

# GitHub Guide

Daniel Scarbrough  
Email: dscarbro@mines.edu

May 31, 2019

## Contents

<b>1</b>	<b>Overview</b>	<b>2</b>
<b>2</b>	<b>Installation</b>	<b>2</b>
2.1	Windows, Mac, and Linux . . . . .	2
2.2	Linux Subsystem for Windows - (Suggestion) . . . . .	3
<b>3</b>	<b>Getting Started with Git</b>	<b>4</b>
3.1	Git in the Terminal . . . . .	4
3.2	Git Config . . . . .	5
3.3	Local Git Basics . . . . .	6
3.3.1	Starting a Local Repository . . . . .	6
3.3.2	Making Commits . . . . .	7
3.3.3	Defining Files to Ignore . . . . .	9
3.3.4	Removing Files . . . . .	12
3.3.5	Viewing the Log and Reverting to a Previous Commit . . . . .	14
3.4	Moving Online with GitHub . . . . .	17
3.4.1	GitHub Account . . . . .	17
3.4.2	Putting an Existing Local Repository on GitHub . . . . .	17
3.4.3	Pushing Commits to Your Remote Repository . . . . .	23
3.4.4	Cloning a Remote GitHub Repository . . . . .	24
3.4.5	Pulling Updates from a Remote Repository . . . . .	27
<b>4</b>	<b>Branching</b>	<b>27</b>
4.1	Merging . . . . .	27
4.1.1	Merge Conflicts . . . . .	27

## Future Additions

- Branching
- Git diff
- Git blame
- Collaboration
- Merge conflicts

## 1 Overview

*GitHub* and *Git* are extremely useful tools when programming in any language from Python to Matlab. *Git* and *GitHub* are distinct because *Git* is the language we use to perform all of our version control operations from the terminal, and *GitHub* is where we store our repositories remotely for access anywhere and easy collaboration. This document is meant to introduce the basic usage of GitHub and Git.

## 2 Installation

### 2.1 Windows, Mac, and Linux

This website has everything you need to install Git on the three main operating systems

<https://www.linode.com/docs/development/version-control/how-to-install-git-on-linux-mac-and-windows/>

Most Linux distributions are ready to go with Git, but if not it usually takes one line to get it installed. I don't have a Mac so I haven't been able to test those instructions myself, but it shouldn't be too difficult. For the Windows

instructions just follow through to step 14 - we won't be using the Git GUI in this guide.

## 2.2 Linux Subsystem for Windows - (Suggestion)

Windows 10 now offers a way to install a Linux Subsystem, which lets you use a Linux bash shell on your system without installing a virtual machine or dual booting. The best part of this is that you can access all of your Windows files through this shell. For example you could use your favorite code editor to write a Python file then open up the Linux shell, navigate to the script and run it. If the script generates an image or anything you can then access it through your regular Windows programs. This is what I do for all of my projects and it is extremely convenient. Installing packages is so much easier through the subsystem than it is trying to install them through Windows.

If you're interested, this guide is good to get setup:

<https://www.howtogeek.com/249966/how-to-install-and-use-the-linux-bash-shell-on-windows-10/>

Once it's setup, you'll probably want to get to your Windows directories from the subsystem. When the shell starts you'll be in the root directory of the subsystem shown by `~`. To get to the Windows directories you can access any drive on your computer by changing directories as such:

```
cd /mnt/c
```

Which will put you in your C drive. Change the letter if you need another drive. At anytime to get back to the Linux home directory, just enter `cd`

If you're using graphics while in the subsystem (e.g. using matplotlib in Python) you'll need to install one more thing. Follow this guide:

<https://www.howtogeek.com/261575/how-to-run-graphical-linux-desktop-applications-from-windows-10s-bash-shell/>

If you do this, you might find it annoying to enter `export DISPLAY=:0` every

time you open a new terminal and need to run something with graphics. You can have the bash shell do this automatically when it opens. To do so, open up your bash terminal and ensure you're in the root directory denoted by `~`. If not just enter `cd` to get there. Now you'll need to open up the configuration file. Enter the following command:

```
vi .bashrc
```

This will open up the `vi` text editor in your terminal. ***DON'T DELETE ANYTHING***. All you're going to do is add a line. To begin typing in `vi`, you need to go into insert mode by just hitting the `I` key. Now you can use the arrow keys to navigate and enter to make a new line. On an empty line, just type

```
export DISPLAY=:0
```

Now you just have to save and exit `vi`. Press `escape` on your keyboard to get out of insert mode. Now just type

```
:wq
```

This should just show up at the bottom of the window. If it gets typed into the document you are still in edit mode and need to hit `escape`. `w` indicates that the file should be saved (write), and `q` indicates that you are quitting `vi`. Now every time you open the bash shell this display variable will already be set and you'll just have to be sure Xming is running.

## 3 Getting Started with Git

### 3.1 Git in the Terminal

It's best to be comfortable using Git from a terminal. While there are visual applications to use Git, being able to quickly open up a terminal on any machine to get to your repositories hosted on GitHub is a major feature of Git. To get started, open a terminal (PowerShell on Windows, or the Linux subsystem bash shell if you went for the subsystem). Check that your

installation worked by entering

```
git --version
```

If there's an error, then something in the installation went wrong and we'll have to troubleshoot.

## 3.2 Git Config

Before you get started using Git, you need to do some quick things to apply your identity to your work. This way the Git logs will show your name and a contact email in case you are working collaboratively. You can set this information globally on your machine so you only have to do it once. The following two commands let you set the name and email for Git:

```
git config --global user.name "Joseph Fourier"
git config --global user.email joe_fourier@example.com
```

If for some reason you want to change your name/email within specific projects, as long as the terminal is within that project you can enter these commands without the `--global` option. If at any time you'd like to see what config options git has in your current location you can enter

```
git config --list
```

If you want to do any comment editing for Git within the terminal you may want to change the editor config option. For example if I wanted to use VIM for any editing I would enter

```
git config --global core.editor vim
```

Note that you can use Git without ever needing to edit in the terminal, but there are occasions where it may be convenient to do so. You won't need to for this guide.

### 3.3 Local Git Basics

The core of Git is managing your projects, and each project is stored as a repository. A repository should always be started in the top-most directory of your project. Beyond that it can have as many subfolders as needed. You should also avoid doing things like having a repository within a repository - good organization is key.

#### 3.3.1 Starting a Local Repository

To follow along I recommend making a folder to store all of these practice repositories. I made a folder called `testEnvironments` to get started. When you're working with a terminal, you'll want to avoid having spaces in your file and folder names. Within this folder I made another one called `localRepo` where we'll get started.

Navigate your terminal to the `localRepo` folder. Add some files of different types to this folder. For my example, I made two Python files and a text file. You can make whatever you want, and can even leave the files empty for now. Use the `ls` command to verify that your files are in the right place.

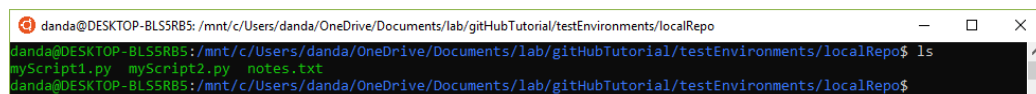
A screenshot of a terminal window. The title bar shows the path: danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo. The terminal content shows the user running the 'ls' command, which lists the files: myScript1.py, myScript2.py, and notes.txt. The prompt is danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo\$.

