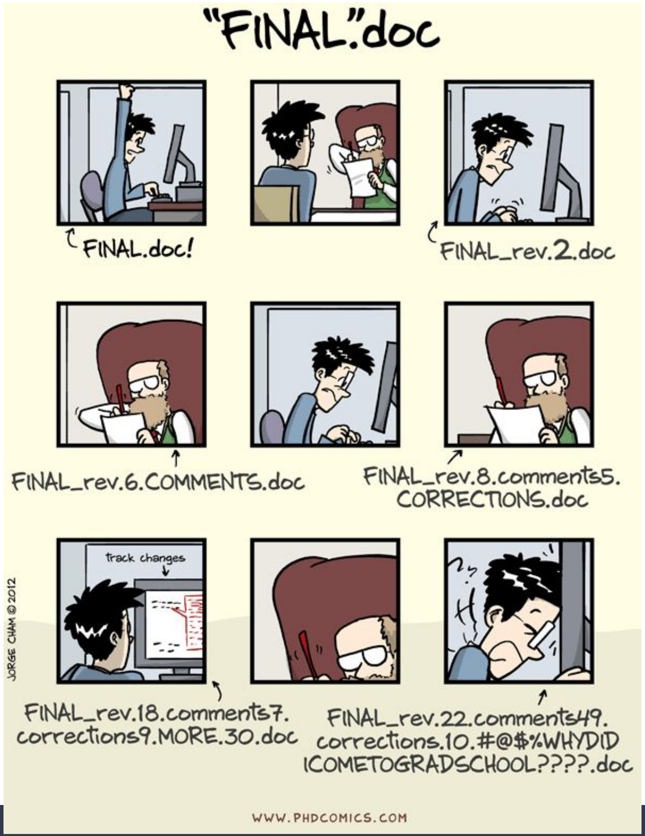
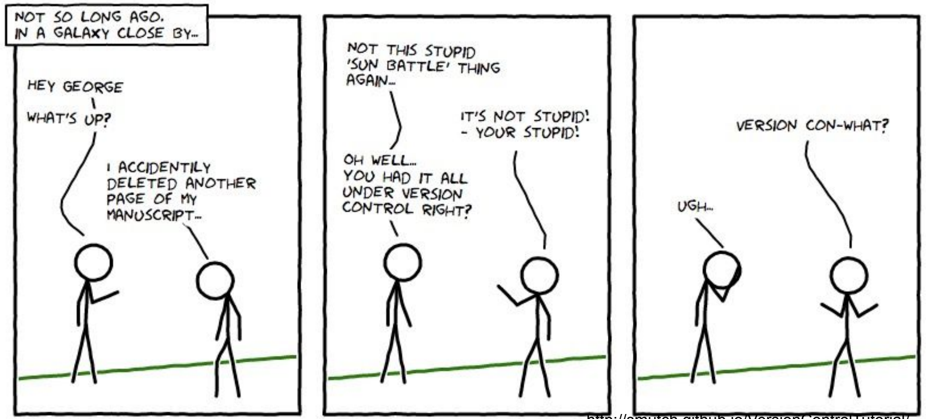
Introduction to bash and version control

Adela Sobotkova, Cultural Data Science, Aarhus 2022

The material in this tutorial is inspired by & adapted from the Software Carpentry lesson on version control & Dany Vohl's tutorial from the ANITA astroinformatics school in 2017.

The Problem




NOT SO LONG AGO.
IN A GALAXY CLOSE BY...
HEY GEORGE
WHAT'S UP?
I ACCIDENTILY DELETED ANOTHER PAGE OF MY MANUSCRIPT...
NOT THIS STUPID 'SUN BATTLE' THING AGAIN...
OH WELL... YOU HAD IT ALL UNDER VERSION CONTROL RIGHT?
IT'S NOT STUPID! - YOUR STUPID!
UGH...
VERSION CON-WHAT?

http://smutch.github.io/VersionControlTutorial/

Version control, a.k.a. revision control / source code management, is basically a system for **recording and managing changes made to files and folders.**

You can track:

- source code (e.g. Python, R, Bash scripts),
- other files containing mostly text (e.g. LaTeX, csv, plain text),
- work by a lone developer, or
- collaboration on projects (track who's done what, branch to develop different streams, etc).

Why Version Control?

As data scientist, we spend much of our time writing code, whether it be for data cleaning, machine learning, or visualisation. As such, our codes are often constantly evolving. By putting all of our code under version control we can:

- **tag code** versions for later reference *(via tags)*.
- record a **unique identifier** for the exact code version used to produce a particular plot or result *(via commit identifiers)*.
- **roll back** our code to previous states *(via checkout)*.
- **identify** when/how **bugs** were introduced *(via diff/blame)*.
- **keep multiple versions** of the same code in sync with each other *(via branches/merging)*.
- efficiently **share and collaborate** on our codes with others *(via remotes/online hosting)*.

Why Version Control?

It's important to also realise that many of the advantages of version control are not limited to just managing code. For example, it can also be useful when writing papers/reports. Here we can use version control to:

- **bring back** that paragraph we accidentally deleted last week.
- **try out a different structure** and simply disregard it if we don't like it.
- **concurrently work on a paper** with a collaborator and then **automatically merge** all of our **changes** together.

The upshot is ***you should use version control for almost everything***. The benefits are well worth it...

Introducing Git

In this tutorial we will be using Git for version control.

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.[Git website]

- *Distributed* -> everyone has their own complete copy of the entire repository and can make changes as they like
- Commiting to a 'central' (e.g. Github) repository can be done once happy with the changes
- Git is fast (primarily written in C & shell script) and lightweight (as you only track changes)
- Written by Linus Torvalds (creator of Linux)

Why Github

- ❑ Remote repository
- ❑ Version Control
- ❑ Visible code and reproducibility
- ❑ Open code and reuse
- ❑ Collaborative code development
- ❑ Open code development

Short intro to bash

What is a command shell and why would I use one?

At a high level, computers do four things:

- run programs
- store data
- communicate with each other, and
- interact with us

They can do the last of these in many different ways, including through a keyboard and mouse, or touch screen interfaces, or using speech recognition systems.

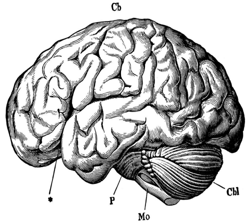
We are all familiar with **graphical user interfaces** (GUI): windows, icons and pointers. They are easy to learn and fantastic for simple tasks where a vocabulary consisting of "click" translates easily into "do the thing I want". But this magic relies on wanting a simple set of things, and having programs that can do exactly those things.

