GitHub

# GIT CHEAT SHEET

V 1.1.1

Git is the open source distributed version control system that facilitates GitHub activities on your laptop or desktop. This cheat sheet summarizes commonly used Git command line instructions for quick reference.

## INSTALL GIT

GitHub provides desktop clients that include a graphical user interface for the most common repository actions and an automatically updating command line edition of Git for advanced scenarios.

**GitHub for Windows**
https://windows.github.com

**GitHub for Mac**
https://mac.github.com

Git distributions for Linux and POSIX systems are available on the official Git SCM web site.

**Git for All Platforms**
http://git-scm.com

## CONFIGURE TOOLING

Configure user information for all local repositories

```
$ git config --global user.name "[name]"
```

Sets the name you want attached to your commit transactions

```
$ git config --global user.email "[email address]"
```

Sets the email you want attached to your commit transactions

```
$ git config --global color.ui auto
```

Enables helpful colorization of command line output

## CREATE REPOSITORIES

Start a new repository or obtain one from an existing URL

```
$ git init [project-name]
```

Creates a new local repository with the specified name

```
$ git clone [url]
```

Downloads a project and its entire version history

## MAKE CHANGES

Review edits and craft a commit transaction

```
$ git status
```

Lists all new or modified files to be committed

```
$ git diff
```

Shows file differences not yet staged

```
$ git add [file]
```

Snapshots the file in preparation for versioning

```
$ git diff --staged
```

Shows file differences between staging and the last file version

```
$ git reset [file]
```

Unstages the file, but preserve its contents

```
$ git commit -m "[descriptive message]"
```

Records file snapshots permanently in version history

## GROUP CHANGES

Name a series of commits and combine completed efforts

```
$ git branch
```

Lists all local branches in the current repository

```
$ git branch [branch-name]
```

Creates a new branch

```
$ git checkout [branch-name]
```

Switches to the specified branch and updates the working directory

```
$ git merge [branch]
```

Combines the specified branch's history into the current branch

```
$ git branch -d [branch-name]
```

Deletes the specified branch

# GIT CHEAT SHEET

## REFACTOR FILENAMES
Relocate and remove versioned files

```
$ git rm [file]
```
Deletes the file from the working directory and stages the deletion

```
$ git rm --cached [file]
```
Removes the file from version control but preserves the file locally

```
$ git mv [file-original] [file-renamed]
```
Changes the file name and prepares it for commit

## SUPPRESS TRACKING
Exclude temporary files and paths

```
*.log
build/
temp-*
```
A text file named `.gitignore` suppresses accidental versioning of files and paths matching the specified patterns

```
$ git ls-files --other --ignored --exclude-standard
```
Lists all ignored files in this project

## SAVE FRAGMENTS
Shelve and restore incomplete changes

```
$ git stash
```
Temporarily stores all modified tracked files

```
$ git stash pop
```
Restores the most recently stashed files

```
$ git stash list
```
Lists all stashed changesets

```
$ git stash drop
```
Discards the most recently stashed changeset

## REVIEW HISTORY
Browse and inspect the evolution of project files

```
$ git log
```
Lists version history for the current branch

```
$ git log --follow [file]
```
Lists version history for a file, including renames

```
$ git diff [first-branch]...[second-branch]
```
Shows content differences between two branches

```
$ git show [commit]
```
Outputs metadata and content changes of the specified commit

## REDO COMMITS
Erase mistakes and craft replacement history

```
$ git reset [commit]
```
Undoes all commits after `[commit]`, preserving changes locally

```
$ git reset --hard [commit]
```
Discards all history and changes back to the specified commit

## SYNCHRONIZE CHANGES
Register a repository bookmark and exchange version history

```
$ git fetch [bookmark]
```
Downloads all history from the repository bookmark

```
$ git merge [bookmark]/[branch]
```
Combines bookmark's branch into current local branch

```
$ git push [alias] [branch]
```
Uploads all local branch commits to GitHub

```
$ git pull
```
Downloads bookmark history and incorporates changes

fournova

# COMMAND LINE CHEAT SHEET

presented by **TOWER** › Version control with Git - made easy

## DIRECTORIES

```
$ pwd
```
Display path of current working directory

```
$ cd <directory>
```
Change directory to <directory>

```
$ cd ..
```
Navigate to parent directory

```
$ ls
```
List directory contents

```
$ ls -la
```
List detailed directory contents, including hidden files

```
$ mkdir <directory>
```
Create new directory named <directory>

## OUTPUT

```
$ cat <file>
```
Output the contents of <file>

```
$ less <file>
```
Output the contents of <file> using the less command (which supports pagination etc.)

```
$ head <file>
```
Output the first 10 lines of <file>

```
$ <cmd> > <file>
```
Direct the output of <cmd> into <file>

```
$ <cmd> >> <file>
```
Append the output of <cmd> to <file>

```
$ <cmd1> | <cmd2>
```
Direct the output of <cmd1> to <cmd2>

```
$ clear
```
Clear the command line window

