Pro Git(https://git-scm.com/book/ko/v2/)를 바탕으로 작성하였습니다.

YoonJoon Lee
SoC KAIST

Fall 2018

What we will take a look in this series

1. Getting Started
2. Git Basics
3. Git Branch
4. Git Server - GitLab

What we will take a look today

1. Getting Started
2. Git Basics
3. Git Branch
4. Git Server - GitLab

What I will talk about in this section

1. Getting a Git Repository
2. Recording Changes to the Repository
3. Viewing the Commit History
4. Undoing Things
5. Working with Remotes
6. Tagging
7. Git Aliases

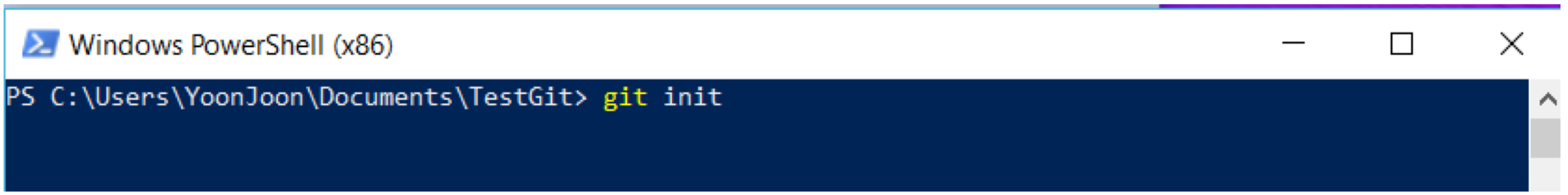
Getting a Git Repository

Typically obtain a Git repository in one of two ways:

1. Take a local directory that is currently not under version control, and turn it into a Git repository, or
2. *Clone* an existing Git repository from elsewhere.

Initializing a Repository in an Existing Directory

You have a project directory that is currently not under version control and you want to start controlling it with Git.



This creates a new subdirectory named `.git` that contains all of your necessary repository files — a Git repository skeleton.

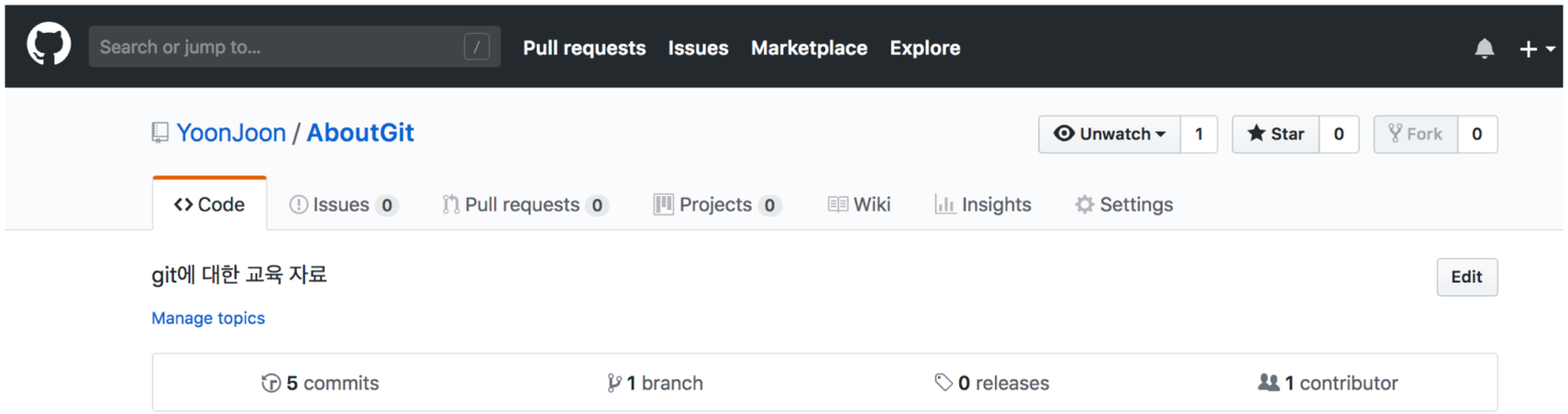
If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit.

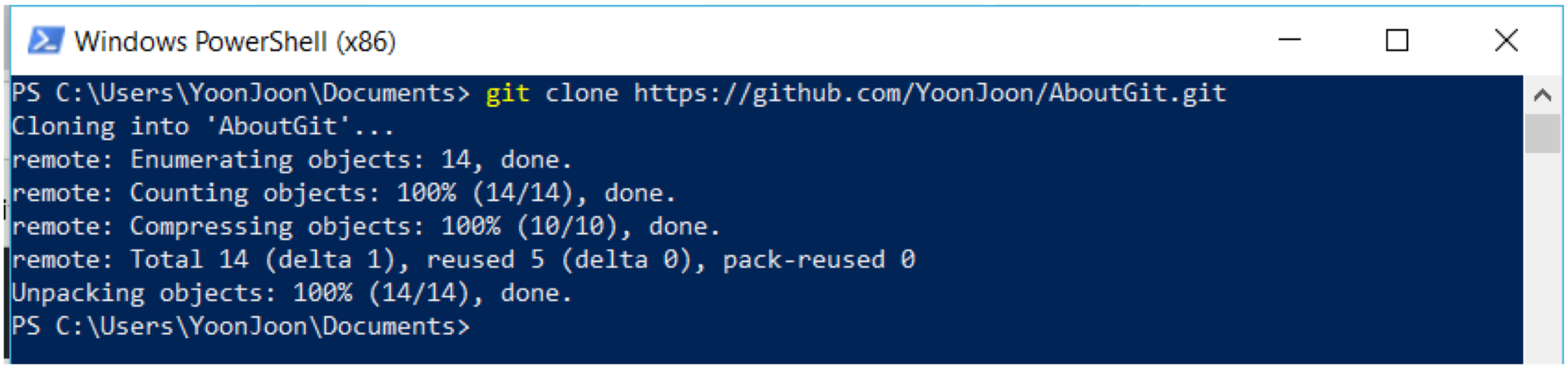
```
Windows PowerShell (x86)
PS C:\Users\YoonJoon\Documents\TestGit> ls


    Directory: C:\Users\YoonJoon\Documents\TestGit


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        2018-10-06   오후 2:18             37 README.md
```

```
Windows PowerShell (x86)
PS C:\Users\YoonJoon\Documents\TestGit> git add *
PS C:\Users\YoonJoon\Documents\TestGit> git commit -m "initial project version"
[master (root-commit) 90bc483] initial project version
 1 file changed, 3 insertions(+)
 create mode 100644 README.md
PS C:\Users\YoonJoon\Documents\TestGit>
```

Cloning an Existing Repository

If you want to get a copy of an existing Git repository, the command you need is `git clone`.



