Advanced Python for Data Science

Shell Basics and Git

Contents

- Shell
- Git
- GitHub

References

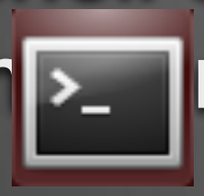
- https://swcarpentry.github.io/shell-novice/02-filedir/
- Pro Git by Scott Chacon, Apress, 2009, ISBN 978-1430218333, http://git-scm.com/book
- Atlassian Git Tutorial, https://www.atlassian.com/git/tutorials
- GitHub Help, https://help.github.com
- http://www.tldp.org/LDP/Bash-Beginners-Guide/html/

The Shell

About the Shell

- The “shell” is a program that takes your commands from the keyboard and gives them to the operating system
- On Linux the shell program is usually called `bash` (for Bourne Again Shell), but there are others
- Commands are just the names of programs
- Commands take zero or more arguments (separated by spaces) which are passed to the program
- The shell recognizes some special characters and may modify arguments before they are passed to the program

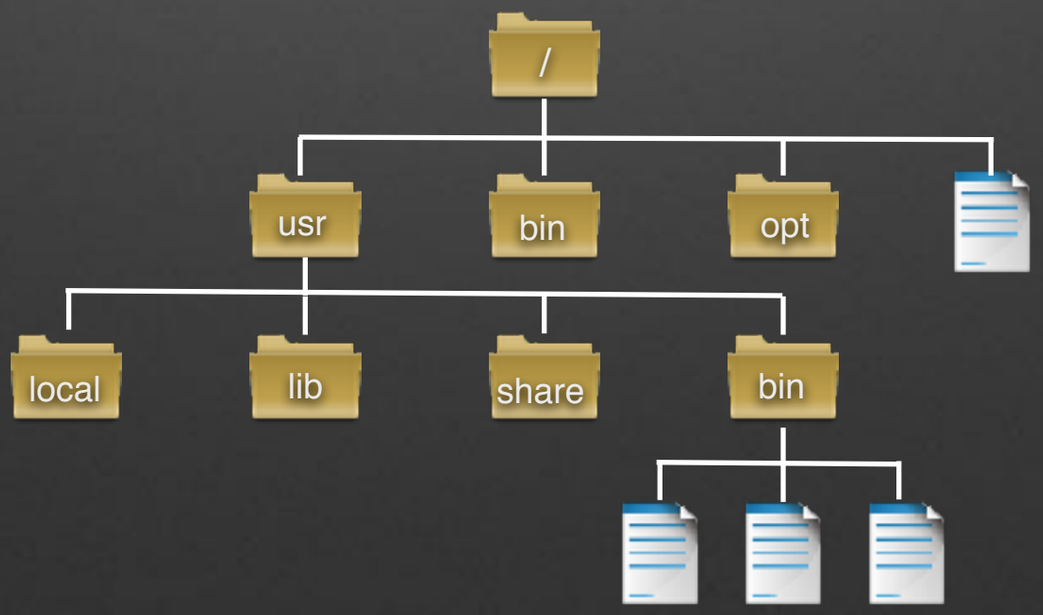
Shell Prompt

- The shell displays a prompt to let you know it is waitir  r input
- The prompt message can be set to almost anything
- You can change the prompt message with a command like

```
> PS1=“my new prompt$ “
my new prompt$
```

Files and Directories

- Linux (and Unix) uses the notion of files and directories
  - Files contain data
  - Directories can contain files and other directories (analogous to a folder)
- A *filesystem* is a hierarchical arrangement of files and directories
- The top of the filesystem is called the “root” and is denoted by a single slash “/“



Paths

- A *path* is a list of directories that lead to the location of a file (or directory)
  - A path can be absolute (starts with “/“) or relative to the *current directory*
    - Directory and file names in the path are separated by “/“
    - The special names “.” and “..” are used to denote the current and parent directories respectively
- Examples

  /home/ds-ga-1007 ./eclipse

  /usr/bin/python usr/sbin/../bin/yes

Navigation Commands

pwd

- Show the current directory (also known as the working directory)

cd *path*

- Change the current directory to the directory specified by *path*
- If no path is specified, sets the current directory to your *home directory*

The ls Command

ls

- List the files in the current directory

ls path

- List the contents of the path
- If path is a file, then just list the file

ls -l path

- List the contents of path in long format. This is a very useful command

ls -la path

- List the contents of path including hidden files

Command Format

- Most commands use the format

  command -option … argument …

- *command* is the name of the command
- *-option …* are used to alter the command's behavior
- *argument …* are the "objects" on which the command operates
- Sometimes (particularly Linux) command options are specified with two dashes

  less --help

- Sometimes no dashes are used at all! (e.g. chmod)