If you wish to do complex, purpose-specific things it helps to have a richer means of expressing your instructions to the computer. It doesn't need to be complicated or difficult, just a vocabulary of commands and a simple grammar for using them.

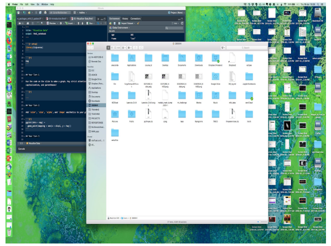
This is what the shell provides - a simple language and a **command-line interface** to use it through.

Short intro to the command line

1. run programs
2. store data
3. communicate with each other, and
4. interact with us

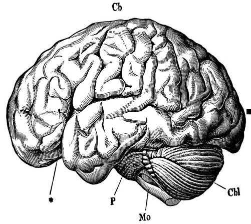


Human thought



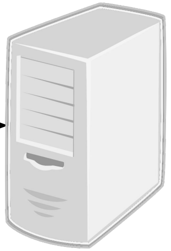
- Simple tasks
- One at a time
- Hard to automate or reproduce

Machine language



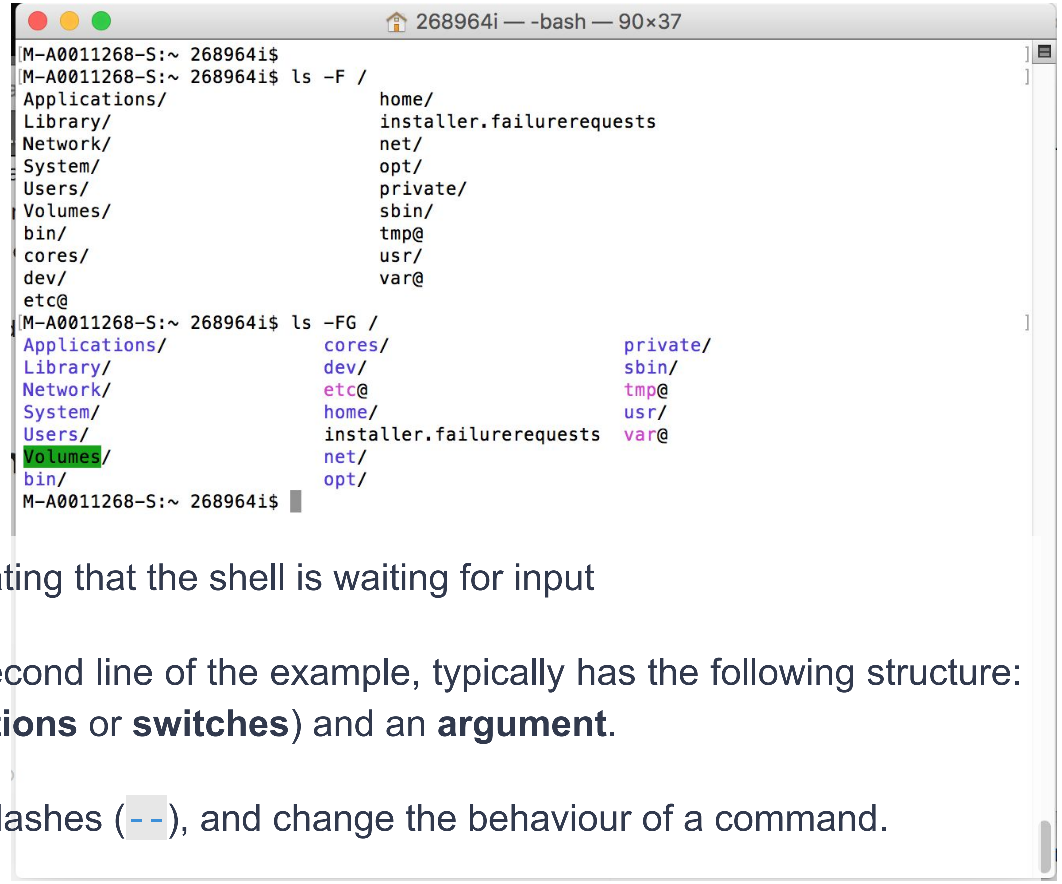
Human thought

- Simple language + command line interface
- Read-evaluate-print loop (**REPL**)
- The shell is a program which runs other programs instead of doing it's own calculation
- Great for automating tasks
- Scripting allows for easy reproducibility



Machine language

Short intro to Bash



```
M-A0011268-S:~ 268964i$
M-A0011268-S:~ 268964i$ ls -F /
Applications/                   home/
Library/                        installer.failurerequests
Network/                        net/
System/                         opt/
Users/                          private/
Volumes/                        sbin/
bin/                            tmp@
cores/                          usr/
dev/                            var@
etc@
M-A0011268-S:~ 268964i$ ls -FG /
Applications/              cores/                     private/
Library/                   dev/                       sbin/
Network/                   etc@                       tmp@
System/                    home/                      usr/
Users/                     installer.failurerequests  var@
Volumes/                   net/
bin/                       opt/
M-A0011268-S:~ 268964i$
```

The first line shows only a **prompt**, indicating that the shell is waiting for input

The part that you type, `ls -F /` in the second line of the example, typically has the following structure: a **command**, some **flags** (also called **options** or **switches**) and an **argument**.

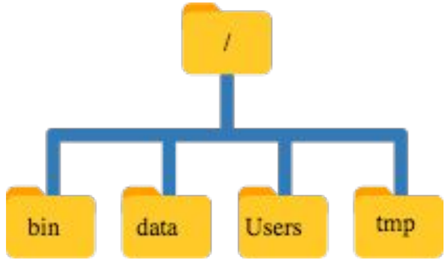
Flags start with a single dash (`-`) or two dashes (`--`), and change the behaviour of a command.

Arguments tell the command what to operate on (e.g. files and directories).

A command can be called with more than one flag and more than one argument: but a command doesn't always require an argument or a flag!

