An Introduction to `git` (and GitHub):
A Distributed Version Control System

Marty Kandes, Ph.D.

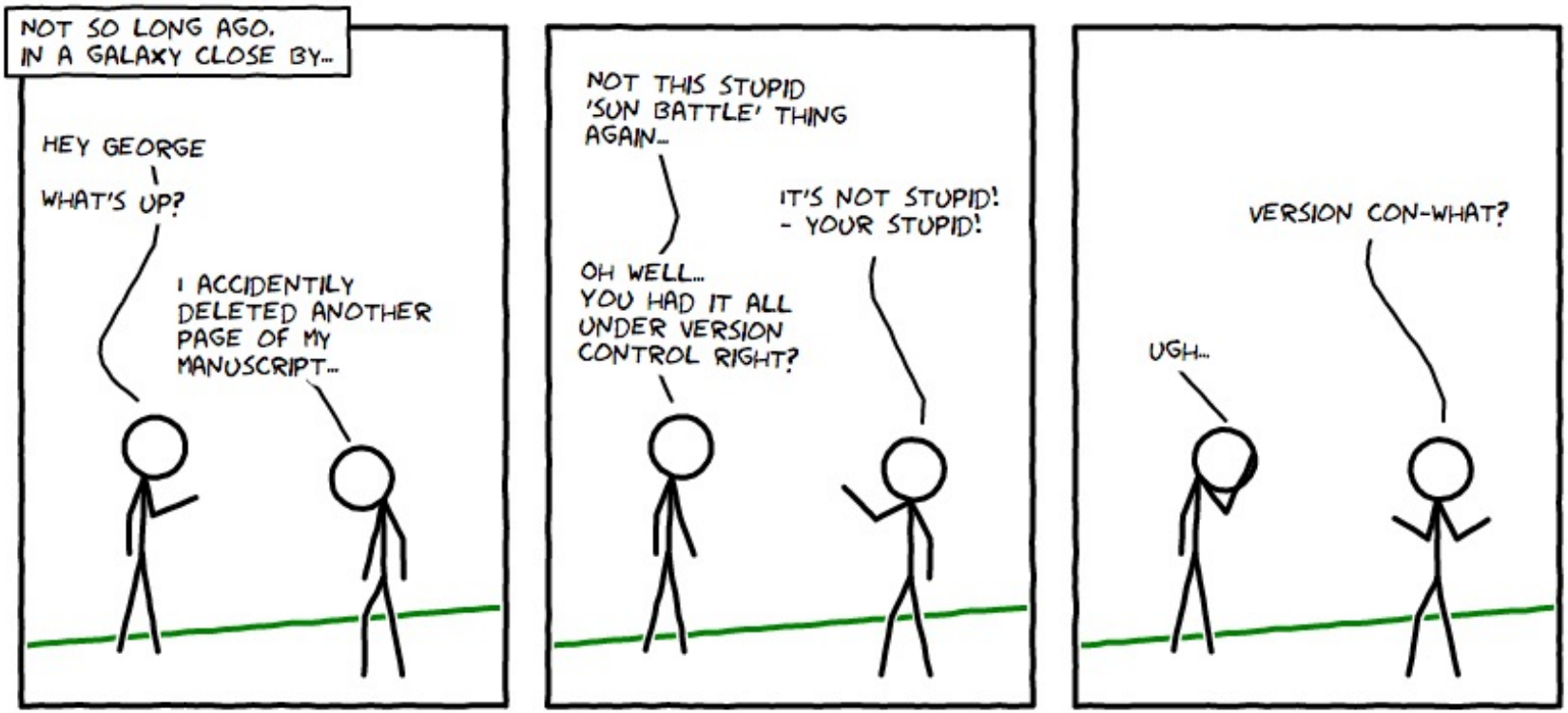
High-Performance Computing User Services Group
San Diego Supercomputer Center
University of California, San Diego

SDSC Summer Institute 2020
Monday, August 3rd, 2020
10:45AM - 12:45PM PDT

Today's Session

- An Overview of Version Control
- About `git`
- Getting Started with `git`
- Working with `git`
- Getting Started with GitHub
- Summary and Conclusion
- Q&A

An Overview of Version Control



What is version control?

Version control is the process of recording and managing changes to a file or set of files over time so that you can recall specific versions later, if needed.

What do we version control?

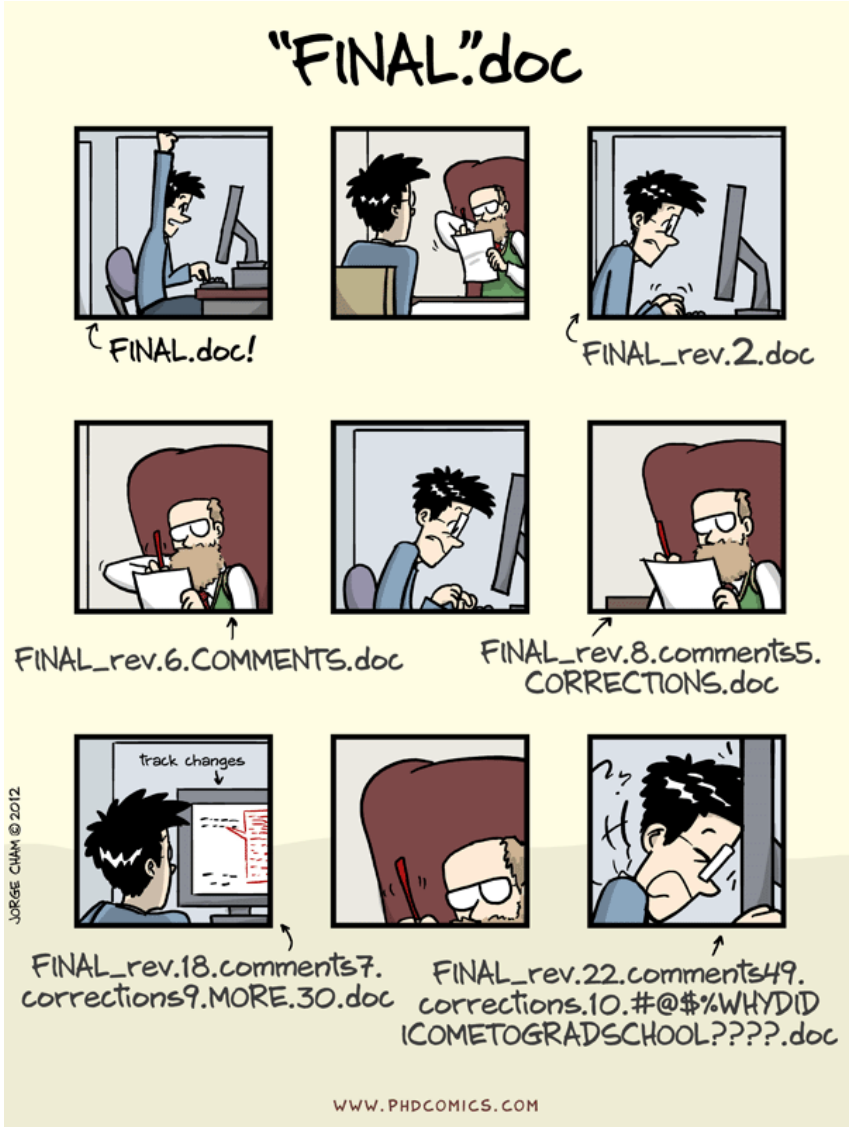
The files we're typically interested in being version controlled are *software source code* files. However, you can version control almost any type of file.

What *should* we version control?

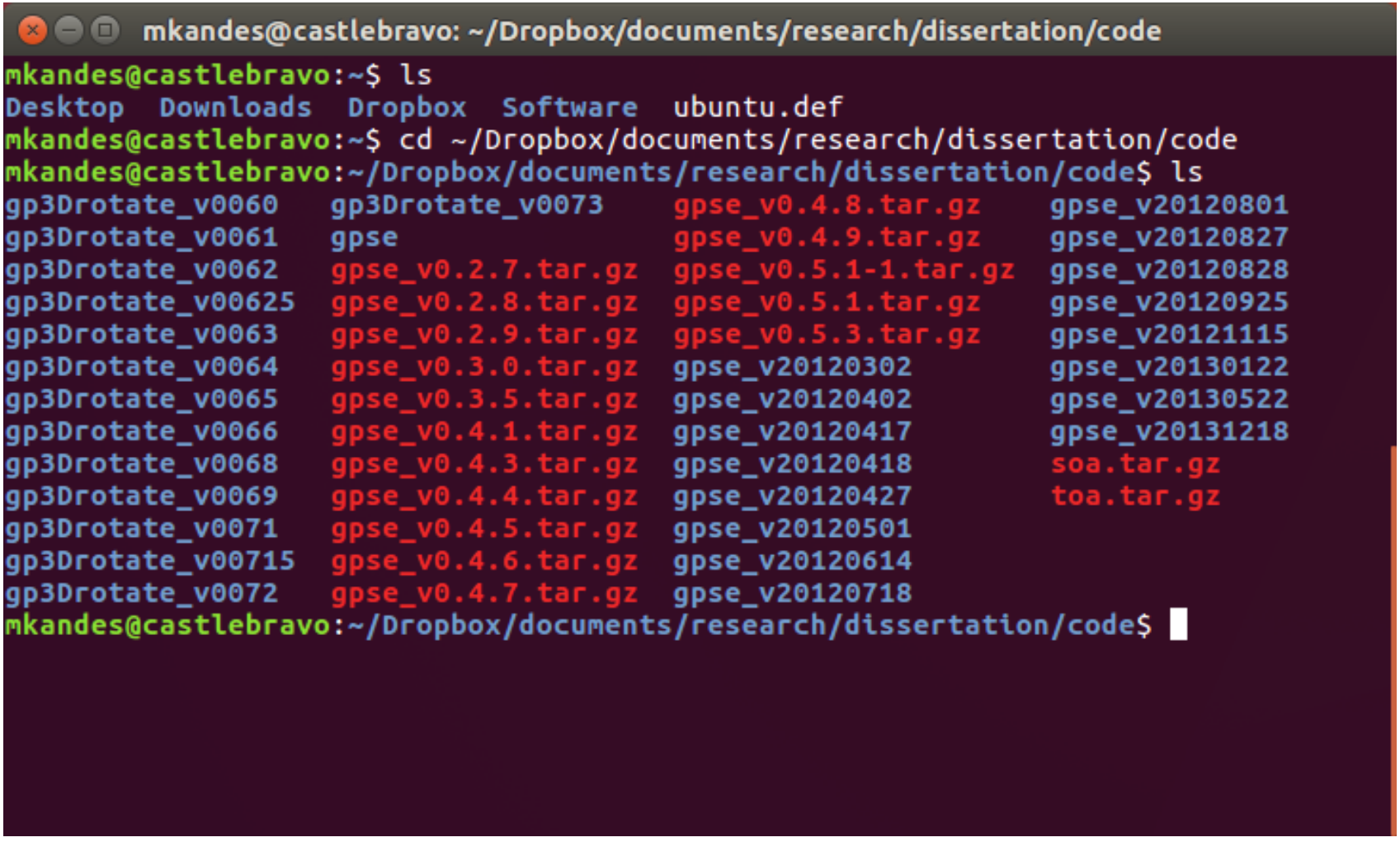
"In practice, everything that has been created manually should be put in version control, including programs, original field observations, and the source files for papers."

Best Practices for Scientific Computing
Wilson et al. 2012 (arXiv:1210.0530)

Have you used version control before?



Yes, yes I have ...

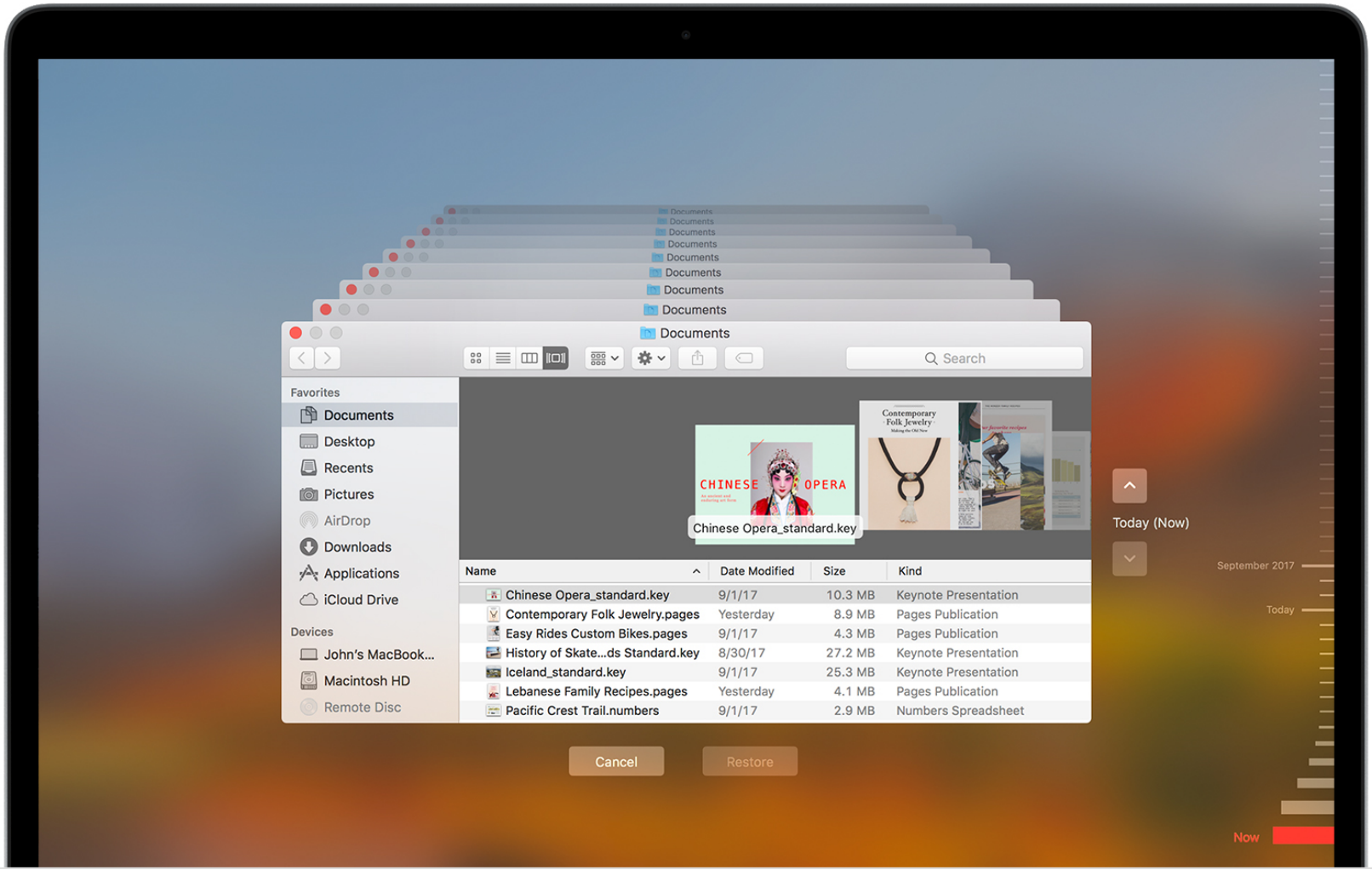


https://github.com/mkandes/gpse

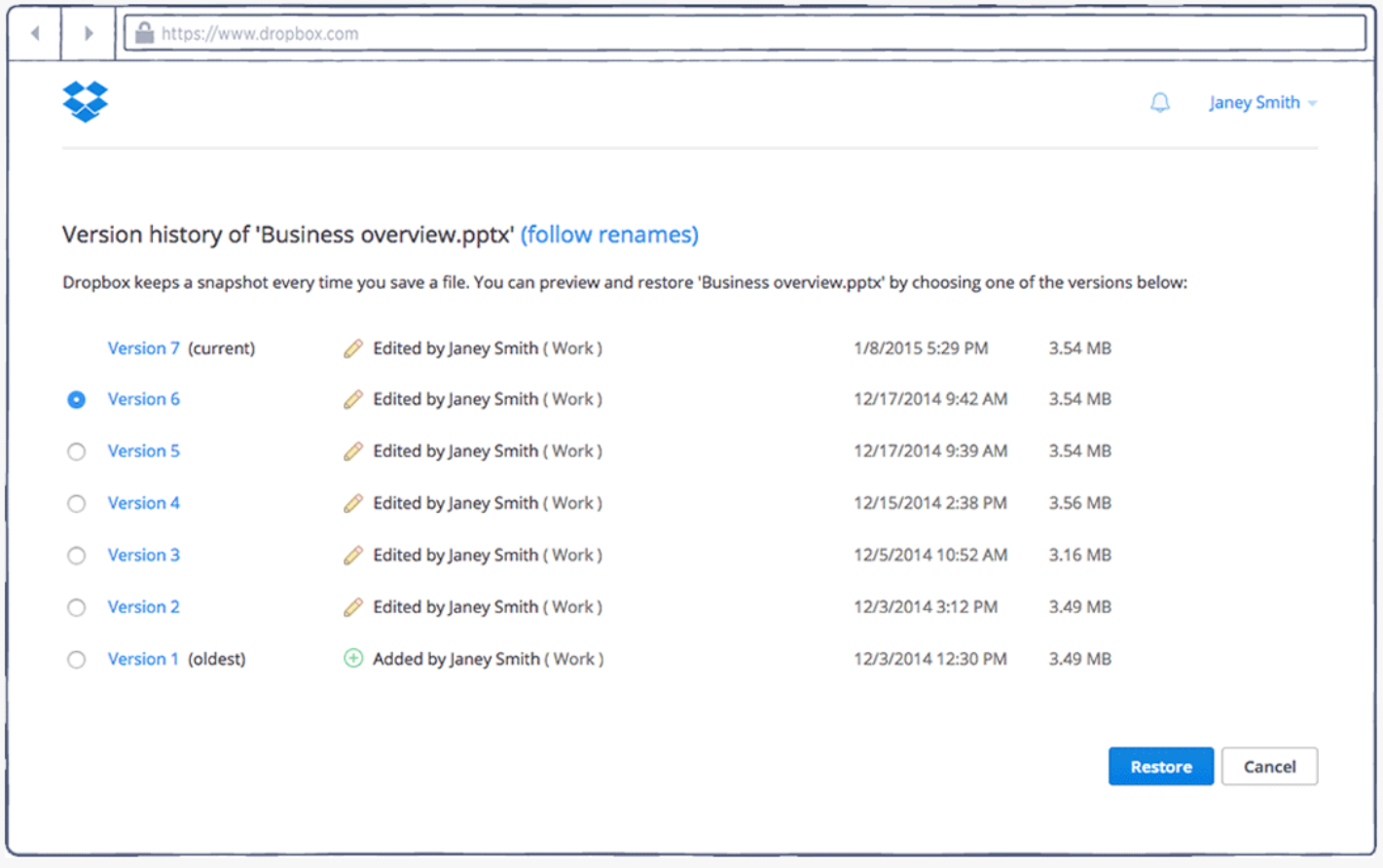
What is a version control system?