## FILES

```
$ rm <file>
```
Delete <file>

```
$ rm -r <directory>
```
Delete <directory>

```
$ rm -f <file>
```
Force-delete <file> (add -r to force-delete a directory)

```
$ mv <file-old> <file-new>
```
Rename <file-old> to <file-new>

```
$ mv <file> <directory>
```
Move <file> to <directory> (possibly overwriting an existing file)

```
$ cp <file> <directory>
```
Copy <file> to <directory> (possibly overwriting an existing file)

```
$ cp -r <directory1>
  <directory2>
```
Copy <directory1> and its contents to <directory2> (possibly overwriting files in an existing directory)

```
$ touch <file>
```
Update file access & modification time (and create <file> if it doesn't exist)

## PERMISSIONS

```
$ chmod 755 <file>
```
Change permissions of <file> to 755

```
$ chmod -R 600 <directory>
```
Change permissions of <directory> (and its contents) to 600

```
$ chown <user>:<group> <file>
```
Change ownership of <file> to <user> and <group> (add -R to include a directory's contents)

## SEARCH

```
$ find <dir> -name "<file>"
```
Find all files named <file> inside <dir> (use wildcards [*] to search for parts of filenames, e.g. "file.*")

```
$ grep "<text>" <file>
```
Output all occurrences of <text> inside <file> (add -i for case-insensitivity)

```
$ grep -rl "<text>" <dir>
```
Search for all files containing <text> inside <dir>

## NETWORK

```
$ ping <host>
```
Ping <host> and display status

```
$ whois <domain>
```
Output whois information for <domain>

```
$ curl -O <url/to/file>
```
Download <file> (via HTTP[S] or FTP)

```
$ ssh <username>@<host>
```
Establish an SSH connection to <host> with user <username>

```
$ scp <file>
  <user>@<host>:/remote/path
```
Copy <file> to a remote <host>

## PROCESSES

```
$ ps ax
```
Output currently running processes

```
$ top
```
Display live information about currently running processes

```
$ kill <pid>
```
Quit process with ID <pid>

# COMMAND LINE TIPS & TRICKS

presented by **TOWER** › Version control with Git - made easy

## GETTING HELP

On the command line, help is always at hand: you can either type `man <command>` or `<command> --help` to receive detailed documentation about the command in question.

## FILE PERMISSIONS

On Unix systems, file permissions are set using three digits: the first one representing the permissions for the owning user, the second one for its group, and the third one for anyone else.

Add up the desired access rights for each digit as following:

`4` – access/read (r)
`2` – modify/write (w)
`1` – execute (x)

For example, `755` means "rwx" for owner and "rx" for both group and anyone. `740` represents "rwx" for owner, "r" for group and no rights for other users.

## COMBINING COMMANDS

If you plan to run a series of commands after another, it might be useful to combine them instead of waiting for each command to finish before typing the next one. To do so, simply separate the commands with a semicolon ( `;` ) on the same line.

Additionally, it is possble to execute a command only if its predecessor produces a certain result. Code placed after the `&&` operator will only be run if the previous command completes successfully, while the opposite `||` operator only continues if the previous command fails. The following command will create the folder "videos" only if the `cd` command fails (and the folder therefore doesn't exist):

```
$ cd ~/videos || mkdir ~/videos
```

## THE "CTRL" KEY

Various keyboard shortcuts can assist you when entering text: Hitting `CTRL+A` moves the caret to the beginning and `CTRL+E` to the end of the line.

In a similar fashion, `CTRL+K` deletes all characters after and `CTRL+U` all characters in front of the caret.

Pressing `CTRL+L` clears the screen (similarly to the `clear` command). If you should ever want to abort a running command, `CTRL+C` will cancel it.

## THE "TAB" KEY

Whenever entering paths and file names, the `TAB` key comes in very handy. It autocompletes what you've written, reducing typos quite efficiently. E.g. when you want to switch to a different directory, you can either type every component of the path by hand:

```
$ cd ~/projects/acmedesign/docs/
```

...or use the `TAB` key (try this yourself):

```
$ cd ~/pr[TAB]ojects/
  ac[TAB]medesign/d[TAB]ocs/
```

In case your typed characters are ambiguous (because "ac" could point to the "acmedesign" or the "actionscript" folder), the command line won't be able to autocomplete. In that case, you can hit `TAB` twice to view all possible matches and then type a few more characters.

## THE ARROW KEYS

The command line keeps a history of the most recent commands you executed. By pressing the `ARROW UP` key, you can step through the last called commands (starting with the most recent). `ARROW DOWN` will move forward in history towards the most recent call.

Bonus tip: Calling the `history` command prints a list of all recent commands.