Intro to Bash – Navigating

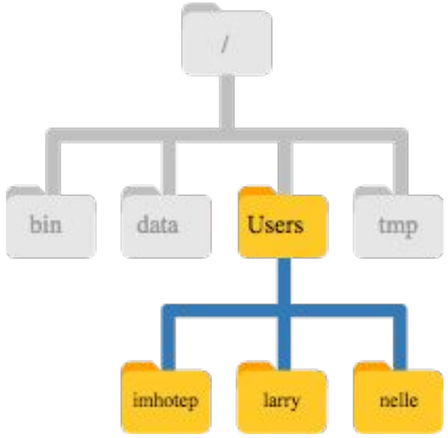
The part of the operating system responsible for managing files and directories is called the **file system**. It organizes our data into files, which hold information, and directories (also called “folders”), which hold files or other directories.



Every user on a computer will have a **home directory**.

The home directory path will look different on different operating systems. On Linux it may look like `/home/nelle`, and on Windows it will be similar to `C:\Documents and Settings\nelle` or `C:\Users\nelle`

To understand how we can navigate through our **file system** we need to have a look at how the file system as a whole is organized.



At the top is the **root directory** that holds everything else. We refer to it using a slash character, `/`, on its own.

Inside that directory are several other directories, in which are other directories, and so on.

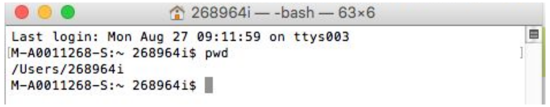
Notice that there are two meanings for the `/` character. When it appears at the front of a file or directory name, it refers to the root directory. When it appears *inside* a name, it’s just a separator.

Intro to Bash – Navigating

When we open the Bash terminal we start out in our **home directory.**

Let’s find out where this is exactly by running a command called `pwd` (which stands for “print working directory”).

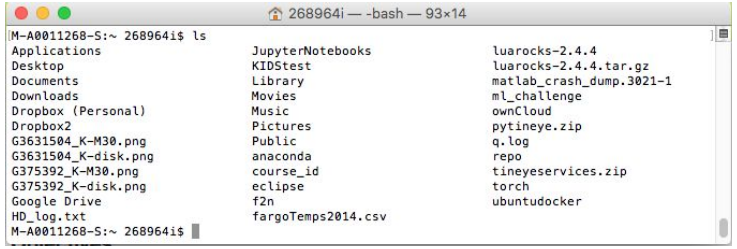
```
$ pwd
```



```
Last login: Mon Aug 27 09:11:59 on ttys003
M-A0011268-S:~ 268964i$ pwd
/Users/268964i
M-A0011268-S:~ 268964i$
```

We can see what’s in our home directory by running `ls`, which stands for “listing”

```
$ ls
```



```
M-A0011268-S:~ 268964i$ ls
Applications                    JupyterNotebooks                luarocks-2.4.4
Desktop                         KIDStest                        luarocks-2.4.4.tar.gz
Documents                       Library                         matlab_crash_dump.3021-1
Downloads                       Movies                          ml_challenge
Dropbox (Personal)              Music                           ownCloud
Dropbox2                        Pictures                        pytineye.zip
G3631504_K-M30.png              Public                          q.log
G3631504_K-disk.png             anaconda                        repo
G375392_K-M30.png               course_id                       tineyeservices.zip
G375392_K-disk.png              eclipse                         torch
Google Drive                    f2n                             ubuntudocker
HD_log.txt                      fargoTemps2014.csv
M-A0011268-S:~ 268964i$
```

Remember a **command** can often be followed by **flags** and/or **argument**, e.g.:

```
$ ls -FG
```

`ls` has lots of other **flags**. There are two common ways to find out how to use a command and what flags it accepts:

`$ ls --help` or `$ man ls`

There is also a handy *tldr* online, explaining the most commonly used command options: https://tldr.ostera.io/

Intro to Bash – Navigating

Next let’s **change our location** to a different directory, so we are no longer located in our home directory.

The command to change locations is `cd` (“change directory”) followed by a directory name to change our working directory:

```
$ cd workshop
```

To check this worked:

```
$ pwd
```

And to see the content of the folder, including hidden files and directories:

```
$ ls -aFG
```

Special names:

`.`	→	this location
`..`	→	the directory above
`~`	→	the current user’s home directory, has to be at the start of specified path
`-`	→	the previous directory I was in

Intro to Bash – your turn

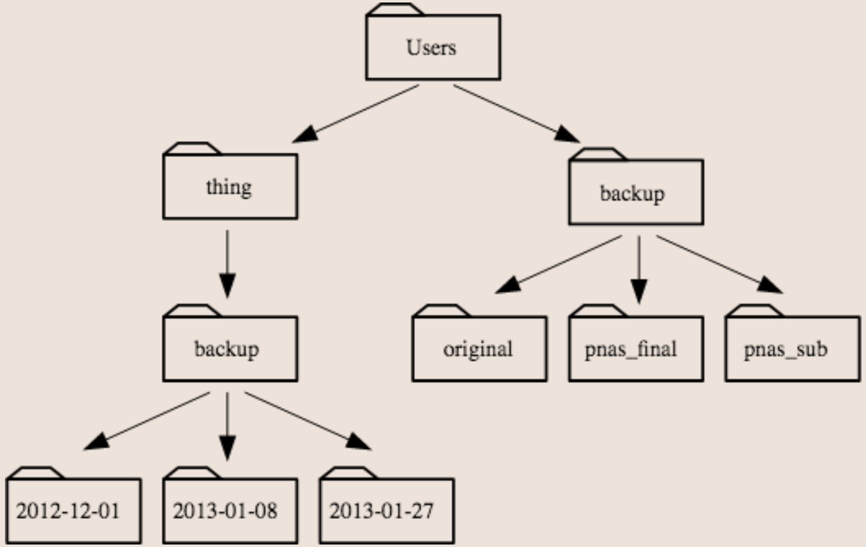
5:00
Stop

Starting from `/Users/amanda/data/`, which of the following commands could be used to navigate to the home directory, which is `/Users/amanda`?

1. `cd .`
2. `cd /`
3. `cd /home/amanda`
4. `cd ../..`
5. `cd ~`
6. `cd home`
7. `cd ~/data/..`
8. `cd`
9. `cd ..`

Using the filesystem diagram on the right:
if `pwd` displays `/Users/thing`,
what will `ls -F ../backup` display?