- A *version control system* is a type of software tool that helps record and manage the changes you make to the file or set of files you want version controlled.
- Version control systems keeps track of *every* modification you make to the files being tracked in a special kind of database.
- If a mistake is made, you can then turn back the clock and compare earlier versions of the files to help fix the mistake.

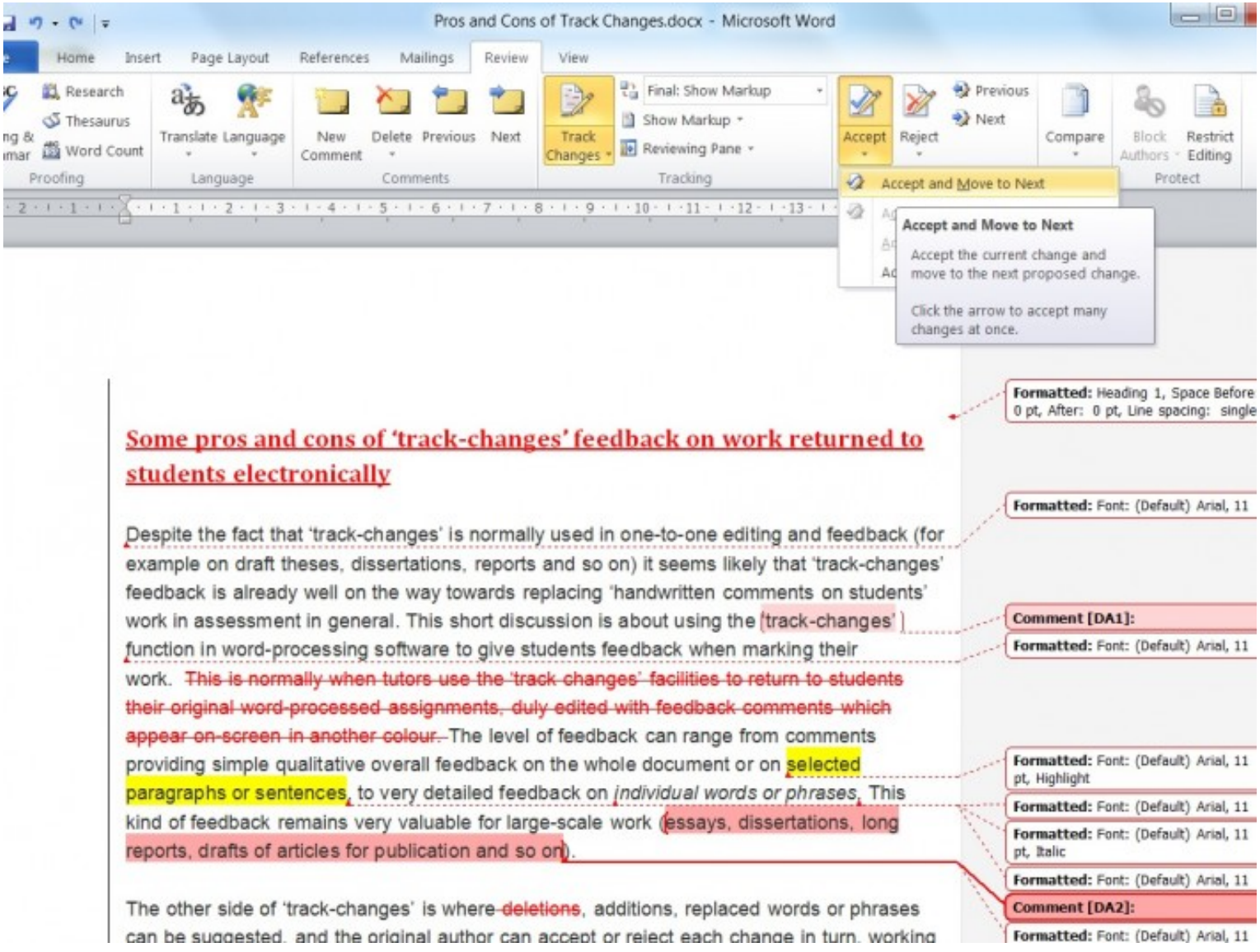
Example: Apple's Time Machine



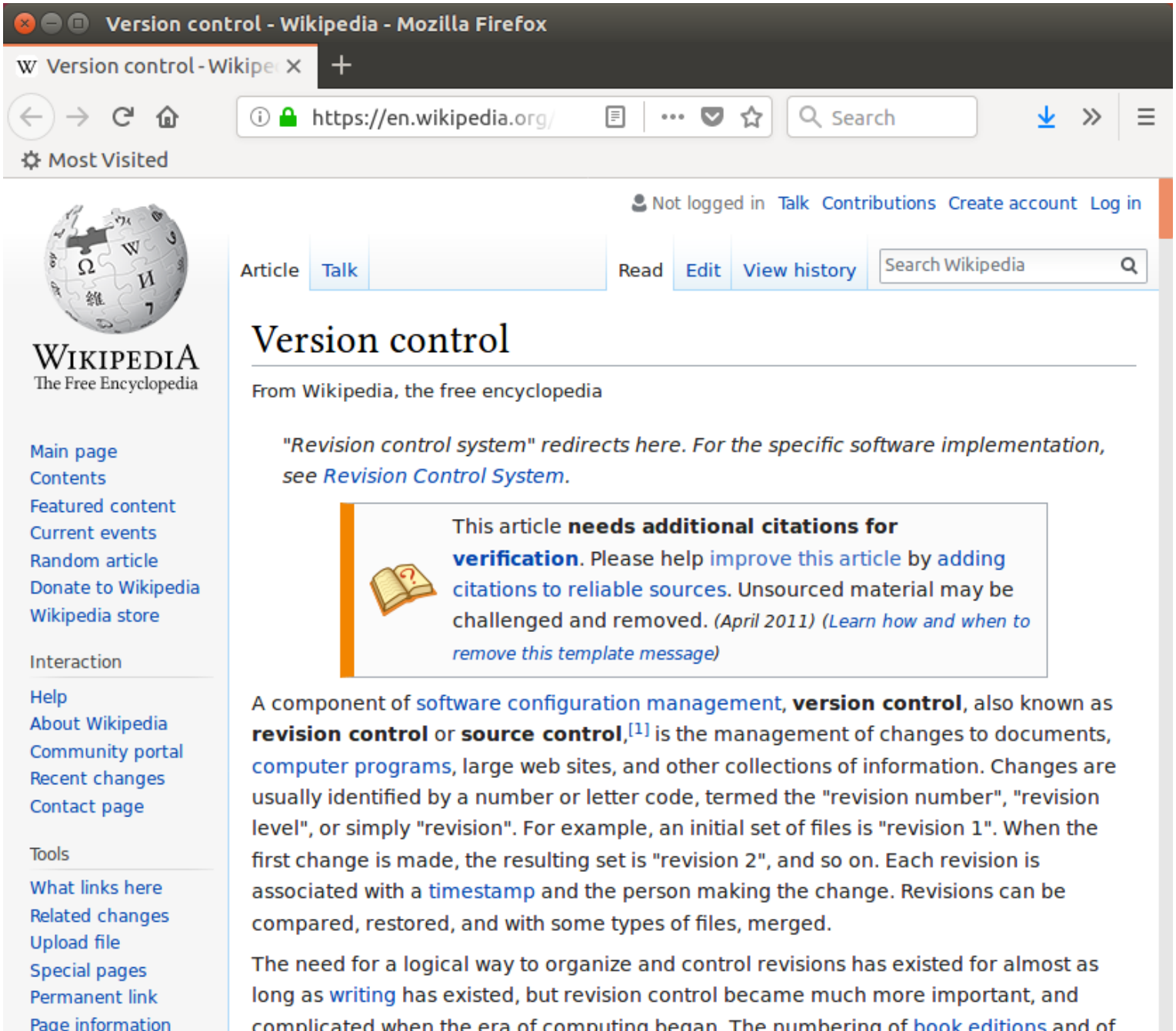
Example: Dropbox



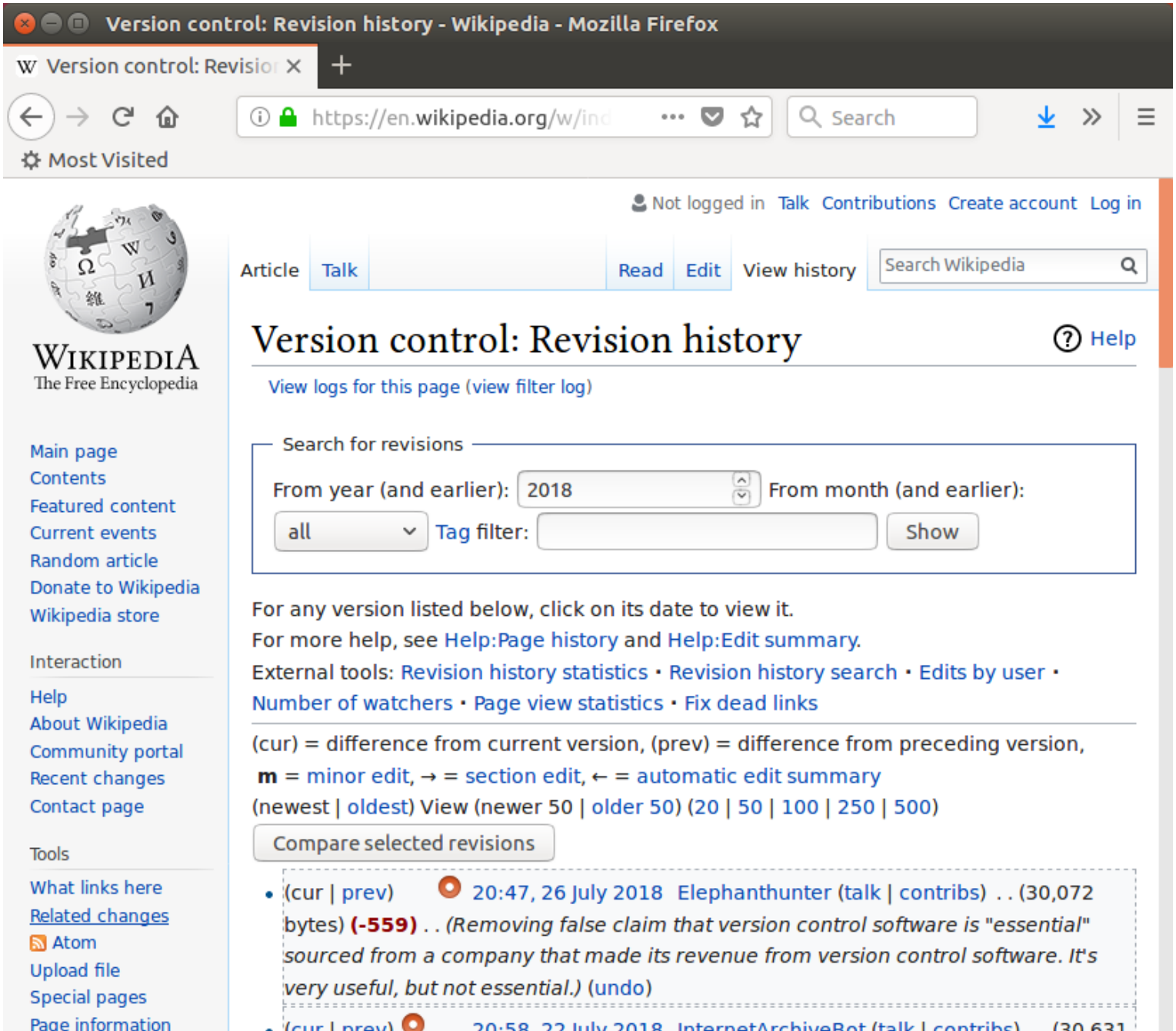
Example: Microsoft Word's Track Changes



Example: Wikipedia's View History



Example: Wikipedia's View History



How does version control work? Record and Playback.

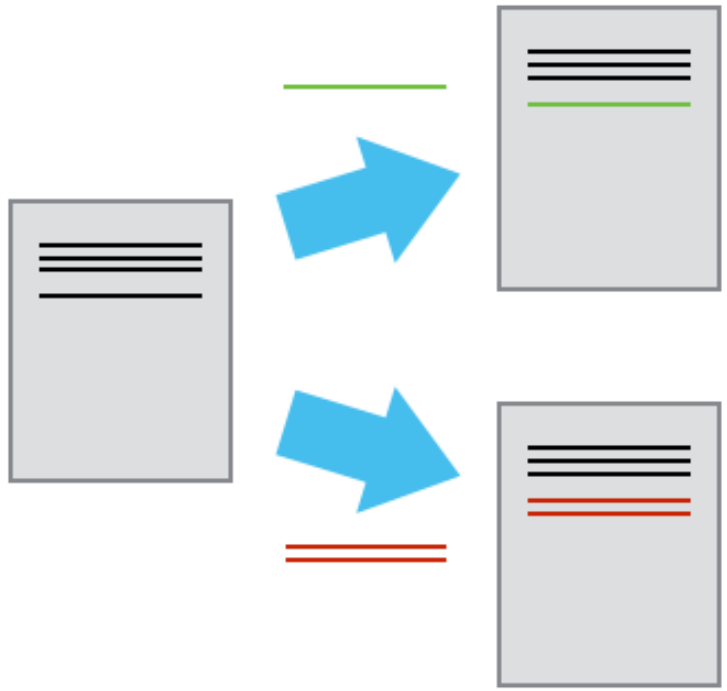
Version control systems start with a base version of a document and then simply record the changes you make each step of the way.

This allows you to rewind to start of the base document and playback each change you made, allowing you to recreate any version of the document in its recorded history.

How does version control work? Branching.

By recording the entire history of changes to a document and using the playback of those changes to recreate any version of it, the management and creation of new, independent versions or *branches* of the document itself becomes quite easy.

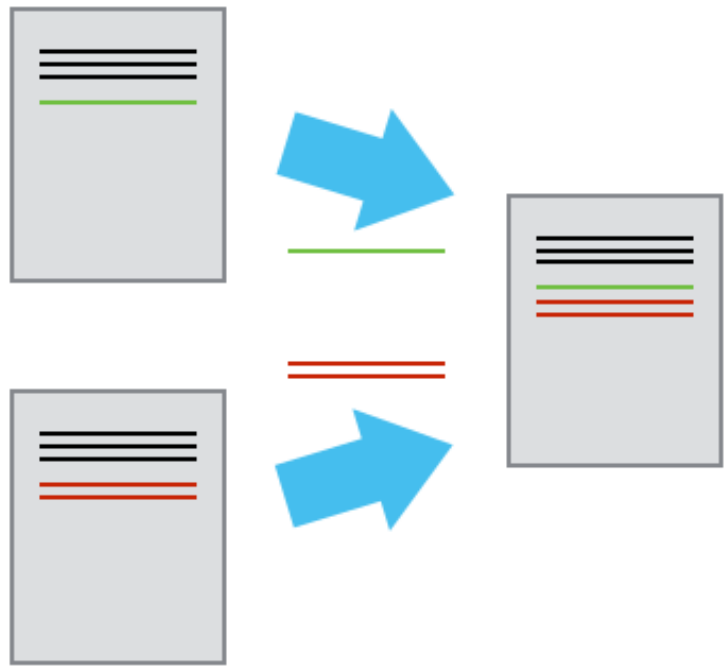
This *branching* helps facilitate the collaboration amongst multiple contributors to a single, shared document since each contributor simply needs to record their independent history of changes.