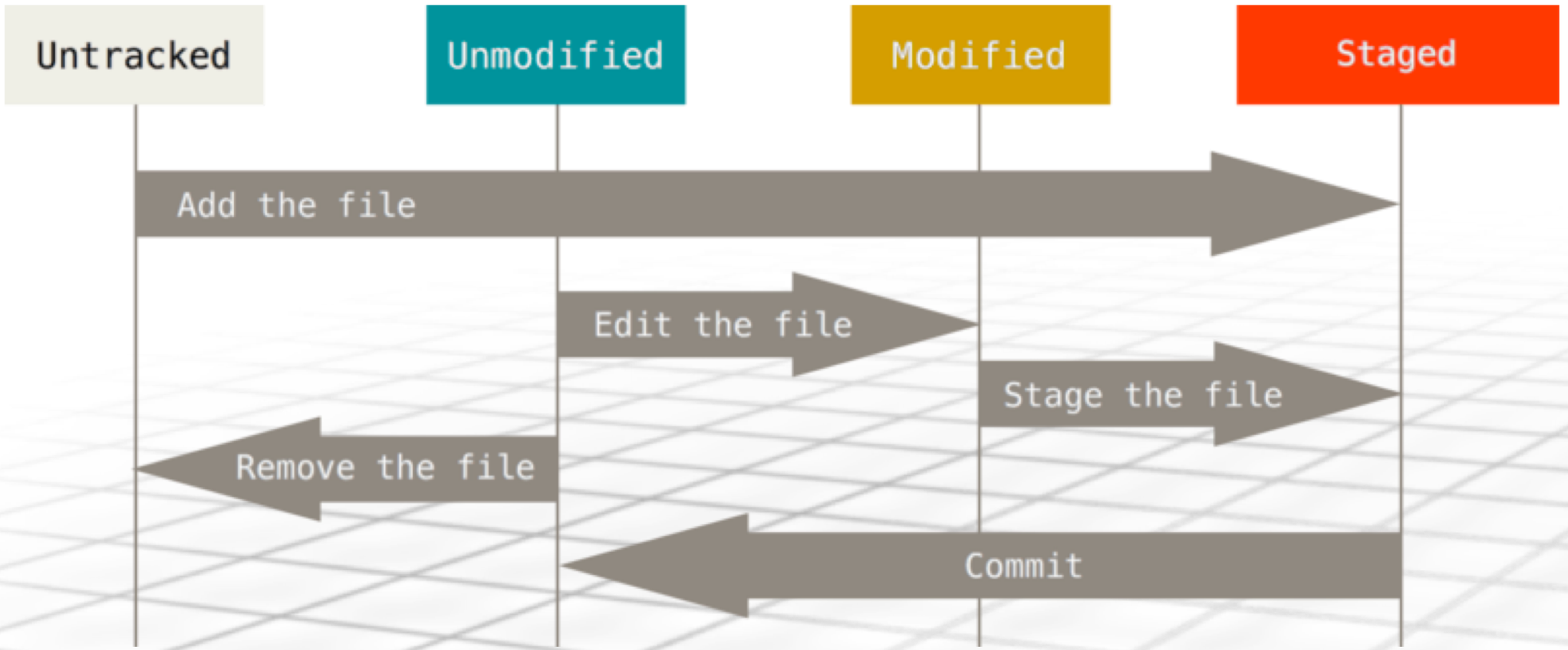


Recording Changes to the Repository

At this point, we should have a *bona fide* Git repository on our local machine.

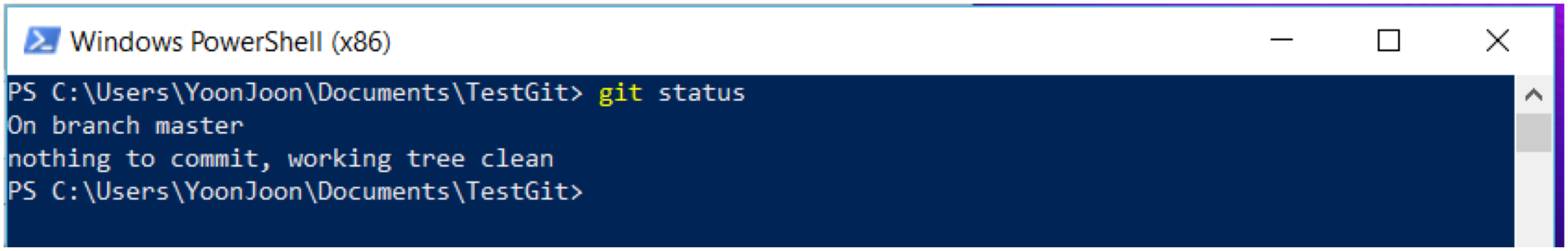
Each file in the working directory can be in one of two states: *tracked* or *untracked*.
Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged.

When you first clone a repository, all of your files will be tracked and unmodified


Untracked
Unmodified
Modified
Staged
Add the file
Edit the file
Stage the file
Remove the file
Commit

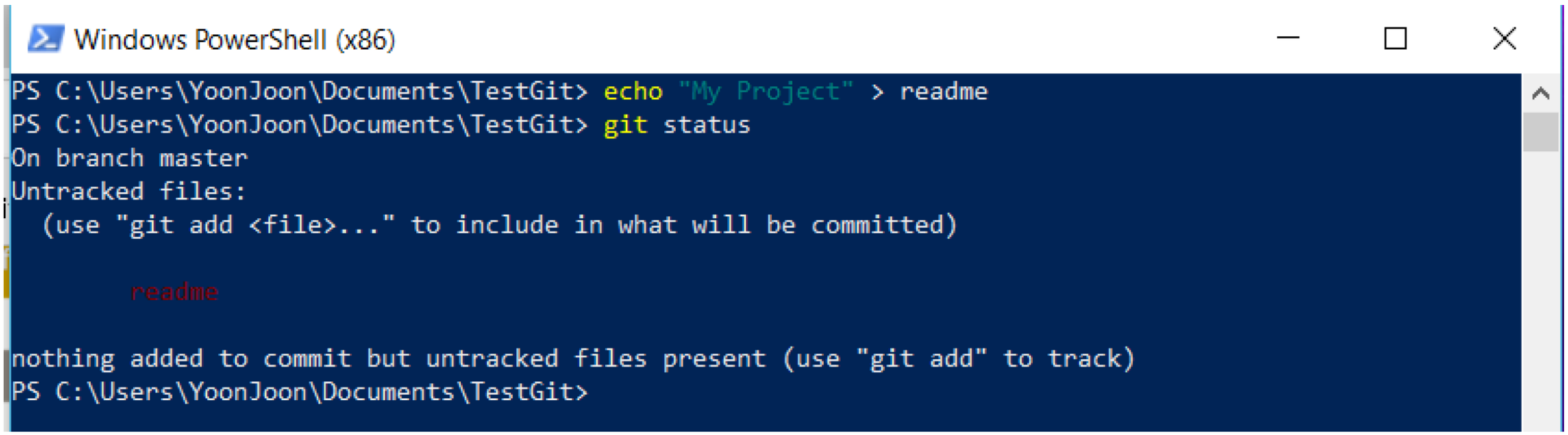
Checking the Status of Your Files

The `git status` command shows which files are in which state is



The command tells you which branch you're on and informs you that it has not diverged from the same branch on the server. For now, that branch is always "master", which is the default.

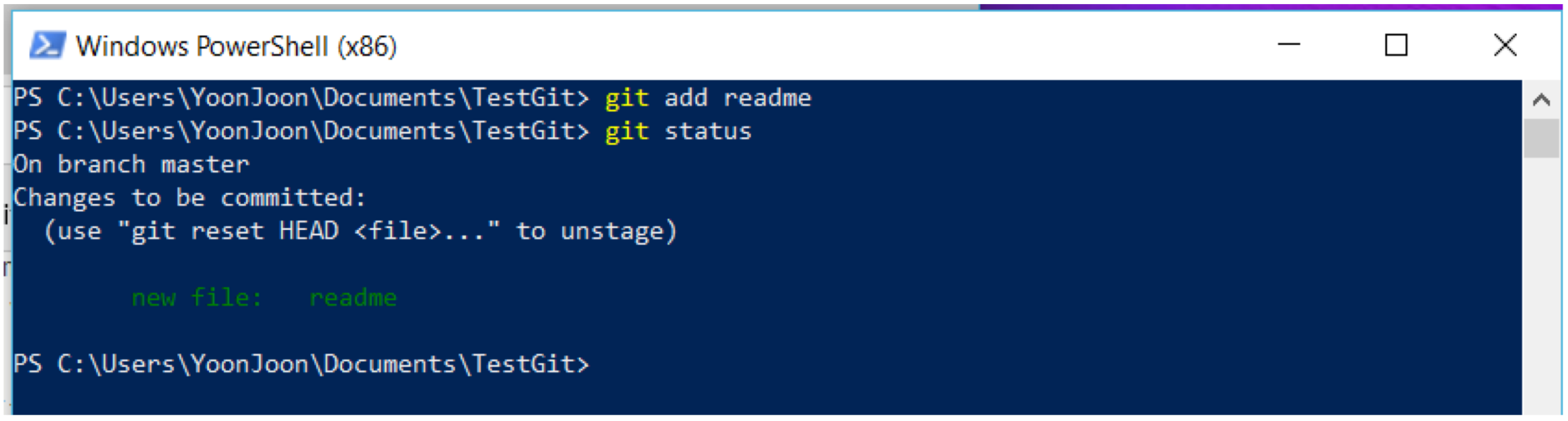
We add a new file to our project, a simple **`readme`** file. If the file didn't exist before, and you run **`git status`**:



We can see that your new **`readme`** file is untracked, because it's under the "Untracked files" heading in your status output. Untracked basically means that Git sees a file you didn't have in the previous snapshot (commit);