1. `../backup: No such file or directory`
2. `2012-12-01 2013-01-08 2013-01-27`
3. `2012-12-01/ 2013-01-08/ 2013-01-27/`
4. `original/ pnas_final/ pnas_sub/`



Intro to Bash – Working with files

Now that we know how to move around and explore our filesystem we need to learn how to **create** folders and files and **inspect their content**:

First check we are in the correct folder:

```
$ pwd
```

Then we can **make** a new **directory** (good file names do not use whitespace)

```
$ mkdir report
```

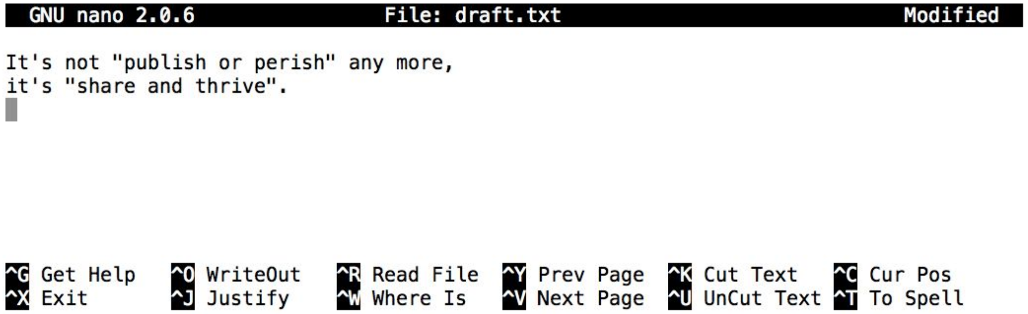
Check it was created and then move into it

```
$ ls -FG
```

```
$ cd report
```

Then we open a (new) text file to start a draft report

```
$ nano draft.txt
```



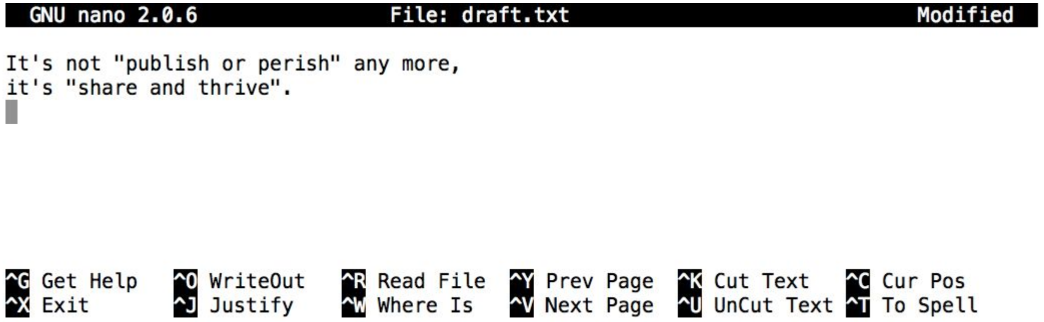
```
  GNU nano 2.0.6                File: draft.txt                         Modified

It's not "publish or perish" any more,
it's "share and thrive".


^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text ^T To Spell
```

Intro to Bash – Working with files

In nano, along the bottom of the screen you'll see `^G Get Help ^O WriteOut`. This means that you can use `Control-G` to get help and `Control-O` to save your file.

The Control key or "Ctrl" key, can be described in many ways, e.g. for press Ctrl and X:

- `Control-X`
- `Control+X`
- `Ctrl-X`
- `Ctrl+X`
- `^X`
- `C-x`



```
  GNU nano 2.0.6                 File: draft.txt                         Modified

It's not "publish or perish" any more,
it's "share and thrive".





^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell
```

Let's type a few lines, then save and exit the file.

We can use `ls` to check the file was created, to see the content we can use `cat` to print to screen (or concatenate):

```
$ cat draft.txt
```

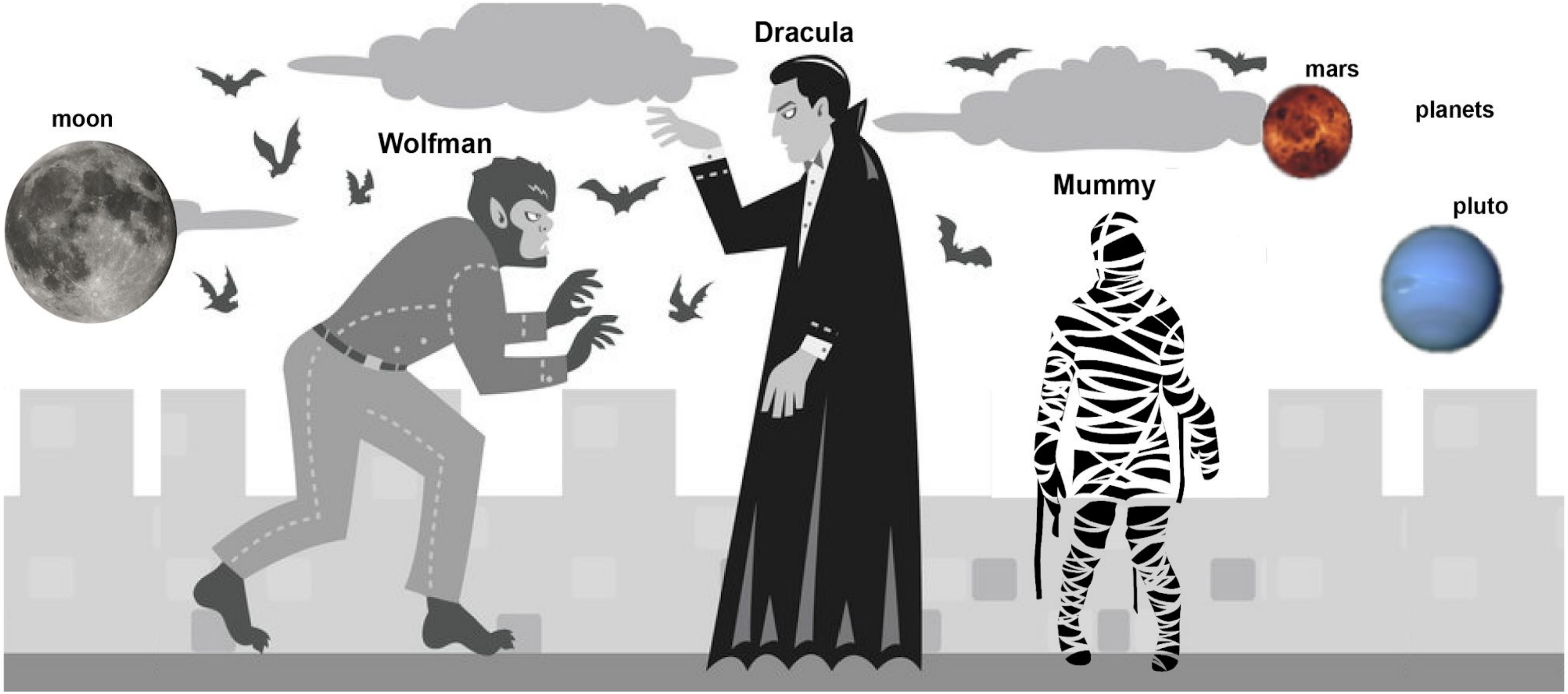
Another way to create files is the `touch` command, this creates an empty file with the specified name.

```
$ touch my_file.txt
```

