Git

What is Git and why bother?

Git is a version control system (VCS). A VCS tracks a project and all of its files. Meaningful changes to a project's files are recorded by the VCS, and its users can switch between all of the recorded versions. Using a VCS is all about making continuous and small improvements to our projects, while knowing we can go back to previous versions if we do not like the changes. VCS's also allow for multiple people to collaborate on the same project, facilitating the distribution and maintenance of all files related to the project.

In this class, we will use a specific VCS: Git. You will use Git to track and submit your projects, and, in the process, you will learn an important tool that allows for multiple people across the world to work on the same projects at the same time (for example, see Linux, which has thousands of contributors, both individuals and companies).

Basics of a Git Repository

A Git repository is a collection of all the versions of your project. Each of these versions is called a commit. A commit is a snapshot of your entire project at a point in time, and the repository collects these commits through time.

Even though Git stores a history of all commits of your project, it is very efficient in doing so and can be used both with small and large projects (the linux project is an example of a project with a huge history and a few gigabytes of size). If you have a Git repository for a project, you have all versions of this project available to you, and you can move back and forth between versions at any time.

All commits (snapshots) belong to a branch. A branch is an independent line of development of the project. The default line is called a `master` branch, and it is always available. New branches are created to work on different parts of the project. In this course, you will most of the time work on the `master` branch, since the project exercises are dependent on each other. However, you could, for example, open a new branch to work on an exercise that is independent from the others.

When you finish the work you set out to do on a branch, you can merge that branch back to the `master` branch. To do so, you create a pull request. A pull request is a request to have your branch merged with the master branch. In a team setting, a pull request would involve discussions with the rest of your team and having your teammates testing what you did.

Git has a command line interface, but graphical user interfaces (GUI) are also available (like GitHub Desktop and Sourcetree). However, we will use the command line interface. The reasons are:

- If you say you know Git it is assumed that you have the command line skills
- Git commands via the command line can be automated
- Fast and easy

The GUI for Git can be useful at times. They provide a good visual representation of the branches of a repository and some interactive commands can be easier to use with a GUI. Remember to choose the right tool for the right task.

To understand how Git works you must know the 4 locations Git uses:

1. Working Tree: this is the main directory of your project. It starts out empty or with the files of a commit (snapshot). You can modify the files in the working tree.
2. Staging Area (Index): this place stores the files that you plan on including in your next commit (meaningful snapshot of project).
3. Local Repository: it is a hidden directory (`.git/`) inside your main folder, it contains the history of all commits.
4. Remote Repository: contains the commits of the project; usually the source of truth of the project.

The basic Git workflow goes through all of the 4 locations above. You start by working on the files in your working tree until you have a meaningful change to the project. Then you stage the files that are related to the change in the project, and you create a commit of the project, which is stored in your local repository along with a history of all your commits. You then synchronize the changes with a remote repository, so that your project is available to others.

Working Tree   →   Staging Area   →   Local Repository   ↔   Remote Repository

`git add`   `git commit`   →   `git push`
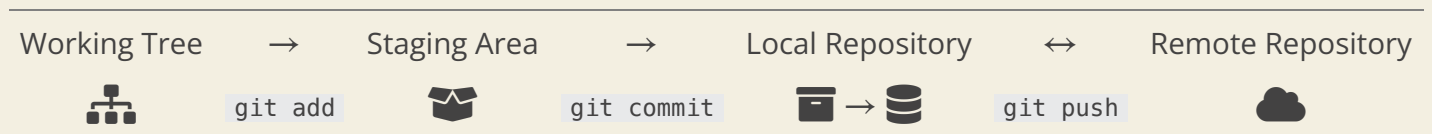
Syntax for Git commands

The syntax for all Git commands follows:

```
git [command] [--flags] [arguments]
```

You can find the documentation for Git and for specific commands via:

```
git help [command]          # full documentation
git [command] -h            # concise help
```

If you never used a command before it is helpful to use `git help [command]` and read the description of the command.