Tracking New Files

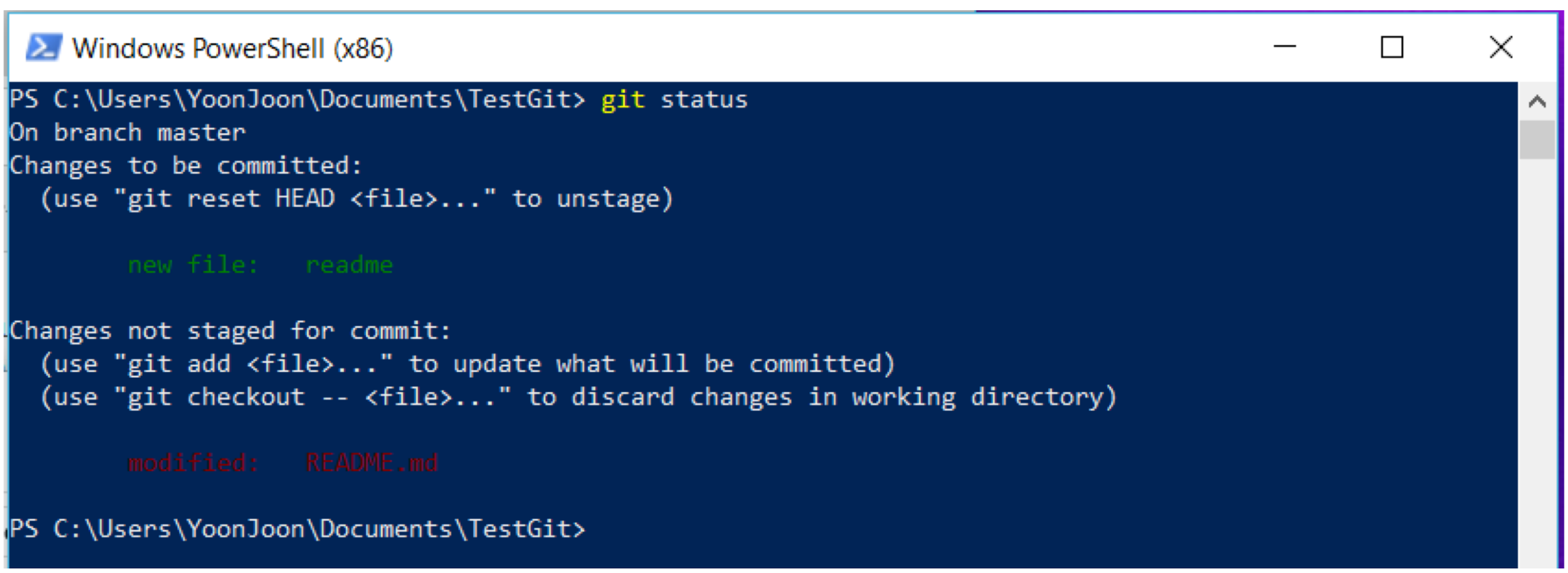
In order to begin tracking a new file, we use the command `git add`.



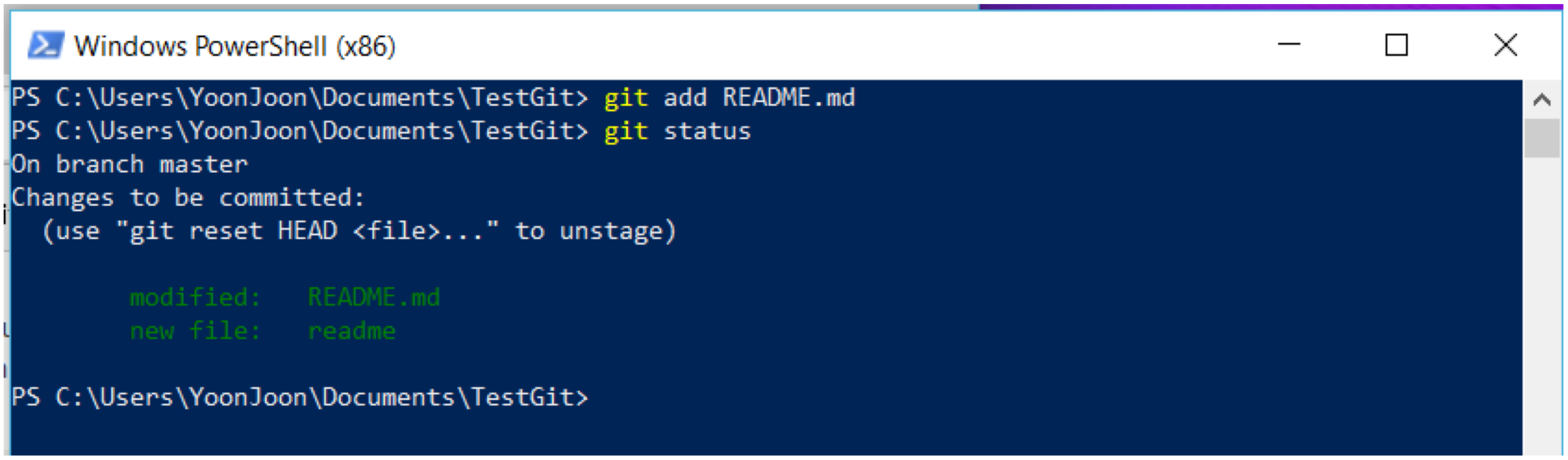
We can tell that it's staged because it's under the "Changes to be committed" heading.

Staging Modified Files

Let's change a file that was already tracked. If we change a previously tracked file called `README.md` and then run our `git status` command again,



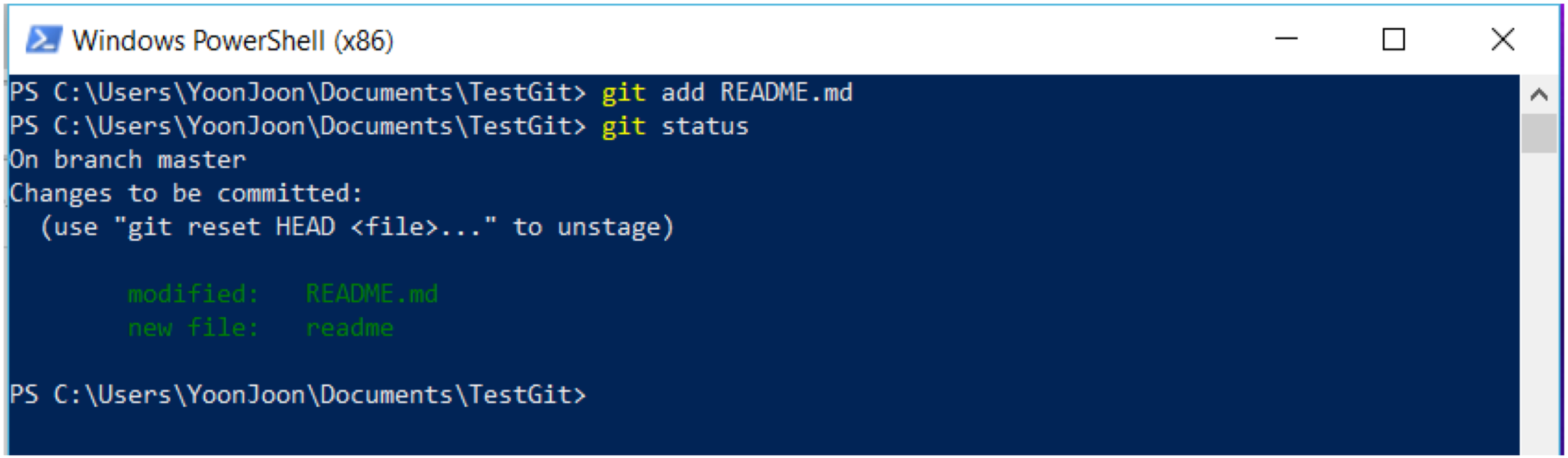
Let's run `git add` now to stage the `README.md` file, and then run `git status` again:

