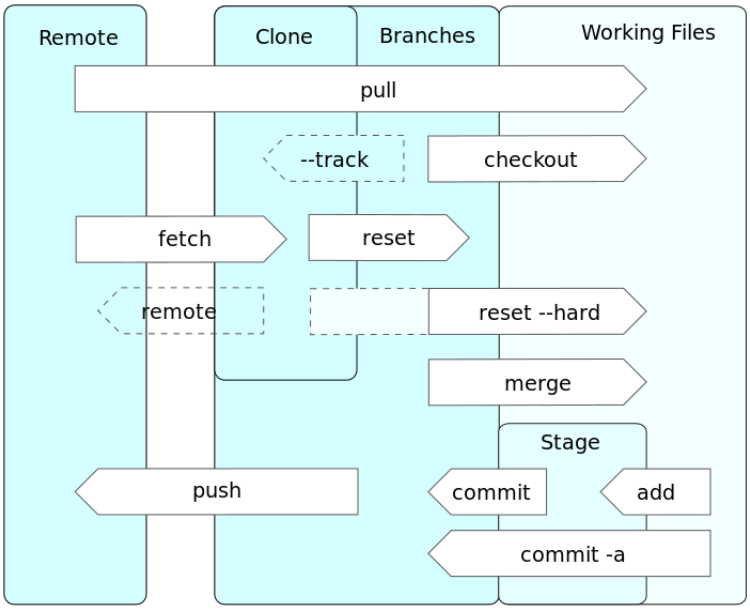
Git concepts

Git thinks of data like a set of snapshots of a file system.
HEAD is a pointer to the last commit on the currently checked out branch.
HEAD^ is a parent, HEAD^^ is a grandparent, HEAD^2 is a merged parent, not grandparent



(Source for the graphic: Wikipeadia)

State:	**modified**	**staged**	**committed**
Meaning	Changed but not committed	File marked to go into your next commit	Safely stored in local db

Sections	Working Directory	Staging Area	Git Directory (Repository)
Meaning	Where you work; single checkout of one version of project	Stores information about what will go into next commit	Stores metadata and object db
Workflow	Modify files in working directory	Stage the files, adding snapshots to staging area	Commit which stores files permanently
	add →	**commit →**	**push →**
	`git add <filename>` **`git add .`** recursive **`git add *`** not recrsv	**`git commit –m "Commit message"`**	**`git push origin master`** Sends changes to remote repository

Quick start run through

```
$ mkdir myproject
$ cd myproject
$ git init
$ touch README
$ git add .
$ git commit –m 'initial commit'
$ git remote add origin https://github.com:userName/myproject.git
$ git push –u origin master
```

Git commands

Help

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

Ignoring files https://github.com/github/gitignore/blob/master/Python.gitignore

$ touch .gitignore **Use this link for a start**

```
*.log     # no log files
*.[oa]    # no o or a files
*~        # no tilde files
!lib.a    # but do track lib.a
/TODO     # only ignore root TODO, not sub /TODO
build/    # ignore all files in build/ directory
doc/*.txt # ignore doc/notes.txt, but include doc/server/arch.txt
```

for comments, / for directory, ! creates exceptions

Glob patterns work: simplified regular expressions

*	matches zero or more characters
[abc]	matches any character in the brackets
?	matches a single character
[0-9]	matches any character in range

Staging files

Add options

```
$ git add -A    : stages All
$ git add .     : stages new and modified, without deleted
$ git add -u    : stages modified and deleted, without new
$ git add -i    : interactive mode
```

Seeing what's going on

Super high level

```
$ git status -s
```

Shows which files have been modified and/or staged. Short flag to limit output

High level

```
$ git status
```

Main tool to determine file state; plenty of output and hints

Mid level

```
$ git diff --stat
```

Summarizes changed but not staged to staged

Line level

```
$ git diff
```

Compares changed but not staged to staged line by line

All changes, staged and unstaged

```
$ git diff HEAD
```
Compares changed to committed, skip staged