How does version control work? Merging.

When a contributor to a document wants to share their changes with their collaborators, they simply need to *merge* their changes into some common branch of the document shared by the team.

As long as there are no conflicts when the contributor's independent history of changes are played back on the common, shared branch of the document, the changes are *merged* and become a part of the shared history of the document.



Benefits of Using a Version Control System

- **Archiving**: You *must* regularly save the changes you make.
- **Reproducibility**: Creating a history of saved changes allows you to revert selected files or your entire project back to any previous state in its recorded history.
- **Collaboration**: You and a team of contributors can work on a project independently, but share your changes amongst one another as the project takes shape.
- **Accountability**: Compare changes, see who last modified something, and who introduced a problem and when.
- **Recoverability**: Each contributor has their own local, recent copy of the project and its complete history, making it highly unlikely you'll ever lose a significant portion of the project.

About git

https://git-scm.com

What is `git`?

- `git` is a *distributed* version control system.
- `git` was designed to be *simple*, *fast*, and *fully-distributed*, with support for large projects and nonlinear development workflows.
- `git` was originally created in 2005 by Linus Torvalds and other Linux kernel developers when the free use of the proprietary version control system they had been using for kernel development was revoked by its copyright holder.

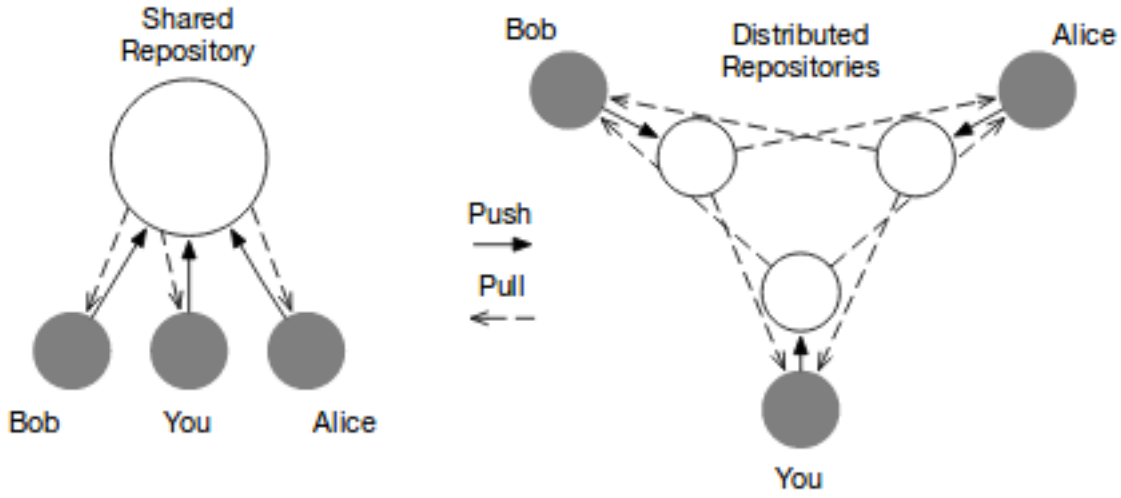
How is `git` different? Centralized vs. Distributed

In general, there are two models for version control systems:

- **Centralized**: Whenever you make a change and want to `commit` that change, you must send it back to a centralized *repository* server to be tracked. However, this makes sharing your changes with others who have access to the server immediate once the change is committed.

- **Distributed**: The process of committing changes and sharing them with others broken up into a two-step process. This allows you to commit changes in a *private repository* on your local computer, which eliminates the need to be connected to a network where the centralized server is accessible in order to commit a change.

How is git different? Private vs. Public Repositories



Gray circles are the private repositories; outlined circles are public repositories.

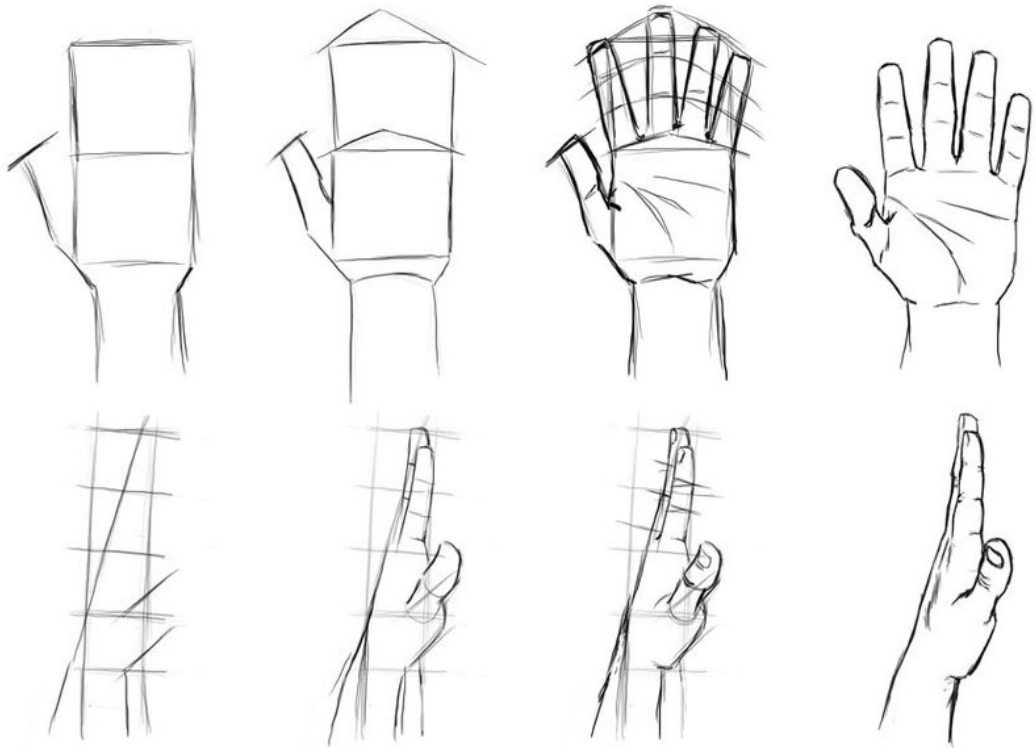
When using git, each contributer who is sharing changes with others on a project has at least two repositories:

- a **private repository** is the one that exists on your local computer and is the one you make commits to;
- a **public repository** is the one that you use to share your changes with collaborators.

The `git` workflow

1. Update your private repository by `fetching` all recent changes committed by your team of collaborators from the shared public repository (or their individual public repositories).
2. Make changes and `commit` them to your private repository.
3. Review your committed changes.
4. Share your new changes with the team by `pushing` your committed changes back to the team's shared public repository (or your individual public repository)
5. Rinse and repeat.

Getting Started with `git`



Installing `git` on Linux

If you want to install `git` on a Linux-based system, you should be able to do so via your operating system's standard package management tool.

For example, on any RPM-based Linux distribution, such as Fedora, RHEL, or CentOS, you can use `dnf`:

```
$ sudo dnf install git-all
```

On any Debian-based distribution, such as Ubuntu, try `apt`:

```
$ sudo apt install git-all
```

Installing `git` on Mac OS X

There are several ways to install `git` on your Mac. Probably the easiest way is to install the Xcode Command Line Tools, which you can do by simply trying to run `git` from the Terminal:

```
$ git --version
```

If not already installed, you should be prompted to install it.

If the above option does not work or you need a more up-to-date version of `git`, you can always install it via a binary installer maintained by the `git` team, which is available for download at:

`https://git-scm.com/download/mac`

Installing git on Windows

There are also a few ways to install git under Windows operating systems. However, the official build is available for download on the git website. If you go to

`http://git-scm.com/download/win`

the download should start automatically.

Note: This is a project called "git for Windows", which is separate from git itself. Please see

`https://gitforwindows.org`

for more information.

Windows Subsystem for Linux

The *Windows Subsystem for Linux* lets you run a GNU/Linux environment – including most command-line tools, utilities, and applications – directly on Windows, unmodified, without the overhead of a traditional virtual machine or dualboot setup.

`https://docs.microsoft.com/en-us/windows/wsl`

Setting up `git` with `git config`

When we start using `git` for the first time on a new computer, there are always a few things we need to configure before we begin working with `git`.

The `git config` command lets you look up and set configuration variables that control all aspects of how `git` works.

Setting up `git` with `git config`

These configuration variables may be stored in a few different places on your system:

1. `/etc/gitconfig`: This file contains configuration variables applied to every user on the system and all of their repositories.

2. `~/.gitconfig`: This file contains configuration variables that apply only to you and **all** of your repositories on the system.

3. `~/path/to/your/repo/.git/config`: This file contains configuration variables that apply only to this repository.

Setting up your identity with `git config`

The first configuration variables you should set right after you install `git` for the first time are your `user.name` and `user.email` variables, which identify who you are and how to contact you.

```
$ git config --global user.name "Marty Kandes"
$ git config --global user.email "mkandes@sdsc.edu"
```

These configuration variables are important because this information is included in every `git commit` you make in order to provide a history of who made what changes.

Note: The `--global` option used here places the values of these configuration variables in your personal `~/.gitconfig`.

Setting up your text editor with `git config`

After you've set up your identity, you can configure the default text editor that will be used by `git` when you need to edit a `commit` message, etc.

For example, if you want to use a different text editor, such as Emacs, you can type the following:

```
$ git config --global core.editor "emacs"
```

If not configured, `git` will use your system's default editor.

Setting up colors with `git config`

In addition to your identity and default text editor, you'll also want to make sure that you turn on color highlighting wherever possible in `git`'s user interface. You can do so with the following command:

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

Checking your `git config`

To check the configuration variables you've set in your `git config`, you can run the command

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

to list all of the variable values.

If you only need to check one variable in the configuration, you can always replace the `--list` option with the name of the configuration variable itself. For example, if you wanted to check the value of `user.email` you've set, you'd run the command:

```
$ git config user.email
```

Getting `git help`

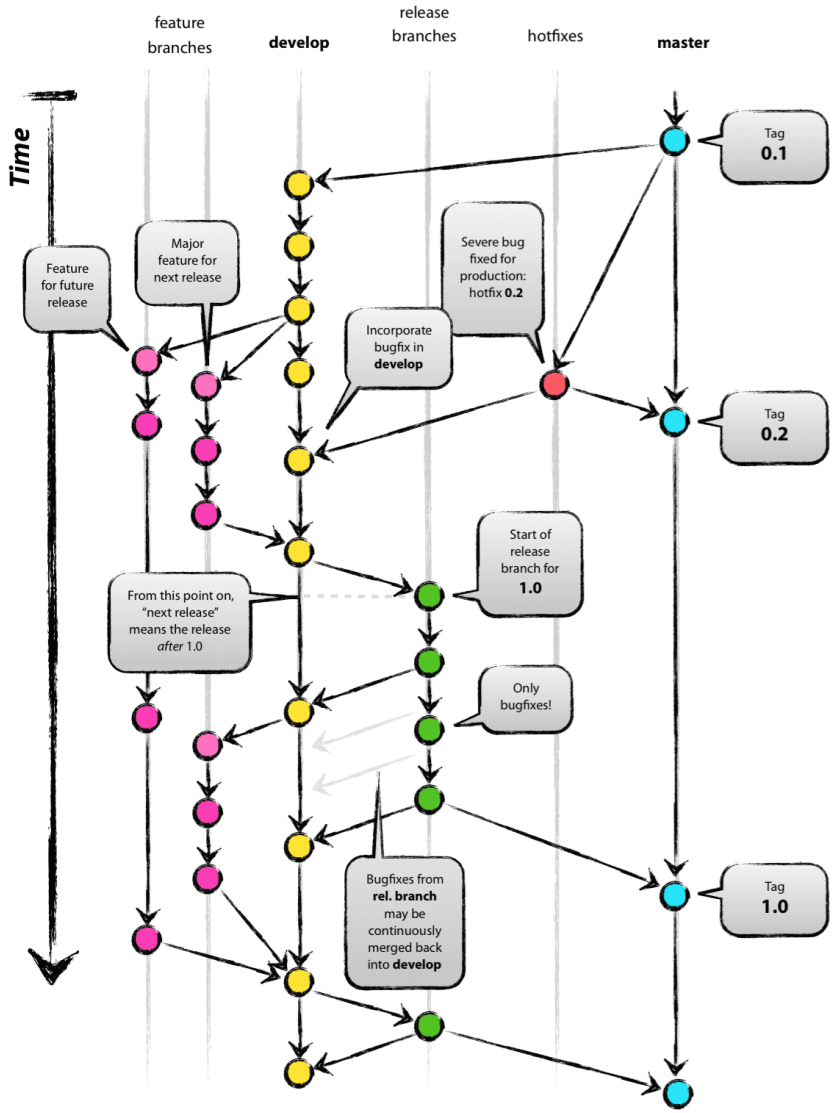
If you ever need `help` while using `git`, you can access the comprehensive manual page (manpage) for any `git` command via the `git help` command:

```
$ git help <command>
```

If you only need a quick reference to the options available for a given `git` command, you can use the `-h` or `--help` options appended to the command itself:

```
$ git <command> -h
$ git <command> --help
```

Working with git



Creating your first `git` repository with `git init`

If you have a project on your local system that is not currently under version control and you want to start tracking it `git`, you can create a git repository for that project by first going to that project's directory

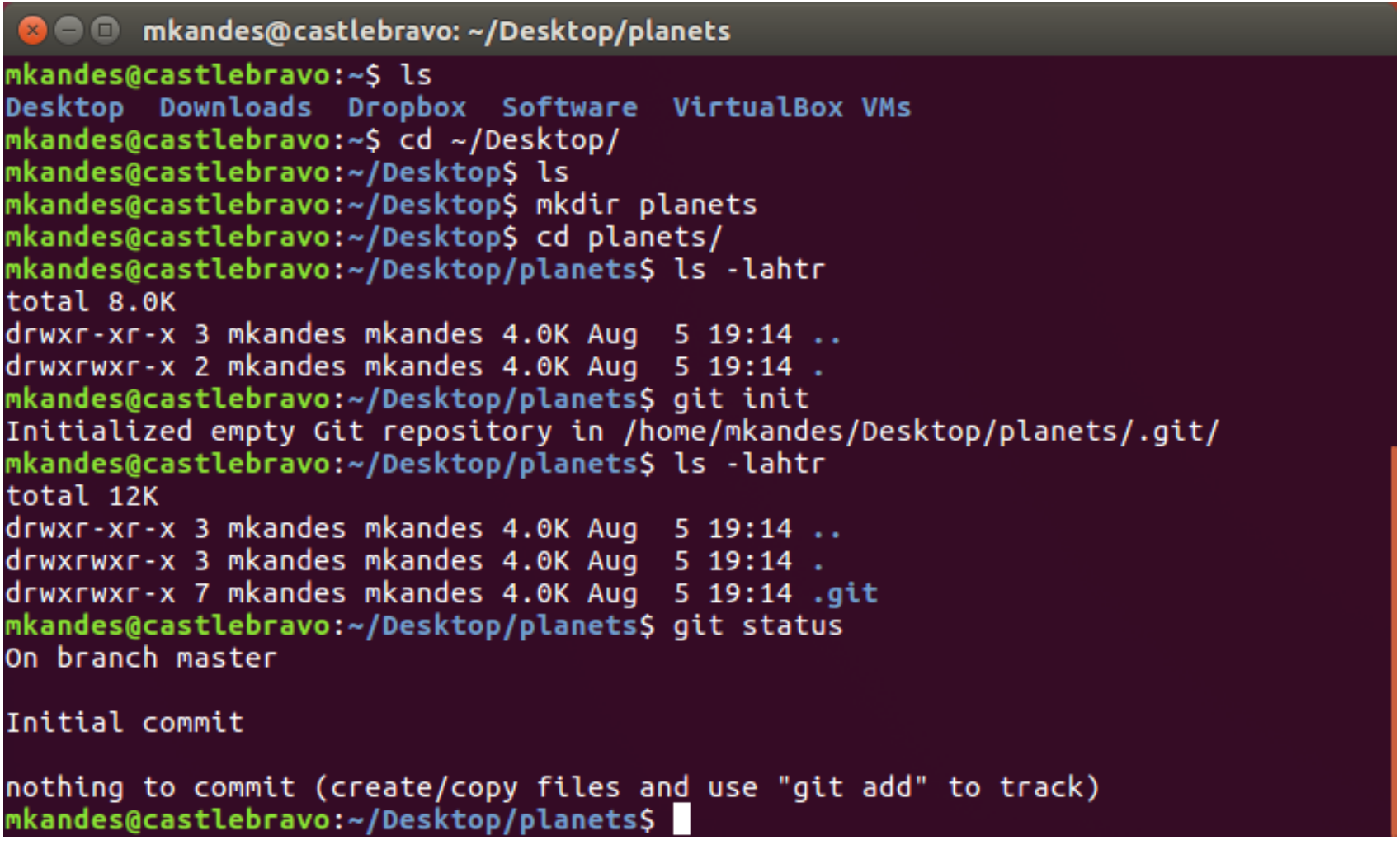
```
$ cd /path/to/your/project
```

and then typing the command:

```
$ git init
```

This creates a new subdirectory named `.git` in the project directory where the repository will live. If you ever want to stop tracking the project with `git`, you simply need to delete this subdirectory.