Recap

`$ ls -Flag [location]`	list content of specified location, using specified flags
`$ pwd`	print working directory → current location in filesystem
`$ cd [location]`	change directory to specified location, relative paths work
`.` *and* `..`	special characters denoting *here* and *directory above*
`~` *and* `-`	special characters denoting *HOME* and *previous directory*
`$ mkdir [name]`	make directory with specified name (can include paths)
`$ nano [filename]`	open specified file using the *nano* text editor
`CTRL-O` *then* `<Enter>`	*nano* command to save content of file
`CTRL-X`	*nano* command to close file (asks confirmation if file changed)
`$ touch [filename]`	creates empty file with specified name if file does not exist

Getting started with Git – Premise



https://commons.wikimedia.org/wiki/File:Planets_are_us.png https://img00.deviantart.net/95f6/i/2009/156/f/8/werewolf_vs_dracula_by_b_maze.jpg
https://commons.wikimedia.org/wiki/File:Mummy_icon_-_Noun_Project_4070.svg https://commons.wikimedia.org/wiki/File:Lune_ico.png

Getting started with Git

Git commands are written as `git verb`, where `verb` is what we actually want to do:

```
$ git config --global user.name "Vlad Dracula"
$ git config --global user.email "vlad@tran.sylvan.ia"
```

The flag `--global` tells *Git* to use the settings for every project, in your user account, on this computer.

You can check your settings at any time:

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

Creating a repository

First, let's **create** a directory in `Desktop` folder for our work and then move into that directory:

```
$ cd ~/Desktop
$ mkdir planets
$ cd planets
```

Then we tell *Git* to make `planets` a **repository** - a place where *Git* can store versions of our files:

```
$ git init
```

To check if everything was successful:

```
$ ls -a
```

You should see a *.git* folder.

Warning: *Unless you are familiar with* Git *it is generally best to avoid touching the .git folder or it's contents.*

We can check that everything is set up correctly by asking Git to tell us the **status** of our project:

```
$ git status
```

Tracking changes – creating new files

Let’s **create** a file called `mars.txt` that contains some notes about the red planet’s suitability as a base.
I’ll use `nano` to edit the file; you can use whatever editor you like.

```
$ nano mars.txt
```

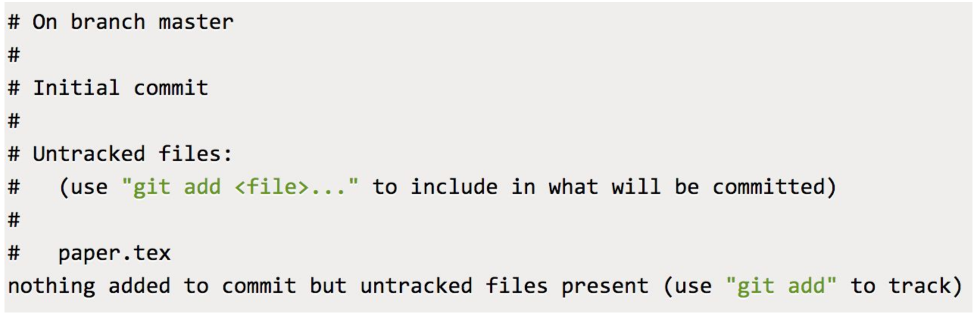
Type the text below into the `mars.txt` file, save and close it:

```
Cold and dry, but everything is my favorite color
```

To check all was saved correctly: `$ cat mars.txt`

If we check the **status** of our project again, Git tells us that it’s noticed the new file:

```
$ git status
```



```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   paper.tex
nothing added to commit but untracked files present (use "git add" to track)
```

Tracking changes - adding files

We can **add** our file to the **staging area**:

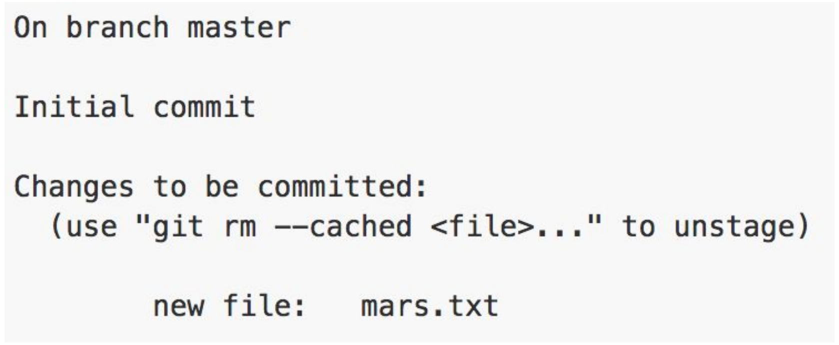
```
$ git add mars.txt
```

- You can add more than one file by listing the ones you want to add
- You can also use -A flag to add all files to the staging area

Note: all files staged before a commit will have the same commit message

If we check the **status** of our project again, Git tells us that it's noticed the new file:

```
$ git status
```



```
On branch master

Initial commit

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

        new file:   mars.txt
```

Tracking changes - commiting files

To **commit** our changes to the repository we need to run one more command:

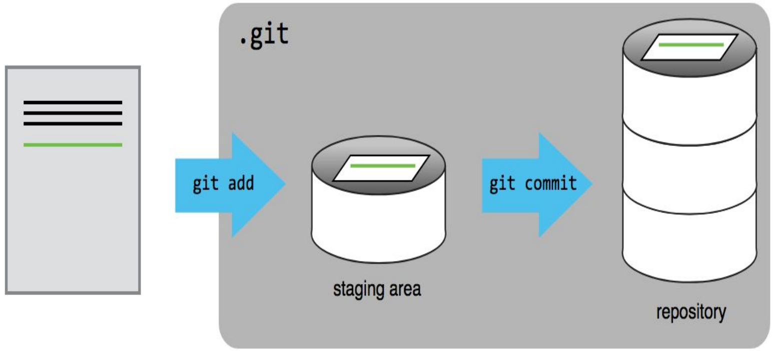
```
$ git commit -m "Start notes on Mars as a base"
```

```
[master (root-commit) f22b25e] Start notes on Mars as a base
 1 file changed, 1 insertion(+)
 create mode 100644 mars.txt
```