Staged and ready to commit
```
$ git diff --cached   or   $ git diff --staged
```
Compares staged to last commit

Removing files
Keep file, stop tracking
```
$ git rm --cached readme.txt
```
Removes from staging area and keeps it, then fix .gitignore

Delete file stop tracking
```
$ git rm <file>
```
Removes file from staging area at next commit and deletes it

```
$ git rm -f <file>
```
For modified files staged already

```
$ git rm log/\*.log
```
\ is escape character. Command removes all .log files in log directory

```
$ git rm \*~
```
removes all files ending in ~

Moving, renaming files
```
$ git mv <old file> <new file>
```
Move or rename this way to keep staging happy

Committing files
Commit options
`$ git commit` : Launches editor to write commit message
`-v` : For verbose, puts the diff in the editor
`-m "msg"` : Bypasses the editor and includes a message
`-a` : Skips staging area, adds everything tracked for you

Undoing things
```
$ git checkout -- <file>
```
Revert to last committed version. Throw away current modified version

```
$ git reset --hard HEAD
```
Undoes last commit, unstage files, and undo changes to working dir. Lose the changes and go back.

```
$ git reset --soft HEAD~
```
Last commit will be undone, files touched will be back on the stage (HEAD~ is parent of HEAD). Leaves work in working dir and staging area, rolls back HEAD to previous commit. Rolls back to just before commit.

```
$ git commit --amend
```
Commits your current staging area and allows you to edit the commit message

Debugging

Binary search

```
$ git bisect start
$ git bisect bad
$ git bisect good <tag>
```

Current version has a bug, <tag> version was working. Many commits in between. Git checks out middle one for you to test.

```
$ git bisect good     or    $ git bisect bad
```

Until version that introduced the bug is found.

```
$ git bisect reset
```

Resets HEAD to where you were before you started. Important!

File annotation

```
$ git blame –L 21, 31, <file>
```

Annotates line change history from lines 21 to 31

Finding something

```
$ git grep "hello"
```

Search all files git tracks for "hello"

```
$ git grep "hello" v2.5
```

Search all files git tracks in v2.5 for "hello"

Stashing work

Save work when interrupted

```
$ git stash save "msg"
```

Saves uncommitted work in process

See different stashes

```
$ git stash list
```

Lists stashes in a stack

Reapply changes and drop stash

```
$ git stash pop
```

Reapplies the stashed changes, previously staged files will not be restaged, and deletes stash

Reapply changes only

```
$ git stash apply
```

Reapplies the stashed changes, previously staged files will not be restaged

Drop stash

```
$ git stash drop stash@{0}
```

Deletes stash

Dump whole stash

```
$ git stash clear
```

Deletes all stash

Viewing history

```
$ git log
```

Shows commit history

```
$ git log --graph --all --oneline
```
Shows commit tree view history

```
$ git log -p -2
```
Shows diff of last two entries

```
$ git log --stat
```
shows a list of modified files and +- lines

```
$ git log --pretty=format:"%h - %an, %ar : %s"
```
- %h abbreviated commit hash
- %an author name
- %ar author date, relative
- %s subject

```
$ git log --pretty=format:"%h %s" --graph
```
Shows a branch and merge history

```
$ git log --since=2.weeks
```
Changes in last 2 weeks

```
$ git log --author=gitster --since="2008-10-01"
```
Filters by author specified

--grep option matches keywords in commit messages

--all-match for both --grep and --author

Visual History
```
$ gitk
```
Opens a history visualizer window; pip install gitk first

Branching workflow
See available branches
```
$ git branch
```
shows various branches available

Create new branch
```
$ git checkout -b newBranch
```
Creates & switches to newBranch branch

Switch to master
```
$ git checkout master
```
Switches back to master branch

Preview changes
```
$ git diff <source_branch> <target branch>
```
Preview changes pre merge