Creating your first `git` repository with `git init`

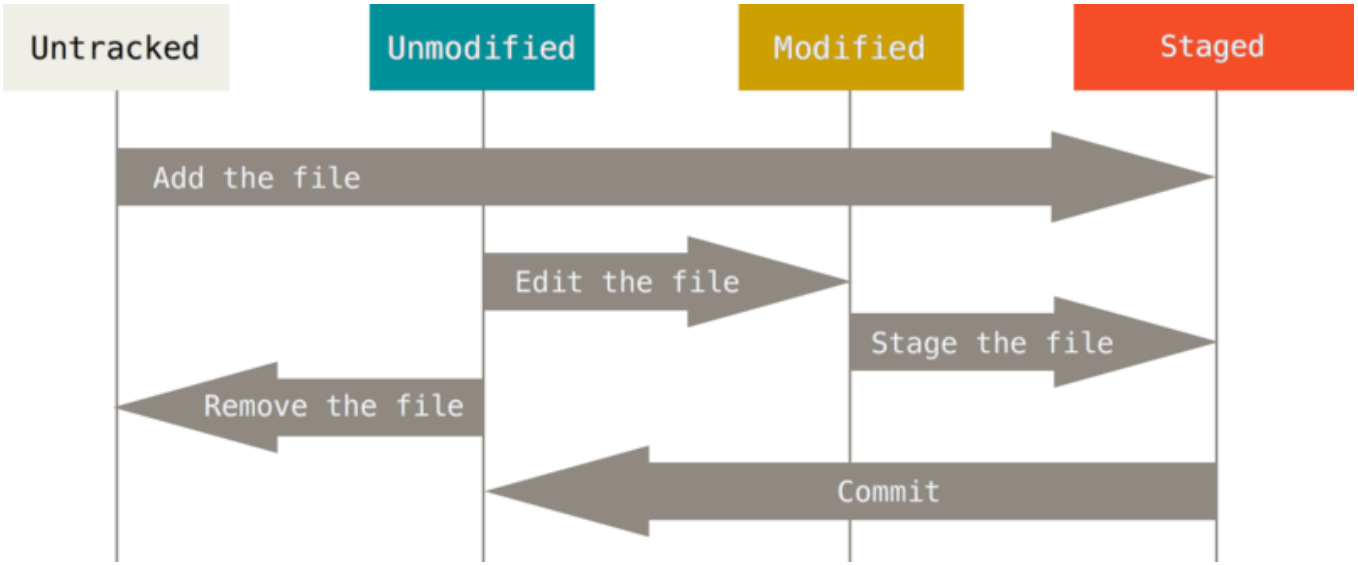


Checking the state your repository with `git status`

Each file in your repository's working directory may take on one of a few different states. The main tool you use to determine which files are in which state is with the command:

```
$ git status
```

Different states of a file in a `git` repository



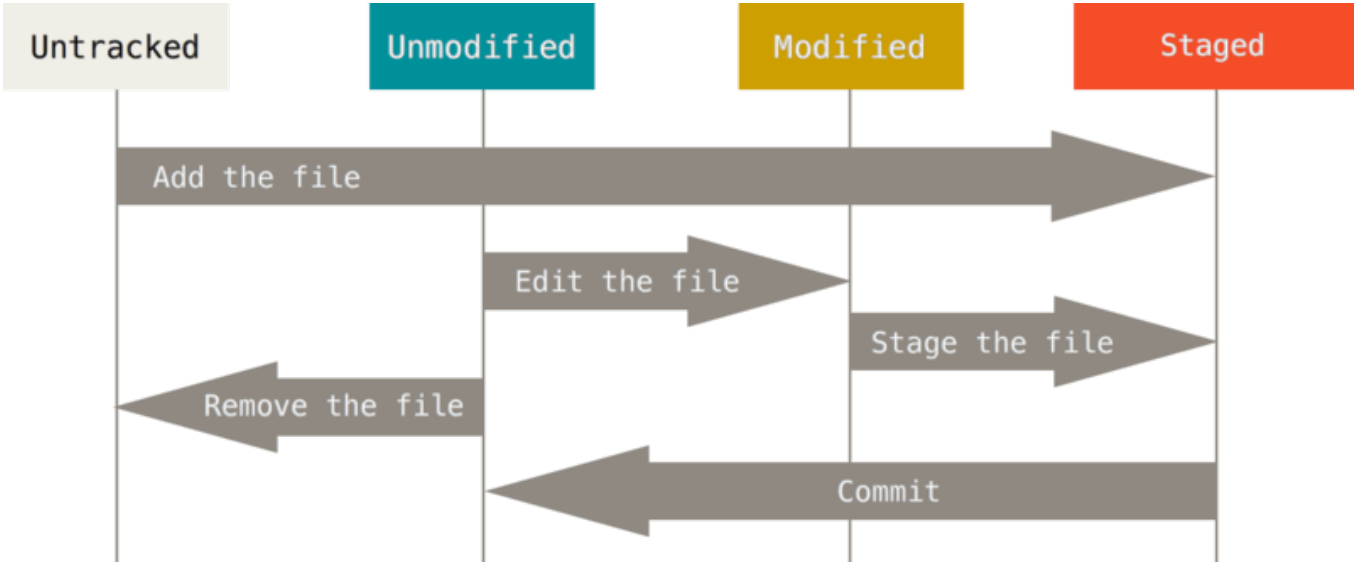
There are two primary file states you will find in your repository:

1. **Tracked files** are all of the files that `git` already knows about and has under version control; they have secondary states, where they can be either *unmodified*, *modified*, or *staged*.
2. **Untracked files** are everything else — any files in your project's working directory not yet under version control.

Adding files to your `git` repository with `git add`

```
mkandes@castlebravo: ~/Desktop/planets

Initial commit

nothing to commit (create/copy files and use "git add" to track)
mkandes@castlebravo:~/Desktop/planets$ echo 'Cold and dry, but everything else i
s my favorite color.' > mars.txt
mkandes@castlebravo:~/Desktop/planets$ ls -lahtr
total 16K
drwxr-xr-x 3 mkandes mkandes 4.0K Aug  5 19:14 ..
drwxrwxr-x 7 mkandes mkandes 4.0K Aug  5 20:07 .git
-rw-rw-r-- 1 mkandes mkandes   56 Aug  5 20:08 mars.txt
drwxrwxr-x 3 mkandes mkandes 4.0K Aug  5 20:08 .
mkandes@castlebravo:~/Desktop/planets$ git status
On branch master

Initial commit

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

        mars.txt

nothing added to commit but untracked files present (use "git add" to track)
mkandes@castlebravo:~/Desktop/planets$
```

Adding files to your `git` repository with `git add`

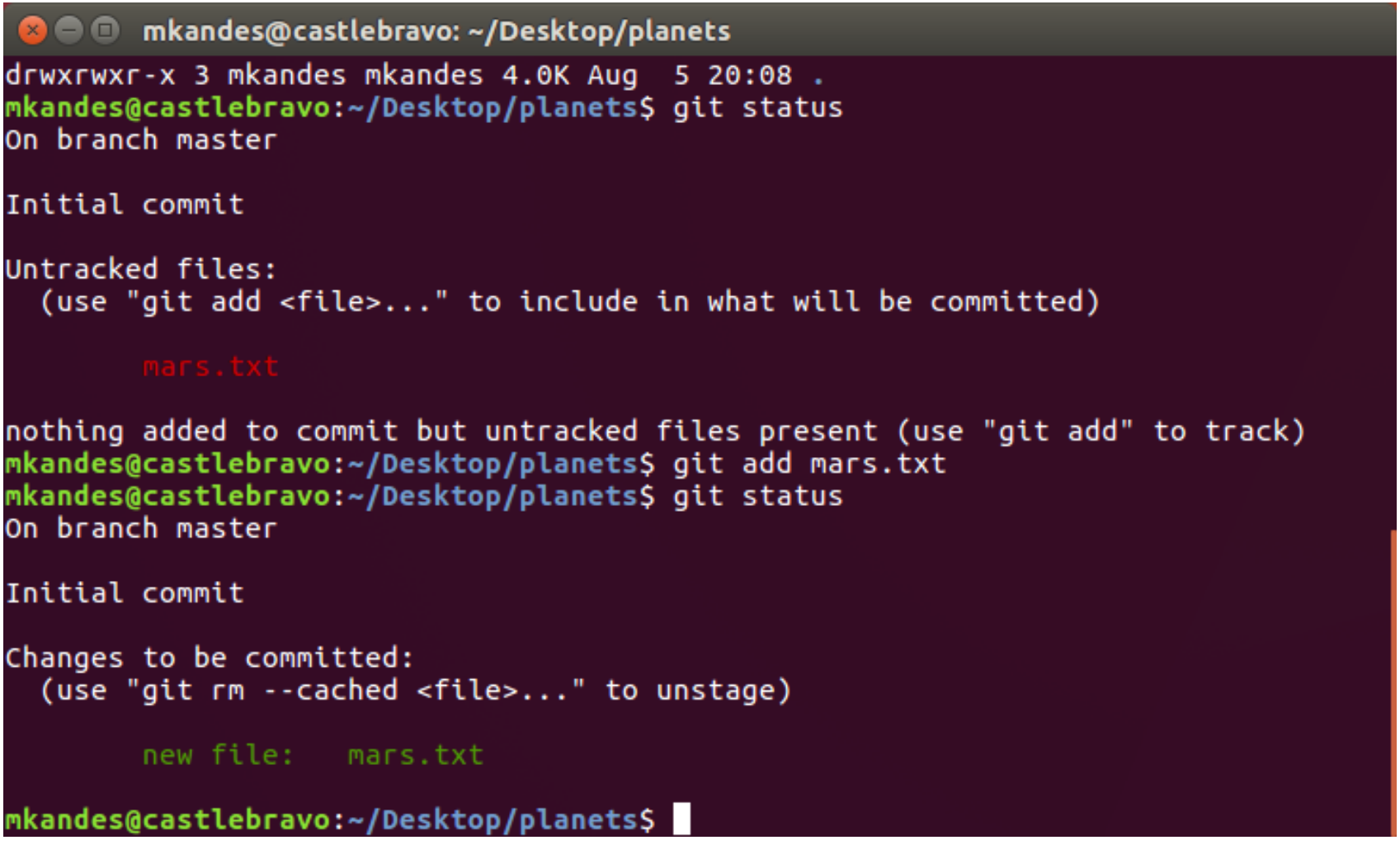


To begin tracking a new file in your project's working directory, you must first *stage* the new file to be committed to the repository with the command:

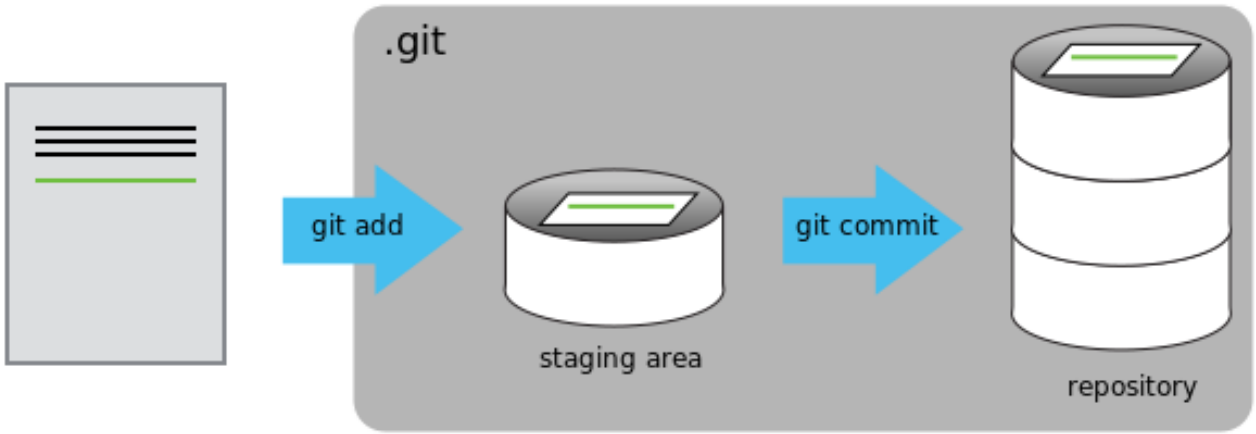
```
$ git add <path/to/file/filename>
```

If you run `git status` again, you should see that the newly added file is now tracked and staged to be committed.

Adding files to your `git` repository with `git add`



Committing changes with `git commit`



Changes to files in your repository are tracked through commits, which you make with the

```
$ git commit
```

command. Prior to a commit, you need to stage the files whose changes you want to commit using the `git add` command.

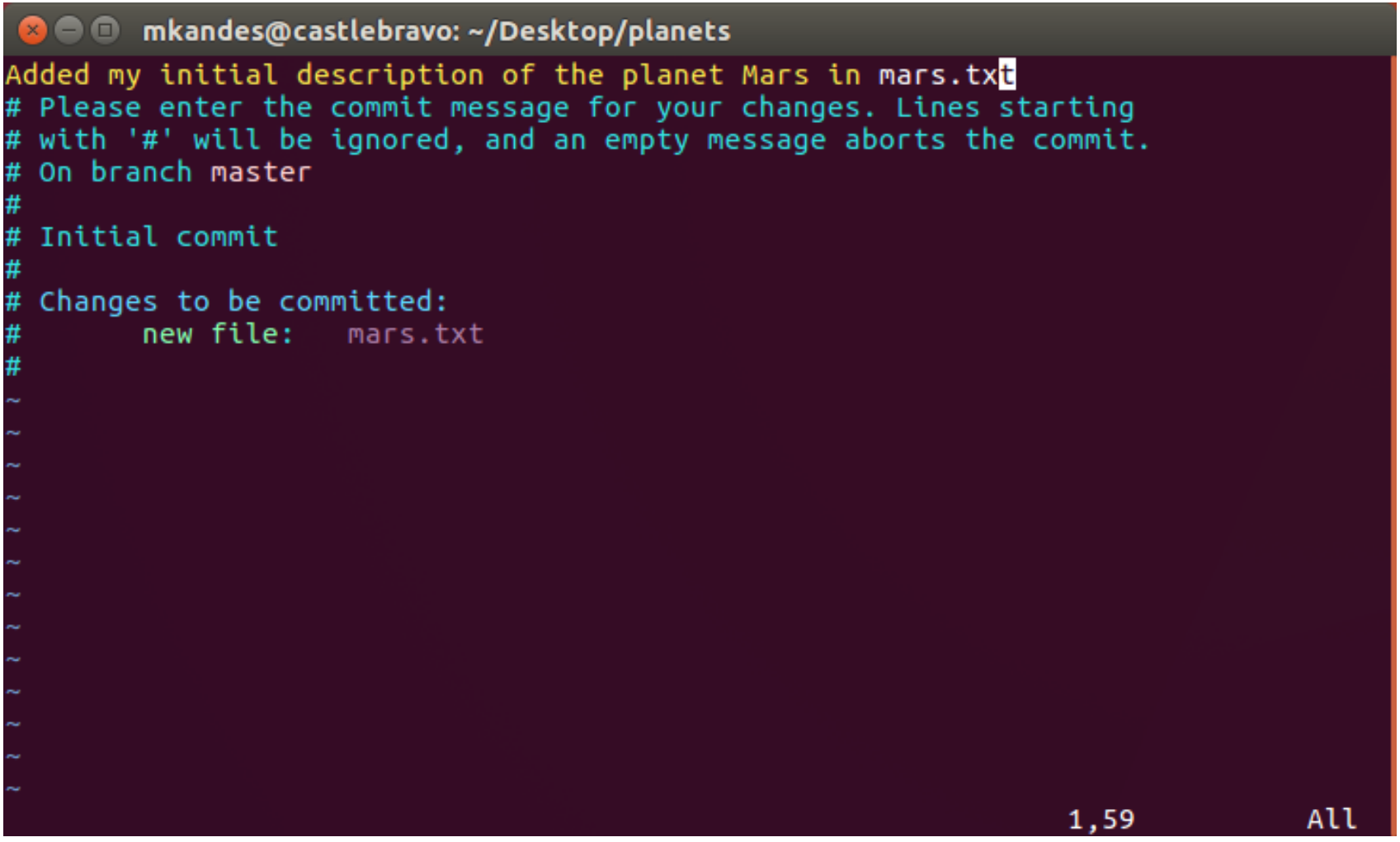
Writing a commit message

Each commit requires a commit message. They provide the *context* about *why* a change was made. This may be to inform other collaborators working on the project about the change or, more often, to remind yourself why you made the change.

