, do the actual coding, see the tests change to passing! — https://wiki.c2.com/?CodeUnitTestFirst (rabbit hole warning for that wiki)
- Each unit test should test as few units as possible without relying on the correctness of other units else it's more an integration test than a unit test
- Resources (APIs, databases, IO etc.) should generally be mocked in unit tests to avoid wasting resources or accidentally modifying production data
    - https://pytest-mock.readthedocs.io/en/latest/usage.html + https://www.pythontutorial.net/python-unit-testing/python-unittest-mock/
    - However, sometimes the resources doesn't need to be part of the test and sometimes we need to test against live resources especially in integration tests — https://blog.boot.dev/clean-code/writing-good-unit-tests-dont-mock-database-connections/#have-some-spice

Git

- In case of emergency: https://ohshitgit.com/ — `git reflog` in particular will save your bacon if you accidentally lose all (visible to you) references to <u>commited</u> changes
- HEAD is a pointer to the parent of your next commit

- A branch is a pointer to a commit, provided git status says you are on a branch that pointer will be updated along with HEAD as you make changes
- Each commit has a hash that identifies it — alongside the message and changes, each commit contains the hash of its parent(s) forming a directed acyclic graph of commits
- To minimise synchronisation issues and so the risk of merge conflicts, separable work should be done on different branches (and as few people working on each feature branch as possible) — https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow
    - Once a feature has been entirely developed and tested, the branch can be merged into main using a pull request (poorly named by GitHub, GitLab calls them merge requests)
        - A pull request (PR) should contain a brief description of the changes and some justification that they are ready to be merged (i.e. evidence of tests)
        - Ideally a PR should be reviewed by someone other than the author but just the act of putting one together and trying to look at it from the perspective of a reviewer can be useful
            - Reviewers should check that code style is reasonable, no tests are failing, and (ideally automated when backend) tests have been added to cover the changes
- Ideally commits should be atomic, that is be small enough to only make one change to enable easy rollbacks and big enough to leave the code in a state as least as good as you found it to keep the history clean — https://www.pauline-vos.nl/atomic-commits/
    - `git add --patch` is useful for maintaining atomicity as you go along: https://nuclearsquid.com/writings/git-add/
- I talk a lot about trying to avoid merge conflicts but as long as they're small they're basically fine
- If you aren't logged into GitHub on git yet, the easiest way since GitHub disabled git password authenticaiton tends to be SSH keys https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent — you will need to switch the clone command from HTTPS to SSH
- Pimp your git!
    - Use https://github.com/Wilfred/difftastic as your difftool — https://difftastic.wilfred.me.uk/git.html#difftastic-by-default
    - Use https://jonas.github.io/tig/ when raw git is too painful
    - Define bash aliases for frequently used commands e.g.
        - alias ga='git add *' — don't forget to make and commit a .gitignore
        - alias gc='git commit'
        - alias gca='git commit --amend --no-edit'
        - alias gd='git diff'
        - alias gl='git log --oneline --graph'
        - alias gp='git push'
        - alias gpf='git push --force-with-lease'
        - alias grc='git rebase --continue'
        - alias gri='git rebase -i'
        - alias gs='git status'
    - Config your git (https://git-scm.com/docs/git-config) e.g.
      `git config --global user.name YOUR_NAME_FOR_COMMITS`

```
    git config --global core.editor YOUR_PREFFERED_EDITOR
    git config --global fetch.fsckObjects true
    git config --global fetch.prune true
    git config --global fetch.all true
    git config --global push.autoSetupRemote true
    git config --global submodule.recurse true
    git config --global diff.external difft
    git config --global difftool.prompt false
    git config --global pager.difftool true
```