Figure 1

Now I want to manage this project with Git, so I'm going to initialize Git with the following command:

```
git init
```

A good command to use frequently to verify what's going on is `git status`. Go ahead and run that and you should see that you are on what's called the *master* branch. This is the top level of your repository. For now we'll make changes here but later in Section 4 we'll cover good practices with branches.

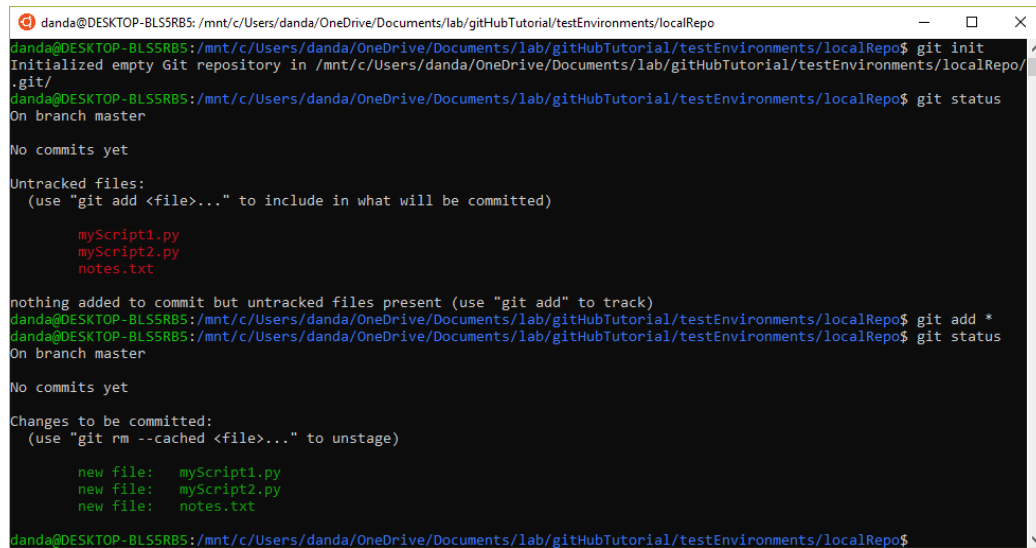
### 3.3.2 Making Commits

#### Tracked and Untracked Files

You'll also notice from the `status` command that there are no commits yet and there are *untracked* files. Until you tell git that you want to track files as part of your repository it won't do anything with them. In a lot of cases you're going to want to track everything you have (but not always - more on that later). If this is the case, you can just tell Git to track everything.

```
git add *
```

Now run `git status` again and you'll see your files are now green and ready to be committed.



```
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git init
Initialized empty Git repository in /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo/.git/
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git status
On branch master

No commits yet

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

    myScript1.py
    myScript2.py
    notes.txt

nothing added to commit but untracked files present (use "git add" to track)
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git add *
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git status
On branch master

No commits yet

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

    new file:   myScript1.py
    new file:   myScript2.py
    new file:   notes.txt

danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$
```

Figure 2

It's ready for commit. When committing it is really important that you provide a concise but descriptive comment about what you've done. This is helpful if you need to look at old versions and for others to see what has been updated or if something still needs some work. We can perform the commit and comment in one step with a command option. Doing so looks like

```
git commit -m "comment on commit"
```

The `-m` option tells the commit command that we want to add a message and the next part will be that message. It is necessary to put that message in quotations or else you'll get an error. Since we have only just started this repository, we'll just commit with a basic `"initial commit"` message.

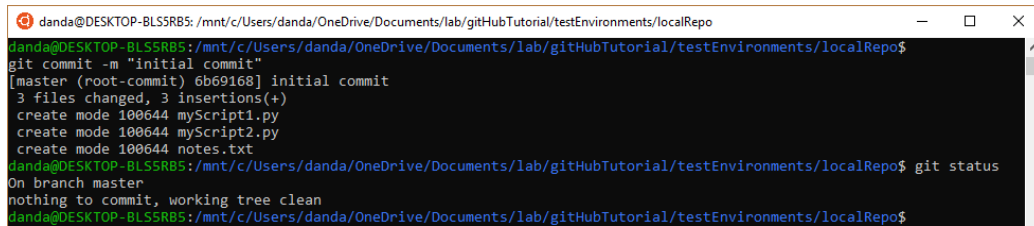
A screenshot of a terminal window with a dark background. The window title is "danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo". The terminal shows the following commands and output:  
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo\$ git commit -m "initial commit"  
[master (root-commit) 6b69168] initial commit  
3 files changed, 3 insertions(+)  
create mode 100644 myScript1.py  
create mode 100644 myScript2.py  
create mode 100644 notes.txt  
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo\$ git status  
On branch master  
nothing to commit, working tree clean  
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo\$

Figure 3

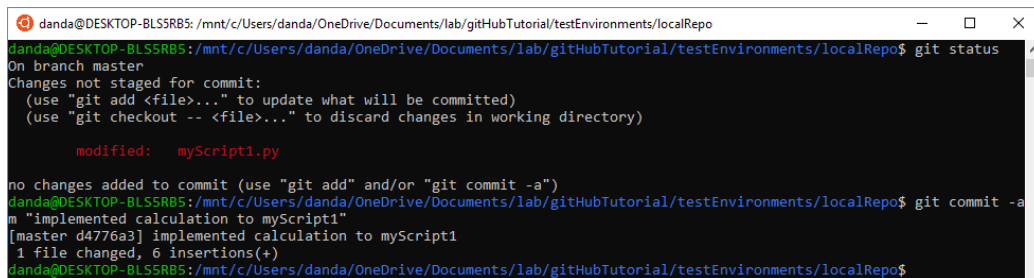
Now as your project develops, you'll want to periodically save your progress and commit your repository again. Like many things it is important to commit at appropriate intervals. Committing too often will make it more difficult to traverse your version history if you need to revert to a previous version, and not committing often enough may leave you forced to revert too far back if you run into an issue. Commits are also a good record keeping method of tracking progress and what features were added when. So in this example I will make some changes to `myScript1.py`. After making the changes I can see what Git is doing with the status command again. Doing so shows that Git recognizes the changes but they're not staged for commit - so you have to add it again. We can do the `add` command then commit

```
git add *  
git commit -m "implemented calculation to myScript1"
```

Or we can do it in one go by telling `commit` that we want to commit everything by adding `"a"` to the command options.

```
git commit -am "implemented calculation to myScript1"
```



A screenshot of a terminal window with a dark background. The window title is "danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo". The terminal shows the following commands and output:

```
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ 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:   myScript1.py

no changes added to commit (use "git add" and/or "git commit -a")
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git commit -a
[master d4776a3] implemented calculation to myScript1
1 file changed, 6 insertions(+)
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$
```

Figure 4

**IMPORTANT NOTE:** Using the `-am` command will only commit *changes* to files that are *already tracked*. If there are new files then you will have to use `git add` as well.

### 3.3.3 Defining Files to Ignore

Now let's say I've been working on this project and have some additional files I've created:

- Two more Python scripts
- A text file of test data I'm using
- A couple spreadsheets for project planning and scheduling
- A folder of images that my code has generated that I need for debugging

Outside of the Python scripts, these are all things that I don't want to be part of the repository when I have it on GitHub for others to use. But we've been using `git add *` and `git commit -am`, so this won't work in this case. I could do

```
git add myScript1.py
git add myScript2.py
git add myScript3.py
git add myScript4.py
```

Or

```
git add *.py
```

The first option is clearly non-ideal, and the second is a good way to add all of the Python files, but if there are other file types I want to add then this is going to be annoying as well. This is where creating a `.gitignore` file is really helpful. First lets look at the list of files in the repository and the status of Git.

