Lab: Managing Your Worktree

Ignoring Files

What if we have files that we do not want Git to track for us, like backup files created by our editor or intermediate files created during data analysis? Let's create a few dummy files:

```
$ cd ~/work/planets
$ mkdir results
$ touch a.dat b.dat c.dat results/a.out results/b.out
```

and see what Git says:

```
$ git status
```

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

    a.dat
    b.dat
    c.dat
    results/

nothing added to commit but untracked files present (use "git add" to track)
```

Putting these files under version control would be a waste of disk space. What's worse, having them all listed could distract us from changes that actually matter, so let's tell Git to ignore them.

We do this by creating a file in the root directory of our project called `.gitignore`:

```
$ nano .gitignore
$ cat .gitignore
```

```
*.dat
results/
```

These patterns tell Git to ignore any file whose name ends in `.dat` and everything in the `results` directory. (If any of these files were already being tracked, Git would continue to track them.)

Once we have created this file, the output of `git status` is much cleaner:

```
$ git status
```

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

    .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

The only thing Git notices now is the newly-created `.gitignore` file. You might think we wouldn't want to track it, but everyone we're sharing our repository with will probably want to ignore the same things that we're ignoring. Let's add and commit `.gitignore`:

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

```
On branch main
nothing to commit, working tree clean
```

As a bonus, using `.gitignore` helps us avoid accidentally adding files to the repository that we don't want to track:

```
$ git add a.dat
```

```
The following paths are ignored by one of your .gitignore files:
a.dat
Use -f if you really want to add them.
```

If we really want to override our ignore settings, we can use `git add -f` to force Git to add something. For example, `git add -f a.dat`. We can also always see the status of ignored files if we want:

```
$ git status --ignored
```

```
On branch main
Ignored files:
 (use "git add -f <file>..." to include in what will be committed)

        a.dat
        b.dat
        c.dat
        results/

nothing to commit, working tree clean
```

> ## *Ignoring Nested Files*
>
> *Given a directory structure that looks like:*
>
> ```
> results/data
> results/plots
> ```
>
> *How would you ignore only `results/plots` and not `results/data` ?*
>
> > ## *Solution*
> >
> > *If you only want to ignore the contents of `results/plots`, you can change your `.gitignore` to ignore only the `/plots/` subfolder by adding the following line to your .gitignore:*
> >
> > ```
> > results/plots/
> > ```

This line will ensure only the contents of `results/plots` is ignored, and not the contents of `results/data`.

As with most programming issues, there are a few alternative ways that one may ensure this ignore rule is followed. The "Ignoring Nested Files: Variation" exercise has a slightly different directory structure that presents an alternative solution. Further, the discussion page has more detail on ignore rules.

Including Specific Files

How would you ignore all `.dat` files in your root directory except for `final.dat`? Hint: Find out what `!` (the exclamation point operator) does

Solution

You would add the following two lines to your .gitignore:

```
*.dat           # ignore all data files
!final.dat      # except final.data
```

The exclamation point operator will include a previously excluded entry.

Note also that because you've previously committed `.dat` files in this lesson they will not be ignored with this new rule. Only future additions of `.dat` files added to the root directory will be ignored.

Ignoring Nested Files: Variation

Given a directory structure that looks similar to the earlier Nested Files exercise, but with a slightly different directory structure:

```
results/data
results/images
results/plots
results/analysis
```

How would you ignore all of the contents in the results folder, but not `results/data`?

Hint: think a bit about how you created an exception with the `!` operator before.

Solution

If you want to ignore the contents of `results/` but not those of `results/data/`, you can change your `.gitignore` to ignore the contents of results folder, but create an exception for the contents of the `results/data` subfolder. Your .gitignore would look like this:

```
results/*               # ignore everything in results folder
!results/data/          # do not ignore results/data/ contents
```

Ignoring all data Files in a Directory

Assuming you have an empty .gitignore file, and given a directory structure that looks like:

```
results/data/position/gps/a.dat
results/data/position/gps/b.dat
results/data/position/gps/c.dat
results/data/position/gps/info.txt
results/plots
```

What's the shortest `.gitignore` rule you could write to ignore all `.dat` files in `result/data/position/gps`? Do not ignore the `info.txt`.

Solution

Appending `results/data/position/gps/*.dat` will match every file in `results/data/position/gps` that ends with `.dat`. The file `results/data/position/gps/info.txt` will not be ignored.

Ignoring all data Files in the repository

Let us assume you have many `.dat` files in different subdirectories of your repository. For example, you might have:

```
results/a.dat
data/experiment_1/b.dat
data/experiment_2/c.dat
data/experiment_2/variation_1/d.dat
```

How do you ignore all the `.dat` files, without explicitly listing the names of the corresponding folders?

Solution

In the `.gitignore` file, write:

```
**/*.dat
```

This will ignore all the `.dat` files, regardless of their position in the directory tree. You can still include some specific exception with the exclamation point operator.

The Order of Rules

Given a `.gitignore` file with the following contents:

```
*.dat
!*.dat
```

What will be the result?

Solution

The `!` modifier will negate an entry from a previously defined ignore pattern. Because the `!*.dat` entry negates all of the previous `.dat` files in the `.gitignore`, none of them will be ignored, and all `.dat` files will be tracked.

> ## *Log Files*
>
> *You wrote a script that creates many intermediate log-files of the form `log_01`, `log_02`, `log_03`, etc. You want to keep them but you do not want to track them through `git`.*
>
> 1. *Write **one** `.gitignore` entry that excludes files of the form `log_01`, `log_02`, etc.*
> 2. *Test your "ignore pattern" by creating some dummy files of the form `log_01`, etc.*
> 3. *You find that the file `log_01` is very important after all, add it to the tracked files without changing the `.gitignore` again.*
> 4. *Discuss with your neighbor what other types of files could reside in your directory that you do not want to track and thus would exclude via `.gitignore`.*
>
> > ## *Solution*
> >
> > 1. *append either `log_*` or `log*` as a new entry in your .gitignore*
> > 2. *track `log_01` using `git add -f log_01`*

Mastering Git Stash Workflow

Git is a powerful tool that makes up for a lot of use cases on our development workflow. One such case is to isolate the changes of a certain branch to itself. Let me elaborate.

Suppose you are working on a branch called `admin-dashboard` implementing ,well, an administrative dashboard. But you are not done yet, and the project manager wants a quickfix for the login implementation. Now you want to switch to the `login` branch and fix the issue but don't want to carry the changes you are doing on the `admin-dashboard` branch. Well this is where git stash comes in.

Git stash what?

Git stash allows you to quickly put aside your modified changes to a LIFO (Last In First Out) stack and re-apply them when feasible. We will be doing a rough walkthrough to see this in action.

Git stash walkthrough

Initiate git on an empty directory. Add a file named `add.py` and put the following code.

```
# add.py

def add(a, b):
    return a + b
```

Let's add the file to git and commit it .

```
git add add.py && git commit -m "Add function"
```

Next let's create and checkout to a new branch called `mul` and create a file called `mul.py` \

```
git checkout -b mul
```

Add the following code to the file.

```
# mul.py

def mul(a, b):
    return a * b
```

Add the file to git and commit it.

```
git add mul.py && git commit -m "Mul function"
```

Now suppose, we need to update the mul function to take in a third argument and you just edited the function like so:\

```
# mul.py

def mul(a, b, c):
    return a * b
```

Take note that, we still haven't updated the return value with `c`. While we were making our changes, the project manager called-in to update the `add` function immediately with a third argument. Now you can't waste a second on the `mul` function which you haven't completed. **What are you going to do?**

If you try to checkout to master where add function resides, git won't let you because you have unfinished changes that hasn't been committed.

Stashing Changes

Well this is the situation you should be using the `git stash` command. We want to put away the changes on our current branch so that we can come back to it later.

Stash the file with a message on the `mul` branch.