Commands

- Double check your current state with `git status` before pretty much every git command you use!
- To create and switch to a new branch b2 pointing to start with at current HEAD (so you should switch to main before this) : `git switch -c b2`
    - You will need to use `git push --set-upstream origin b2` for the first push (if you haven't used config to enable `autoSetupRemote)` but git will tell you this if you forget
- To start working on an existing branch b: `git switch b`
- Don't forget `git pull` to fetch the latest changes (for all branches) from github and apply them to your local files (for your current branch) before you start working on it or you increase the risk of merge conflicts!
    - Pull whenever you start work or switch to a different branch
- To see how your (not yet staged for commit) local copies of files differ from the versions at HEAD: `git diff`
- To add f1.py f2.png to the next commit : `git add f1.py f2.png`
    - To undo this: `git restore --staged f1.py f2.png`
- To see how your staged for commit local copies of files differ from the versions at HEAD: `git diff --cached`
- To make a commit: `git commit`
- To view changes made in commit abc12: `git show abc12`
- To update github with your commits on the current branch: `git push`
- To add more changes (or change the commit message) to a commit with no children: `git commit --amend --date=now`
    - Try to avoid amending a pushed commit, especially if it's possible someone has started working on top of them!
        - If you are amending pushed commits you will need to do your next push with `git push --force-with-lease`
            - --force-with-lease is safer than --force in that it checks that no one has pushed commits that descend from it, whenever I refer to a force push I mean `--force-with-lease`
        - If someone has force pushed to a branch b2 and you have made commits on top of the commit(s) they have amended: Instead of `git pull` do `git pull --rebase` and hope for no merge conflicts
- To amend a commit `abc12` that may have children (and implicitly amend its descendants as they will have a different parent and potentially different diffs): `git rebase -i abc12~` and hope the changes made by descendants of abc12 don't conflict with your new changes

    - Rewriting history sounds complicated and can create a mess if you are not careful but the instructions given by git when you run these commands are reasonably good
    - If you have uncommitted changes you will likely need to resolve this before you can rebase — use `git stash` to lose your changes from your local files but push them onto a stack in git for safe keeping then `git stash pop` to reapply them once you finish your rebase
    - E.g. Change abc12 from pick to edit, git will rewind head and your local files to abc12, make your changes, `git commit --amend, git rebase --continue`
        - Useful things that aren't edit: `squash, reword`, reordering the commits in the file
    - If you have pushed the commit you amended be very mindful of any other people currently working on the branch before doing this and you will need to force push
- To view commit hashes, authors, and times on current branch: `git log`
    - To view across all branches: `git log —-all`
    - For pretty picture: `git log --all --decorate --oneline --graph`
- To move HEAD and branch back to abc12 without affecting your local files: `git reset abc12`
- To move HEAD and branch back to abc12 <u>and overwrite your local files that existed there to reflect their state at abc12</u>: `git reset --hard abc12`
- To <u>overwrite</u> your local version of a particular file f1 with it at commit abc12 (e.g. HEAD if you want to lose your uncommitted changes to f1): `git checkout abc12 -- f1`
- You may find you need to incorporate into a branch b changes that have been merged into main since b was made: `git switch b` then `git rebase main`
    - This will alter all commits that are on the branch and not on main to be descendants of the most recent commit to main, hope there are no conflicts!
        - You will need to force push so be very mindful of any other people currently working on the branch
- If a commit does conflict during a rebase, git will pause with HEAD and local files there and tell you — modify the file(s) to fix the conflict (search for conflict markers >>> or === or <<<) then `git commit --amend` and `git rebase --continue`
- Further reading:
    - Useful but not essential git commands (git bisect and git cherry-pick): https://www.pauline-vos.nl/git-legit-cheatsheet/
    - To get familiar with rebase and cherry-pick: https://learngitbranching.js.org/
    - The official manual if you don't like my approach: https://git-scm.com/docs/user-manual#Developing-With-git

Deployment (for free using Github Student Pack)

- FAIR WARNING: THIS GENUINELY IS COMPLETELY OUT OF SCOPE FOR THE COURSEWORK BUT IT IS VERY COOL TO HAVE AND WILL GIVE YOU SOME CLOUD EXPERIENCE
- https://medium.com/@hobegi/flask-app-on-digital-ocean-9b6db466079e + https://www.digitalocean.com/github-students
- https://www.name.com/partner/github-students