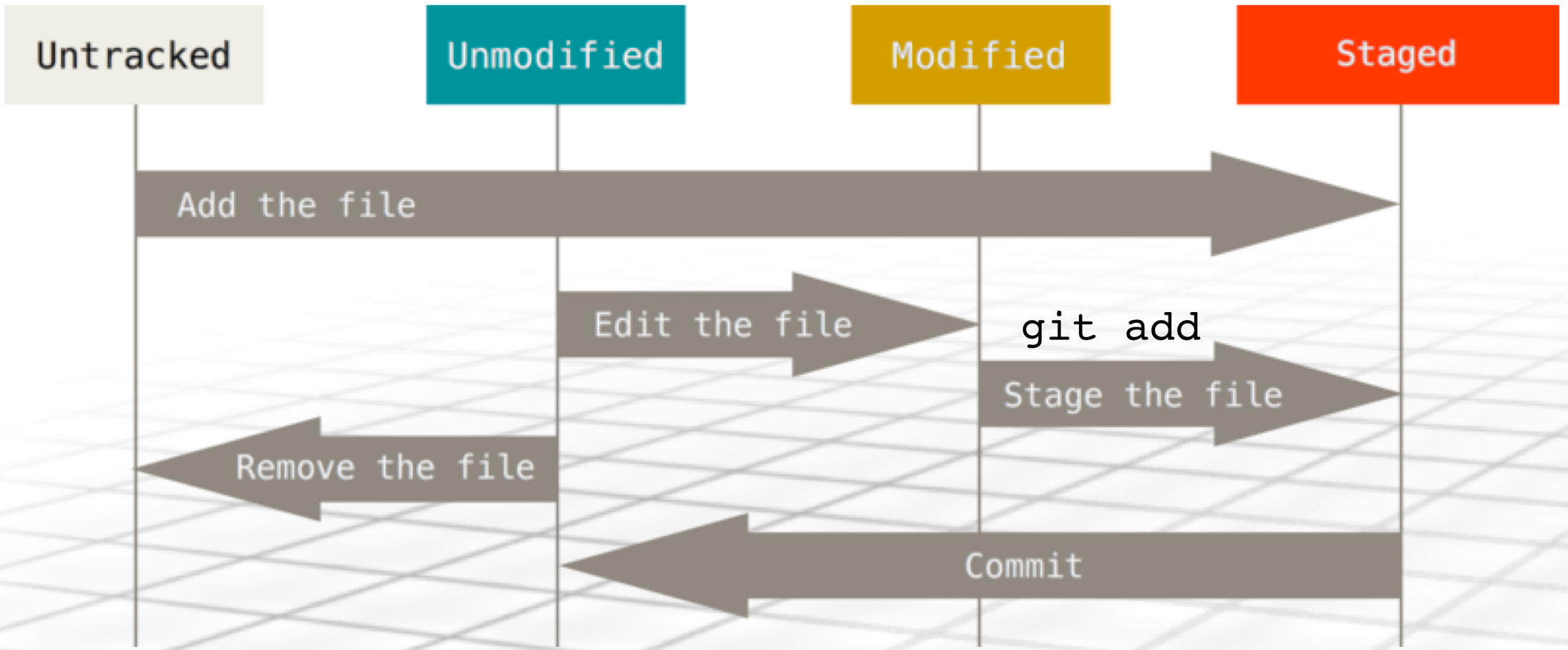


We want to make change in **`README.md`** before we commit it. We open it again and make that change, and we're ready to commit. However, let's run **`git status`** one more time:

```
Windows PowerShell (x86)
PS C:\Users\YoonJoon\Documents\TestGit> git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md
        new file:   readme

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:   README.md

PS C:\Users\YoonJoon\Documents\TestGit>
```

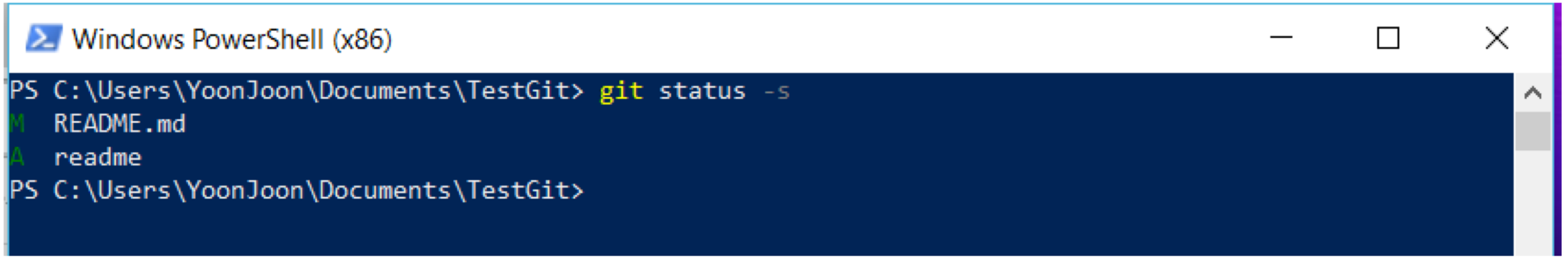
If we modify a file after we run `git add`, we have to run `git add` again to stage the latest version of the file:




Untracked
Unmodified
Modified
Staged
Add the file
Edit the file
git add
Stage the file
Remove the file
Commit

Short Status

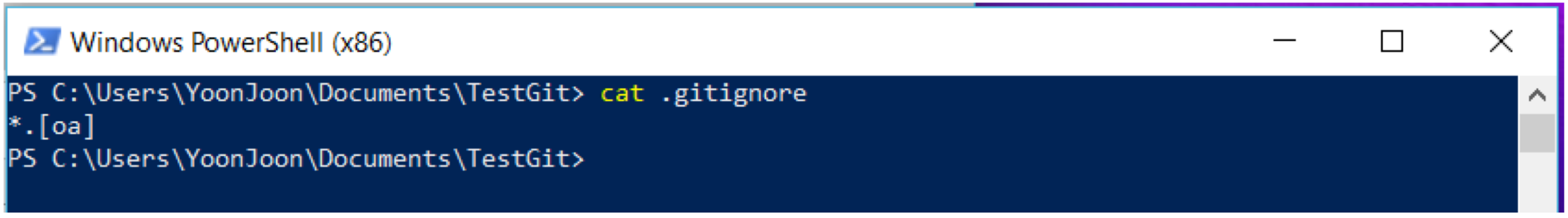
Git also has a short status flag so we can see our changes in a more compact way. If we run `git status -s` or `git status –short`.



New files that have been added to the staging area have an `A`, modified files have an `M` and so on.

Ignoring Files

Often, we'll have a class of files that we don't want Git to automatically add or even show we as being untracked. We can create a file listing patterns to match them named `.gitignore`.



The first line tells Git to ignore any files ending in ".o" or ".a" — object and archive files that may be the product of building our code.

The rules for the patterns we can put in the `.gitignore` file are as follows:

- Blank lines or lines starting with # are ignored.
- Standard glob patterns work, and will be applied recursively throughout the entire working tree.
- We can start patterns with a forward slash (`/`) to avoid recursivity.
- We can end patterns with a forward slash (`/`) to specify a directory.
- We can negate a pattern by starting it with an exclamation point (`!`).

Windows PowerShell (x86)

```
PS C:\Users\YoonJoon\Documents\TestGit> cat .gitignore
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf

PS C:\Users\YoonJoon\Documents\TestGit>
```

Viewing Your Staged and Unstaged Changes

We want to know exactly what we changed, not just which files were changed — we can use the `git diff` command.

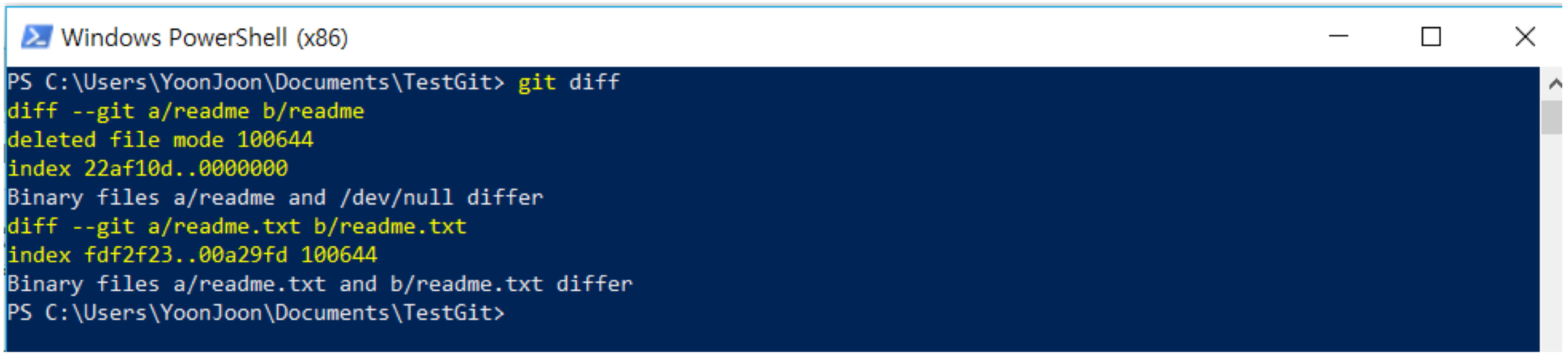
- What have you changed but not yet staged?
- And what have you staged that you are about to commit?