Viewing Commands

file path

- Determine the contents of a file before you try to view it

less path

- Display the contents of the file at path
- Make sure path is a text file otherwise you will see garbage on your screen

man *command*

- Display a description of the command and its arguments

Manipulation Commands

cp *from to*

- Copy the file specified by the path from to the path specified by to
- If to is a directory, then the result will be a file called from in the to directory

mv *from to*

- Move (i.e. rename) the file from to the file to
- If to is a directory, then the result will be a file called from in the *to* directory

rm *path*

- Delete the file at location path
- If path is a directory, *it and all its contents* can be removed using the -r option

mkdir *path*

- Create a new directory at path

rmdir *path*

- Remove the directory at location path

File Permissions

- Permissions are used to control access to your files and directories
- The ls -l command can be used to show the permissions of all files in the current directory or a specific *path*
- Permissions are displayed as a string of 10 characters

  drwxr-x---
- Permissions are grouped as follows

File Type (character 1)	Owner Access (characters 2-4)	Group Access (characters 5-7)	Other Access (characters 8-10)
- = regular file d = directory	r = readable w = writable x = executable	r = readable w = writable x = executable	r = readable w = writable x = executable

Modifying Permissions

- The chmod command is used to change the permissions on a file or directory
- The arguments specify the desired access permissions to apply to the file or directory

Entity	Operator	Access Rights
u = owner (user)	+ = grant	r = readable
g = group	- = revoke	w = writable
o = others	= = set	x = executable
a = all of the above		- = no access

- For example, to grant the owner read/write access, the group read access, and others no access

  chmod u=rw,g=r--,o-rwx *path*

Wildcards

- The shell provides special characters to easily specify groups of path names
- Wildcards allow you to select path names based on patterns of characters

*	Matches any characters
?	Matches any single character
[*characters*]	Matches any character that is in the set of characters
[!*characters*]	Matches any character that is not in the set of characters

Wildcard Examples

*	All filenames
g*	All filenames beginning with “g”
b*.txt	All filenames that begin with “b” and end with “.txt”
Data???	Any filename that begins with “Data” followed by exactly three more characters
[abc]*	Any filename that begins with “a”, “b”, or “c” followed by any other characters
Data[!ef]*	Any filename that begins with “Data” followed by any characters that are not “e” or “f” followed by any other characters

Redirecting Input and Output

- Every command is able to read input and send output (even if it chooses not to)
- Normally, input comes from the keyboard, and this is called *standard input*.
- Output is normally sent to the screen. There are two kinds of output, *standard output* and *standard error*.
- It is possible to change where the input comes from and output goes to using shell *redirection*.

Output Redirection

- Output redirection is enabled using the '>' character

```
$ who > who.out
$ ls -l who.out
-rw-rw-r-- 1 ds-ga-1007 ds-ga-1007 92 Feb  3 16:00 who.out
```

- In this case, the output from the who command has been redirected to a file called who.out
- Use the '>>' characters to append the output rather than overwriting the file
- Note that the redirection comes *after* the command

Input Redirection

- Input redirection is enabled using the '<' character
- The cat command just sends its input to its output

```
$ cat < who.out
ds-ga-1007 :0           2015-02-03 15:43 (:0)
ds-ga-1007 pts/0        2015-02-03 15:48 (:0)
```

- In this case, the input for the cat command comes from the file called who.out
- Input and output redirection can be combined into one command

Pipes

- Sometimes it is useful to be able to connect the output of one command to the input of another
- This is done with a *pipe*, which is enabled using the '|' character

```
$ ls -l | wc
      12     101     699
```

- Here, the output from the ls -l command is sent to the input of the wc (word count) command
- Input and output redirection can also be combined with pipes

Shell Scripts

- You can create your own shell commands by placing shell commands in a text file called a *script*
- A shell script must start with the line

  #!/bin/sh
- The remaining lines in the script will be executed one a a time
- Arguments can be passed using special shell variables
  - $1, $2, etc. are replaced with the corresponding argument to the script
  - $# is replaced with the number of arguments
  - $@ is replaced with all the arguments
- Lines beginning with # are comments and will be ignored (apart from the first line)

Shell Variables

- Shell variables are strings
- Variables can be created using the syntax

  name=value
- A variable is referenced using $name
- If name has a special meaning, use ${name}
- If value contains special characters, enclose it in single or double quotes (use single quotes if you want to include a double quote, and vice versa)
- Variables, or in fact any string, can be display using the echo command

  echo "my variable has the value $my_variable"

Shell Programming

- Various control constructs are available

if *list* then *list* elif *list* fi	for *variable* in *word* do *list* done	while *list* do *list* done

- list is a sequence of commands separated by newlines or semicolons

Shell Programming

- The actions depend on the *exit status* of the last command in the list
- An exit status of 0 means success and anything else (usually) means failure
- In order to evaluate a condition, we use the test command
- Returns 0 if the test is successful
- The test command is implied by enclosing it's arguments in [and]

```
if [ "$my_variable" = "some string" ]; then
    echo "hi there"
fi
```

