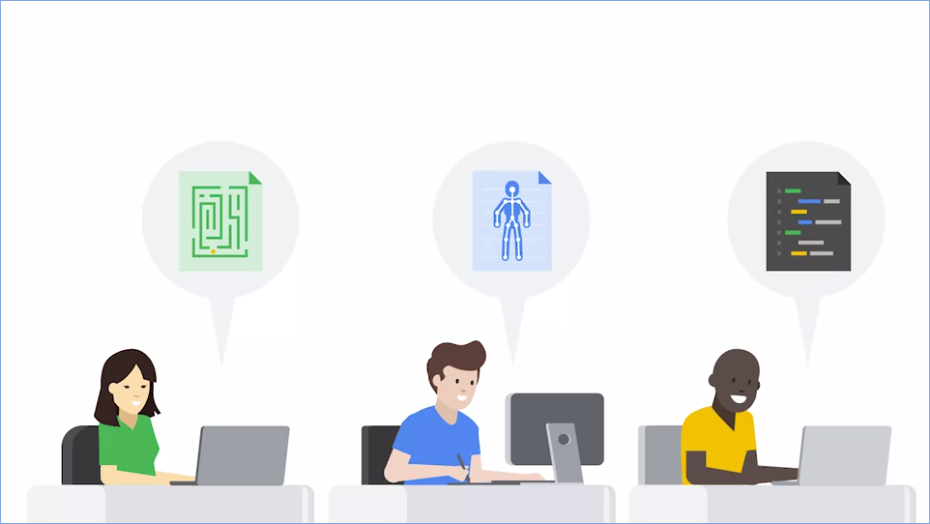
Using a Remote Repository

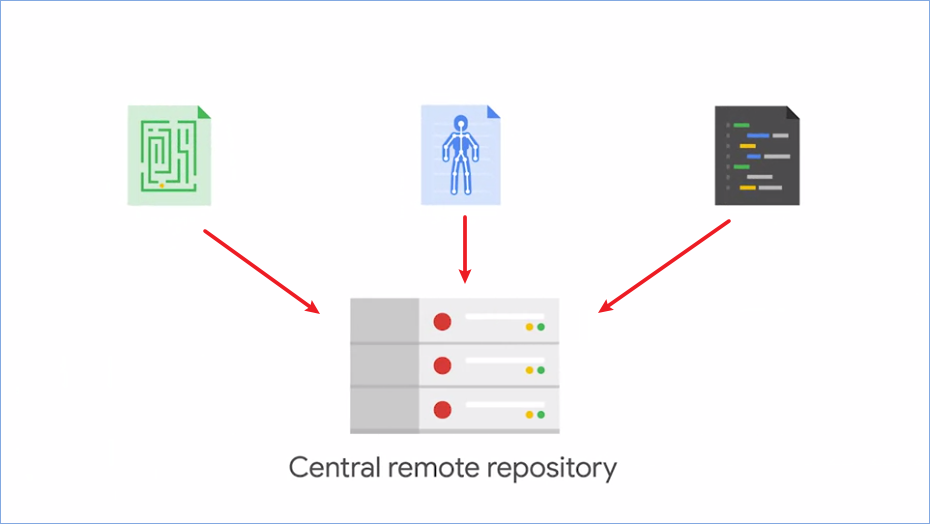
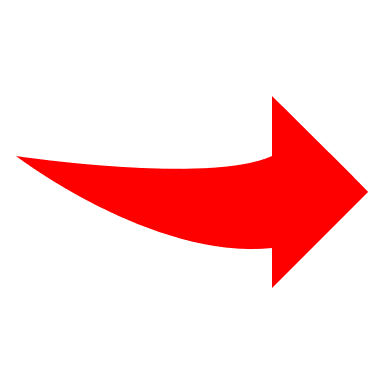
**What is a Remote?**

When we clone the newly created GitHub repository, we had our local Git Repo interact with a remote repository. Remote repositories are a big part of the distributed nature of Git collaboration. It let lots of developers contribute to a project from their own workstations making changes to local copies of the project independently of one another. When they need to share their changes, they can issue git commands to pull code from a remote repository or push code into one.