We edit and stage the **`README.md`** file again and then edit the **`reademe.txt`** file without staging it. If you run your **`git status`** command,

Windows PowerShell (x86)

```
PS C:\Users\YoonJoon\Documents\TestGit> git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md
        new file:   readme
        new file:   readme.txt

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:    readme
        modified:   readme.txt

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

        .gitignore
```

To see what we've changed but not yet staged, type `git diff` with no other arguments:

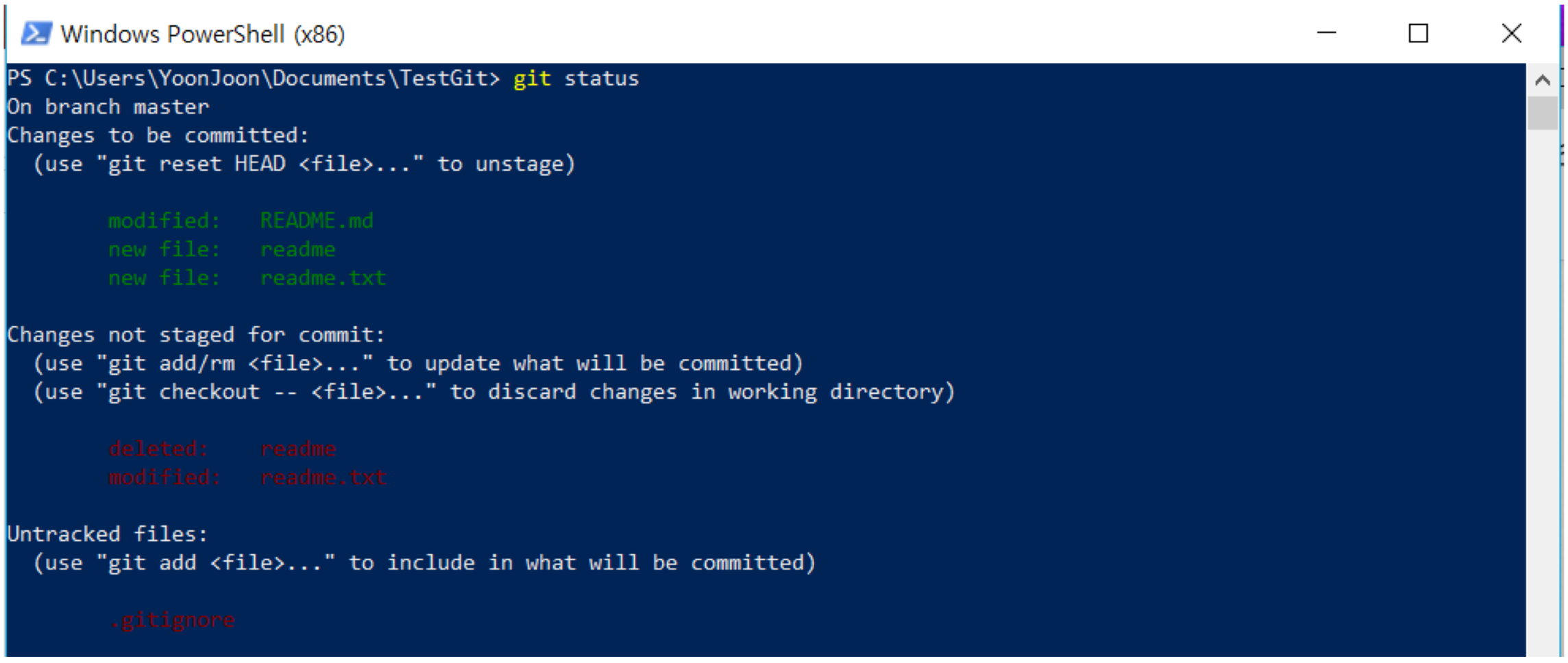


If you want to see what we've staged that will go into our next commit, we can use `git diff --staged`. This command compares our staged changes to your last commit:

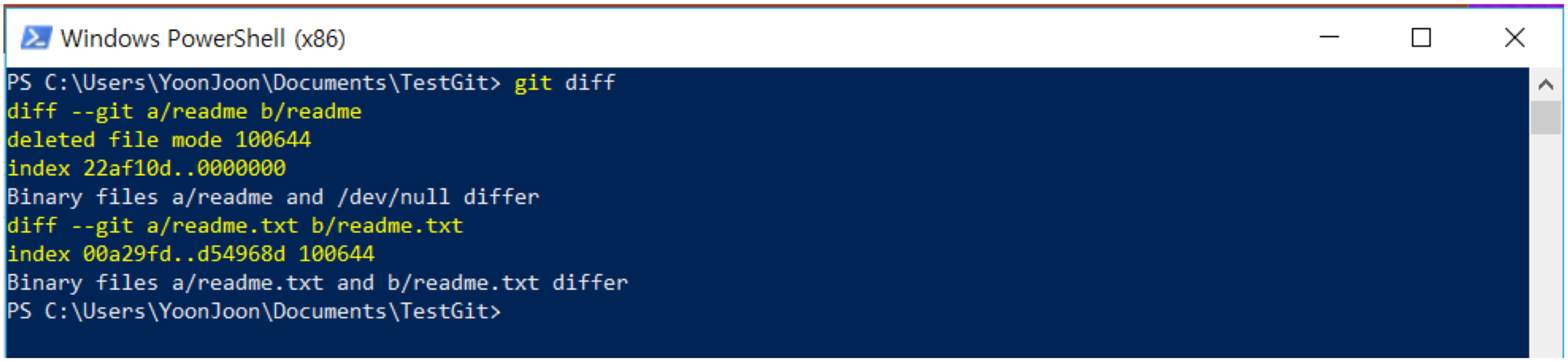
Windows PowerShell (x86)

```
PS C:\Users\YoonJoon\Documents\TestGit> git diff --staged
diff --git a/README.md b/README.md
index 58850fb..f6fd8a3 100644
--- a/README.md
+++ b/README.md
@@ -1,3 +1,8 @@
 # Git Test Text FIle

 a test file
+
+2nd line
+
+3rd line
+
diff --git a/readme b/readme
new file mode 100644
index 0000000..22af10d
Binary files /dev/null and b/readme differ
diff --git a/readme.txt b/readme.txt
new file mode 100644
index 0000000..fdf2f23
Binary files /dev/null and b/readme.txt differ
PS C:\Users\YoonJoon\Documents\TestGit>
```