Merge branch
```
$ git merge newBranch
```

Merges newBranch branch

Delete merged branch

```
$ git branch –d newBranch
```

Deletes newBranch branch post merge

Merge conflicts

Open file in editor, resolve conflict, delete other code and <<<<, =====, >>>>>>; or

```
$ git mergetool
```

Opens the system merge tool

More branch commands

```
$ git branch newName2
```

creates another branch, newName2

```
$ git checkout newName2
```

switches to newName2 branch

```
$ git branch ––merged
```

Shows what's already merged

```
$ git branch ––no–merged
```

Shows what's not already merged

Tagging

```
$ git log
```

To get the first 10 characters of the hash commit ID

```
$ git tag 1.0.0 1b2e1d63ff
```

Tags with the commit ID

```
$ git tag
```

Shows tags in alpha order, i.e., out of order

```
$ git tag –a v1.4 –m ‘my version 1.4’
```

Creates annotated tag, which is a full object in Git db

```
$ git show v1.4
```

Shows info on a specific tag

```
$ git describe
```

Shows how many commits since last tag

```
$ git push origin  ––tags
```

Tags don’t get pushed to remote servers unless you specify

Remote repositories: GitHub

github repositories

URL `https://github.com/username/myproject`

Public clone URL: `git://github.com/username/myproject.git` is read-only

Your clone URL: `git@github.com:username/myproject.git`

GitHub workflow

Fork a repo, clone the repo down locally, make changes, commit them, push them back to your public copy, then send the owner a pull request.

Clone a repository

```
$ git clone username@host:/path/to/repository
$ git clone [url]
$ git clone /path/to/repository
$ git clone git://github.com/schaon/grit.git mygrit
$ git clone username@host:/path/to/repository
```

Clone a repository on a remote server. Creates a directory called mygrit, initialilzes a .git directory inside it. Pulls down all the new data for that repository. Checks out a working copy of the latest version.

```
$ git remote
origin
```

Shows which remote servers you configured

```
$ git pull
```

Automatically fetches and merges a remote branch into your current path.

```
$ git push origin master
```

origin = [remote name], master=[branch name] Works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime.

```
$ git remote –v
origin git://github.com/blahblah.git
```

Show URL of cloned project. This one is pull only

```
$ git remote add origin git://github.com/username/filename.git
```

Adds a repository with the short name origin. If not clones existing repo & you want to connect your repo to remote server

```
$ git fetch paul
```

Fetches all the information that Paul has but you don't have in your repository. Paul's master branch is accessible as paul/master. You can merge it into one of your branches or checkout a branch.

```
$ git fetch origin
```

fetches any new work pushed to a server since you cloned (or last fetched from) it. Fetch pulls data to your local repository. It does not merge it with your work. You have to merge it manually.

```
$ git remote show origin
```

Shows information about a particular remote, URL and tracking branch information

```
$ git remote rename pb paul
```

Changes a remote's short name

```
$ git remote rm paul
```

Removes a reference

Delete github repository

Navigate to the repository, on right side select Settings from action bar, click Delete from inside Danger Zone ™. Enter the name again to confirm. Click "I understand the consequences…"

Configuration

Check git settings

```
$ git config --list
```

Local configuration

```
$ git config --list
```

Checks your user settings

```
$ git config --global core.editor subl
```

Changes one of the user settings, editor to sublime text

Initial configuration

```
$ git config --global user.name "Your Name "

$ git config --global user.email you@yourdomain.example.com
```

Auto completion

`contrib/completion` directory find `git-completion.bash` Copy this to your home directory and add `source ~/.git-completion.bash` to your .bashrc file

Aliases

```
$ git config --global alias.co checkout
```

Sets up `$ git co` as the `checkout` command

```
$ git config --global alias.visual "!gitk"
```

Sets up `$ git visual` as the `gitk` shell command