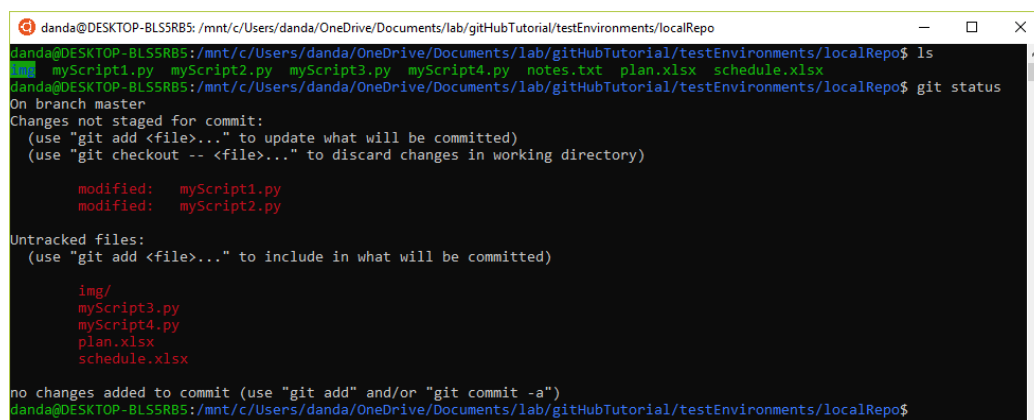
A terminal window titled 'danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo'. The terminal shows the following commands and output:  
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo\$ ls  
myScript1.py myScript2.py myScript3.py myScript4.py notes.txt plan.xlsx schedule.xlsx  
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo\$ 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: myScript1.py  
 modified: myScript2.py  
  
Untracked files:  
 (use "git add <file>..." to include in what will be committed)  
  
 img/  
 myScript3.py  
 myScript4.py  
 plan.xlsx  
 schedule.xlsx  
  
no changes added to commit (use "git add" and/or "git commit -a")  
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo\$

Figure 5

What we'll do here is create a new file called `.gitignore` which you can create in whatever way you prefer (VIM, Notepad, etc.) - but it is essential that the file does not end up with some extension after it like `.gitignore.txt`. Within this file we will just list all of the things we want git to ignore. If you use Notepad it might force you to create a `.txt` file first but then you can rename it to `.gitignore`. To ignore the files I want in this project the ignore file looks like:

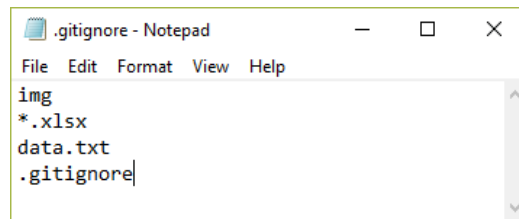


Figure 6

1. The first line will tell Git to ignore the folder named `img` and everything inside of it.
2. This line tells Git to ignore every file with the Excel extension `xlsx`.
3. This line tells Git to ignore the file named `data.txt`, which will not affect the `notes.txt` file from before.
4. The last line tells git to ignore this file as well. Sometimes you may actually want to track the ignore file if you have some repository that will generate large folders upon compilation so that collaborators don't end up tracking unnecessary files too.

Now if we check the status of Git we will see the ignore file is working and we can just commit everything with `git add` since we also have new files.

```
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ 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:   myScript1.py
        modified:   myScript2.py

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

        myScript3.py
        myScript4.py

no changes added to commit (use "git add" and/or "git commit -a")
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git add *
The following paths are ignored by one of your .gitignore files:
data.txt
img
plan.xlsx
schedule.xlsx
Use -f if you really want to add them.
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   myScript1.py
        modified:   myScript2.py
        new file:   myScript3.py
        new file:   myScript4.py

danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git commit -a
[m "Modified scripts 1 and 2, added scripts 3 and 4"
[master fd9e8d0] Modified scripts 1 and 2, added scripts 3 and 4
4 files changed, 14 insertions(+)
create mode 100644 myScript3.py
create mode 100644 myScript4.py
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$
```

Figure 7

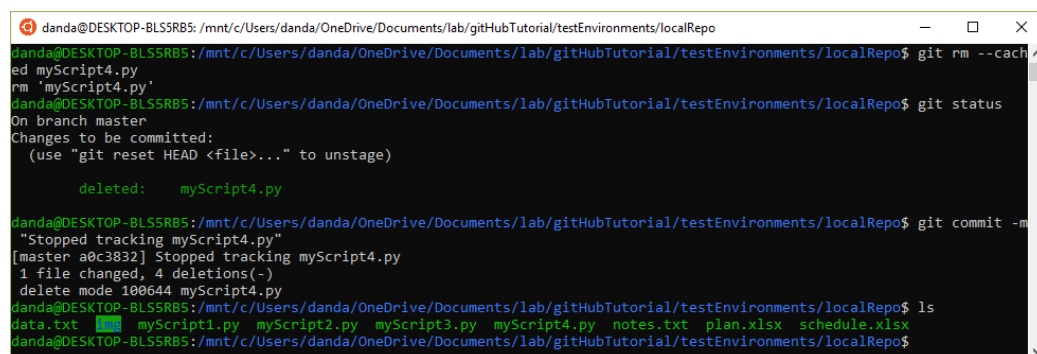
**Note:** It is possible to make commits with modified files not staged for commit or untracked files, but make sure you're not going to run into some discrepancy down the road if you are working on an online repository. Git also won't let you go to a different version of the repository if it would mean losing unsaved information. If you run into this problem, go ahead and make a commit before going to another version, or look into using `git stash`

### 3.3.4 Removing Files

Now what if `myScript4.py` uses something in `myScript1.py`, but recent improvements to `myScript1.py` causes errors in `myScript4.py`? You'll want to make it so that people using this repository online aren't being given the broken script, but you want to keep it locally so you can fix it and recommit it later. You'll want to add this file to `.gitignore` so that it isn't showing

up every time you want to commit in the mean time, but if you only do this the script will still be tracked. The ignore file only prevents *untracked* files from being tracked in an add command. In order to stop tracking a file you'll do the following:

```
git rm --cached myScript4.py
```



```
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git rm --cached myScript4.py
rm 'myScript4.py'
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    myScript4.py
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git commit -m "Stopped tracking myScript4.py"
[master a0c3832] Stopped tracking myScript4.py
1 file changed, 4 deletions(-)
delete mode 100644 myScript4.py
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ ls
data.txt  myScript1.py  myScript2.py  myScript3.py  myScript4.py  notes.txt  plan.xlsx  schedule.xlsx
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$
```

Figure 8

You can see that `myScript4.py` is still in the folder locally, but has been deleted from the GitHub commits. It was also added to the ignore file so it will not show up in the status or be tracked by an `add *` until it is removed from the ignore list. If it is determined that a file is completely unnecessary then it can be deleted locally just as you would with any file and Git will recognize this. Deleting the script and recommitting will update the repository to reflect the deletion. The file will be recoverable if it was previously committed - this is covered in Section 3.3.5

```
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ rm myScript4.py
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ ls
data.txt  myScript1.py  myScript2.py  myScript3.py  notes.txt  plan.xlsx  schedule.xlsx
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    myScript4.py

no changes added to commit (use "git add" and/or "git commit -a")
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git commit -a
[master 96cc787] Removed myScript4.py
1 file changed, 4 deletions(-)
delete mode 100644 myScript4.py
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$
```