Running Scripts

- A script can be run by using the sh (or bash) command and passing the script as an argument

  sh my_command

- You can also run the command directly by setting the execute permission on the file

  chmod +x my_command

- Now just enter an absolute or relative path to the command

  /…/my_command

  ./my_command

  my_command # if "." is in my PATH

Version Control Using Git

Overview

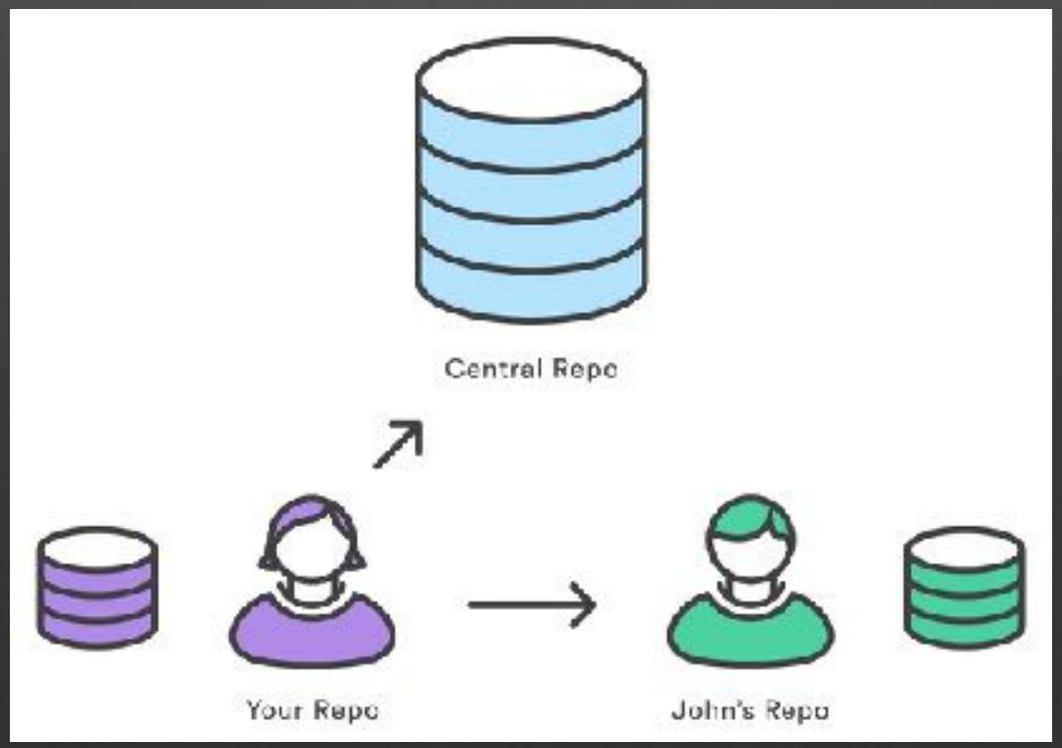
- A *version control system (VCS)* is a system for managing changes to source code over time
- Allows developers to track changes to a large project, and if necessary, revert back to a previous state
- Records *what* the change was, *who* made the change, and *when* the change was made
- The source code, changes, and other information are stored in a *repository*

Types

- There are three main types of VCS
  - **Local** - all developers must use the same computer system that contains a local repository
  - **Centralized** - developers share a single central repository on a remote system
  - **Distributed** - each developer works with their own local repository, and changes are distributed to the separate repositories

Git

- Git is a distributed VCS
- Each developer has an isolated local copy of the repository
- Developers can share changes between any two repositories
- Often there is a “central repository” that contains a master copy of the project, but this is not necessary

