Git

An introductory knowledge

What is Git?

- Git is a Version Control System (VCS) that allows users to track and manage changes to file repositories
- It was created in 2005 by Linus Torvalds to manage the creation of the Linux kernel (as needs were not met by other VCS's at the time, Linus created his own)

Basic Mechanisms of Git

- Git effectively stores all changes to a repository (repo) when users create “snapshots” of the current state of the project
- At its heart, Git uses the mechanisms behind the built-in `diff` and `patch` commands
  - `diff` takes in two files and outputs the differences between them
  - `patch` takes in a file and the output of a `diff` involving that file and changes the file according to the `diff`
- Git can be thought of as storage of `diff` outputs that are executed each time the user takes a “snapshot”
  - These stored `diff` outputs can, in turn, be used to revert projects to previous versions via the `patch` command

First Steps with Git

- Set name and email
  - `git config –global user.name “NAME”`
  - `git config –global user.email “EMAIL”`
- Initialize a new Git repo in a directory
  - `git init`
- Clone a git repo in a directory
  - `git clone [URL]`

Basic Git Workflow

- Three-step process
  1. A file in the **working tree** (current state of the directory that exists in a computer) is changed
  2. For Git to track those changes, that file is added to the **staging area** (collection of files to be used in the next “snapshot”)
  3. Once all changed files that a user wishes to include are in the staging area, the user takes a “snapshot” of this new state of the project by doing a **commit**

Example

1. Initialize a Git repo in “git_practice”

```
C:\Users\joshs\Desktop\Userful\git_practice>git init
Initialized empty Git repository in C:/Users/joshs/Desktop/Userful/git_practice/.git/

C:\Users\joshs\Desktop\Userful\git_practice>
```

2. A call to git status shows the only file in the directory, “git_practice.py”, is currently not being tracked by Git

```
C:\Users\joshs\Desktop\Userful\git_practice>git status
On branch master

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)

C:\Users\joshs\Desktop\Userful\git_practice>
```

3. To track the file, add it to the staging area using `git add`

```
C:\Users\joshs\Desktop\Userful\git_practice>git add git_practice.py

C:\Users\joshs\Desktop\Userful\git_practice>git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   git_practice.py


C:\Users\joshs\Desktop\Userful\git_practice>
```

4. Commit the new changes to the repository (with a small commit message)

```
C:\Users\joshs\Desktop\Userful\git_practice>git commit -m "Added git_practice.py"
[master (root-commit) 2f3b97b] Added git_practice.py
 1 file changed, 1 insertion(+)
 create mode 100644 git_practice.py

C:\Users\joshs\Desktop\Userful\git_practice>
```

5. New commit is added! (shown using git log)

```
C:\Users\joshs\Desktop\Userful\git_practice>git log
commit 2f3b97b3ef4e17064f52af16851e22993d159885 (HEAD -> master)
Author: Josh Strand <josh.strand@userful.com>
Date:   Mon Aug 8 08:41:00 2022 -0400

    Added git_practice.py

C:\Users\joshs\Desktop\Userful\git_practice>
```

Commit Messages

- Long commit messages can be made using git commit without –m flag
- Format may vary by company, but generally should have a short summary followed by a new paragraph with a long summary of changes made in the commit
- The more descriptive a user is when making a commit, the easier it is for that user to understand what they were doing at this time in the future

Extra Git Commands

- Skipping the staging area:
  - `git commit -a` will automatically stage and commit all modified and deleted files (but not new files)
- Moving files:
  - `git mv [file to be moved] [desired location]`
- Removing files:
  - `git rm [file to be deleted]`
- Renaming files:
  - `git mv [file to be renamed] [new name]`

Undoing Things Before Committing

- To unstage something before committing
  - `git reset HEAD [file to be unstaged]`

- NOTE: “HEAD” is used to identify the current, checked out snapshot in the Git directory

Amending Commits

- To amend the previous commit, stage changes in the staging area to be consistent with desired commit, then run:
  - `git commit --amend`

- Restrictions:
  - Should not be used on shared repos, as it may lead to confusion
  - Can only be used on previous commit