Figure 9

### 3.3.5 Viewing the Log and Reverting to a Previous Commit

Sometimes you may want to go back to a previous version of your project. Let's say you had a working algorithm and spent some time working with it and you think you got it working. You commit this new algorithm but haven't quite tested it fully. Now, a couple days later you find a serious error in it but can't remember what the algorithm was before and you can't CTRL-Z back all the way.

```
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ 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:   myScript1.py

no changes added to commit (use "git add" and/or "git commit -a")
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ git commit -a
[master 3760fe6] Modified algorithm in script 1 - UNTESTED
1 file changed, 2 insertions(+), 1 deletion(-)
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$ python myScript1.py
hello world
Traceback (most recent call last):
  File "myScript1.py", line 7, in <module>
    x = z / y + y - x**2 / a
ZeroDivisionError: integer division or modulo by zero
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$
```

Figure 10

A good first step is to check the logs to see your commit history and figure out

where you want to go back to. This is done with the `git log` command. This command has a ton of options to specify the info format and get what you need (see <https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History>). For this simple task we will use the "pretty" format with the following command:

```
git log --pretty=oneline
```

This will show you all of the commits starting with its unique hash which is basically just an ID for each commit and the comment that you've written for it. In this case, just going back to the previous commit will suffice. You can do so with the `revert` command, specifying that you just want to revert once from the HEAD (pointer to your current location in the history).

```
git revert HEAD --no-edit
```

Since the `revert` command also creates a new commit, it requires a message. What `--no-edit` does is it automatically generates the comment for you. If you want to write your own message, then you will have to be familiar with editing in the terminal with something like VIM (you'll also have to tell git which editor you prefer in `git config`). Reverting will automatically change the files on your machine to what they were in that commit, including deleting files, adding files, etc. This change should immediately be reflected in any text editor or IDE you are using. If you have uncommitted changes you will likely be warned about this and Git will hold off on reverting. You can either commit the changes you have made before reverting or use the `--hard` option to discard changes.

```
danda@DESKTOP-BLS5RBS: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$
danda@DESKTOP-BLS5RBS: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$
git log --pretty=oneline
3760fe628675d2e43cc53fa9dc982fee55c2cc9 (HEAD -> master) Modified algorithm in script 1 - UNTESTED
96cc787f62094b49ee09f6fbf0b961b212806d76 Removed myScript4.py
ec6236624b09364505b6b6f074f6f026ba02db0b Return my Script4.py to repo
a0c3832a3c3e7c7c49a71703f70c27d7a2397f6 Stopped tracking myScript4.py
fd9e8d0128819a516c87ac645a4a932286b66f10 Modified scripts 1 and 2, added scripts 3 and 4
be6477bb6644647100ddd7c3b07596857e969cc4 Added scripts 3 and 4, modified scripts 1 and 2
8cc0616c854364ed788ddd8b41db13fc428c6e8a Added scripts 3 and 4
d4776a384132610433775096161eadf024dc5293 implemented calculation to myScript1
6b691683fda4cd5c54cf7dc34e24751405fa40c0 initial commit
danda@DESKTOP-BLS5RBS: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$
git revert HEAD --no-edit
[master 1a9ae0c] Revert "Modified algorithm in script 1 - UNTESTED"
Date: Thu May 30 17:53:36 2019 -0600
1 file changed, 1 insertion(+), 2 deletions(-)
danda@DESKTOP-BLS5RBS: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$
git log --pretty=oneline
1a9ae0c616be2a48ee9c1f5ce5418a69d7fdbc62 (HEAD -> master) Revert "Modified algorithm in script 1 - UNTESTED"
3760fe628675d2e43cc53fa9dc982fee55c2cc9 Modified algorithm in script 1 - UNTESTED
96cc787f62094b49ee09f6fbf0b961b212806d76 Removed myScript4.py
ec6236624b09364505b6b6f074f6f026ba02db0b Return my Script4.py to repo
a0c3832a3c3e7c7c49a71703f70c27d7a2397f6 Stopped tracking myScript4.py
fd9e8d0128819a516c87ac645a4a932286b66f10 Modified scripts 1 and 2, added scripts 3 and 4
be6477bb6644647100ddd7c3b07596857e969cc4 Added scripts 3 and 4, modified scripts 1 and 2
8cc0616c854364ed788ddd8b41db13fc428c6e8a Added scripts 3 and 4
d4776a384132610433775096161eadf024dc5293 implemented calculation to myScript1
6b691683fda4cd5c54cf7dc34e24751405fa40c0 initial commit
danda@DESKTOP-BLS5RBS: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localRepo$
```

Figure 11

This might seem odd to have a new commit that's just a copy of a previous version, but it is important to retain your history in Git. You may want to revisit the version with the error if you realize what the problem was, and using `git reset` rather than `git revert` will delete everything after the version you return to. This is especially problematic if the repository is shared - mismatching histories can cause tons of problems and will be surprisingly difficult to reconcile.

## Going back further

If you want to go more than just one commit back, things can get pretty complicated. One option is to use `revert` to revert a range of commits, and the other is to use branching which will be discussed in Section 4. Each will accomplish a similar result but one may be preferable over the other depending on your situation. Git also allows you to revert changes in a commit or commits further back without reverting changes in most recent commits. This isn't a common need, and it will likely be a lot less confusing to use the branching method to just go to a previous state and move that forward with a new commit. For more information on these finer details of `revert`, see <https://git-scm.com/docs/git-revert>.



## 3.4 Moving Online with GitHub

Using Git locally provides huge advantages for version control and project management on one machine. Where Git really shines though is using it with GitHub to backup your code, share code with others, work collaboratively, and quickly pull code and updates to other machines.

### 3.4.1 GitHub Account

First you'll need a GitHub account. It's best that everyone has their own account for logistical reasons with collaboration which will be clearer later. Go to <https://github.com> and click on sign-up. As we collaborate GitHub will show names, but it is nice to have your username related to your name just so everyone doesn't have to remember who has the username `lasersAreCool`. For example my username is `dscarbro21`.

Put in your email and a secure password and continue to the next step. When prompted to select a plan, just choose the free one - it's all you'll need. It will ask you a few questions about yourself then you will need to verify your email address. After that you're in and brought to the homepage. There are several guides out there, but I've written this one to get you all of the essentials I've found very useful in one shorter summary. This won't cover everything though, Git is very robust and it takes time and practice to learn and expand beyond the basics.

### 3.4.2 Putting an Existing Local Repository on GitHub

If you have an existing project that you want to backup, share and work on with others, or deploy to multiple machines with easy updating then you'll want to put it on a new *remote* repository hosted on GitHub.

For this example I have just made a new copy of the project from the *Local Git Basics* section. If you copy the whole directory, it copies all of the Git history along with it. So running `git status` will show that we are on the master branch with nothing to commit, and checking the log would show the

same commit history.