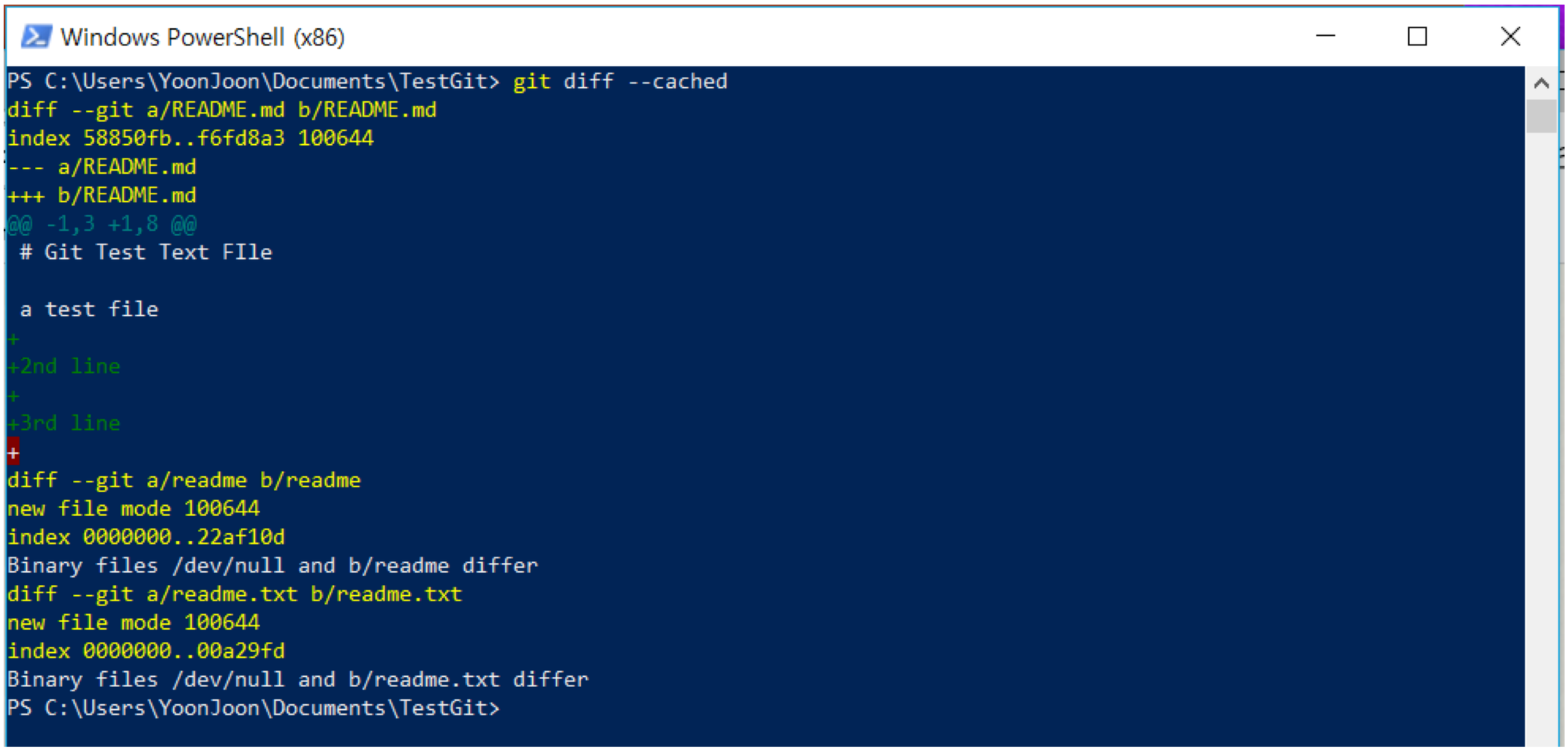
If we stage the **`readme.txt`** file and then edit it, you can use **`git diff`** to see the changes in the file that are staged and the changes that are unstaged.



`git diff` to see what is still unstaged:



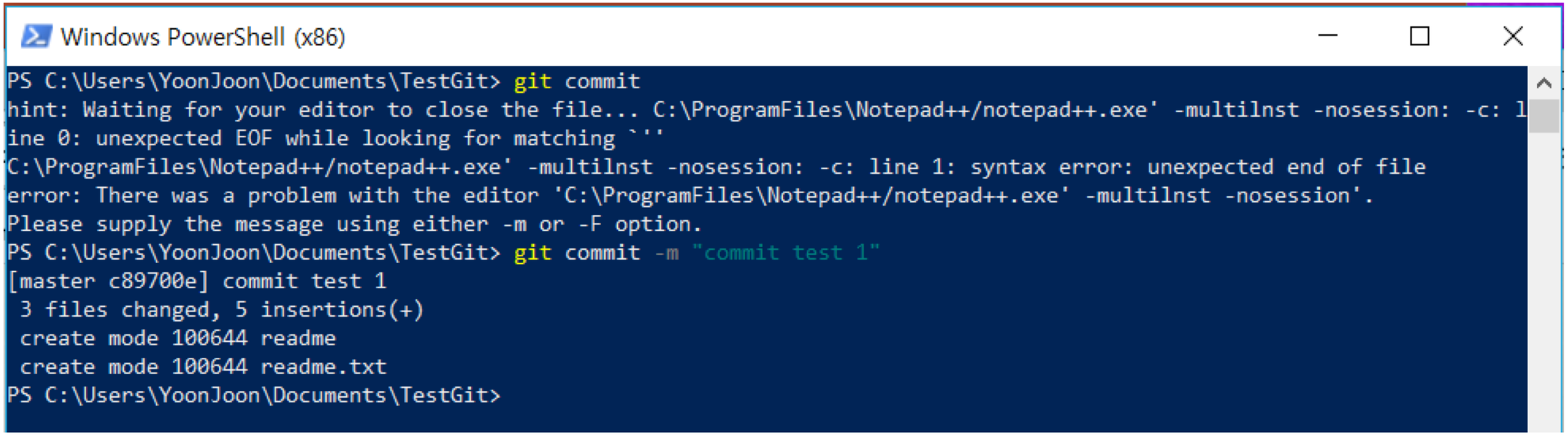
and `git diff --cached` to see what we've staged so far (`--staged` and `--cached` are synonyms):

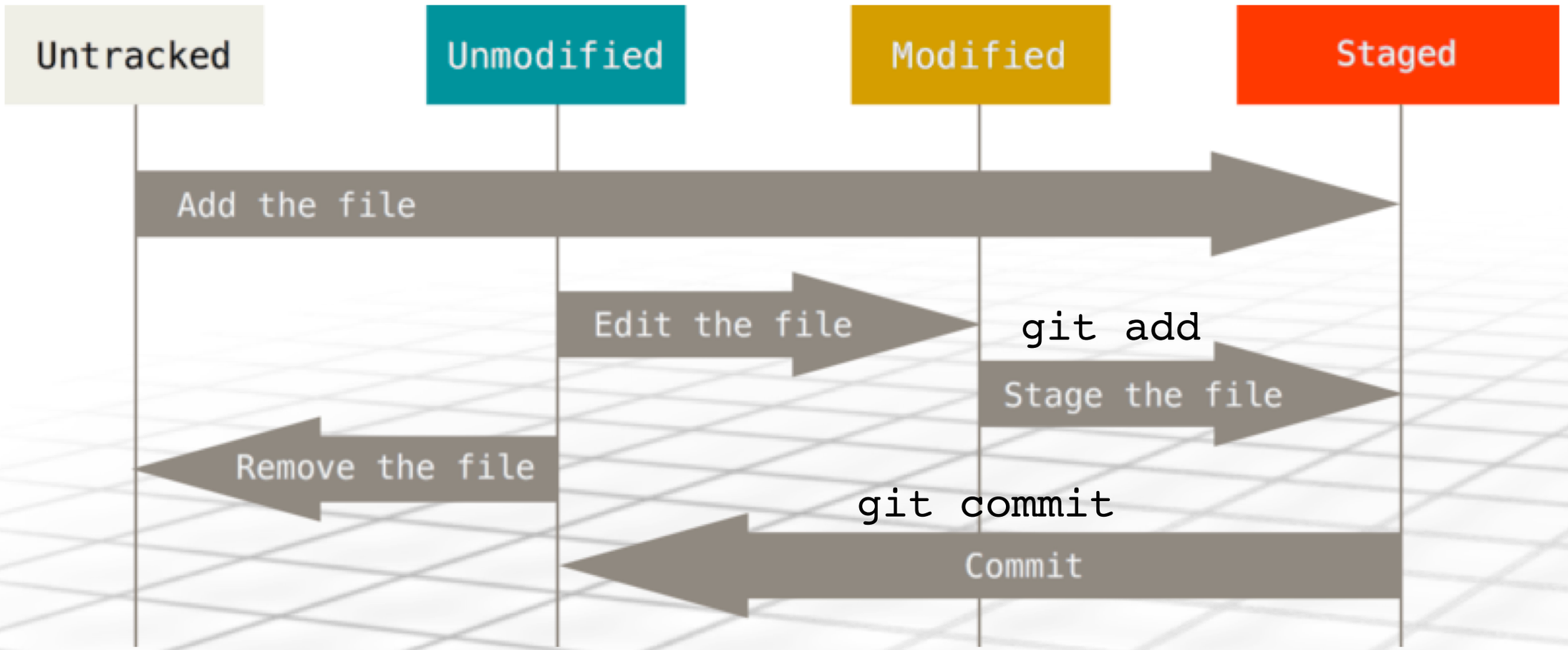


Committing Your Changes

Our staging area is set up the way you want it, you can commit your changes.

