Python Chases Monkey

An Introduction to Python Programming

Jason M. Pittman

Appendix

Python Gits Good

Introduction

done Learning how to leverage version control is almost as important as learning how to program. The two go hand in hand with an added bonus that version control is useful for more than our program source code. Moreover, once you are comfortable with version control, using it becomes no more of a burden than manually saving a word processing document as you work.

As I mentioned, version control has uses beyond programming. We can basically put any file object we want into version control. That, however, assumes we understand what version control is and how to use it. Well, that's why we're having this discussion. That and I couldn't resist the meme potential.

Get it, Python gits good?

Version control philosophy

done I'd like to explain why I think version control is important before we start exploring Git. Put simply, I see version control (Git specifically) as an antidote to two things which make managing file objects harder than what you'd imagine: changes and time.

Based on that, I think the secret to learning how to effectively use version control is to think about file objects, any object, and how change over time affects it. More specifically, think about being in the future and wanting to revisit an older version. Maybe we want to do this because we need to see how we did something. Maybe we want to do this because a *new* change erased an important statement. The reasons are as endless as the variety of objects we can track with version control.

We can get even more specific (i.e., technical) and say that version control is a way to future-proof our data's integrity.

Well, what's integrity? The simple explanation is integrity is a measure of an object's state. We can perform the measure at different points in time and, unless the object has changed states, the measure will produce the same output as before. The output is commonly referred to as a *checksum* or *fingerprint*.

There have been a variety of version control systems over the years (SVN, Visual Source Safe, etc.) but Git is the dominant system in modern software by a huge margin. As new programmers, we shouldn't expect to master Git but knowing what Git is and how to use basic features will serve us well. With that said, let's get started.

Git, good

done Just like with Python, we have to start somewhere and that somewhere is installing Git. Fortunately, there is a standard build for the major operating systems. Further, take heart that the processes and git commands we're going to review are identical across operating systems. Personally, I love when software works that way.

Speaking of work, if we want to get good at Git we need to get some work done. First up, installing the applicaiton.

Installing

done We can grab the latest build from `https://git-scm.com/`. Run the installation and take the defaults unless you are absolutely certain you need something non-standard.

A little forewarning: the default installation assumes we'll be using a command shell of some sort. There are a myriad of Git graphical user interfaces and every GUI I've seen follows normal Git command names fortunately. However, I suggest the most straightfoward way to develop our Git knowledge is through command line. For one thing, our learning will be unencumbered by extra friction such as figuring out menus, finding commands, and so forth. Further, chances are we'll have a shell or terminal running already since our language selection demands it.

After installation, we can execute `git help` from command line to test our implementation. What is the first command in the ouput?

Next, we have two post-installation steps to finish. Read through following commands before moving forward.

```
(1) git config --global user.name "my name is"
(2) git config --global user.email "myemail@is.com"
```

These lines are global configurations which dictate the name and email attached to every *commit*. We don't know what a commit is yet but that's okay. For now, we can observe that (1) Git has its own configuration as an application and (b) how we use it has meaning outside of that configuration. We'll call back to this when we examine the `commit` command.

Let's verify by looking at the content of the global configuration file, `.gitconfig`. Confusingly, the file is user specific (two users on the same system do not share the same file). Where do you think the file is located?

Once we're done, let's move onto making a repository.

Making a repository

done The basic process to make a repository we should use is as follows:

Example 1

```
(1) mkdir [repository name]
(2) cd [repository name]
(3) git init
```

In case you're not familiar, the first command is how we create new directories at command line. The general standard is to use all lowercase with words separated by a `-`. For example, `python-chases-monkey`.

Given the above, we can say that a repository is a directory that has been initialized for use with Git. The initialization process (the `git init` command) establishes three critical components for us: an list of file objects, a *log* of all changes over time to those objects, and a *pointer* of sorts to our current objects' state. This will make sense once we learn how to use our repository.

We can run `ls -laht` after line two in **Example 1**. Is there a hidden directory? If so, what is within any such directory?

Using a repository

done Naturally, we need to be able to use a repository for the *repo* to have value. We have two steps, two commands to digest.