How you choose to format your commit messages is up to you and your team. But if you need some guidance, here are seven rules to writing a great `git` commit message:

`https://chris.beams.io/posts/git-commit`

Writing a commit message



Committing changes with `git commit`

```
mkandes@castlebravo: ~/Desktop/planets
On branch master

Initial commit

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

        new file:   mars.txt

mkandes@castlebravo:~/Desktop/planets$ git commit
[master (root-commit) d9995d3] Added my initial description of the planet Mars i
n mars.txt
 1 file changed, 1 insertion(+)
 create mode 100644 mars.txt
mkandes@castlebravo:~/Desktop/planets$ git status
On branch master
nothing to commit, working directory clean
mkandes@castlebravo:~/Desktop/planets$ git log
commit d9995d3e77f7c68f7a80b13bfca997008524ca3e
Author: Marty Kandes <mkandes@sdsc.edu>
Date:   Sun Aug 5 20:49:22 2018 -0700

    Added my initial description of the planet Mars in mars.txt
mkandes@castlebravo:~/Desktop/planets$
```

Viewing your commit history with `git log`

The power of `git` is that it tracks the changes to files in your project over time. You can use the

```
$ git log
```

command to view that history.

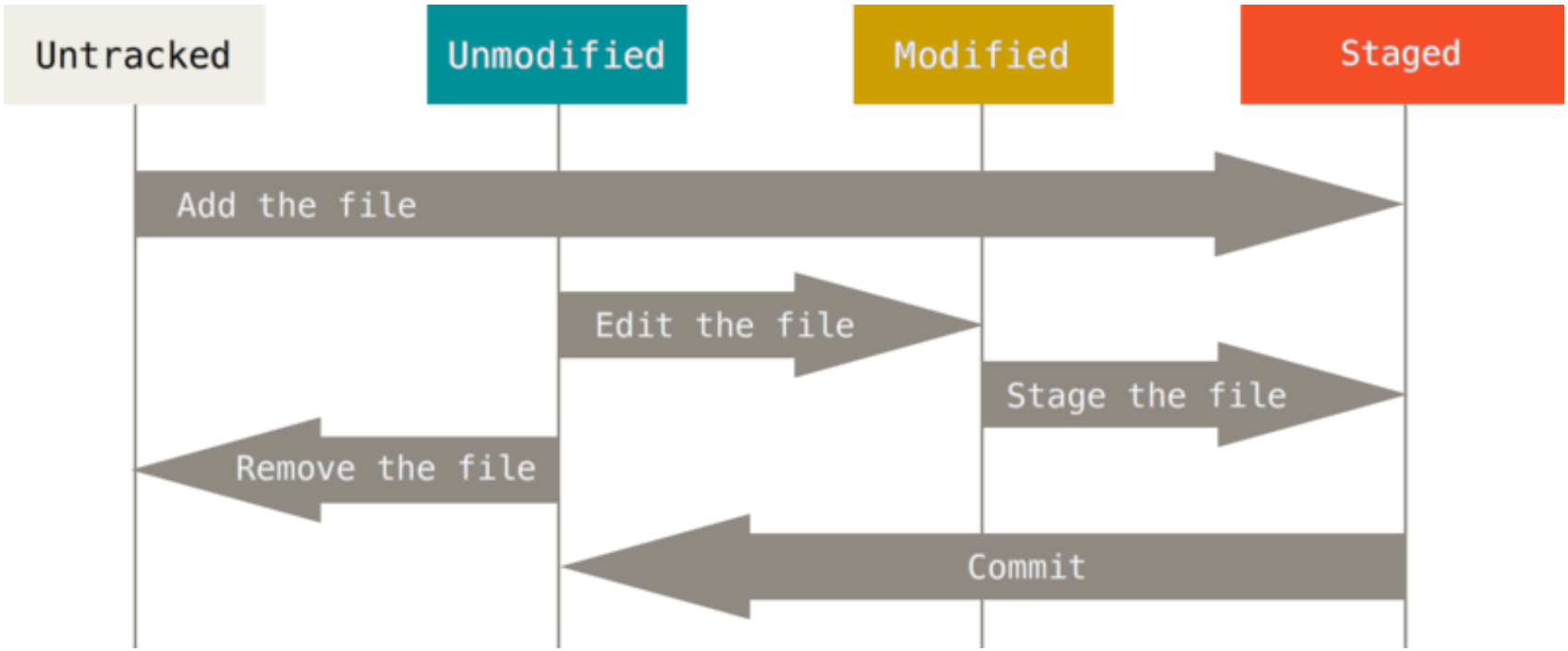
In the standard output of the `git log` command, you will see the commit ID of each commit, the author who made it, the date it was committed, and the commit message explaining the reason for the change(s).

By default, the `git log` command will list commit history in reverse chronological order.

Let's add some more files and make some changes ...

```
mkandes@castlebravo: ~/Desktop/planets
mkandes@castlebravo:~/Desktop/planets$ git status
On branch master
nothing to commit, working directory clean
mkandes@castlebravo:~/Desktop/planets$ echo 'A pale blue dot, but it is home.' >
 earth.txt
mkandes@castlebravo:~/Desktop/planets$ echo 'Not a planet.' > pluto.txt
mkandes@castlebravo:~/Desktop/planets$ echo 'Well, not the sunsets -- they are b
lue.' >> mars.txt
mkandes@castlebravo:~/Desktop/planets$ git status
On branch master
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:   mars.txt

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

        earth.txt
        pluto.txt

no changes added to commit (use "git add" and/or "git commit -a")
mkandes@castlebravo:~/Desktop/planets$
```

... then begin tracking the new files and stage the changes for the next commit ...



Staging new, untracked files and committing them ...

```
mkandes@castlebravo: ~/Desktop/planets
mkandes@castlebravo:~/Desktop/planets$ git add earth.txt pluto.txt
mkandes@castlebravo:~/Desktop/planets$ git commit -m "Added my initial descripti
ons of the planet Earth in earth.txt; also added my initial description of Pluto
, even though it is only a minor planet now"
[master a1c283b] Added my initial descriptions of the planet Earth in earth.txt;
 also added my initial description of Pluto, even though it is only a minor plan
et now
 2 files changed, 2 insertions(+)
 create mode 100644 earth.txt
 create mode 100644 pluto.txt
mkandes@castlebravo:~/Desktop/planets$ git status
On branch master
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:   mars.txt

no changes added to commit (use "git add" and/or "git commit -a")
mkandes@castlebravo:~/Desktop/planets$ git log
commit a1c283b0eaebf7460ede1576c9525315aee1ae5a
Author: Marty Kandes <mkandes@sdsc.edu>
Date:   Sun Aug 5 22:32:43 2018 -0700

    Added my initial descriptions of the planet Earth in earth.txt; also added m
y initial description of Pluto, even though it is only a minor planet now

commit d9995d3e77f7c68f7a80b13bfca997008524ca3e
Author: Marty Kandes <mkandes@sdsc.edu>
Date:   Sun Aug 5 20:49:22 2018 -0700

    Added my initial description of the planet Mars in mars.txt
```

Comparing differences with `git diff`

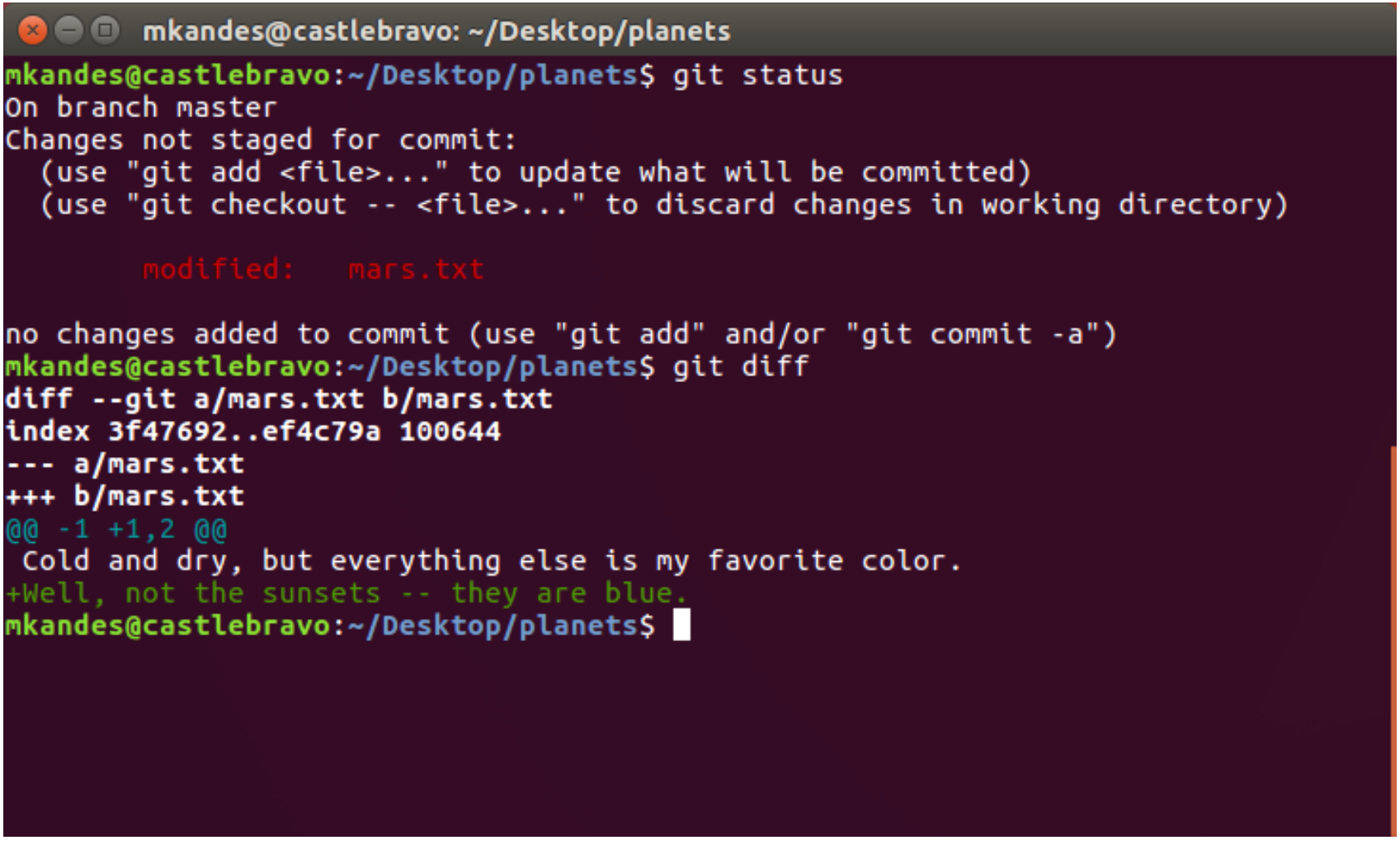
While `git log` provides you with information about a commit – who made it, when they made it, and why they made it — sometimes looking at the specific changes made to the files themselves is what's more important.

You can use the

```
$ git diff
```

command to see the changes to any *modified* files in your working directory before you stage them to be committed.

Comparing differences with `git diff`



Breaking down a `git diff` line-by-line

1. The first line tells us that `git` is comparing the old and new versions of the file with the `diff` command.

2. The second line tells us exactly which versions of the file are being compared.

3. The next two lines once again show the name of the file being changed.

4. The remaining lines show us the actual differences between the files and the lines on which they occur, with the + character in the first column showing where we added a new line of text.

After reviewing our change, it's time to commit it ...

```
mkandes@castlebravo: ~/Desktop/planets
diff --git a/mars.txt b/mars.txt
index 3f47692..ef4c79a 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,2 @@
 Cold and dry, but everything else is my favorite color.
+Well, not the sunsets -- they are blue.
mkandes@castlebravo:~/Desktop/planets$ git add mars.txt
mkandes@castlebravo:~/Desktop/planets$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   mars.txt

mkandes@castlebravo:~/Desktop/planets$ git commit -m 'I forgot there is at least
 one thing on Mars that is not my favorite color.'
[master 64b3df5] I forgot there is at least one thing on Mars that is not my fav
orite color.
 1 file changed, 1 insertion(+)
mkandes@castlebravo:~/Desktop/planets$ git status
On branch master
nothing to commit, working directory clean
mkandes@castlebravo:~/Desktop/planets$
```

Moving files and directories with `git mv`

Performing tasks such as reorganizing files under new directories, changing filenames, and so on requires that files or sometimes entire directories get moved from once place to another in your repository.

You can move a file (or directory) into another one with the command:

```
$ git mv <path/to/old/fname> <path/to/newfname>
```

`git` will stage the change for you after you call `git mv`. You must then call `git commit` afterwards to make the move permanent.

Moving files and directories with `git mv`

```
mkandes@castlebravo: ~/Desktop/planets
mkandes@castlebravo:~/Desktop/planets$ ls
earth.txt  mars.txt  pluto.txt
mkandes@castlebravo:~/Desktop/planets$ mkdir minor-planets
mkandes@castlebravo:~/Desktop/planets$ git mv pluto.txt minor-planets/pluto.txt
mkandes@castlebravo:~/Desktop/planets$ ls
earth.txt  mars.txt  minor-planets
mkandes@castlebravo:~/Desktop/planets$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    pluto.txt -> minor-planets/pluto.txt

mkandes@castlebravo:~/Desktop/planets$ git commit -m "Demoted pluto.txt to the n
ew minor-planets subdirectory."
[master 40ee458] Demoted pluto.txt to the new minor-planets subdirectory.
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename pluto.txt => minor-planets/pluto.txt (100%)
mkandes@castlebravo:~/Desktop/planets$ git status
On branch master
nothing to commit, working directory clean
mkandes@castlebravo:~/Desktop/planets$
```

Removing files and directories with `git rm`

Files and directories in your repository sometimes outlive their usefulness. When this occurs, you can tell `git` to stop tracking changes to these files and remove them from the working tree of your repository — your copy of the repository at a particular point in its commit history; normally at the last commit.

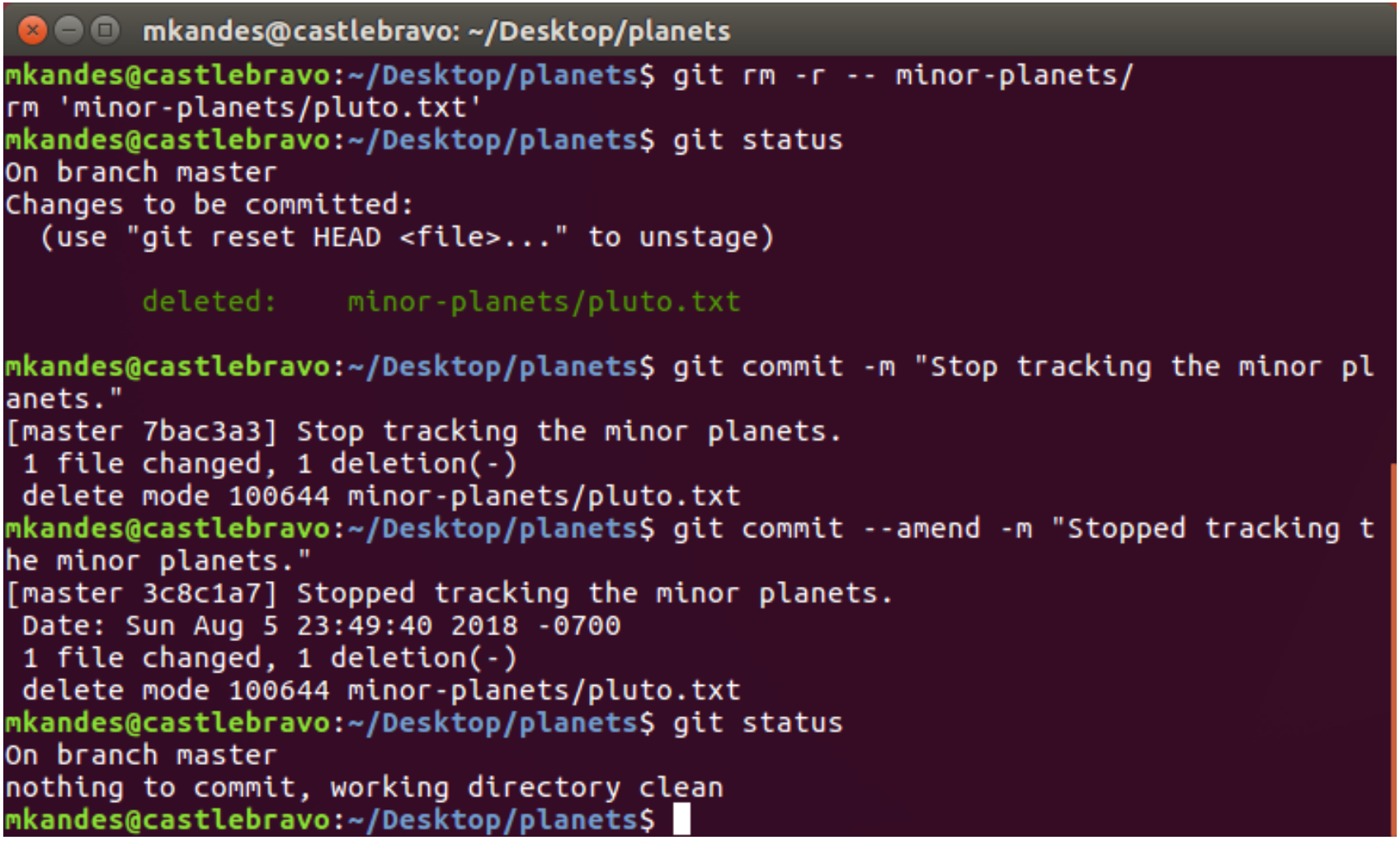
To an remove an outdated file (or a directory) from the working tree of your repository, you use the command:

```
$ git rm -- <path/to/file/filename>
```

This **does not** remove the file from your repository's history; it only removes it from your working tree going forward

Like other `git` actions, you must run `git commit` after `git rm` to finalize the change.