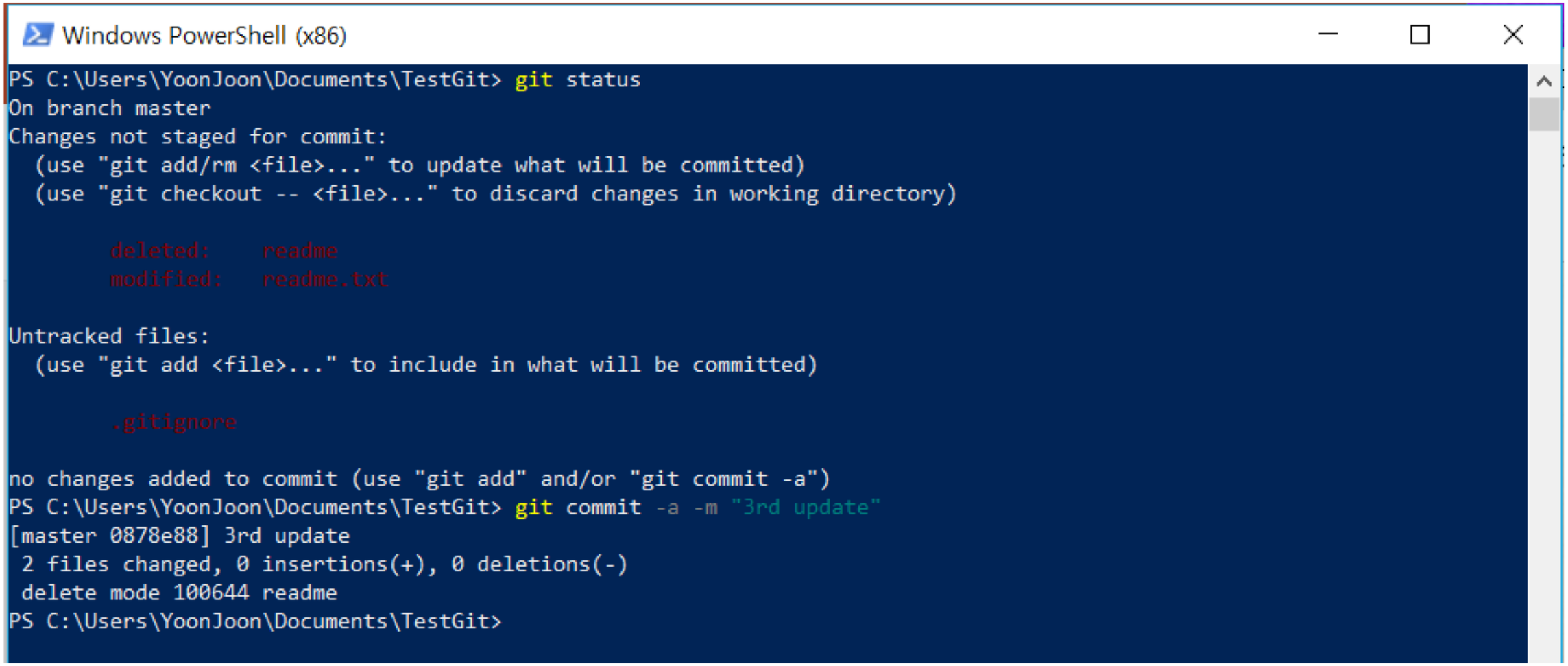
The simplest way to commit is to type `git commit:`




Untracked
Unmodified
Modified
Staged
Add the file
Edit the file
git add
Stage the file
Remove the file
git commit
Commit

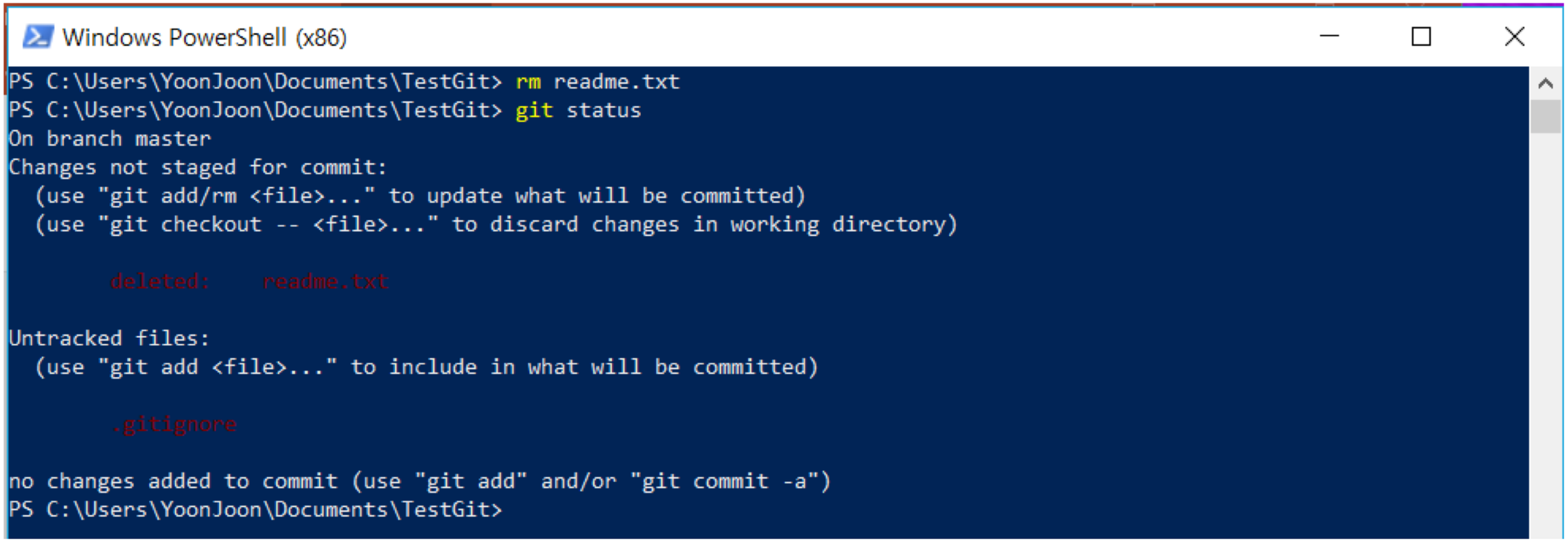
Skipping the Staging Area

Adding the `-a` option to the `git commit` command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the `git add` part:



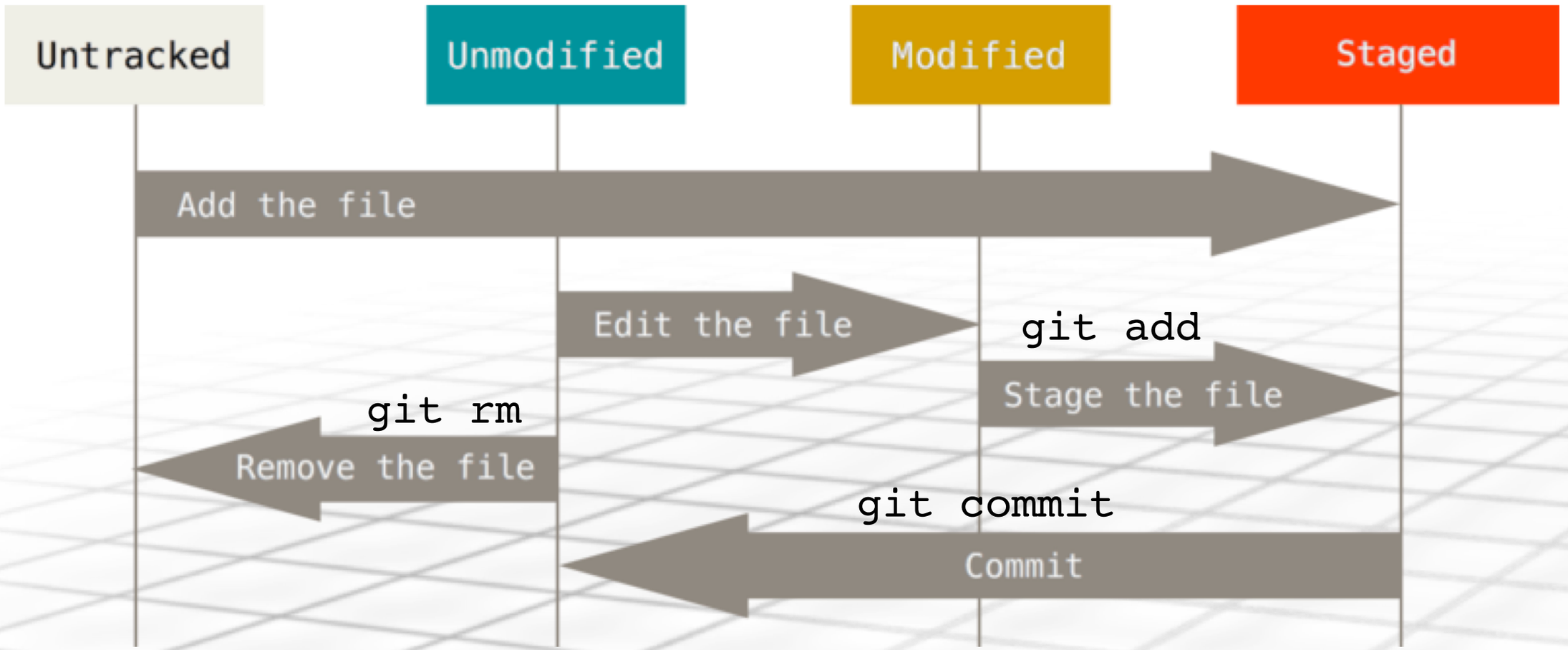
Removing Files

To remove a file from Git, we have to remove it from our tracked files (more accurately, remove it from your staging area) and then commit. The `git rm` command does that, and also removes the file from our working directory