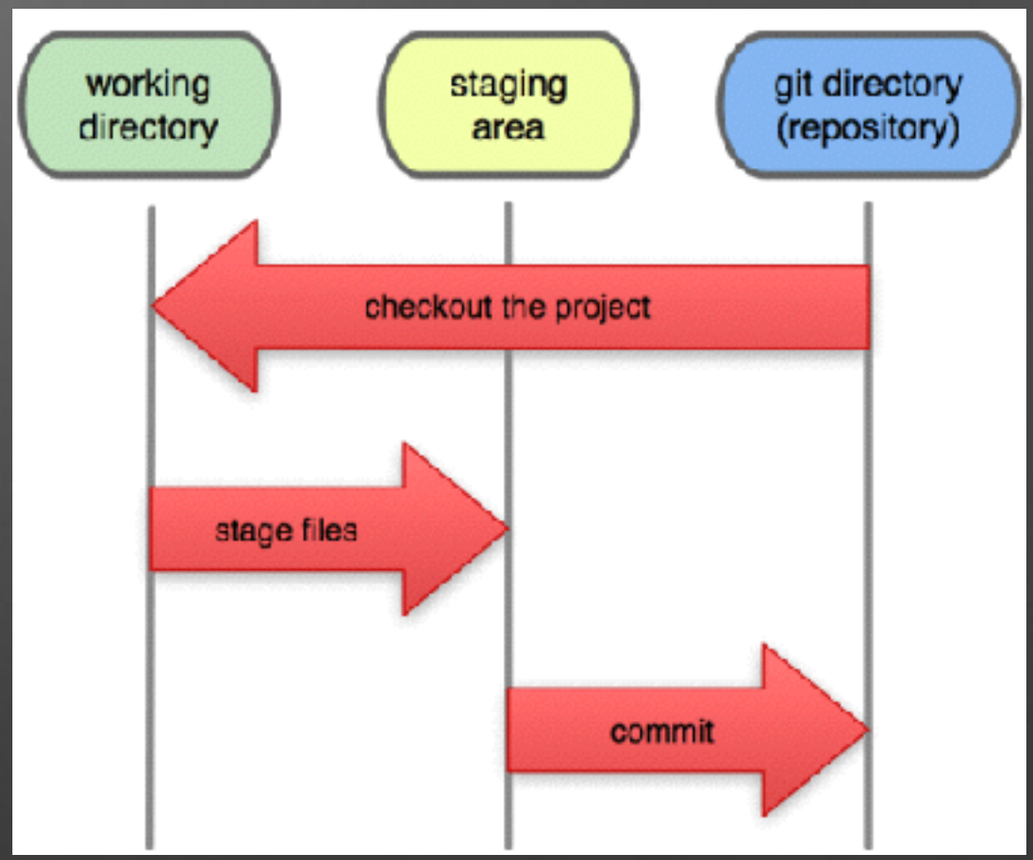


Structure

- There are three main “components” to git
  - The *Git directory,* or repository, is where all the files and other *metadata* reside
  - The *working directory* is one version of the project containing files extracted from the repository. Developers make changes to the files in the working directory.
  - The *staging area* (also know as the *index*) stores temporary information about the changes that will be transferred to the repository

Workflow

1. **Checkout** - Files are extracted from the repository into the working directory. This is usually only done once.
2. Files in the working directory are modified, removed, or added.
3. **Stage** - A snapshot of the modified files is added to the staging area.
4. **Commit** - The snapshot is stored permanently in the Git directory.
5. Steps 2-4 are repeated.



File States

- A file that is in the Git directory is *committed*
- A file that is modified and has been added to the staging area is *staged*
- A file that has been modified since it was checked out but has not been staged, is *modified*
- Files that have not been staged or committed are *untracked*

Installing Git

- First check if Git is already installed

  ds-ga-1007$ git --version

  bash: /usr/bin/git: No such file or directory

- Fetch and install Git

  ds-ga-1007$ sudo apt-get install git

  [sudo] password for ds-ga-1007: python

  …

Setting Up a Repository

- There are three ways to set up a repository
  - Create a new empty repository from the current directory

    git init
  - Create one from an existing (non Git) directory

    git init

    git add .

    git commit -m “Initial commit”
- Copy an existing Git repository (usually from a remote location)

  git clone URL

Configuring a Repository

- A new repository requires configuration before it can be used
- The minimum configuration is to set the user name and email

```
ds-ga-1007$ git config --global user.email "greg.watson@nyc.edu"

ds-ga-1007$ git config --global user.name "Greg Watson"
```

- There are many other configuration options that can be set

Committing Changes

- When changes are made to files, or files are created or removed, they are staged using the command

  git add path

  where path is either a file or a directory (in which case all the files below the directory are added)

- When it is time for the snapshot to be committed to the repository

  git commit

  which will launch a text editor prompting for a commit message

- To supply a commit message without launching an editor, use the command

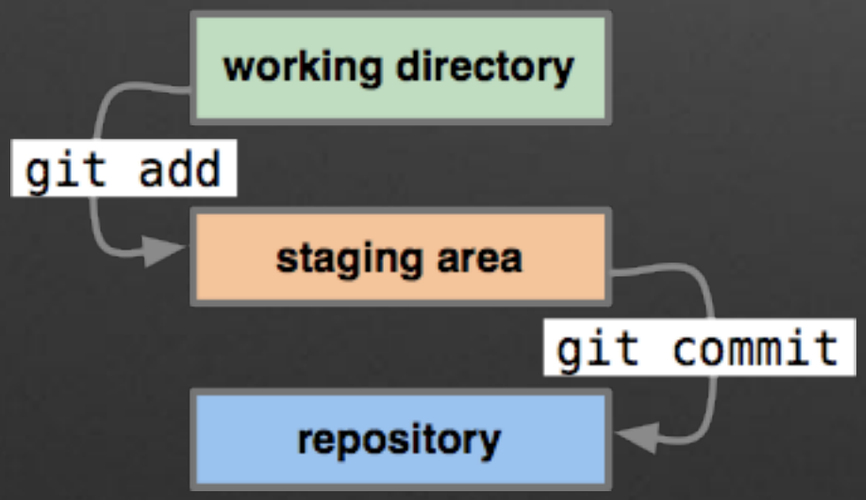
  git commit -m “commit message”