To get this project and its files on GitHub, we will have to first create a new repository on the website. Log in to your account on <https://github.com> and click the green "Create Repository" button. If you already have one or more repositories, they will be listed on the left panel. To make a new one just click the green "New" button. You will be greeted with the screen shown in Figure 12. Enter a descriptive name for the repository and a short description. You can select to have the repository be public or private. You can always give access to other users as needed in a private repository. Since we already have a working local repository, don't check the box to create a README; we will first bring the remote repository up to date with the local one to avoid a discrepancy in files right off the bat. Finish up by clicking "Create Repository"

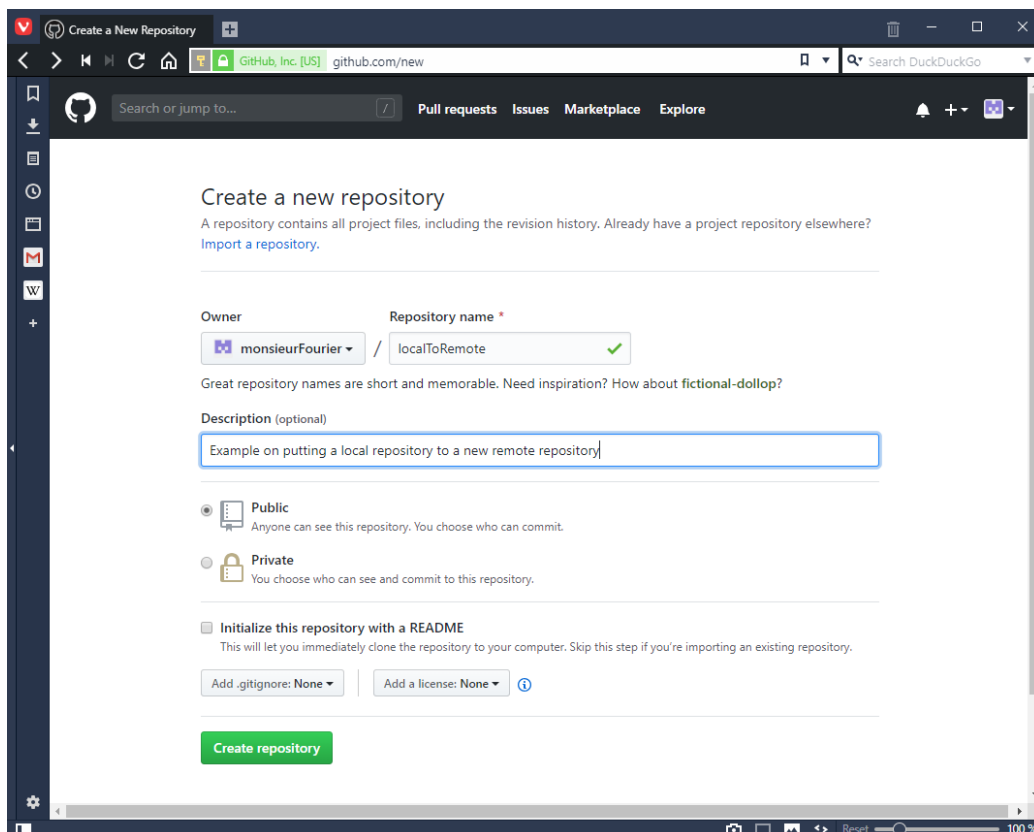


Figure 12

Now there will be a few sets of instructions for how we can get started in this repository. For this one we will want to push an existing repository from the command line. This is the section outlined in red in Figure 13. You can also see in the blue outline that your username is part of the URL. I've created this *Monsieur Fourier* to provide some example repositories which we will use later on.

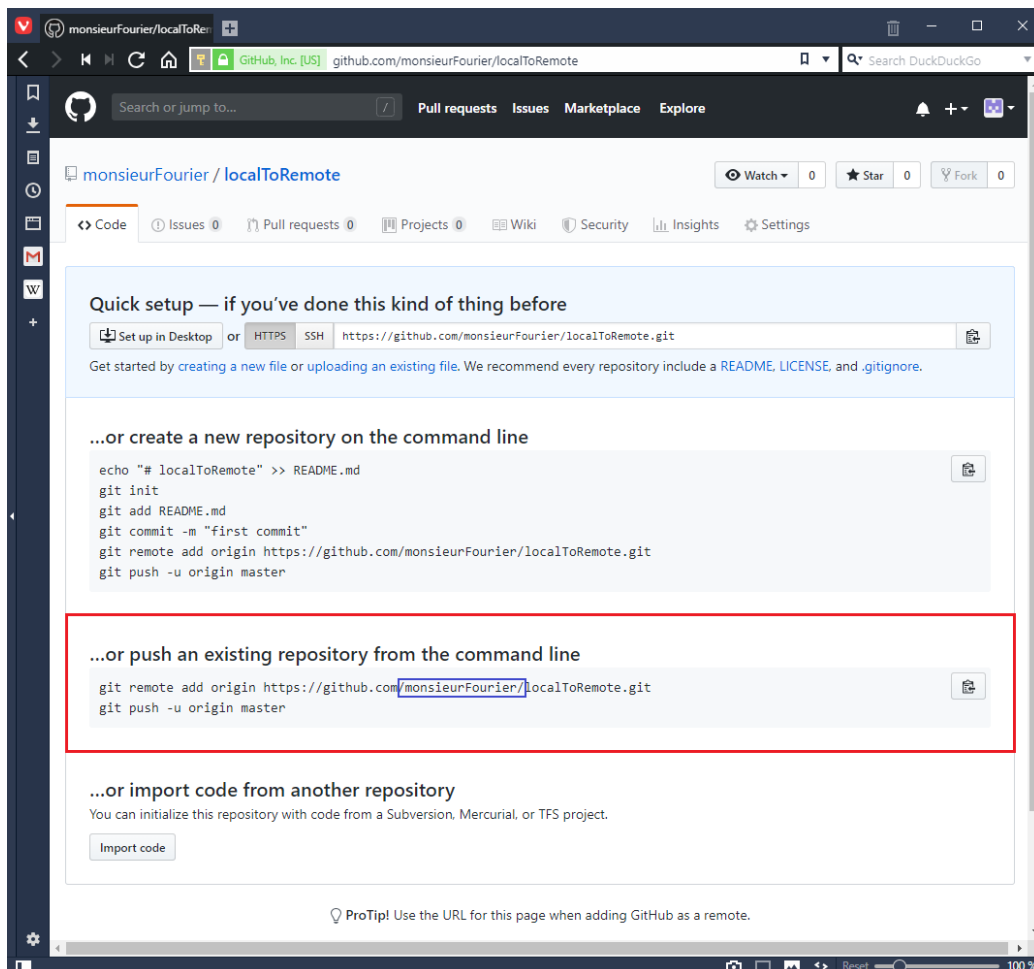
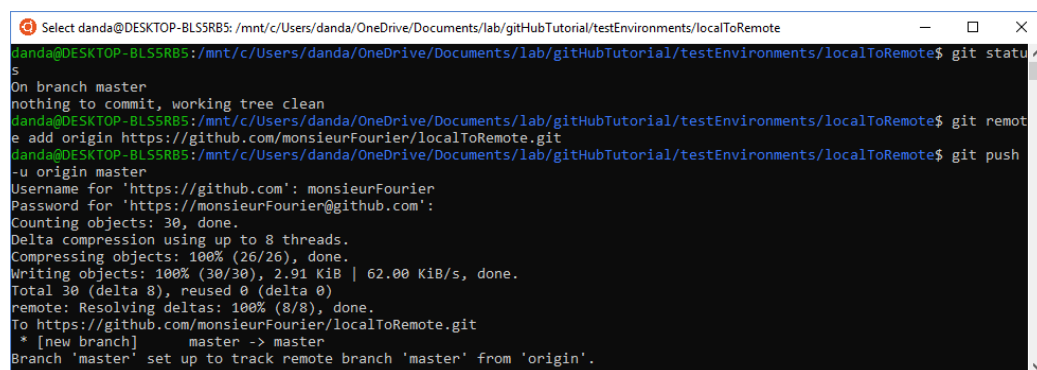