Removing files and directories with `git rm`



Creating and switching between `git` branches

When you want to begin modifying the contents of your repository, but only want to do so to a limited subset of the files in it, you can begin using the *branching* and *merging* mechanisms of `git`.

To create a new *branch* in your repository, you use the command:

```
$ git branch <branch-name>
```

To then begin working on this new branch, you must switch over to this branch of this project using the command:

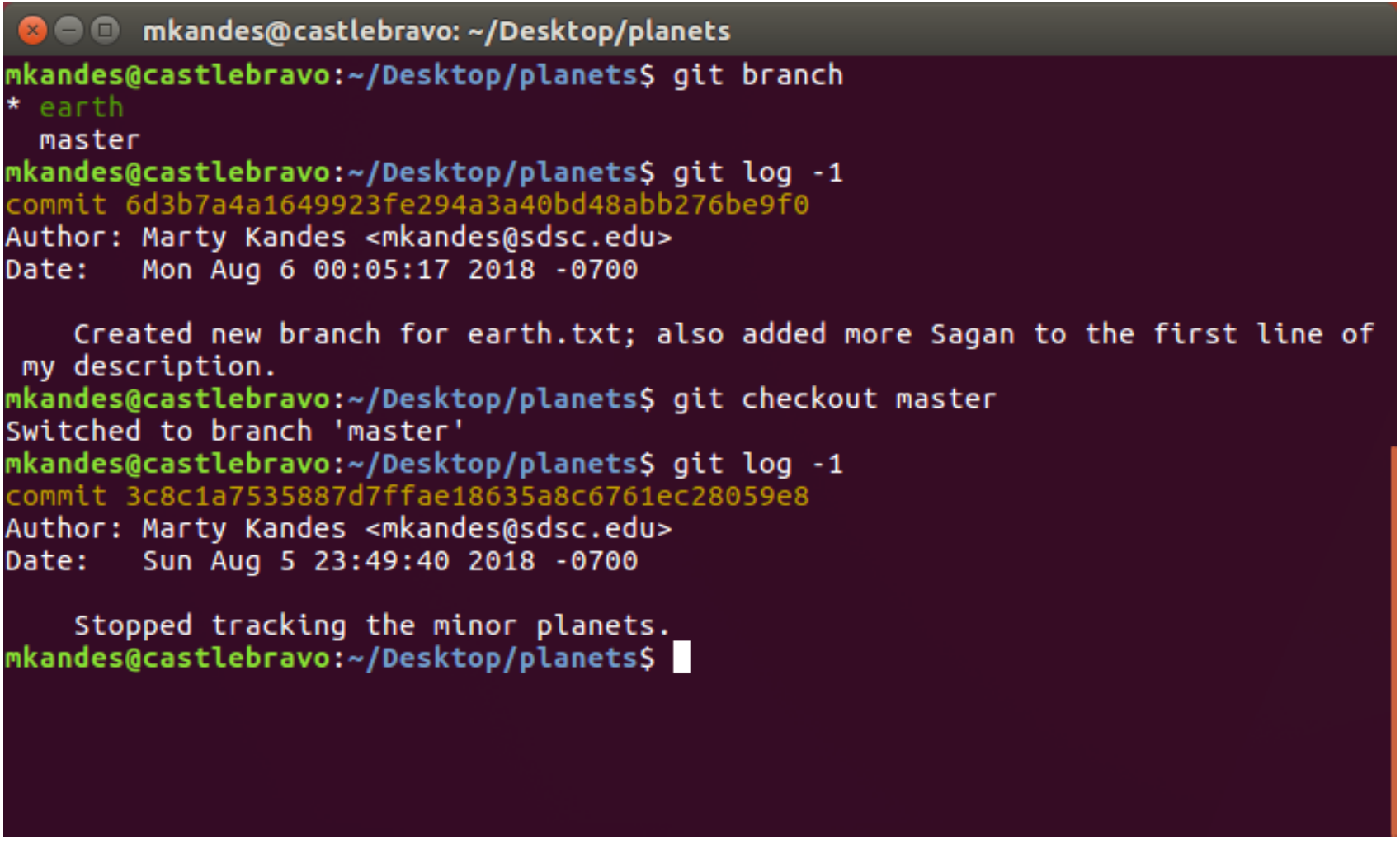
```
$ git checkout <branch-name>
```

The default branch in every `git` repository is the `master` branch.

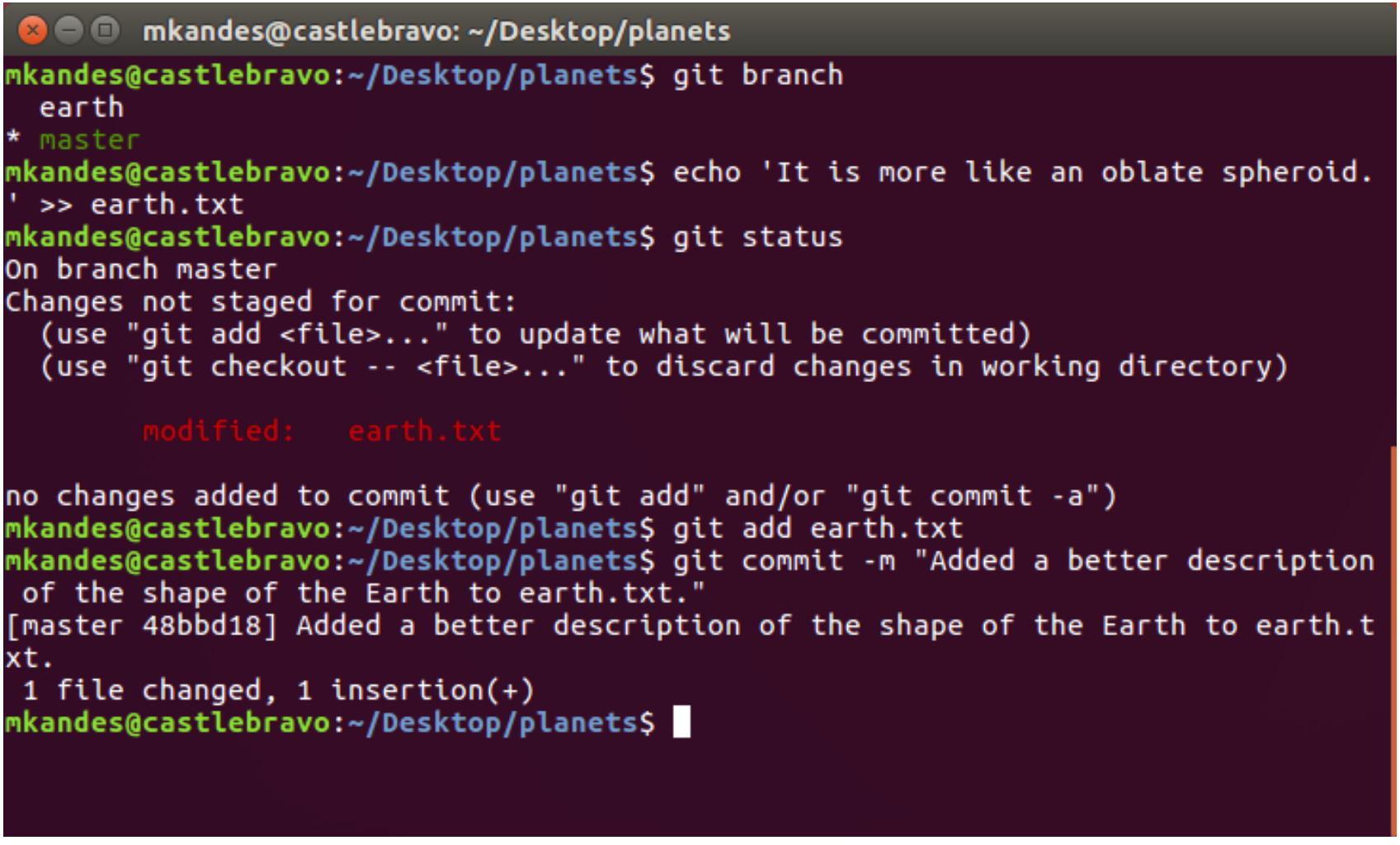
Creating and switching between `git branches`

```
mkandes@castlebravo: ~/Desktop/planets
mkandes@castlebravo:~/Desktop/planets$ git branch earth
mkandes@castlebravo:~/Desktop/planets$ git checkout earth
Switched to branch 'earth'
mkandes@castlebravo:~/Desktop/planets$ echo 'It is not flat.' >> earth.txt
mkandes@castlebravo:~/Desktop/planets$ vi earth.txt
mkandes@castlebravo:~/Desktop/planets$ git status
On branch earth
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:   earth.txt

no changes added to commit (use "git add" and/or "git commit -a")
mkandes@castlebravo:~/Desktop/planets$ git add earth.txt
mkandes@castlebravo:~/Desktop/planets$ git commit -m "Created new branch for ear
th.txt; also added more Sagan to the first line of my description."
[earth 6d3b7a4] Created new branch for earth.txt; also added more Sagan to the f
irst line of my description.
 1 file changed, 2 insertions(+), 1 deletion(-)
mkandes@castlebravo:~/Desktop/planets$ git log
commit 6d3b7a4a1649923fe294a3a40bd48abb276be9f0
Author: Marty Kandes <mkandes@sdsc.edu>
Date:   Mon Aug 6 00:05:17 2018 -0700
```

Creating and switching between `git branches`



Creating and switching between `git branches`



Merging commits between branches with `git merge`

Once you and/or your collaborators have made changes on a project in different branches, at some point you'll want to *merge* your changes together.

To *merge* the changes from different branches, you first need to *checkout* the branch you want the changes to be merged into.

```
$ git checkout <branch-name>
```

You then select the changes from another branch to be merged into this current branch.

```
$ git merge <branch-name-to-be-merged>
```

Note, however, if there is a *conflict* between the content the two branches, you must first resolve these conflicts prior to committing the merge.

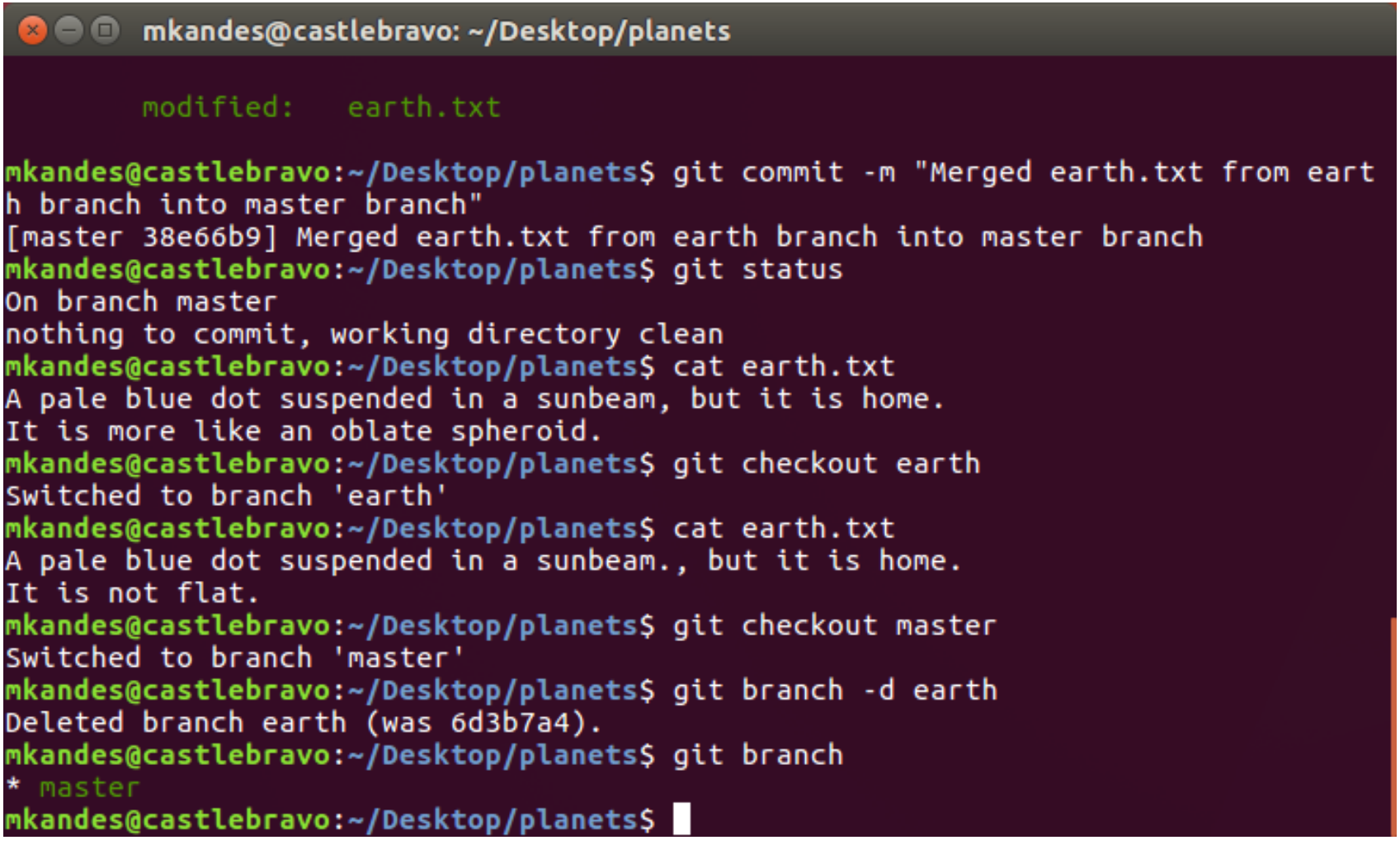
Merging commits between branches with `git merge`

```
mkandes@castlebravo: ~/Desktop/planets
mkandes@castlebravo:~/Desktop/planets$ git branch
  earth
* master
mkandes@castlebravo:~/Desktop/planets$ git merge earth
Auto-merging earth.txt
CONFLICT (content): Merge conflict in earth.txt
Automatic merge failed; fix conflicts and then commit the result.
mkandes@castlebravo:~/Desktop/planets$ cat earth.txt
<<<<<<< HEAD
A pale blue dot, but it is home.
It is more like an oblate spheroid.
=======
A pale blue dot suspended in a sunbeam., but it is home.
It is not flat.
>>>>>>> earth
mkandes@castlebravo:~/Desktop/planets$ git checkout earth
earth.txt: needs merge
error: you need to resolve your current index first
mkandes@castlebravo:~/Desktop/planets$ vi earth.txt
mkandes@castlebravo:~/Desktop/planets$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
```

Merging commits between branches with `git merge`

```
mkandes@castlebravo: ~/Desktop/planets
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   earth.txt

no changes added to commit (use "git add" and/or "git commit -a")
mkandes@castlebravo:~/Desktop/planets$ git add earth.txt
mkandes@castlebravo:~/Desktop/planets$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

        modified:   earth.txt

mkandes@castlebravo:~/Desktop/planets$ git commit -m "Merged earth.txt from eart
h branch into master branch"
[master 38e66b9] Merged earth.txt from earth branch into master branch
mkandes@castlebravo:~/Desktop/planets$
```