First-Time Git Setup

The first command we will use is `config` . It is used to setup certain variables that customize how Git behaves. The basic syntax for this command is as follows:

```
git config [--local|--global|--system] <key> [<value>]
# --system: applies to every repository and all users of computer
# --global: applies to every repository for your current user
# --local: applies only to the current repository and current user
```

Let's start by adding your username and email:

```
git config --global user.name "Guilherme"
git config --global user.email "gfs8@duke.edu"
# setup an editor to edit messages
```

This allows Git to store your information when creating commits. When you create commits of your project, Git will ask you to enter a message explaining what is being committed and why. This message is created in a text editor, and we can define which editor to use by default:

```
git config --global core.editor nano
# other options: emacs, vim
# can also specify full path to editor
git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -nosession"
```

You can check the current value of the configuration variables with:

```
git config user.name          # value of a single variable
git config --list
```

Creating a Repository

The first project will cover step by step how to set up the repository for the projects, but here we discuss how to set up repositories in general. There are two ways to create a repository:

1. Start one from scratch in a directory
2. Clone an existing repository

From Scratch

To start a repository from scratch, go to the terminal and either create a new folder for your repository, or move to an existing folder where you would like to start a repository:

```
mkdir ~/Repositories        # ~/ represents the user folder
cd ~/Repositories           # move to Repositories folder
mkdir TestRepo              # create a folder called TestRepo
cd TestRepo                 # move into the folder
git init                    # initialize a git repository
```

The last command creates a new Git repository that will be responsible for storing the history of your project. The repository is kept on a hidden folder named `.git`. You can verify the creation of this folder by listing the directory contents:

```
ls -a                       # -a to show names beginning with a dot
info ls                     # to get description of the ls command
```

You can now add files to the working tree, and then stage and commit those files. For example:

```
touch README.md                     # creates a markdown file
git add README.md            # adds the file to the staging area
git commit -m "Initial commit."      # creates a commit
```

From an existing remote repository

To start a repository based on a repository that already exists (remote repository) we use the `git clone` command, which clones a remote repository to our computer. Remote repositories are usually hosted on a website, like GitHub and Bitbucket, and can be private or public. In a corporate environment repositories can be hosted on-premise (usually with GitHub enterprise, Bitbucket Server, Gogs, Gitlab or others).

In this course, projects will be hosted on GitHub (you will receive an email with an invitation link soon). Remote repositories have an address given by a URL. For example, consider this test repository. If you go to the link above, you will see a green button saying `Clone or download` . Clicking on that button reveals the URL to the repository. Notice that the name of the repository ends with `.git` . We can use that URL to make a clone of the repository.

```
cd ~/Repositories
git clone <url/to/projectname.git> [localprojectname]
# if no [localprojectname], then git creates a project with name projectname
git clone https://github.com/Salompas/test.git
```

This will create a folder with the name `test` and copy the contents of the repository to your computer. This clone contains all history of the project.

Next, let's discuss in detail how to use the Working Tree and the Staging Area.

The Working Tree and the Staging Area

The working tree is the main directory of your repository. Let's create a new repository and a couple of files in the working tree:

```
cd ~/Repositories
mkdir StagingAreaBasics          # create a folder for our new repo
cd StagingAreaBasics/
touch README.md                  # add a README file for the repository
touch python.py                  # add a Python file
touch report.tex         # add a LaTeX file
git init                 # initialize the repository
```

When we initialize the repository in this folder, Git will create the `.git` folder, but it will not have any commits yet. The working directory, `StagingAreaBasics/` contains our files and the repository folder.

In a typical workflow you would start editing your files until you have created a meaningful addition to your project. In the case of our projects, you would edit the files until you made all the computations required for an exercise, for example. Let's assume we have made those changes. At this point we would like to commit the state of our project. Right now, the files are in the Working Tree, and we need to let Git know what files we want to include in the next commit, that is what the Staging Area is for.