Figure 13

Copy these two lines or type them yourself into a terminal that is in your local Git repository. Git will prompt you for your username and password,

and the password won't show as you type it for security. After verifying your login, you should see the following indicating a successful push. Explaining these commands a little: the first command tells your Git repository that you are adding a *remote* reference to a repository called *origin* and it is at the provided URL. The second line tells Git to push all of your commits to the new remote repository, and the `-u` option tells git that the **master** branch will be the default branch to push to (more on this later).



```
Select danda@DESKTOP-BLS5RBS: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote
danda@DESKTOP-BLS5RBS:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote$ git status
On branch master
nothing to commit, working tree clean
danda@DESKTOP-BLS5RBS:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote$ git remote
e add origin https://github.com/monsieurFourier/localToRemote.git
danda@DESKTOP-BLS5RBS:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote$ git push
-u origin master
Username for 'https://github.com': monsieurFourier
Password for 'https://monsieurFourier@github.com':
Counting objects: 30, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (26/26), done.
Writing objects: 100% (30/30), 2.91 KiB | 62.00 KiB/s, done.
Total 30 (delta 8), reused 0 (delta 0)
remote: Resolving deltas: 100% (8/8), done.
To https://github.com/monsieurFourier/localToRemote.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Figure 14

Now if you refresh the GitHub page in your browser you should see all of your files are now there on the remote repository. Also note that any files that were in your `.gitignore` were not uploaded. You can also click on the commits tab to see the commit history and even view the repository as it was in each commit. GitHub will also let you read each file within the browser. Back on the main repository page you can see the commit message for when each file was edited last. Every project should have a README so others (and maybe your future self) can read about what the project is for, what bugs it may have, how to use it, etc. Go ahead and click on the green "Add a README" button.

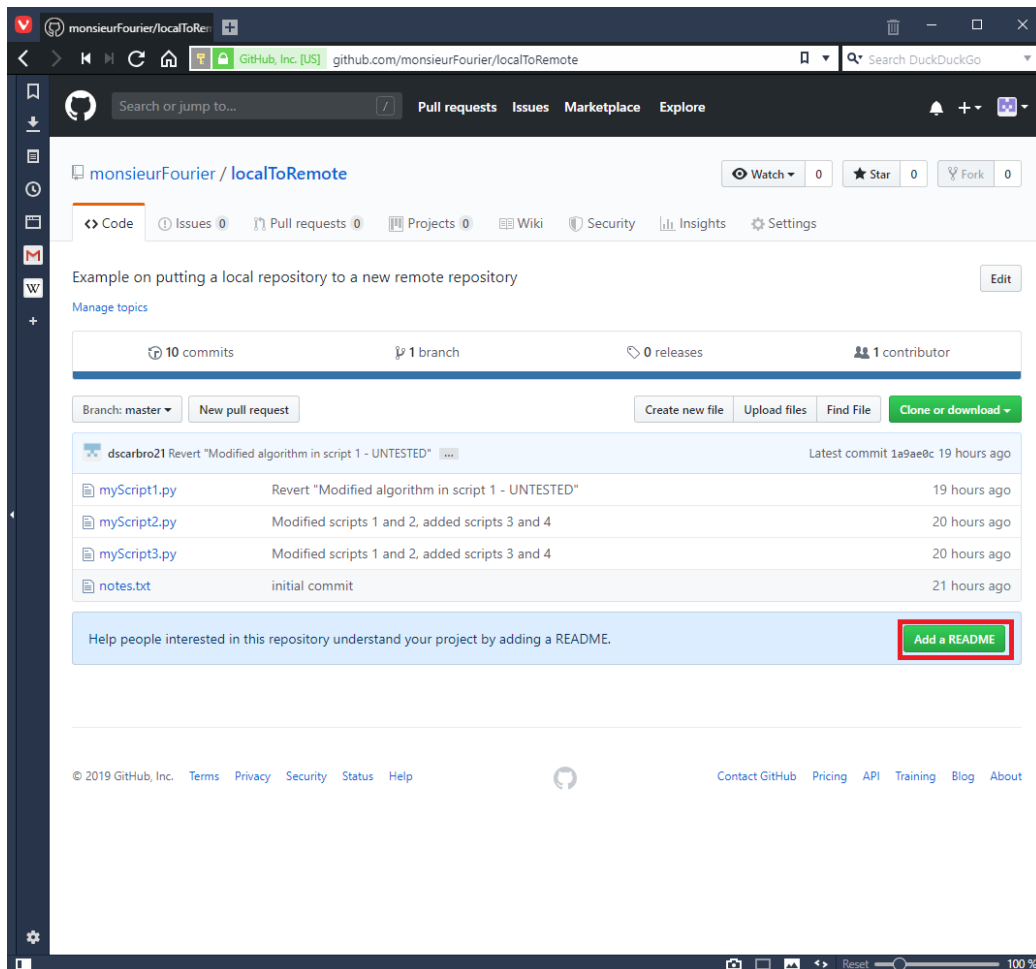


Figure 15

Add some text to the Readme then scroll down to the bottom of the page.

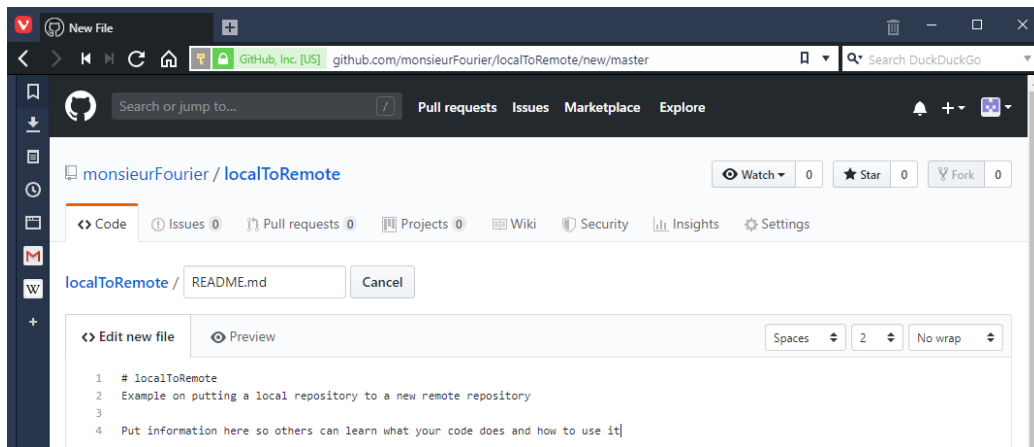


Figure 16

You can change the default commit message if you like, and even add some extra optional details if you like. Leave the "Commit directly to the **master** branch" option selected and click "Commit new file".

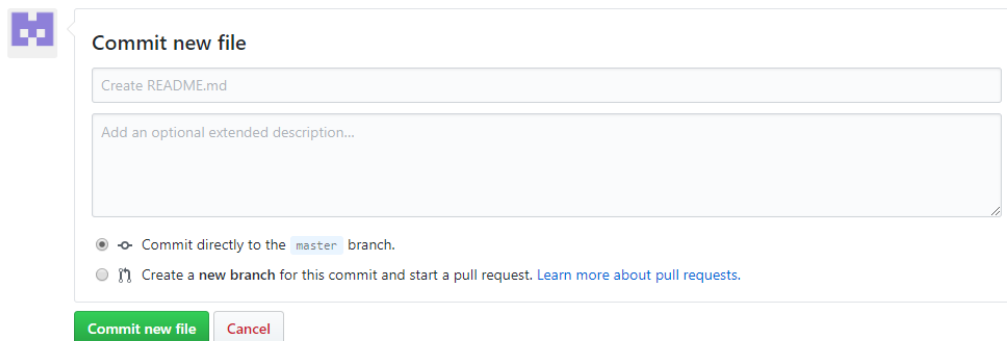


Figure 17

Back on the repository page, you'll now see the README.md file in your repository and GitHub automatically displays it below. This is a really nice feature as you and others view the repository to quickly get an idea of what it's for. Now our remote repository is one commit ahead of our local repository. Before you make any new changes in the local one, you'll want to

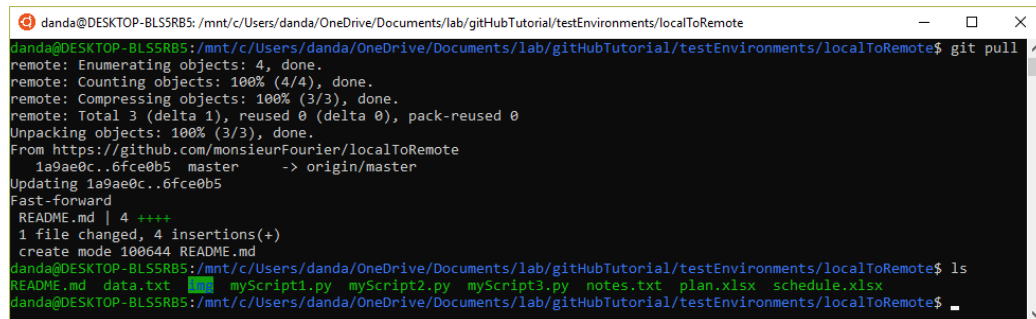
perform a `pull` to avoid any conflicts (which are resolvable but avoidable). Go back to your terminal and enter