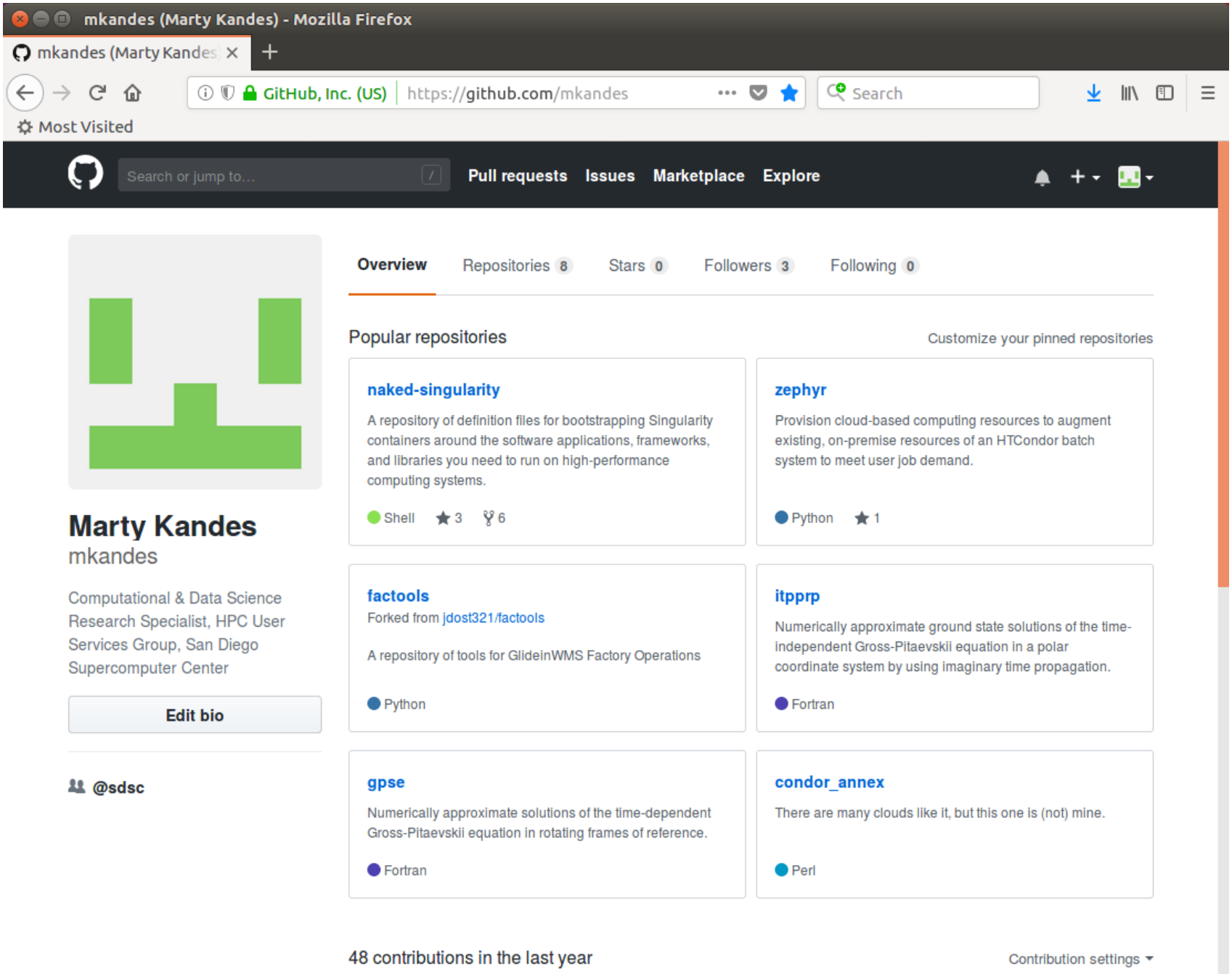
Merging commits between branches with `git merge`