We first use the `git status` command to see the status of the files in the Working Tree from the point of view of the Git repository:

```
> git status                    # show the working tree status
On branch master

No commits yet

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

        README.md
```

```
        main.m
        report.tex

  nothing added to commit but untracked files present (use "git add" to track)
  > git status -s                          # -s for concise status information
  ?? README.md
  ?? main.m
  ?? report.tex
```

The first line shows that we are on the branch `master` . Remember that we can create branches to work on independent parts of the projects, and the default branch is called `master` . The second line shows that there are no commits stored in the repository yet. The third line says we have <u>untracked files</u>. Files in a Working Tree can have 4 states:

1. Untracked: these are files that the Git repository is not tracking, either because we haven't told it to track these files yet, or just because we do not want them tracked.
2. Unmodified: these are files that are being tracked, but have not been modified yet.
3. Modified: these are files that are being tracked and that have been modified.
4. Staged: these are files that we put on the Staging Area and that we plan to use for our next commit (snapshot).

Notice that the more concise `git status -s` indicates untracked files with a preceding `??` .

Let's tell Git to track the `README.md` file.

```
  > git add README.md
  > git status
  On branch master

  No commits yet

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

          new file:   README.md

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

          main.m
          report.tex

  > git status -s
  A  README.md
  ?? main.m
  ?? report.tex
```

Git has added the `README.md` file to the Staging Area, indicating that we plan on including this file to our next commit (snapshot) of the project. This information is displayed under `Changes to be committed:` , or by the preceding `A` in the concise status display.

The `README.md` file contains information about the repository, and usually it explains what is the purpose of the repository. This file is automatically displayed on GitHub as the front page of the repository. So, let's modify this file to make it more meaningful.

```
  > emacs README.md      # add some text to the file
```

When we move a file to the Staging Area, Git saves the state of the file at the time we ran the `git add` command. If we then update the file, Git will not have the updated version in the Staging Area, but only the ver-

sion we added. We can verify that this is the case by getting the status of the files in our Working Tree:

```
> git status
On branch master

No commits yet

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

        new file:   README.md

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:   README.md

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

        #README.md#
        main.m
        report.tex

> git status -s
AM README.md
?? #README.md#
?? main.m
?? report.tex
```

Notice that we have `README.md` under `Changes to be committed`, but it also shows up under `Changes not staged for commit`. In the concise status this is shown by a preceding `M`. We can update the version of `README.md` in the staging area by running `git add README.md` again:

```
> git add README.md
> git status
On branch master

No commits yet

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

        new file:   README.md

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

        #README.md#
        main.m
        report.tex
```

Remember, the Staging Area is like a box where we put copies of the files relevant for the modifications we plan on saving. When we commit, it is like closing that box and storing it in a shelf.

To commit, we use the `git commit` command:

```
> git commit                     # will open the editor to add a commit message
> git commit -m "Commit message."    # pass message as argument (only for short messages)
[master (root-commit) 8b8c33f] Initial commit.
 1 file changed, 2 insertions(+)
 create mode 100644 README.md
```

Git has now stored the commit (snapshot) of the project on the database.

Let's look at the status of the working tree again:

```
> git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        #README.md#
        main.m
        report.tex
```

Notice that the file `README.md` is not under `Untracked files` anymore, nor it is under `Changes to be committed`, since it already has been committed. The `README.md` file is now a tracked and unmodified file, and will not show up under `git status` until we modify it again.

Notice that a new file (`#README.md#`) showed up. This is a backup file that emacs automatically generated when we were editing the original file. It has nothing to do with Git. For Git it is an untracked file, so it shows up in the status of the Working Tree. We will see how to tell Git to ignore these files later on.