```
git pull
```

Once it finishes you should see that you now have the `README.md` file. Since we used the `-u` option earlier when we first pushed our repository, Git knows that a `pull` should default to our `origin` remote repository and its `master` branch. So it basically ran the command

```
git pull origin master
```

So you could use either. If you are working with branches you can define which branch you'd like to pull from - which will be discussed later in the Branching section.



```
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/monsieurFourier/localToRemote
   1a9ae0c..6fce0b5  master    -> origin/master
Updating 1a9ae0c..6fce0b5
Fast-forward
 README.md | 4 +++
 1 file changed, 4 insertions(+)
 create mode 100644 README.md
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote$ ls
README.md data.txt myScript1.py myScript2.py myScript3.py notes.txt plan.xlsx schedule.xlsx
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote$
```

Figure 18

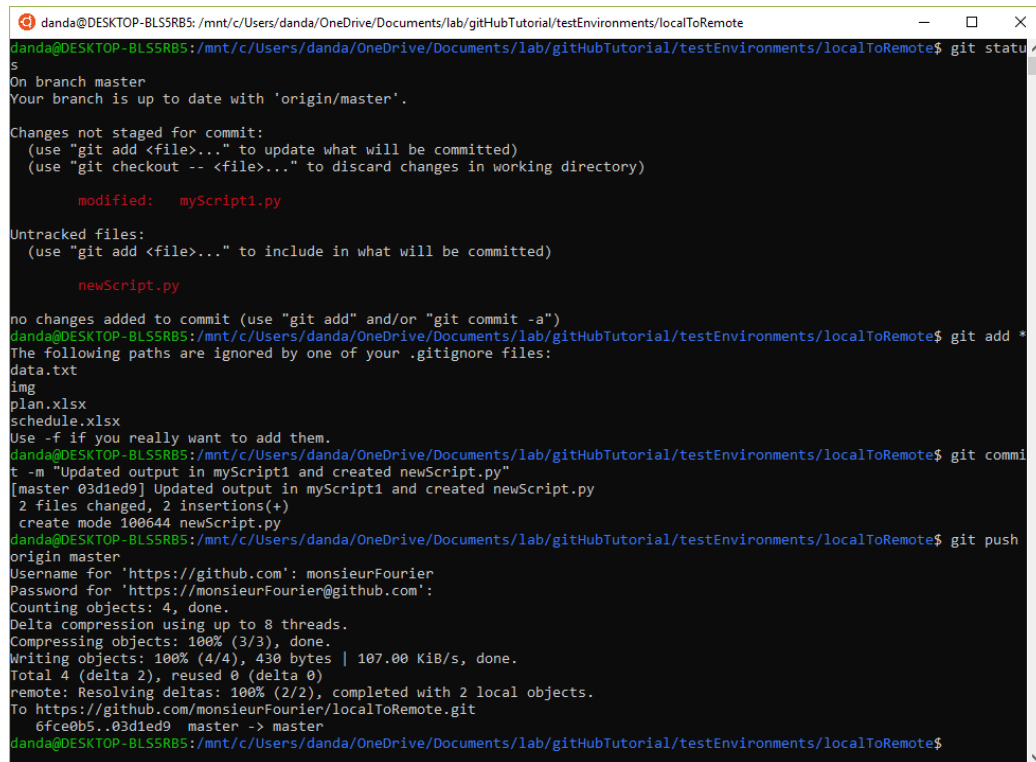
### 3.4.3 Pushing Commits to Your Remote Repository

Now that we have a local repository linked up with a remote repository, you'll want to update the remote version as you make more progress on the project locally. Once you've made a commit or even many commits, you'll want to push it back up to the remote repository. This is really simple to do, just enter

```
git push origin master
```

Assuming you want to push it to the master branch. Now all of your new

changes are backed up and easily accessible on GitHub. If you're using this project on any other machines, you'll have to update their state with a `pull` which is described in Section 3.4.5



```
danda@DESKTOP-BLS5RBS: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote$ git status
On branch master
Your branch is up to date with 'origin/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:   myScript1.py