Example 2

```
(1) git add [file object]
(2) git commit -m ["what changed"]
```

Here, the `add` action appends the indicated file object to an internal tracking mechanism. This is valuable because the tracking is aware of when a change, any change occurs to an added object. The `index` in our `.git` is responsible for holding the objects and objects' state information. That's the list I referred to earlier.

There are three ways to `add`: by specific file names separated by spaces, using the `.` to indicate *all* based on shell or terminal interpolation, or using `git add -A` which is the Git *all* flag.

A `commit` action is what creates a snapshot of our tracked (added) file objects. The idea is that we can incrementally capture changes over time. Knowing *how* this works is to explain the very essence of git. Further, it is absolutely critical to know that our changes are not part of the repository until we make a commit. The `add` only stages the changes.

In short, each commit places our current state into `HEAD` and places the prior commit into a history or log. The global `user.name` and `user.email` values we set earlier are attached to the commit at this point. This is how projects with multiple contributors can track who made which change.

By way of analogy, consider a vinyl record player. The player itself is the repository. The LP record is the file object we have added to verison control with each groove represent an object state or commit. The needle and armature are `HEAD`.

All of this is viewable at any time by using the `git status` command. If I'm being honest, I use this command more frequently than just about anything else in the git ecosystem. You can never know *too much*, right? I also use `git log` a fair amount. This comes in handy when we want to move between commits.

By the way, if you want to use a remote repository, I prefer `https://github.com` but you might like BitBucket or GitLab instead. The fact is you don't need a remote to take advantage of the benefits of Git. Nothing stops us from just using local backups if we're concerned about data loss. All of that is besides the point though and best left for another time.

Switching between commits

Okay, how to review commits and change between them is the last thing we need to know before we can confidently assert we are good with Git or at least good enough to have Git be helpful and not be a burden to our programming toolchain.

Remember our conversation about `HEAD` and functioning much like the needle on a record player? Well, if `commit` moves the `HEAD` forward or towards the center of the LP as we add commits then it stands to reason that we have a command to move backwards or forwards (if we've gone backwards previously). If this occurs every time we commit, then we can surmise there must be a *history* of commits as part of the repository.

Git provides two means to position `HEAD` where we want it: **absolute** and **relative**.

The **absolute** way to move to a commit is by referencing its SHA1 hash. This presumes we know the hash though. Perhaps we don't for some reason. Remember `git log`? There is a *commit* field with a long hexadecimal string... that's the SHA1 hash and the source of our integrity along with serving as a unique identifier.

Enter **relative** positioning and the ability to use references such as `HEAD~1` or `HEAD^3`. That's one commit back (child) relative to current or the third commit forward (parent) in the second case. This presupposes we know the relative positioning of the commit we want across the totoal repository history. Here, `git log` can reveal the information we need as well.

Here's the skinny: if I simply want to go back one or two commits, I use **relative**. Beyond that, I tend to loose mental track and prefer to rely on `git log` keeping track for me since that is the application's entire goal.

To actually move `HEAD`, we simply issue a `git checkout` command in conjunction with the SHA1 hash or relative position.

What do you think `git checkout HEAD^1` will do for us? What about `git checkout 23a64781`?

There it is, we are good at Git now. Remember, commit often and frequently. The cost is miniscule compared to losing changes due to a power drop, hardware failure, or wily cat dancing on a keyboard.

Exercises

1. Create a local repository called `my-python`. Then, add and commit a simple Python program named `FakePhil.py`.
2. Edit the `FakePhil.py` file (add any text you wish). Run the `git status` command from a shell or terminal and note the output. Then, go ahead and commit the change using `git commit -m "updated FakePhil.py"`. Note this output as well.
3. Edit `FakePhil.py` again but this time remove the content you added in **Exercise 2**. Run the `git status` command from a shell or terminal and note the output. Then, go ahead and commit the change using `git commit -m "updated FakePhil.py"`. Note this output as well.

Questions

1. What are two things you can imagine communicating with version control?
2. What might be a type of data you *do not* want to include in version control?
3. What differentiates a directory from a respository?

4. Based on **Exercise 2**, what would you do to resolve the situation?
5. Why do you think we get the output we do in **Exercise 3**?