Now, suppose we made progress on the project and made changes in to the `main.m` and `report.tex` files. Let's stage and commit those files.

```
> git add main.m
> git add report.tex
> git commit -m "Made computations for Exercise 1 and wrote analysis on report."
[master 8b47a1c] Made computations for Exercise 1 and wrote analysis on report.
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 main.m
 create mode 100644 report.tex
```

History of Commits

We can see the history of all commits by using the `git log` command:

```
> git log
commit 8b47a1c1f34a24ab59c8fb1a19c3df2e8a00c7ac (HEAD -> master)
Author: Guilherme Salome <guilhermesalome@gmail.com>
Date:   Tue Aug 28 10:09:30 2018 -0400

    Made computations for Exercise 1 and wrote analysis on report.

commit 8b8c33f977e43668ac46c7676267685ae6f1d0ec
Author: Guilherme Salome <guilhermesalome@gmail.com>
Date:   Tue Aug 28 10:00:12 2018 -0400

    Initial commit.
```

It shows the commits in reverse chronological order (most recent to oldest). Each commit has an identifying number, a field showing the author and date of the commit, and a paragraph with the commit message. It is important to have good commit messages so that you can identify versions of your project more easily.

It is possible to limit the number of commits displayed:

```
git log -2                          # will only show the last 2 commits
```

It is also possible to see what were the incremental changes from one commit to another with the `-p|--patch` option:

```
git log -2 -p              # shows incremental changes of previous 2 commits
```

The `--stat` option adds a summary of the changes when displaying the log:

```
> git log --stat
commit 8b47a1c1f34a24ab59c8fb1a19c3df2e8a00c7ac (HEAD -> master)
Author: Guilherme Salome <guilhermesalome@gmail.com>
Date:   Tue Aug 28 10:09:30 2018 -0400

    Made computations for Exercise 1 and wrote analysis on report.

 main.m     | 0
 report.tex | 0
 2 files changed, 0 insertions(+), 0 deletions(-)

commit 8b8c33f977e43668ac46c7676267685ae6f1d0ec
Author: Guilherme Salome <guilhermesalome@gmail.com>
Date:   Tue Aug 28 10:00:12 2018 -0400

    Initial commit.

 README.md | 2 ++
 1 file changed, 2 insertions(+)
```

The `git log` command can take the option `--pretty` which allows us to decide what we want to see in the log. There are a few pre-built options, like `oneline`, `short`, `full` and `fuller`.

```
git log --pretty=oneline
```

We can add a small visual representation with the `--graph` option:

```
git log --pretty=oneline --graph
```

For more options on controlling the log output (like selecting log by dates) please see the reference.

Undoing Stages

Let's say you discover you can add all files from the Working Tree to the Staging Area with the command `git add -A`, and accidentally end up adding a file that you do not want to track, say `#README.md#`. In order to unstage this file, you can use the command `git reset HEAD #README.md#`

```
> git status -s
A  #README.md#
> git reset HEAD #README.md#        # HEAD refers to the last commit in the repository.
```

Now suppose you make modifications to a file in the Working Tree, but regret the changes and want to go back to how the file was at the last commit. Then you can use the command `git checkout -- <filename>`. Git will substitute the file in the Working Tree for the version of the file in the last commit.

A common problem is committing too early, and then you realize you forgot to stage an important file or missed something in the commit message. To fix this problem, first stage the files you missed (if any) and create a new commit with the option `--amend`. When you run `git commit --amend`, the editor will open up for you to update the commit message. This amended commit will <u>replace</u> the previous commit.

Removing Files

To remove a file from the repository <u>and</u> from the directory use the command `git rm [filename]`. This command immediately deletes the file from your folder, and also untracks this file on the next commit. If you want to keep the file in the directory but have it not be tracked anymore, then use `git rm --cached [filename]`. To rename a file use `git mv [filename] [new_filename]`.

Ignoring Files

Let's go back to our example.

```
> git status -s
?? #README.md#
```

The file `#README.md#` automatically showed up. We do not care about it, since it is just a backup in case the editor freezes, and it will show up every time we make changes to our files. So, we do not want to have to delete it by hand every time, and we do not want it to show up in the output of `git status`.

