Branching With Git

By this point, you have seen many of the basics of Git and GitHub. You know how to fork a repostory on GitHub, clone it locally on your machine, commit changes, and push those changes back up to GitHub. Throughout this work, your code has progressed along a single path. You have always committed changes to your master branch, whether in your local repository or on GitHub. This keeps things simple, but it is also limiting. When your code gets more complicated, or when you work with others, you will need to make multiple versions of your code with different changes in each one. This is where branching comes in.

A branch is just what it sounds like: a copy of your code that you can change without affecting other branches. You can switch back and forth between branches without changing any code; branches do not affect one another until they are explicitly merged.

Let's begin with a simple example. Imagine that you want to try a new version or feature but you do not want to merge your changes into the master branch right away. You might want to see if the feature is any good, you might want to make sure that other people can continue to merge branches into the master branch, or you may just want to observe good coding practices. This is a perfect time to create a new branch.

You have seen that we are on a specific branch when we have run `git status` in the past.

Go ahead and run it in our repository again.

```
git status
On branch master
nothing to commit, working directory clean
```

We can see that we are on the master branch, which is typically the main branch or the "trunk" of your (and most everyone's) repositories. Since we want to start with a feature that we are not sure will be any good, let's create a new branch.

```
git checkout -b super_cool_feature
Switched to a new branch 'super_cool_feature'
```

We can see that we have now switched to a new branch. The `-b` flag means that we want to checkout a new branch that we called super_cool_feature.

Now let's start making these super cool changes. It is important to remember that this branch is completely separate from our other branches; any code, changes, or modifications to this branch are completely isolated from the other branches. We will see exactly what that means in a moment.

Open `hello.py` in your favorite editor. Instead of just having it print some simple statements, let's make it a bit more interactive. Erase what we have and let's put it. *Is "put it" accepted jargon"?*

```
import sys

print(sys.argv)
```

This file is pretty straightforward, and once we run it, you should be able to see what this is doing.

The `sys` package is imported, meaning that we can use some of the tools that it provides. We take advantage of the `argv` part of the `sys` module using the dot syntax. This allows us to execute our Python script from the command line and pass in arguments.

```
python3 hello.py
# Should print ['hello.py']

python3 hello.py world
# Should print ['hello.py', 'world']
```

Do you intend for the two "should print" lines above (and the one below) to print as a top-level heading?

Are you getting a sense for what the `argv` does?

It gives us access to the arguments sent to `python3` in a list format. Now let's look at the real power of what we just did.

```
python3 hello.py "I'm learning python!"
# Should print ['hello.py', 'I'm learning python!']
```

We do not need to explicitly write our messages as we did in the master branch; we can just pass them in as arguments to our program.

Now let's commit the changes to the current branch that we are working on. Note that we are on the super_cool_feature branch.

```
git status
On branch super_cool_feature
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:   hello.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Now we add it and commit it.

```
git add hello.py
git commit -m "Now parses command line arguments"
[super_cool_feature 53c3a1a] Now parses command line arguments
 1 file changed, 5 insertions(+)
```

The pattern of status, adding, and committing is extremely common. You will find yourself doing this all the time.

—-COULD BE UNNECSSARY—-

Let's make some more changes; we will make things a bit more interactive now as well. First make a new file called calc.py and add in the code below.

```
import sys

args = sys.argv
f = args[0]
func = args[1]
nums = args[2:]

print(f)
print(func)
print(nums)
```

You can see that we have made our argument list a bit more complicated now.
—-ENDCOULD BE UNNECSSARY—-

Now that we have committed it, let's check our status. We are still in the super_cool_feature branch.

```
git status
On branch super_cool_feature
nothing to commit, working directory clean
```

We can run another command to see what other branches we have.

```
git branch
* super_cool_feature
  master
```

We can see that we have a master branch as well as our super_cool_feature branch. Let's merge our new feature back into the master branch. This is a formalized way of integrating one branch into another one. First we will checkout the master branch.

```
git checkout master
Switched to branch 'master'
```

Now we are back on the master branch if we run the following.

```
python3 hello.py
```

We will see our unmodified version of hello.py (the one without the feature that we added). Now let's merge the two branches.

```
git merge super_cool_feature
Updating 4407308..53c3a1a
Fast-forward
 hello.py | 5 +++++
 1 file changed, 5 insertions(+)
```

```
python3 hello.py "This is my brand new feature that I just merged."
# Should print ['hello.py', 'This is my brand new feature that I just merged.']
```

Your output may vary slightly, but the general format should be the same.

We just took a branch that we were working on completely separately from another branch and merged it into another one. This format comes up frequently on projects when large or small teams are working on lots of different features, each having their own branch. Eventually it will come time to merge the branches into another branch.

Conclusion

You have now covered the basics of Git and how Git works. You have created repositories on your own computer, you have forked them from other repositories, you have pushed to other ones, and you have understood how you would go about creating a pull request to work with others on projects.

For the purposes of this course, you have gotten an excellent introduction to Git and the GitHub system. Don't worry if you do not understand everything in perfect detail yet. As you gain more experience with GitHub, it will become more and more intuitive.

As a final word of advice, remember that any problem you will want to solve with Git has been solved before. When you try to fix a problem, simply search for the answers on the Internet using Git vocabulary. For example:

- *How do you rollback a commit?*

- *How do you edit a git commit message?*

Those are questions using the contextual vocabulary. This is something that will help you again and again in your technical and data science careers. It is essential to ask a technical question by framing it using the vocabulary of the language or framework in which you are trying to solve the problem.

References:

- Git Website
- Git Cheatsheet
- Better Explained