The Add Interactive Command

Learn how to achieve more granular control over having staged changes and unstaged changes.

We'll cover the following

The `git add -i` command

Let’s demonstrate how you might want to use this with a simple example:

```
mkdir lgthw_add_i
cd lgthw_add_i
git init
echo 'This is file1' > file1
echo 'This is file2' > file2
git add file1 file2
git commit -am 'files added'
cat > file1 << END
> Good change
> This is file1
> Experimental change
> END
cat > file2 << END
> All good
> This is file2
> END
```

Terminal 1

Terminal

Click to Connect...

Now run the following, and input the characters (or just “enter”/“return”) at the appropriate points:

```
git add -i
What now> p
Patch update>> 1
```

Note that in the next line, you only need to input the “enter”/“return” key. Just hit “return”.

```
Patch update>>
Stage this hunk [y,n,q,a,d,/,s,e,?]? s
Stage this hunk [y,n,q,a,d,/,s,e,?]? y
Stage this hunk [y,n,q,a,d,/,s,e,?]? n
What now> q
git status
git diff
```

Now you have staged the good change, but have not lost the other changes you made. This gives you more granular control over the changes committed.

Terminal 1 + ⟳

Terminal

A lot went on in that last block of output so it’s worth reading over it carefully.

First, you were presented with a set of choices:

```
    1: status   2: update   3: revert   4: add untracked
    5: patch    6: diff     7: quit     8: help
```

Patch

We’re only going to focus on “patch” (number 5) here. There’s no point exhaustively listing all the choices and their meanings, as most of them are self-explanatory, and I did not feel the need to use the rest of them.

Hunk

As you have probably figured out already, a “hunk” is a contiguous (or nearly contiguous) section of a diff.

You’re presented with a set of numbered changes. There were files that had changes to them presented to you.

(/learn)

```
               staged     unstaged path
      1:    unchanged        +2/-0 file1
      2:    unchanged        +1/-0 file2
```

Staged

The first “staged” column tells you what has been staged so far.

Unstaged

The second “unstaged” column tells you how many lines have been added/removed. In the above example, two lines have been added and none removed.

Path

The third is the “path” of the file.

An asterisk (`*`) indicates that the option is the chosen one. So by hitting that number, followed by the enter/return key, you only need to further hit the enter/return key once to choose the first hunk. Then you are presented with the hunk itself and a bewildering series of options:

```
diff --git a/file1 b/file1
index 6a00e12..014f6e4 100644
--- a/file1
+++ b/file1
@@ -1 +1,3 @@
+Good change
This is file 1
+Experimental change
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

Splitting

At this point, you chose `s`, which stands for “split” the hunk.

If you’re not sure what’s going on, you can choose the `?` option here, which explains what the various options mean. If I’m stuck, I can never remember what they all do, so I depend heavily on `?`. Read through the options now, and make sure you understand them all.

Once split, you can “stage” the hunks one at a time by choosing `y` for “yes” when prompted.

If you are happy with the changes, you can commit all the changes you have made.

Why split hunks?

Why is this splitting useful? It’s most commonly used to avoid committing lines you might want to keep for local development but not persist in the repository history.

A typical example of this is printed debug lines that you don’t want to get to production but want to keep for local development. Another example might be chunks of notes that only make sense to you while you are developing.

Why stage at all?

Committing will commit all the changes you have staged. What is the point of staging then? It is to confirm that you want to commit *some* changes made locally, but not others. These changes are added to the *index* (as opposed to the repository).

Remember: index == staging == adding

By committing, it *commits* your staged changes to the local repository (as opposed to *adding*). After which, these commits can be pushed to remote repositories.

If (like me) you run `git commit -am "your commit message"` frequently, then you skip over these steps, which can result in commits with stray lines that you would not want to be part of the history.

The `-a` flag stands for “automatically add”. Confusingly, the `-a` flag is aliased to `--all`, even though not all files are necessarily added; only the already-added ones are.

← Back

Introduction: Git Add Interactive

Next →

Introduction: Reflog

☑ Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/the-add-interactive-command__git-add-interactive__learn-git-the-hard-way)