```
git stash save "Multiply function"
```

Now if you `git status`, the working directory will be clean and you can jump into the master branch for the changes. We can view the items in our stash using `git stash list`. We can view the diff of the items on our stack using `git stash show`

Now you are in `master` branch modifying the `add` function, and comes an even higher priority job to include a `subtraction` function.

> ***Note***:
> *Project Managers in real life doesn't put forward tasks on this manner.*

Your `add` function looks pretty much like the `mul` function now.

```
# add.py

def add(a, b, c):
    return a + b   # couldn't include `c` due to a priority task.
```

Stash it with a message.

```
git stash save "Add function third argument"
```

Now let's suppose we have completed our task for `subtraction` branch and we want to continue working on other branches.

Let's move to the `master` branch first to complete our changes for addition.

There are two ways we can re-apply those changes:

1. `git stash pop`
   applies the top most change stored on the stack and removes it from the stack.

2. `git stash apply <item-id>`
   applies the stash based on the index provided, keeps the applied item intact on the stack.

Popping stashed changes 🍾

Like I mentioned earlier, stash follows the LIFO convention. The latest item we save are always on the top. And when we use `pop`, always the top most change is applied to the current branch. Run the following command on `master`.

```
git stash pop
```

Complete the `add` function and commit it.

Applying stashed changes 📥

Next, checkout `mul` branch. We can use `pop` here as well since there is only one item remaining on the stack. But let's see how `git stash apply` works.

```
git stash apply stash@{0}
```

When we apply from the stash, the item still remains on the stack. Complete the changes here and commit it.

Create a new branch with stashed changes

Let's do something fun. Let's say we want to add a `divide` function on a new branch. Well it is kind of similar to our `multiple` function so why not utilize the item in stash and create a `divide` function with it?

We can do that with the `git stash branch` command. It takes the `<item-id>` and a branch name, then ***applies*** those changes to that branch.

```
git stash branch <branch-name> <item-id>

git stash branch divide stash@{0}
```

Now we can rename the file and change the function to perform division.

Viewing the stashed changes

Sometimes we tend to forget what changes we stashed. What we forget can range from which files were stashed to what changes on the files were stashed.

To view a list of files that were stashed, we can run:\

```
# EXAMPLE:
# git stash show stash@{<stash-id>}

git stash show stash@{2}
```

To get a diff view of changes on the files that were stashed, we can run:\

```
# EXAMPLE:
# git stash show -p stash@{<stash-id>}

git stash show -p stash@{2}
```

Clearing the stack 🖌

Now that the stash has served its purpose, we can clear it. There is again two ways of doing it:

1. `git stash clear` wipes the whole stack clean.

2. `git stash drop <item-id>` removes the item from stack based on provided id.

Managing Git Worktree

Git worktree helps you manage multiple working trees attached to the same repository.

> *In short, you can check out multiple branches at the same time by maintaining multiple clones of the same repository.*

OK back to our problem! Update changes? New Feature? Hot Fix? Whatever it is, you need to change to a different branch and work on it without any changes to your current work directory.

Let's say it's a new feature, your workflow would look like this:

1. create an replica of your project and switch to a new branch
2. create a new feature
3. push it
4. back to previous working directory

Create worktree

Let's say the name of your feature is `feature-x` and you want the branch with the same name. You can create additional worktree on the same directory or move it to a desired path, I prefer the later.

`git worktree add` command creates a worktree along with a branch that is named after the final word in your path.

```
git worktree add <PATH>

# Create feature-x directory and branch with the same name.
git worktree add ../feature-x
```

Named Branch

If you want to give you branch a unique name then you can use the `-b` flag with the `add` command.

```
git worktree add -b feature-xyz ../feature-xyz
```

List Worktrees

View the list of worktrees with `git worktree list`

Remove Worktrees

Now that you have created a new worktree, switched to it and made your changes and pushed it. To remove the worktree, we can run:

```
git worktree remove <name-of-worktree>

git worktree remove feature-x
```

Conclusion

Git worktree is a handy feature that let's you context switch in your project to try out things on a completely different environment, without modifying your main work directory.