## HOME FOLDER

File and directory paths can get long and awkward. If you're addressing a path inside of your home folder though, you can make things easier by using the `~` character. So instead of writing `cd /Users/your-username/projects/`, a simple `cd ~/projects/` will do.

And in case you should forget your user name, `whoami` will remind you.

## OUTPUT WITH "LESS"

The `less` command can display *and paginate* output. This means that it only displays one page full of content and then waits for your explicit instructions. You'll know you have `less` in front of you if the last line of your screen either shows the file's name or just a colon ( `:` ).

Apart from the arrow keys, hitting `SPACE` will scroll one page forward, `b` will scroll one page backward, and `q` will quit the `less` program.

## DIRECTING OUTPUT

The output of a command does not necessarily have to be printed to the command line. Instead, you can decide to direct it to somewhere else.

Using the `>` operator, for example, output can be directed to a file. The following command will save the running processes to a text file in your home folder:

```
$ ps ax > ~/processes.txt
```

It is also possible to pass output to another command using the `|` (pipe) operator, which makes it very easy to create complex operations. E.g., this chain of commands will list the current directory's contents, search the list for PDF files and display the results with the `less` command:

```
$ ls | grep ".pdf" | less
```

# MARKDOWN SYNTAX

**Markdown** is a way to style text on the web. You control the display of the document; formatting words as bold or italic, adding images, and creating lists are just a few of the things we can do with Markdown. Mostly, Markdown is just regular text with a few non-alphabetic characters thrown in, like # or *.

## HEADERS

```
# This is an <h1> tag
## This is an <h2> tag
###### This is an <h6> tag
```

## EMPHASIS

```
*This text will be italic*
_This will also be italic_

**This text will be bold**
__This will also be bold__

*You **can** combine them*
```

## LISTS

Unordered

```
* Item 1
* Item 2
  * Item 2a
  * Item 2b
```

Ordered

```
1. Item 1
2. Item 2
3. Item 3
   * Item 3a
   * Item 3b
```

## IMAGES

```
![GitHub Logo](/images/logo.png)

Format: ![Alt Text](url)
```

## LINKS

```
http://github.com - automatic!

[GitHub](http://github.com)
```

## BLOCKQUOTES

```
As Grace Hopper said:

> I've always been more interested
> in the future than in the past.
```

As Grace Hopper said:

> I've always been more interested
> in the future than in the past.

## BACKSLASH ESCAPES

Markdown allows you to use backslash escapes to generate literal characters which would otherwise have special meaning in Markdown's formatting syntax.

```
\*literal asterisks\*
```

*literal asterisks*

Markdown provides backslash escapes for the following characters:

| | | | |
|---|---|---|---|
| \ | backslash | () | parentheses |
| ` | backtick | # | hash mark |
| * | asterisk | + | plus sign |
| _ | underscore | - | minus sign (hyphen) |
| {} | curly braces | . | dot |
| [] | square brackets | ! | exclamation mark |

# GITHUB FLAVORED MARKDOWN

GitHub.com uses its own version of the Markdown syntax that provides an additional set of useful features, many of which make it easier to work with content on GitHub.com.

## USERNAME @MENTIONS

Typing an `@` symbol, followed by a username, will notify that person to come and view the comment. This is called an "@mention", because you're mentioning the individual. You can also @mention teams within an organization.

## ISSUE REFERENCES

Any number that refers to an Issue or Pull Request will be automatically converted into a link.

```
#1
github-flavored-markdown#1
defunkt/github-flavored-markdown#1
```

## EMOJI

To see a list of every image we support, check out **www.emoji-cheat-sheet.com**

```
GitHub supports emoji!
:+1: :sparkles: :camel: :tada:
:rocket: :metal: :octocat:
```

GitHub supports emoji!
👍✨🐫🎉🚀🤘

## FENCED CODE BLOCKS

Markdown coverts text with four leading spaces into a code block; with GFM you can wrap your code with ``` to create a code block without the leading spaces. Add an optional language identifier and your code with get syntax highlighting.

~~~
```javascript
function test() {
 console.log("look ma', no spaces");
}
```
~~~

```
function test() {
  console.log("look ma', no spaces");
}
```

## TASK LISTS

```
- [x] this is a complete item
- [ ] this is an incomplete item
- [x] @mentions, #refs, [links](),
**formatting**, and <del>tags</del>
supported
- [x] list syntax required (any
unordered or ordered list
supported)
```

☑ this is a complete item
☐ this is an incomplete item
☑ @mentions, #refs, links, **formatting**, and ~~tags~~ supported
☑ list syntax required (any unordered or ordered list supported)

## TABLES

You can create tables by assembling a list of words and dividing them with hyphens `-` (for the first row), and then separating each column with a pipe `|`:

```
First Header | Second Header
------------ | -------------
Content cell 1 | Content cell 2
Content column 1 | Content column 2
```

| First Header | Second Header |
|---|---|
| Content cell 1 | Content cell 2 |
| Content column 1 | Content column 2 |