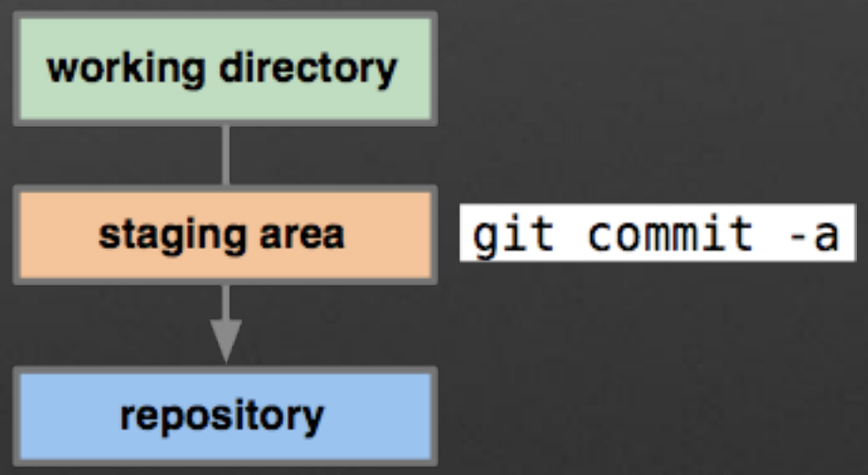
- The add and commit commands can be combined into one (this will *only* work if the file has previously been added with git add)

  git commit -am “commit message”

Why Have a Staging Area?

- Git differs from most VCSs by exposing the staging area
- This allows a subset of modified files to be staged and committed, rather than having to commit all modified files in the working directory
- Can be easily ignored if desired





Inspecting a Repository

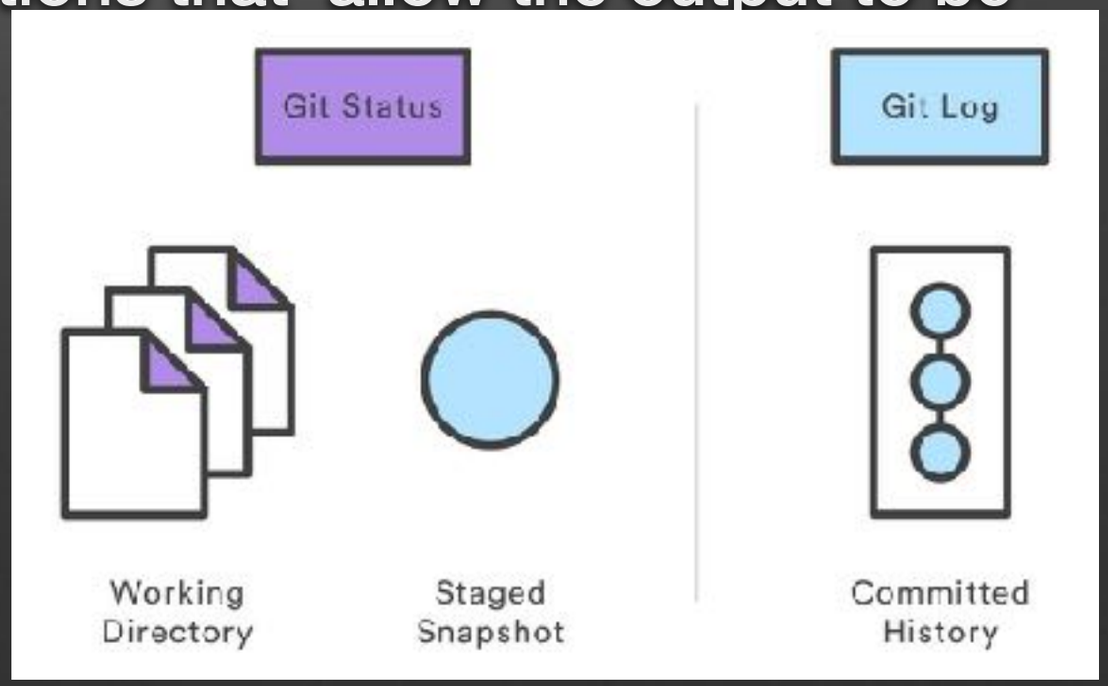
- The main command for displaying the status of the working directory and staging area is

  git status

- To display the history of commits in the repository, use the command

  git log

- The log command has many options that allow the output to be filtered and customized



git status

```
ds-ga-1007$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   yet_another.py
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:   another.py
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hello_world.py
```

staged

modified

untracked

git log

```
ds-ga-1007$ git log
commit 3fd8d3c84439ff2dd6eb27902746940a9fc8a37e
Author: Greg Watson <greg.watson@nyc.edu>
Date:   Fri Sep 12 16:57:04 2014 -0400

    another change

commit fcc6e8fbcf257bca32f9b093230be0ca385d50cd
Author: Greg Watson <greg.watson@nyc.edu>
Date:   Fri Sep 12 16:44:11 2014 -0400

    change to the file
```

commit id

author and date

commit message

Comparing Versions

- To view the differences between the working directory and the staging area

  git diff

- To view the differences between the staging area and the most recent commit

  git diff --cached

- To view the differences between the working directory and the latest commit

  git diff HEAD

- It is also possible to compare with a specific commit id

Undoing Changes

- To remove a file from the staging area, use the command

  git reset path

- To remove all files from the staging area, use the command

  git reset

- To reset the staging area and working directory to match the latest commit

  git reset --hard

  Warning: this will overwrite any changes in the working directory!