Rollbacks

- To effectively change previous commits, or to revert back to a previous project version, a commit must be reverted
- A revert is a new commit that contains the changes needed so that the working tree looks like a previous commit
  - This, as opposed to deleting commits or amending them, keeps a more consistent and complete record of changes

- To revert to an older commit:
  - `git revert [commit id]`

Branching

- Git offers an easy way to test experimental features without having to edit the Master branch (the main project)
- **Branches** are pathways of commits that diverge from the master branch that can be later **merged** into it

- To create a new branch:
  - `git branch [new branch]`
- To switch to a branch:
  - `git checkout [branch]`
- To create a new branch and switch to it:
  - `git checkout –b [new branch]`

Example

1. Look at current list of branches in the project:

```
C:\Users\joshs\Desktop\Userful\git_practice>git branch
* master

C:\Users\joshs\Desktop\Userful\git_practice>
```

2. Create a new branch

```
C:\Users\joshs\Desktop\Userful\git_practice>git branch new-branch

C:\Users\joshs\Desktop\Userful\git_practice>git branch
* master
  new-branch

C:\Users\joshs\Desktop\Userful\git_practice>
```

3. Switch to it

```
C:\Users\joshs\Desktop\Userful\git_practice>git checkout new-branch
Switched to branch 'new-branch'

C:\Users\joshs\Desktop\Userful\git_practice>git branch
  master
* new-branch

C:\Users\joshs\Desktop\Userful\git_practice>
```

4. This branch can now be added to and committed to without affecting the master branch. HEAD is now the current, checked-out commit of new-branch

Merging

- When a user would like to combine a branched commit with the main project, a merge can be done
- Two types of merges:
  - Fast-forward merge: when the current, checked-out branch contains all the commits of the master branch (branches do not diverge). Git updates master branch to point to merged branch; no actual merging
  - Three-way merge: when the history of the branches has diverged, such as when commits are made on both branches after they split. In this case, Git combines the two commits on the branch tips with the most recent shared ancestor (hence the “three-way”)

Example of a three-way merge

- Current, committed git_practice.py from first example:

git_practice.py
```
1   print("hello")
2
```

- New commit on new-branch:

git_practice.py
```
1   print("hello")
2
3   # after switch to new branch
4   print("I am on a new branch now")
```

- New commit on master:

git_practice.py
```
1   print("hello")
2
3   # For branch master:
4   print ("On branch master")
```

1. Attempting merge from diverged branches:

```
C:\Users\joshs\Desktop\Userful\git_practice>git merge new-branch
Auto-merging git_practice.py
CONFLICT (content): Merge conflict in git_practice.py
Automatic merge failed; fix conflicts and then commit the result.

C:\Users\joshs\Desktop\Userful\git_practice>
```

2. Checking information git added to git_practice.py

git_practice.py

```
print("hello")

<<<<<<< HEAD
# For branch master:
print ("On branch master")
=======
# after switch to new branch
print("I am on a new branch now")
>>>>>>> new-branch

```

3. To combine both, merge info is deleted and file is saved

git_practice.py

```
print("hello")


# For branch master:
print ("On branch master")

# after switch to new branch
print("I am on a new branch now")

```

4. Merge is completed by adding edited git_practice.py and committing

```
C:\Users\joshs\Desktop\Userful\git_practice>git merge new-branch
Auto-merging git_practice.py
CONFLICT (content): Merge conflict in git_practice.py
Automatic merge failed; fix conflicts and then commit the result.

C:\Users\joshs\Desktop\Userful\git_practice>git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   git_practice.py

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

C:\Users\joshs\Desktop\Userful\git_practice>git add git_practice.py

C:\Users\joshs\Desktop\Userful\git_practice>git commit -m "merged pathways"
[master f7f2f00] merged pathways
```

5. Checking visualized graph to see branches have been merged

```
C:\Users\joshs\Desktop\Userful\git_practice>git log --graph --oneline
*   f7f2f00 (HEAD -> master) merged pathways
|\
| * 1403da6 (new-branch) New branch first commit
* | f82bf54 New commit on master after divergence
|/
* 2f3b97b Added git_practice.py

C:\Users\joshs\Desktop\Userful\git_practice>
```