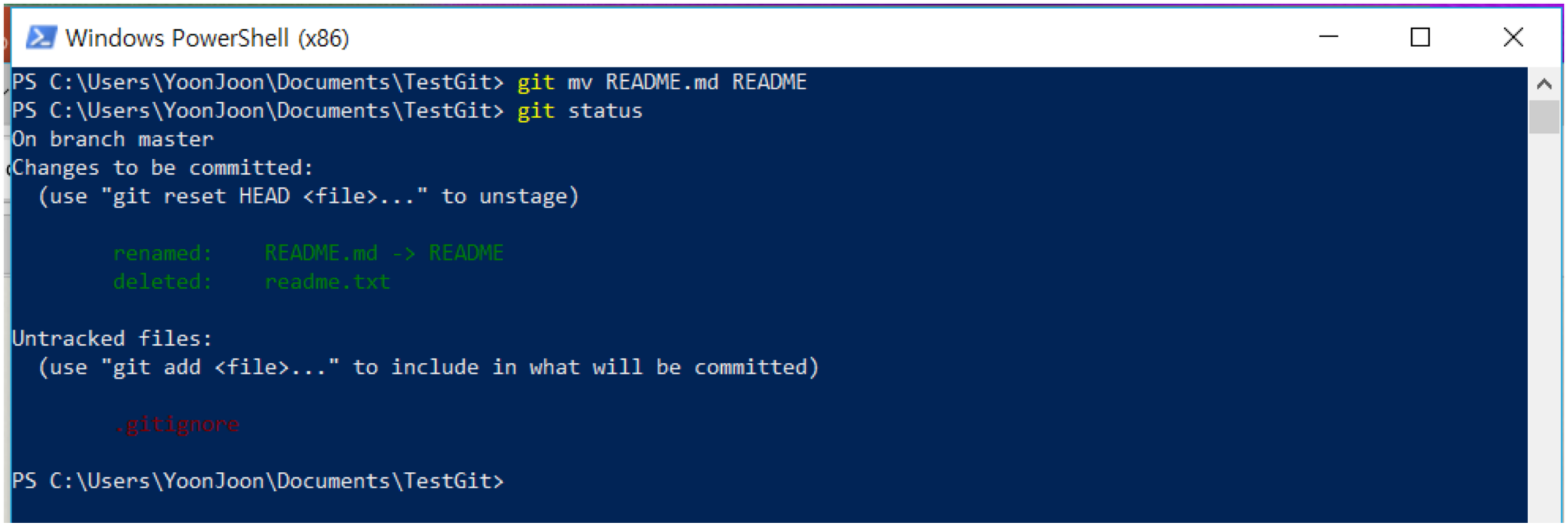
If we run `git rm`, it stages the file's removal:

```
PS C:\Users\YoonJoon\Documents\TestGit> git rm readme.txt
rm 'readme.txt'
PS C:\Users\YoonJoon\Documents\TestGit> git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    readme.txt

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

        .gitignore

PS C:\Users\YoonJoon\Documents\TestGit>
```

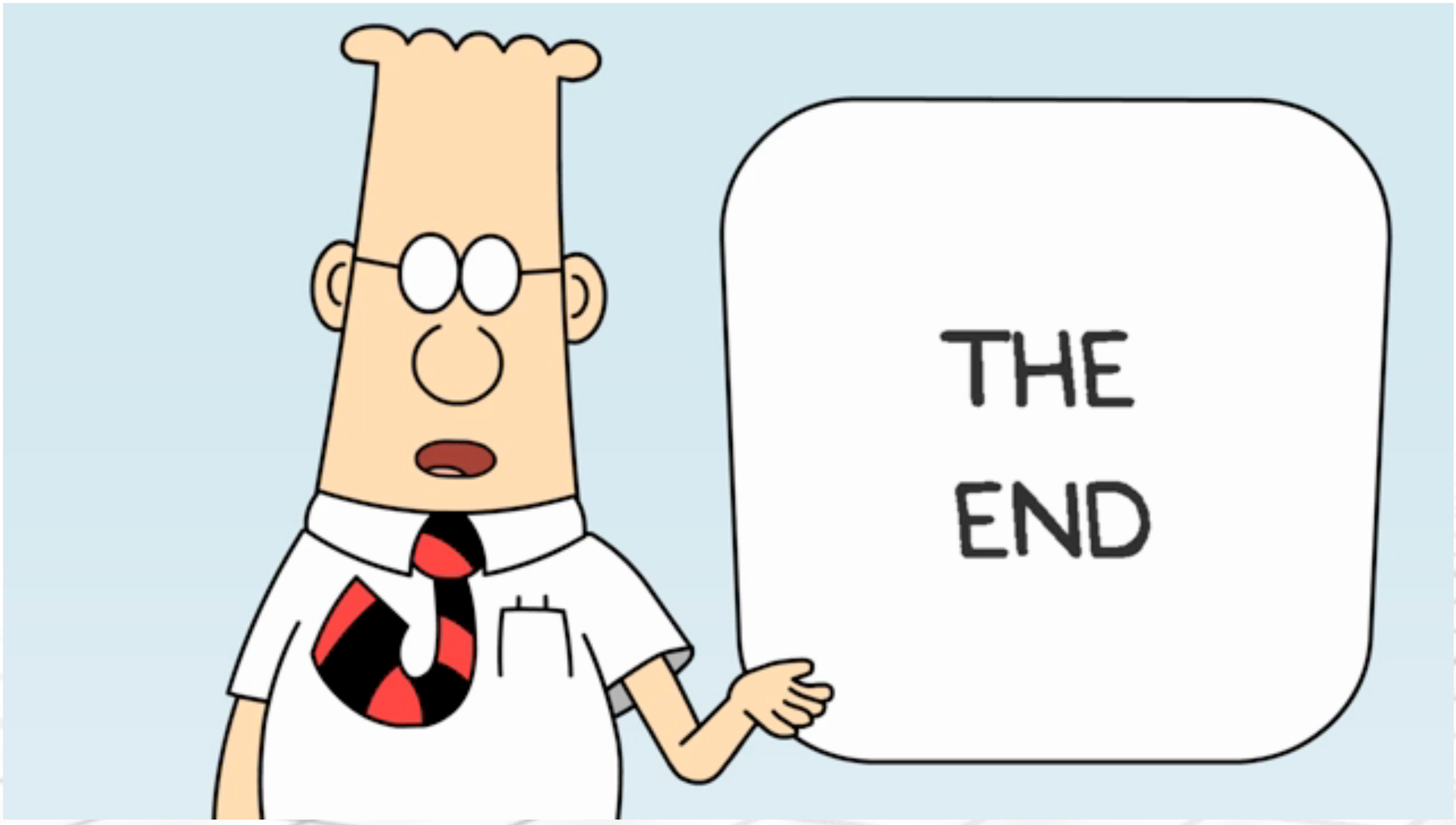

Untracked
Unmodified
Modified
Staged
Add the file
Edit the file
git add
Stage the file
git rm
Remove the file
git commit
Commit

Moving Files

If we want to rename a file in Git, we can run something like:




```
$ mv README.md README
$ git rm README.md
$ git add README
```



출처: metachannels.com

감사합니다