Git is always a **two-step process** to track changes:

```
$ git add [filename]
```

```
$ git commit -m "brief statement (<50 char) about the changes"
```



Tracking changes – your turn

5:00

Stop

Edit the **mars.txt** file so that it reads:

Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity

then **add** and **commit** it to your repository

Tracking changes – your turn

```
$ nano mars.txt
```

[add text]

> The two moons may be a problem for Wolfman

> But the Mummy will appreciate the lack of humidity

[save and exit]

[add file and commit changes]

```
$ git add mars.txt
```

```
$ git commit -m “adding consideration about base suitability for team”
```

Recap

`$ git init`	initiate the repository
`$ git status`	check the current state of the repository
`$ git add [filename]`	stage a file / changes made to a file
`$ git commit -m “useful message”`	commit staged files / changes

Ignoring things

Say we have intermittent output **files** from our program that **we do not want to track**, as they

a) change frequently, and
b) can easily be reproduced

→ tracking them would waste disk space

```
$ mkdir results
```

```
$ touch a.dat b.dat c.dat results/a.out results/b.out
```

To **ignore** files in your repository create a *.gitignore* file listing all files to skip:

```
$ nano .gitignore
```

> *.dat

> results/

[save and exit]

Don’t forget to **track** your .gitignore file!

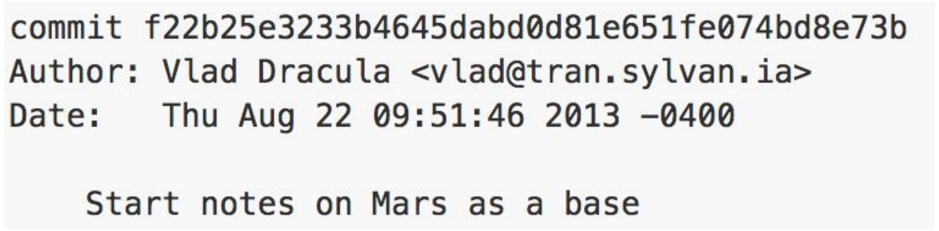
```
$ git add .gitignore
```

```
$ git commit -m "Ignore data files and the results folder."
```

Exploring our changes

If we want to know what we've done recently, we can ask *Git* to show us the **repo history**:

```
$ git log
```



```
commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2013 -0400

    Start notes on Mars as a base
```

`git log` lists:

- all commits made to a repository in reverse chronological order.
- the commit's full identifier,
- the commit's author,
- when it was created, and
- the log message Git was given when the commit was created.

Exploring our changes

You can **see the changes** made to an unstaged file compared to the last commit with

```
$ git diff [filename]
```

Other variations of this:

`$ git diff --staged`	comparing the staged file(s) to the last commit
`$ git diff [commit] [commit]`	comparing two commits using unique identifiers

```
$ git diff HEAD~1

diff --git a/mars.txt b/mars.txt
index df0654a..315bf3a 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,2 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
```

An ill considered change

Let’s start a **new file** on Venus as a base.

```
$ nano venus.txt
```

Add some info on Venus and save the file.

Next we will also add a new line to our mars.txt:

```
$ nano mars.txt
```

> An ill considered change

Save and exit.

Now lets **stage and commit** those files

```
$ git add -A
$ git commit -m “started notes on Venus”
```

An ill considered change

Let’s check everything is up to date and committed

```
$ git status
```

We have realised that our **changes** to mars.txt **were incorrect**, so let’s roll them back!
We want to roll back to the commit before the last, so let’s find out its identifier

```
$ git log --oneline
```

To roll back our changes we will **checkout** the mars.txt file from HEAD~1:

```
$ git checkout HEAD~1 mars.txt
```

Check that the file has changed as expected

`$ cat mars.txt` or `$ git diff mars.txt`

Note, mars.txt has been modified so you will need to stage and commit it again!

```
$ git add mars.txt
$ git commit -m “reverted changes back to commit [number]”
```

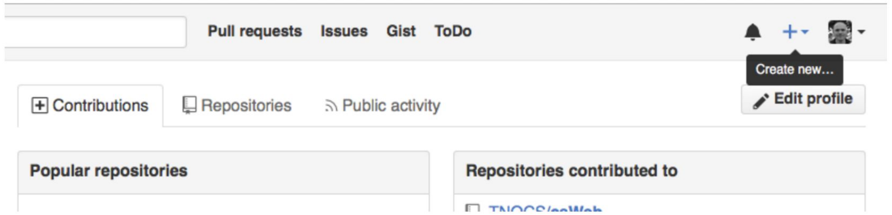
Recap

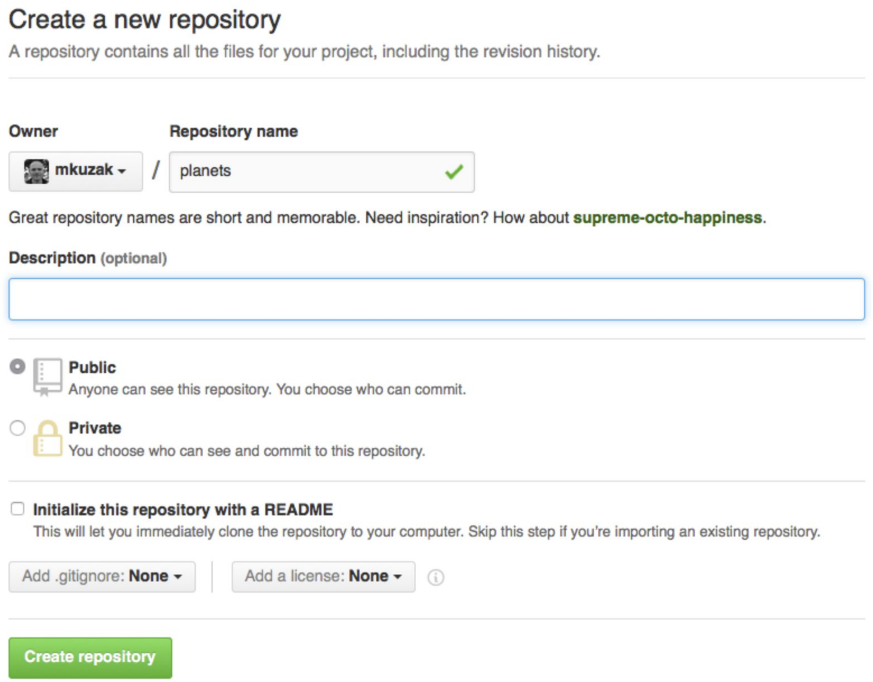
`$ .gitignore`	listed files and folders will not be tracked
`$ git log`	show a log of the commit history
`$ git log --oneline`	show commit history with one line per commit
`$ git diff [filename]`	compare current, unstaged file to latest commit
`$ git diff --staged`	compare staged file(s) to the last commit
`$ git diff [commit] [commit]`	comparing two commits using unique identifiers
`$ git checkout [commit] [file]`	roll (specified file) back to specific commit
`$ HEAD`	denotes the latest commit
`$ HEAD~i`	denotes the *i*th commit before the last