Undoing Commits

- If you have committed a snapshot and wish to undo it, use the following command

  git revert commit_id

- Unlike reset, this does not remove the previous commit, rather it appends another commit that reverses the effect
- This is not only safer, it also preserves the history, which is important for integrity and reliable collaboration
- If you *really* know what you are doing, you can also use

  git reset commit_id

  or

  git reset --hard commit_id

Amending a Commit

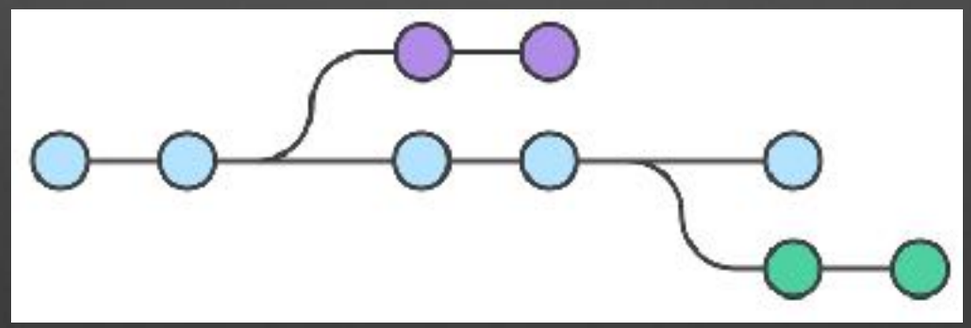
- Git allows you to amend the previous commit
- This is useful if you accidentally left something out of a commit
- First, stage the extra changes

  git add .
- Then commit using the --amend option

  git commit --amend
- If you don’t specify a commit message with -m, you will be prompted with the previous commit messages as the default

Branching

- Most VCSs, Git included, support the notion of *branching*
- A branch is generally used to mark an independent line of development
- For example, a release version might have a different branch from an experimental version of a project



- Think of branching as a way of creating a brand new working directory and staging area

Branching Commands

- The git branch command is used to list, create, rename, and delete branches
- List all branches

  git branch
- Create a new branch

  git branch name
- Rename the current branch

  git branch -m newname
- Delete the specified branch

  git branch -d name

Working With a Branch

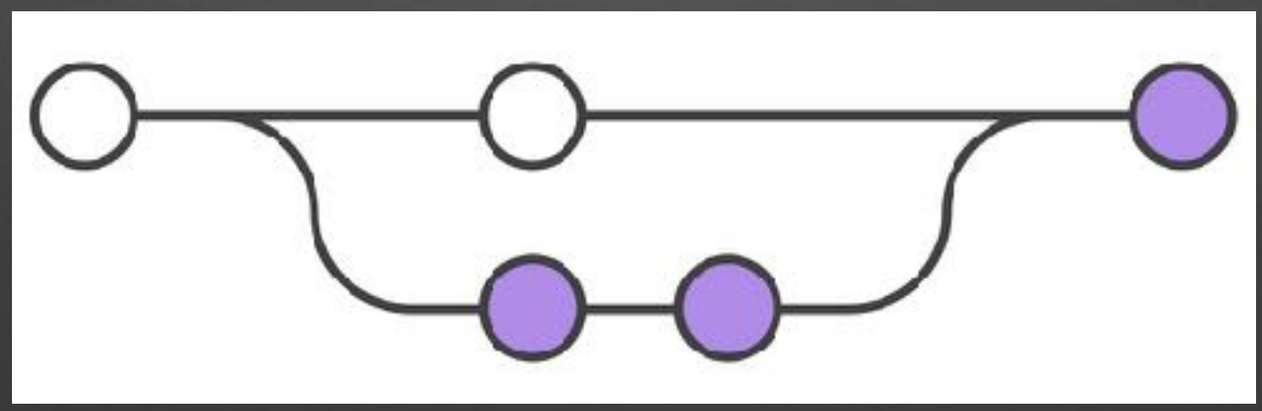
- In order to work with a branch, the working directory needs to be updated with the versions of the files stored in that branch
- In order to do this, the following command is used

  git checkout branch

- The branch branch must already have been created with the git branch command
- This will update the working directory and set the current branch to branch

Merging

- In addition to creating branches, it is also very common to *merge* branches together
- For example, a bug fixed on a release branch should be merged back into the experimental branch



Merging Commands

- The branch branch is merged into the current branch using the command

  git merge branch

- The git checkout command may be used to set the current branch before the merge
- The git branch -d command may be used to remove the branch branch if it is no longer needed after the merge

Types of Merges

- A *fast forward* merge occurs if there is a linear path from the current branch to the target branch
  - The merge just appends the target branch to the current branch.
- A *three-way* merge occurs if the current branch and the target branch have diverged
  - In this case, three commits are required in order to resolve the merge.
- A *merge conflict* occurs when the same part of the same file has been modified in both branches
  - This usually requires the developer to manually resolve the conflict.