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

        newScript.py

no changes added to commit (use "git add" and/or "git commit -a")
danda@DESKTOP-BLS5RBS: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote$ git add *
The following paths are ignored by one of your .gitignore files:
data.txt
img
plan.xlsx
schedule.xlsx
Use -f if you really want to add them.
danda@DESKTOP-BLS5RBS: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote$ git commit -m "Updated output in myScript1 and created newScript.py"
[master 03d1ed9] Updated output in myScript1 and created newScript.py
 2 files changed, 2 insertions(+)
 create mode 100644 newScript.py
danda@DESKTOP-BLS5RBS: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote$ git push
origin master
Username for 'https://github.com': monsieurFourier
Password for 'https://monsieurFourier@github.com':
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 430 bytes | 107.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/monsieurFourier/localToRemote.git
 6fce0b5..03d1ed9  master -> master
danda@DESKTOP-BLS5RBS: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/localToRemote$
```

Figure 19

### 3.4.4 Cloning a Remote GitHub Repository

Now lets say you've written some code on one machine, put it on a remote repository on GitHub, and now you want to use your code on another machine. You could download the repository from GitHub and unzip the download - but if you want to be able to get updates from the remote repository on this machine in the future you'll have to do that all again.

Instead you can quickly `clone` a remote repository and leave it connected to the remote `origin` so you can just run a `git pull` in the future. First login



to your GitHub account in your browser and navigate to the repository you want to clone. Find the green "Clone or download" button and click on it. It will drop down and you'll be able to see a URL for the repository. Go ahead and copy that URL.

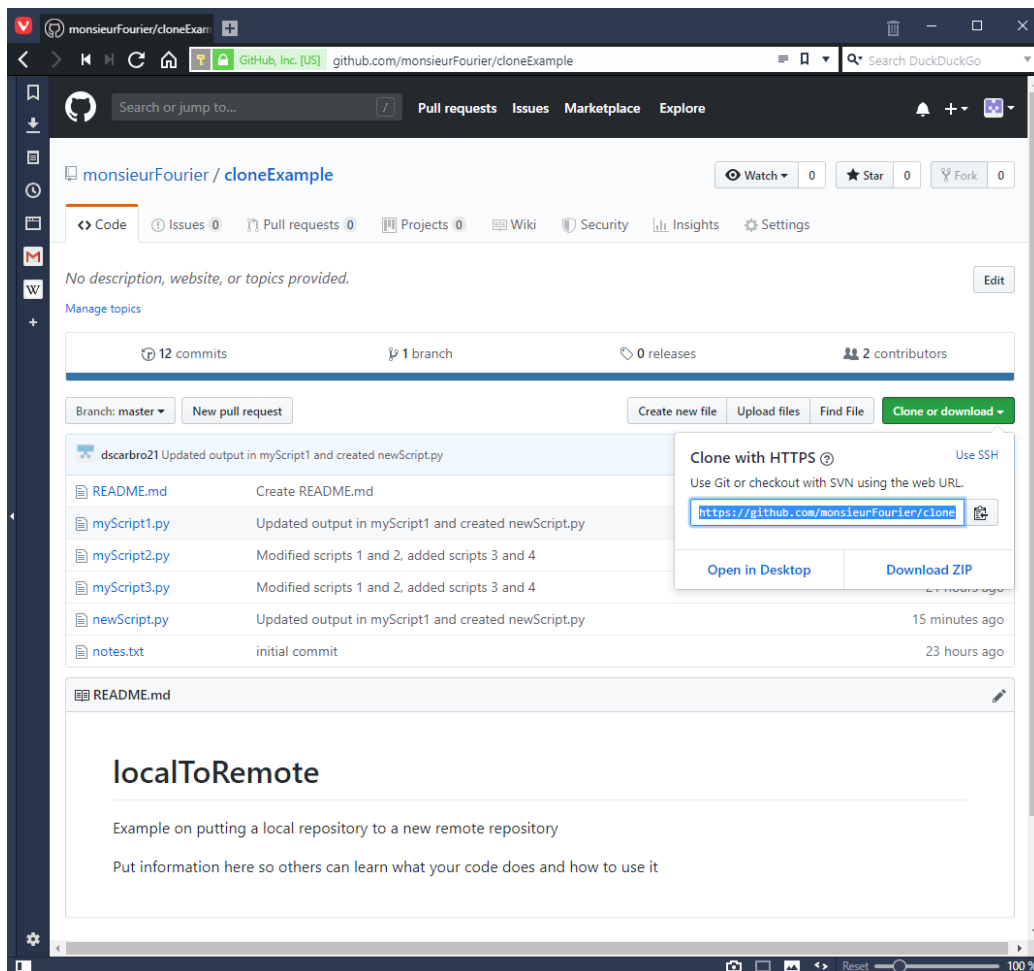


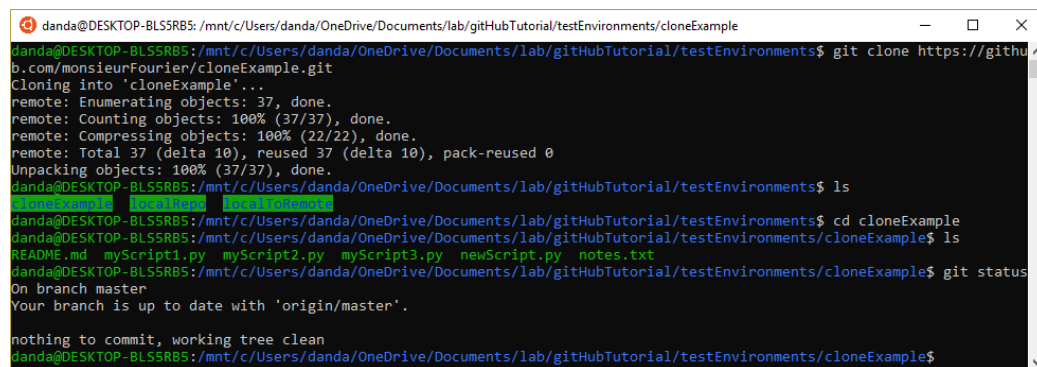
Figure 20

Now open up a terminal and navigate to the place where you want your **project folder** to be. Cloning will automatically make a new folder in your current location with the name of the repository, and all of its files will then be cloned into that folder. I've made an example repository using the files

from the previous section called `cloneExample`. So if I wanted to clone that repository I would just enter

```
git clone https://github.com/monsieurFourier/cloneExample.git
```

Where you would enter the URL for the repository you want. Now if it is your repository you can use the code, make changes (and commit them!) and then push them to your remote repository with a simple `git push origin master` as the `origin` is already set via the cloning.



```
danda@DESKTOP-BLS5RB5: /mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/cloneExample
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments$ git clone https://github.com/monsieurFourier/cloneExample.git
Cloning into 'cloneExample'...
remote: Enumerating objects: 37, done.
remote: Counting objects: 100% (37/37), done.
remote: Compressing objects: 100% (22/22), done.
remote: Total 37 (delta 10), reused 37 (delta 10), pack-reused 0
Unpacking objects: 100% (37/37), done.
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments$ ls
cloneExample  localUser  testExample
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments$ cd cloneExample
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/cloneExample$ ls
README.md  myScript1.py  myScript2.py  myScript3.py  newScript.py  notes.txt
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/cloneExample$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
danda@DESKTOP-BLS5RB5:/mnt/c/Users/danda/OneDrive/Documents/lab/gitHubTutorial/testEnvironments/cloneExample$
```

Figure 21

You can clone others' repositories if you have the URL. If you want to clone the same example use the above URL which can also be found on the public repository page for the example user *Monsieur Fourier*. Since it is a public repository you are allowed to clone it, however only the owner is allowed to push changes to the remote repository. If another user wants to make changes there are a few options:

- The owner can add you as a collaborator under **Settings > Collaborators** on the repository's GitHub page. This is likely the preferred method if the owner trusts you not to wreck their repository.
- Submit a pull request. This allows for a collaborative method for the owner to first review and approve your changes before merging them into their repository. This process can be multi-stepped and the owner can also make edits and commits on your work. This is the preferred method if the owner wants to maintain control over the repository.

- The owner of the repository can review the commits on the editor's machine, and if they are ok with them they can perform the push using their login credentials.
- You can fork the repository from the repository's GitHub page. This creates your own copy of the repository independent of the owner's.

Fine details about working with collaborators and pull requests will be discussed later as there are unique challenges when using either method.

### 3.4.5 Pulling Updates from a Remote Repository

If you have a repository that is connected to a remote repository as we have used in the last couple of examples, there will likely be times where you want to get new updates from the remote repository. All you have to do here is to use `git pull` to request a branch from the remote repository. It is most likely that for now you will just want the `master` branch, so you can enter

```
git pull origin master
```

Or assuming the `-u` option was used on a push to set the default branch to `master` just `git pull` will suffice.

Using `git pull` will also work if you are using someone else's repository that you've cloned. You can always pull updates even if you can't push. If the owner of the repository has a specific branch that they've specified as the default to pull from, `git pull` will get you that branch. You can however request any branch you want.

## 4 Branching

### 4.1 Merging

#### 4.1.1 Merge Conflicts