Remote repositories on GitHub

Let's start by sharing the changes we've made to our current project with the world.

- Log in to GitHub,
- create a new repository called `planets`





Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: mkuzak / Repository name: planets

Great repository names are short and memorable. Need inspiration? How about **supreme-octo-happiness**.

Description (optional)

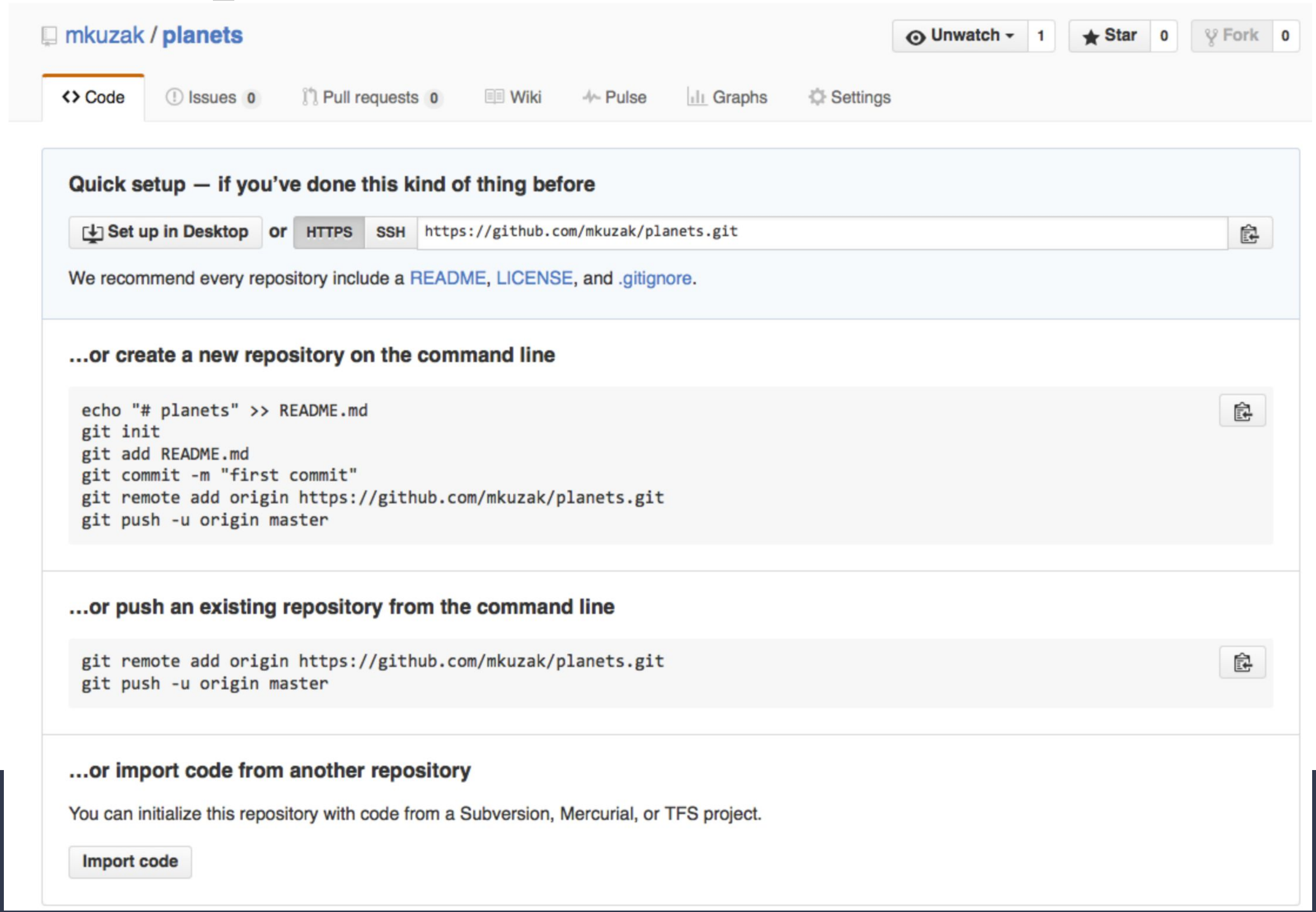
[✓ Public] Anyone can see this repository. You choose who can commit.

Private — You choose who can see and commit to this repository.

Initialize this repository with a README — This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None

Remote repositories on GitHub



mkuzak / planets

Unwatch 1 · Star 0 · Fork 0

Code · Issues 0 · Pull requests 0 · Wiki · Pulse · Graphs · Settings

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH `https://github.com/mkuzak/planets.git`

We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# planets" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/mkuzak/planets.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/mkuzak/planets.git
git push -u origin master
```

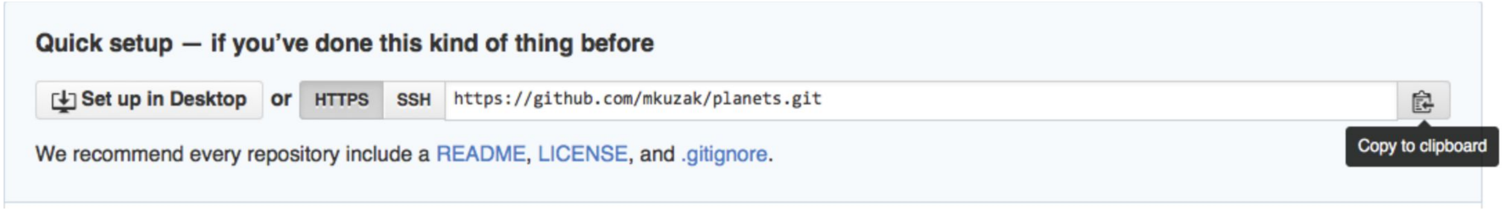
...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

Remote repositories on GitHub

Our remote is still empty, so let's populate it



Quick setup — if you've done this kind of thing before