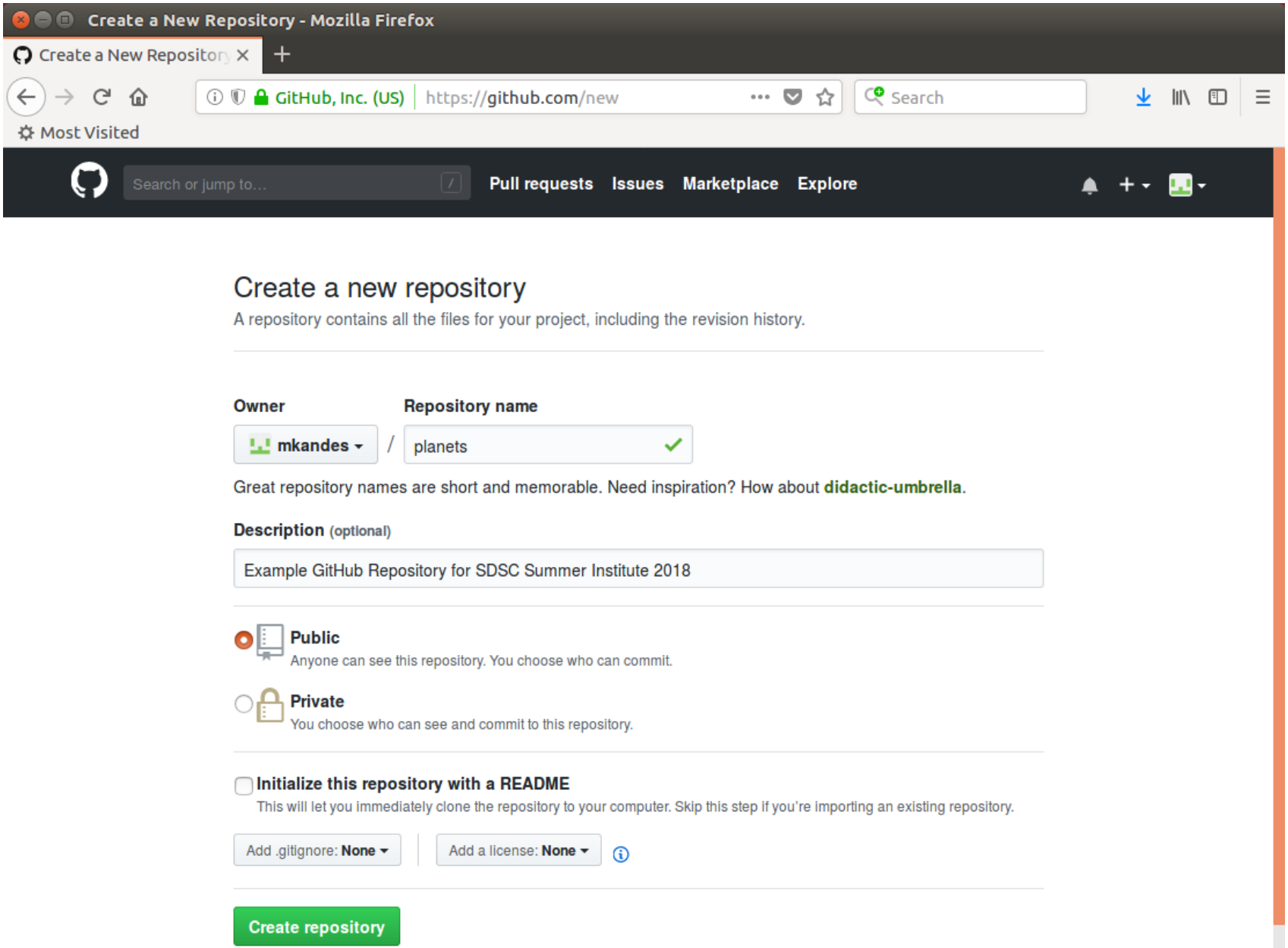
Getting Started with GitHub

https://github.com

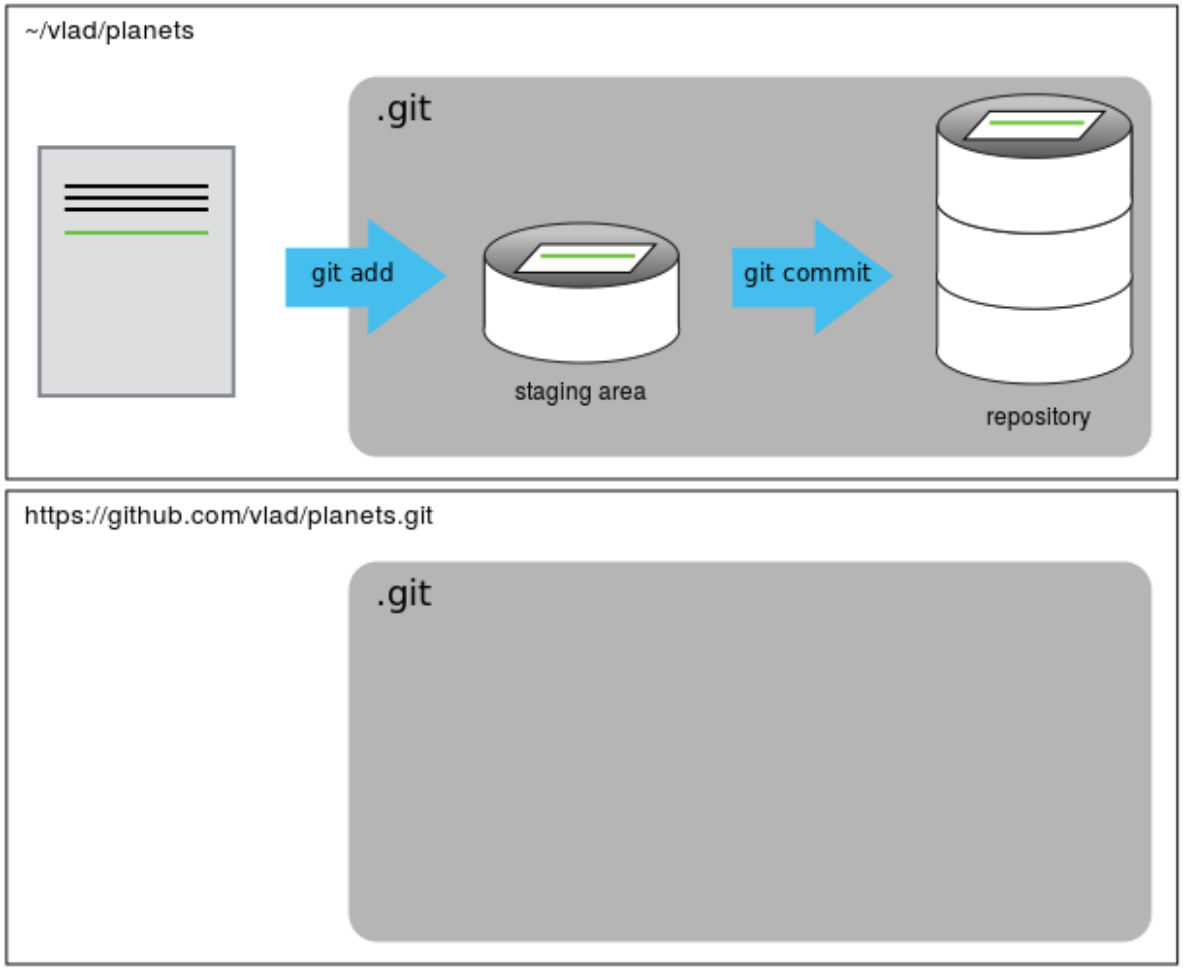
Your GitHub Profile



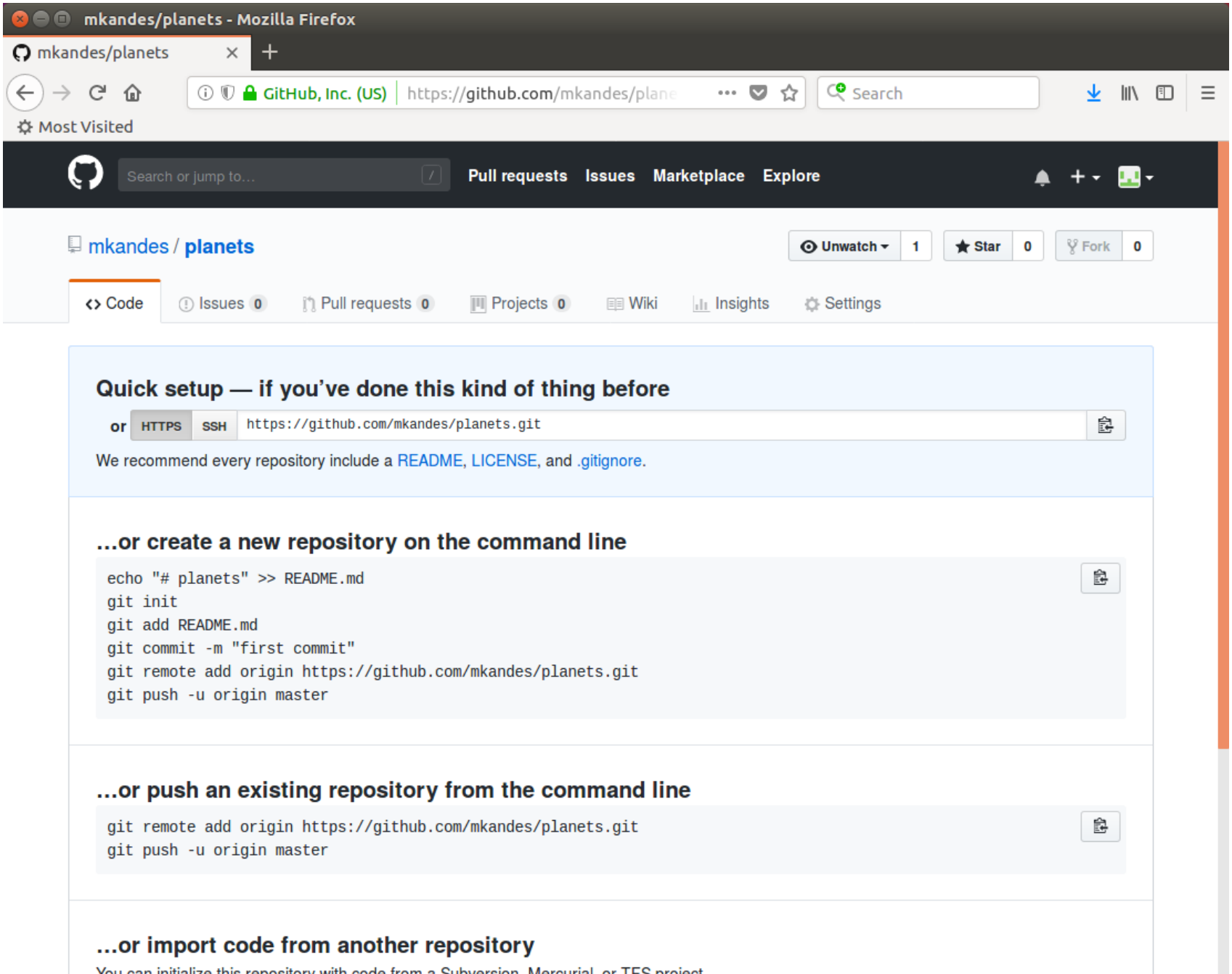
Creating a New GitHub Repository



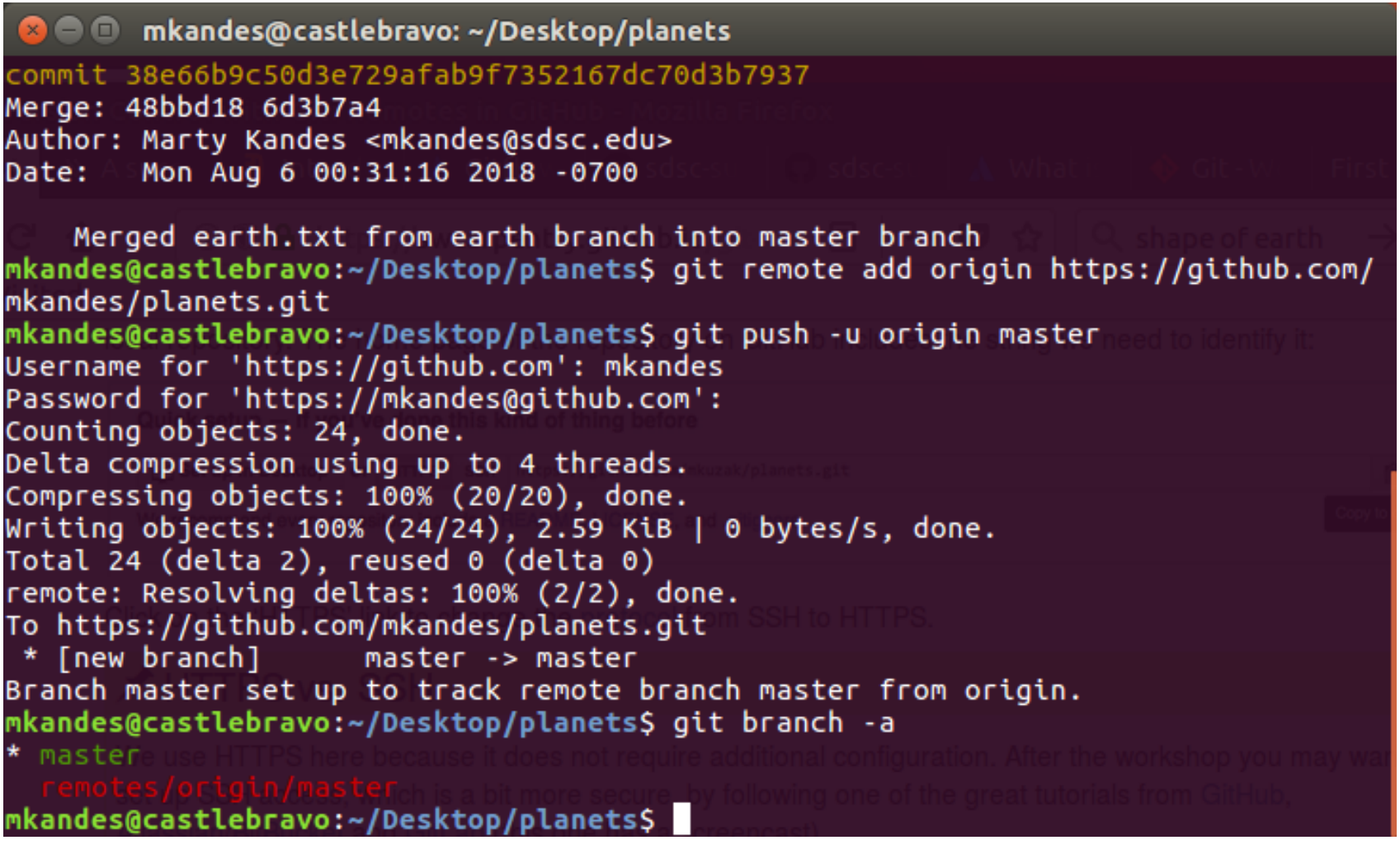
Creating a New GitHub Repository



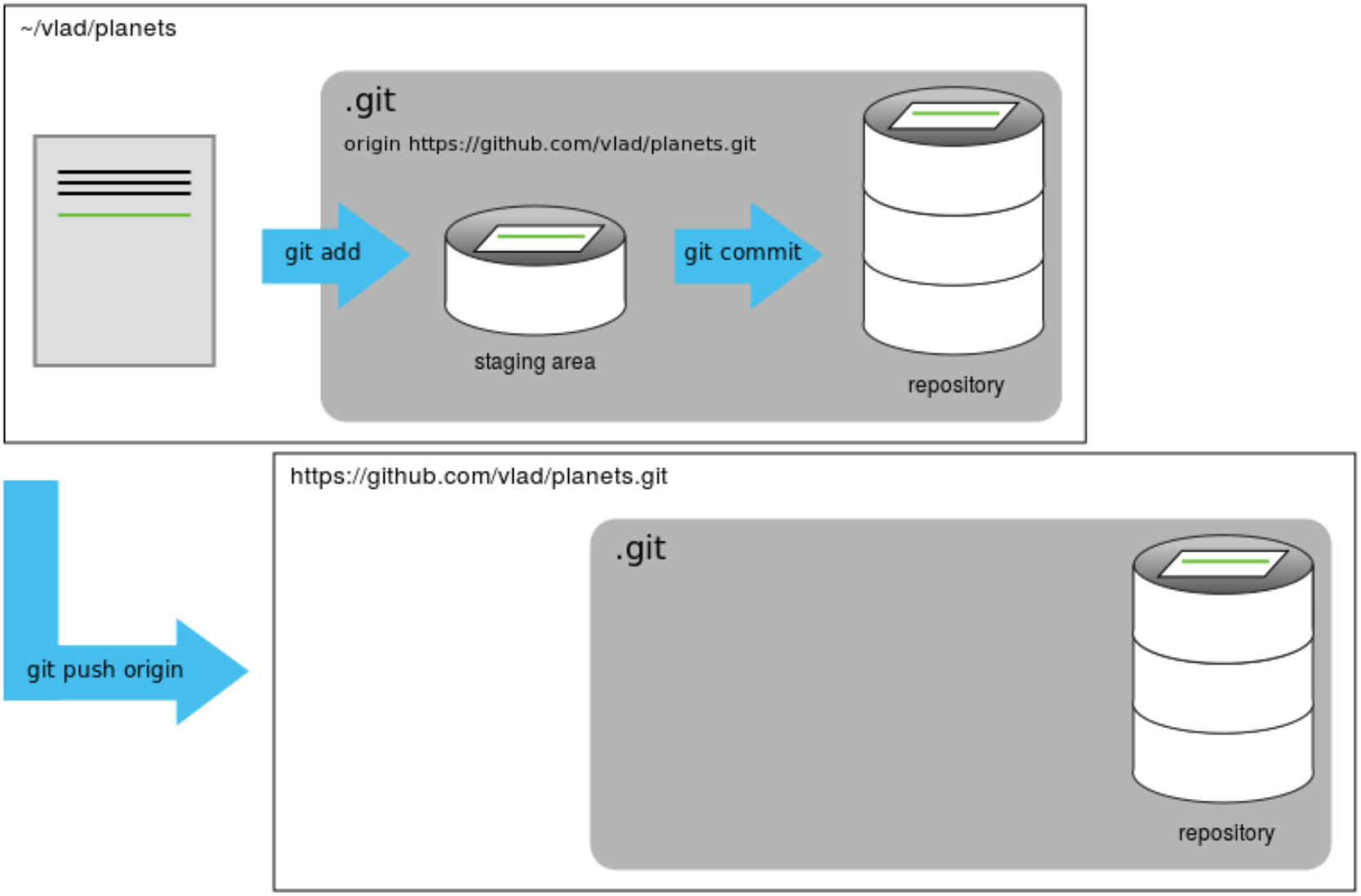
How to make your GitHub repository a `remote`



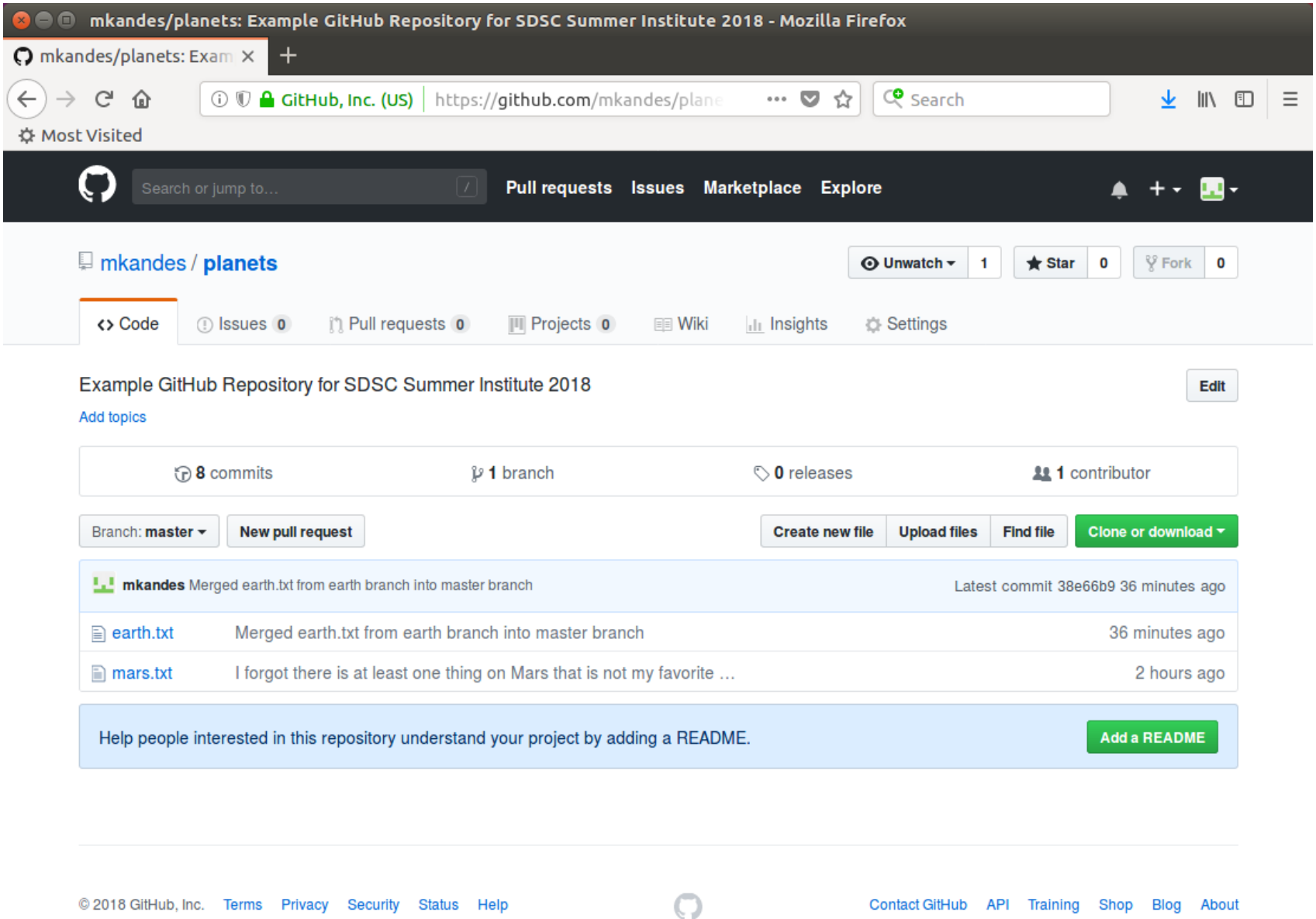
How to make your GitHub repository a `remote`



Your GitHub repository after the first `git push`



Your GitHub repository after the first `git push`



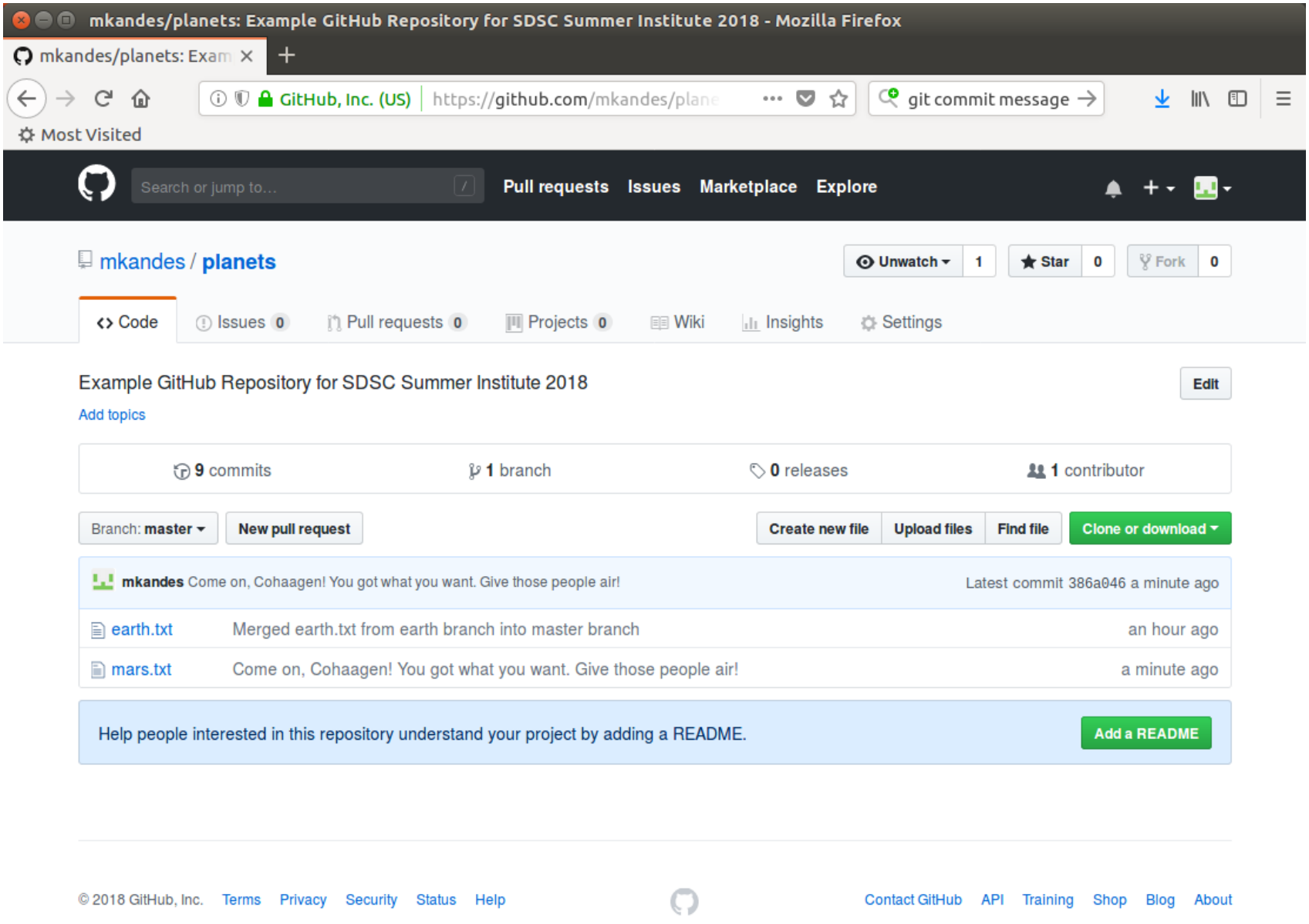
Cloning a GitHub repository with `git clone`

```
mkandes@comet-ln3:~/planets
********************************************************************************
[mkandes@comet-ln3 ~]$ git clone https://github.com/mkandes/planets.git
Cloning into 'planets'...
fatal: unable to access 'https://github.com/mkandes/planets.git/': SSL connect e
rror
[mkandes@comet-ln3 ~]$ git config --global http.sslVersion "tlsv1"
[mkandes@comet-ln3 ~]$ git clone https://github.com/mkandes/planets.git
Cloning into 'planets'...
remote: Counting objects: 24, done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 24 (delta 2), reused 24 (delta 2), pack-reused 0
Unpacking objects: 100% (24/24), done.
[mkandes@comet-ln3 ~]$ cd planets/
[mkandes@comet-ln3 planets]$ git config --global user.name "Marty Kandes"
[mkandes@comet-ln3 planets]$ git config --global user.email "mkandes@sdsc.edu"
[mkandes@comet-ln3 planets]$ git log -1
commit 38e66b9c50d3e729afab9f7352167dc70d3b7937 (HEAD -> master, origin/master,
origin/HEAD)
Merge: 48bbd18 6d3b7a4
Author: Marty Kandes <mkandes@sdsc.edu>
Date:   Mon Aug 6 00:31:16 2018 -0700

    Merged earth.txt from earth branch into master branch
[mkandes@comet-ln3 planets]$
```

Modify cloned repository and push changes back to GitHub

```
mkandes@comet-ln3:~/planets
[mkandes@comet-ln3 planets]$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
[mkandes@comet-ln3 planets]$ echo "It's where Cohaagen's big secret is buried."
>> mars.txt
[mkandes@comet-ln3 planets]$ git diff
diff --git a/mars.txt b/mars.txt
index ef4c79a..e1223fe 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything else is my favorite color.
 Well, not the sunsets -- they are blue.
+It's where Cohaagen's big secret is buried.
[mkandes@comet-ln3 planets]$ git add mars.txt
[mkandes@comet-ln3 planets]$ git commit -m 'Come on, Cohaagen! You got what you
want. Give those people air!'
[master 386a046] Come on, Cohaagen! You got what you want. Give those people air
!
 1 file changed, 1 insertion(+)
[mkandes@comet-ln3 planets]$ git push
Username for 'https://github.com': mkandes
Password for 'https://mkandes@github.com':
Counting objects: 3, done.
Delta compression using up to 24 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 385 bytes | 385.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/mkandes/planets.git
   38e66b9..386a046  master -> master
[mkandes@comet-ln3 planets]$
```

GitHub repository after recent push



Summary and Conclusion: Benefits of Version Control

- **Archiving**: You *must* regularly save the changes you make.
- **Reproducibility**: Creating a history of saved changes allows you to revert selected files or your entire project back to any previous state in its recorded history.
- **Collaboration**: You and a team of contributors can work on a project independently, but share your changes amongst one another as the project takes shape.
- **Accountability**: Compare changes, see who last modified something, and who introduced a problem and when.
- **Recoverability**: Each contributor has their own local, recent copy of the project and its complete history, making it highly unlikely you'll ever lose a significant portion of the project.

"*Version control software is very useful, but it is not essential.*"

– `Elephanthunter`

The End

References

- **Version Control with Git** by D.Huang and I. Gonzalez
  `https://swcarpentry.github.io/git-novice/`
- **Pragmatic Version Control Using Git** by T. Swicegood
- **Pro Git** by S. Chacon and B. Straub
  `https://git-scm.com/book/en/v2`