Stashing Changes

- Sometimes you may need to work on another branch in the middle of making some changes
- It is not possible to switch to the new branch as the changes in the working directory will be lost
- By stashing the changes, it is possible to work on a different branch, then resume working where you left off
- Stash changes using

  git stash
- Change to another branch and do some work

  git checkout another_branch

  git commit -am “finished the work”
- Change back to the original branch and restore

  git checkout original_branch

  git stash pop

Remote Repositories

- Remote repositories are essential in order to collaborate with Git
- Cloning a repository automatically creates a remote connection called *origin* back to the original repository
- You can manage your remote repositories with the commands

  git remote -v

  git remote add name URL

  git remote rm name

Repository URLs

- Git supports a number of different ways to access remote repositories
  - SSH - Uses the ssh protocol for read/write access to the remote repository, requires set up

    ssh://user@server.domain/project.git
  - Git - Uses the Git protocol for fast read/write access but provides no authentication

    git://server.domain/project.git
  - HTTP or HTTPS - Easy to set up and use, but help needed with credentials

    http://server.domain/project.git

Fetching From Remotes

- When developers are working on separate repositories, it is necessary to import the changes at some point
- Fetching is used to see what everyone else is working on
- The git fetch command is used to import commits from a remote repository into the local repository

  git fetch origin

  will import all remote branches
- The resulting commits are stored as remote branches so they have no effect on the local repository

Pulling From Remotes

- It is often necessary to merge the changes from a remote repository into the local repository
- This can be achieved be using a git fetch followed by a git merge
- The git pull command rolls these into one step

  git pull remote

  is the same as

  git fetch remote

  get merge origin/branch

Pushing To Remotes

- After making changes in the local repository, the must be transferred to a remote repository in order to be shared with other developers
- The git push command is used for this

  git push remote name
- This pushes the name to remote and creates a new local branch in the destination repository (if necessary)
- To avoid overwriting other commits, Git will only allow a fast-forward merge in the destination repository
- Non-fast-forward merges must be resolved in the local repository before pushing

Using GitHub

Introduction

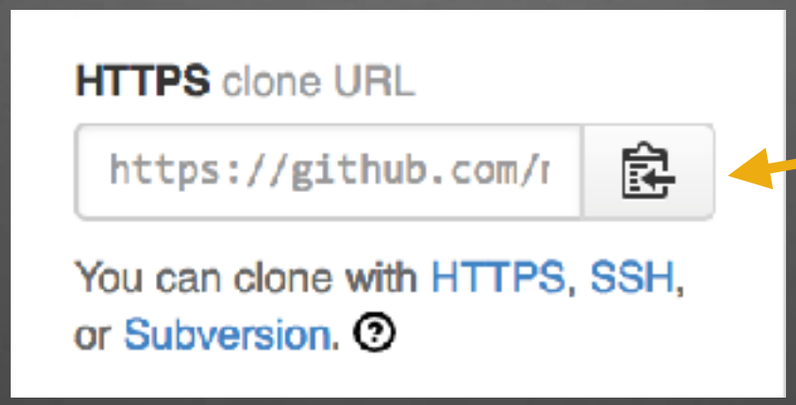
- GitHub is a Git repository hosting service
- Provides a Web-based graphical interface
- Provides access control and several collaboration features, such as a wikis and basic task management tools
- Supports “forking” – copying a repository from one user’s account to another
- Changes to a “forked” repository can be easily shared by sending a “pull request” to the original owner, who then merges the changes

Signing Up

- GitHub is free for small, public, projects
- Go to http://github.com and sign up for an account
  - Use your Net ID or something that allows us to identify you
    - Make sure the email you use is the same email you used to configure Git — this is how GitHub knows who makes a commit
- Check that the free plan is selected
- Click on the **Finish sign up** button

Cloning at GitHub

- GitHub makes it easy to clone by providing a clone URL on the repository page



copy URL to clipboard

- Two main types of URLs are supported

  HTTPS — https://github.com/ds-ga-1007/assignment4.git

  SSH — git@github.com:ds-ga-1007/assignment4.git

HTTPS

- Available on all repositories, public and private
- Will provide you with either read-only or read/write access, depending on your permissions to the repository
- These URLs work everywhere, even if you are behind a firewall or proxy
- When you git fetch, git pull, or git push to the remote repository using HTTPS, you'll be asked for your GitHub username and password
  - You can use a credential helper so Git will remember your GitHub username and password every time it talks to GitHub
  - See the GitHub Help for details on setting up a credential helper
  - https://help.github.com/articles/caching-your-github-password-in-git#platform-linux