To have Git ignore these files, we will create a `.gitignore` file.

```
touch .gitignore              # add the line \#*# to the file
```

This tells Git to ignore all files that start with a hash and end with a hash. Now, if you run `git status -s`, the file `#README.md#` will not be displayed anymore.

We will add to this file when we start working with Latex. This will be necessary because compiling `.pdf` files with Latex generates lots of intermediary files we are not interested in tracking, nor in deleting by hand.

Lastly, let's talk about using a Remote Repository.

Working with a Local and a Remote Repository

We previously used a remote repository with Git to create a new repository on our local computer. Let's leave our current repository and move back to the one we cloned:

```
cd ~/Repositories/test/           # this repository was created with git clone
```

When Git clones a remote repository, it associates the local repository with the remote version. We can use the command `git remote` to find the information Git saved about the remote repository:

```
> git remote --verbose            # same as -v
```

```
origin          https://github.com/Salompas/test.git (fetch)
origin          https://github.com/Salompas/test.git (push)
```

The URLs above are the address from where the remote repository came from. Git uses <u>origin</u> as an <u>alias</u> (name) to the URL of the remote repository. Whenever you see this keyword, it is the same as if you were typing the URL of the remote repository.

Now, you could have started a repository on your local machine (as we did) and want to send it to a remote repository. To do so, we need to associate the local repository with the URL of the remote repository. Let's first move back to the repository we created locally:

```
cd ~/Repositories/StagingAreaBasics/
```

Now, go to GitHub and create a New and Empty repository, and then copy the repository URL. We will use this URL to associate our local repository with the remote repository on GitHub:

```
> git remote add <name> <url> # adds a remote <url> under the alias <name>
> git remote add origin https://github.com/Salompas/StagingAreaBasics.git
> git remote -v
origin          https://github.com/Salompas/StagingAreaBasics.git (fetch)
origin          https://github.com/Salompas/StagingAreaBasics.git (push)
```

Now, we can push the commits from the local repository to the remote repository using the `git push` command.

```
git push [-u] [<repository>] [<branch>]
# <repository>: name (alias) or url, like origin; pushes to this repo
# -u: track this branch; sets the relationship between local and remote repos
# <branch>: name of the branch to push, usually master
git push -u origin master # might ask for username and password
```

Git Workflow for Assignments and Summary of Commands

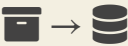Now, let's discuss the Git workflow I expect you to use when solving the projects.

- On your computer, open a terminal and create a new directory called `PythonCourse` where you will clone the remote repository:

```
1: mkdir ~/PythonCourse
2: cd ~/PythonCourse
3: git clone url_to_your_assigned_remote_repository
4: # will create a folder called assignments-YourGithubUsername
```

- Work on solving the assignment exercises.
- Whenever you solve an exercise, use `git add <filename>` to add all relevant files to the staging area.
- Commit the relevant files with `git commit -m "Your commit message"`.
- Push the changes to the remote repository with `git push origin master`

The assignments are due on August 11th by 11 pm. You should push the full contents of your assignments folder to GitHub before then, and it should include a `report.pdf` file for each of the assignments.

The table below summarizes the commands:

Working Tree	→	Staging Area	→	Local Repository	↔	Remote Repository
`git status`	`git add <filename>`	`git rm <file-name>`	`git commit`	`git log -2 -p`	`git push -u origin master`	
`git status -s`		`git rm --cached <file-name>`	`git commit -m "Mes-sage"`	`git log --pret-ty=oneline -graph`		
`.gitignore`		`git mv <file-name> <new filename>`	`git commit --amend`	`git log --stat`		
		`git reset HEAD <filename>`		`git remote -v`		
		`git checkout -- <filename>`		`git remote add origin path/to/repo.git`		
				`git init`		

Reference

A good reference book for Git is Pro Git (free download).