```
https://github.com/mkuzak/planets.git
```

We recommend every repository include a README, LICENSE, and .gitignore.

Copy the URL and go back to your terminal

```
$ git remote add origin [URL to your github repository]
```

We can check the remote was set up correctly with

```
$ git remote -v
```

Remote repositories on GitHub

The name `origin` is a local **nickname** for your remote repository.

To push your local changes to GitHub use:

```
$ git push origin master
```

Note: the `-u` option is synonymous with the `--set-upstream-to` option for the `git branch` command, and is used to associate the current branch with a remote branch so that the `git pull` command can be used without any arguments.

We can **pull changes from the remote** repository to the local one as well:

```
$ git pull origin master
```

Recap

`$ git remote add origin [URL]`	link an empty remote repo to your local repo
`$ git push origin master`	push your local changes to the remote repo
`$ git pull origin master`	pull changes from the remote repo
`$ git remote -v`	show nickname and URL of remote repo(s)

`$ git clone [URL] [location]`	clone a remote repository to your computer
`$ git push -u origin master`	push your local changes to the remote repo and set

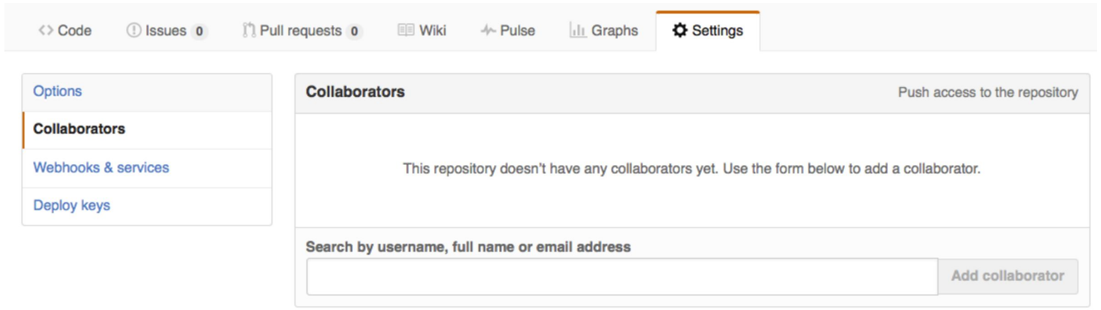
specified remote as you *upstream* → think of it as setting up the default so now you can update without specifying the remote nickname and branch using

`$ git push` or `$ git pull`

Remote repositories – your turn

For the next step, get into pairs. One person will be the "**Owner**" and the other will be the "**Collaborator**".

The Owner needs to give the Collaborator access.



Collaborators

Push access to the repository

This repository doesn't have any collaborators yet. Use the form below to add a collaborator.

Search by username, full name or email address

Next, the Collaborator needs to download a copy of the Owner's repository to her machine. This is called "**cloning** a repo".

```
$ git clone [URL] [local file path]
```

If the *local file path* is not specified then the cloned repo will be saved in the current location using the repo name.

Caution as we all called our repos *planets*, make sure you do not overwrite your own!

Remote repositories – your turn

The Collaborator can now make a change in her clone of the Owner’s repository, exactly the same way as we’ve been doing before:

```
$ cd ~/Desktop/workshop/[new-repo-name]
```

```
$ nano pluto.txt
```

Add some text, then save and exit

```
$ cat pluto.txt
```

Next, stage file and commit changes

```
$ git add pluto.txt
```

```
$ git commit -m “added notes about Pluto”
```

Then push the change to the Owner’s repository on GitHub:

```
$ git push origin master
```

Since we **cloned** the repo it automatically records the URL it was copied from as its remote location.

Remote repositories – collaboration

Take a look at the Owner's repository on its GitHub website now (maybe you need to refresh your browser). You should be able to see the new commit made by the Collaborator.

To download the Collaborator's changes from GitHub, the Owner now enters:

```
$ git pull origin master
```

Everything should now be in sync again.

A basic collaborative workflow

In practice, it is good to be sure that you have an updated version of the repository you are collaborating on, so you should `git pull` before making our changes. The basic collaborative workflow would be:

- update your local repo with `git pull origin master`,
- make your changes and stage them with `git add`,
- commit your changes with `git commit -m “commit message”`, and
- upload the changes to GitHub with `git push origin master`

It is better to make many commits with smaller changes rather than of one commit with massive changes: small commits are easier to read and review.

Remote repositories – collaboration

- Switch roles and repeat the whole process.
- Added challenge, work on the same file.

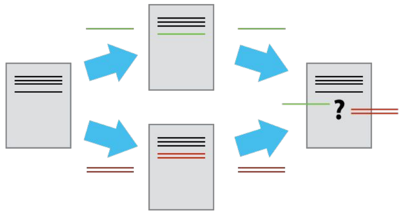
  → What happens if you want to push changes to a remote repo that is out of sync with your local version?

Remote repositories – conflicts

If you try to push changes to a remote repo which contains files out of sync with your local repo your changes will be **rejected**.

```
$ git push origin master
```

```
To https://github.com/vlad/planets.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'https://github.com/vlad/planets.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```



What we have to do is `pull` the changes from GitHub, `merge` them into the copy we're currently working in and then `add`, `commit`, and `push` them.

```
$ git pull origin master
```

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Auto-merging mars.txt
CONFLICT (content): Merge conflict in mars.txt
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
<<<<<<< HEAD
We added a different line in the other copy
=======
This line added to Wolfman's copy
>>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d
```

The conflicting change is preceded by `<<<<<<< HEAD`. Git has then inserted `=======` as a separator between the conflicting changes and marked the end of the content downloaded from GitHub with `>>>>>>>`. (The string of letters and digits after that marker identifies the commit we've just downloaded.)

Your turn: the person who encountered the conflict can now resolve it.
Once their repo and the remote are in sync the other person can pull the changes.

5:00

Stop