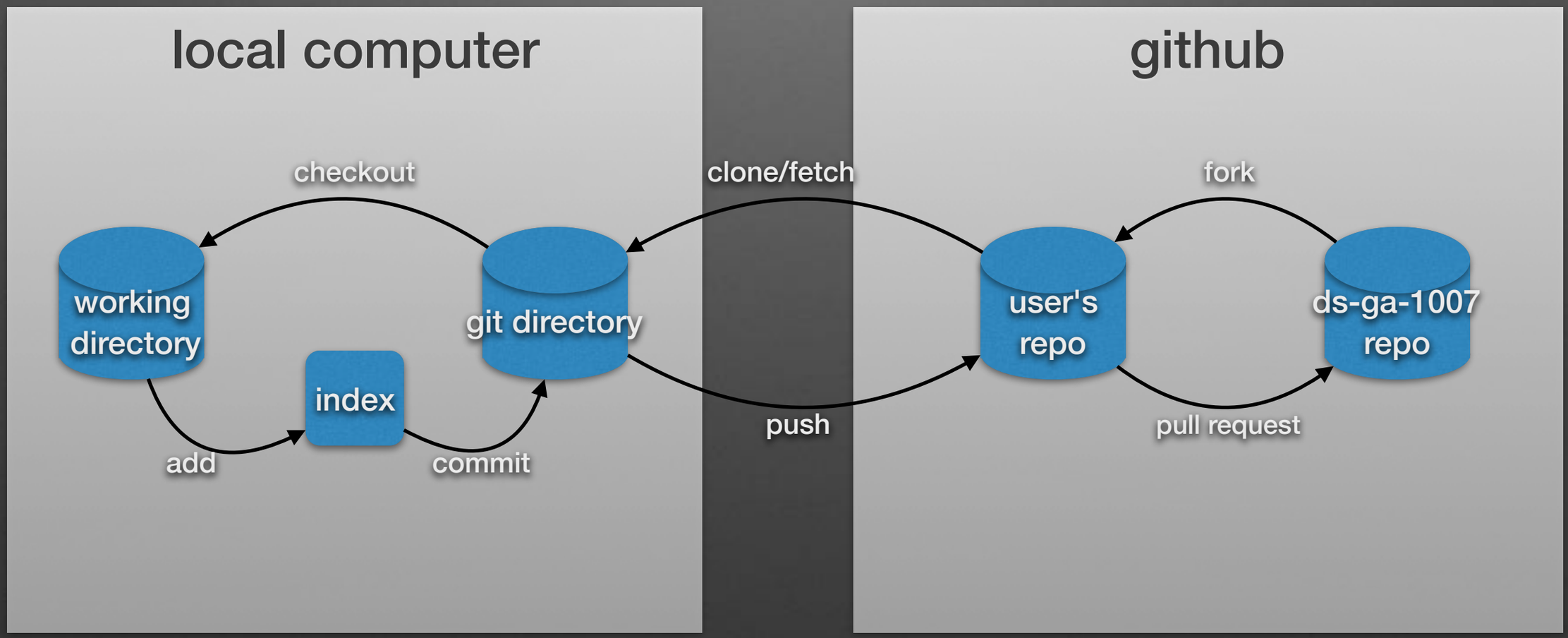
Finding a Repository

- Once you've signed in to GitHub, you can find a repository by using the search box
- Enter ds-ga-1007 into the search box and hit enter
- On the left sidebar, click on **Users**
- Click on the ds-ga-1007 user
- Now click on the assignment4 repository
- You should now see the clone URL for the repository

Forking a Repository

- A fork is a copy of a repository
- Forking creates a new repository in your account that is the clone of another repository
- You can then work on your copy of the repository (e.g. by cloning it on your machine) and push changes to your copy
- GitHub allows your changes to be sent to the original repository using a “pull request”
- More details on forking can be found in the GitHub Help
  - https://help.github.com/articles/fork-a-repo

Fork/Clone Overview



Collaborative Models

- Fork and pull
  - Fork an existing repository and push changes to your local fork
  - Notify the repository owner using the **Compare & review** button, then Create pull request
  - A “pull request” will be created to tell a repository owner that you have changes you’d like to contribute
- Shared repository
  - Project members are granted push access to a shared repository
  - Each member clones the repository and makes changes
  - Members push their changes directly to the shared repository
  - Pull request can also be used to allow contributions to be reviewed and discussed by all project members

Keeping Up To Date

- GitHub also provides the ability to “follow” a user to receive notification of their GitHub activity
  - It might be a good idea to “follow” the ds-ga-1007 user
- You can also “watch” a project to receive notification when the project is updated
  - This only applies to a single project, so use “follow” to receive updates for all of a user’s projects