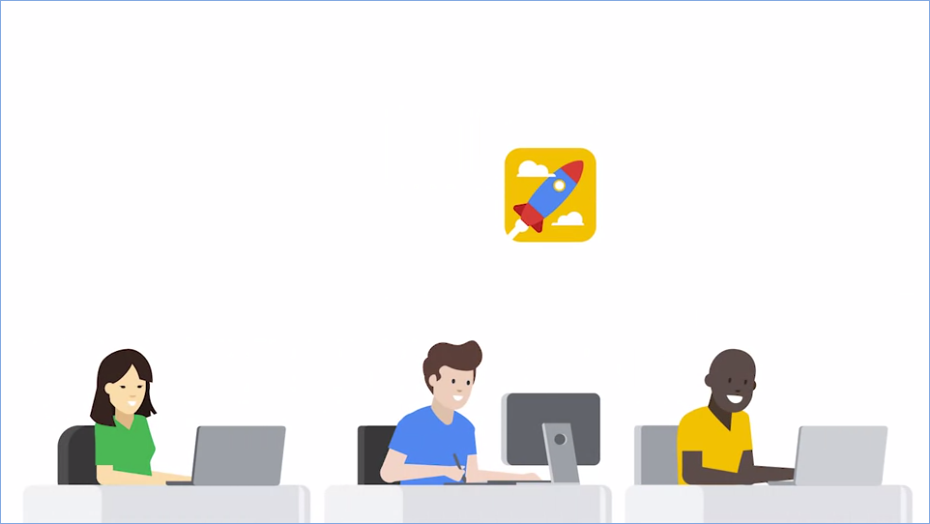
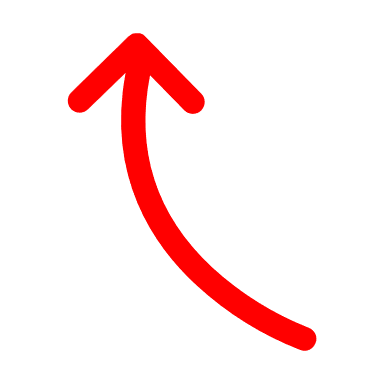
There are a bunch of ways to host remote repositories. As we called out, there is many internet-based Git hosting providers like GitHub, BitBucket or GitLab which offer similar services. We can also set up a Git server on our own network to host private repositories. A locally hosted Git server can run on almost any platform including Linux, mac OS, or Windows. This has benefits like increased privacy, control, and customization.



To understand remote repositories, and Git's distributed nature a bit better, imagine you're working together with some friends to design a computer game, each of you has a different portion of the game you're responsible for. One person is designing the levels, another the characters while others are writing the code for the graphics, physics, and gameplay.

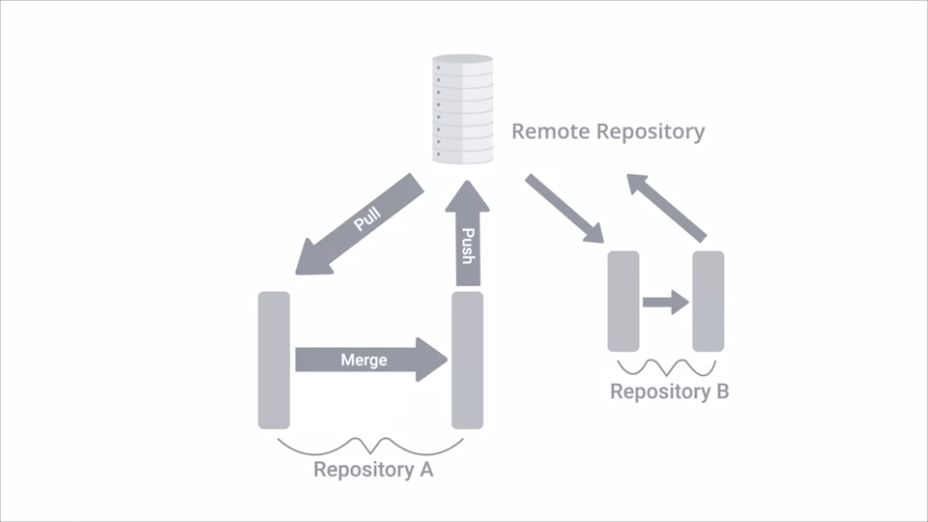


* All these areas will have to come together into a single place for the final product.



* Although your friends might work on their parts by themselves, from time to time, everyone needs to send out progress updates to let each other know what they've been working on. You will then need to combine their work into your own portion of the project to make sure it's all compatible.

Using Git to manage a project helps us collaborate successfully. Everyone will develop their piece of the project independently in their own local repositories maybe even using separate branches. Occasionally they'll push finished code into a central remote repository where others can pull it and incorporate it into their new developments. So how does this work?



* Alongside the local development branches like master, Git keeps copies of the commits that have been submitted to the remote repository and separate branches. If someone has updated a repository since the last time you synchronize your local copy, Git will tell you that it's time to do an update. If you have your own local changes when you pull down the code from the remote repo, you might need to fix merge conflicts before you can push your own changes. In this way Git let's multiple people work on the same project at the same time. When pulling new code it will merge the changes automatically if possible or will tell us to manually perform the integrating if there are conflicts.

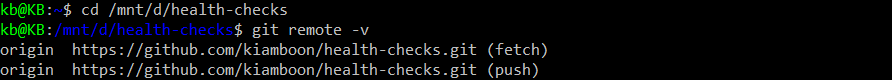
So when working with remotes the workflow for making changes has some extra steps. Will still modify stage and commit our local changes. After committing, we'll fetch any new changes from the remote repo manually merge if necessary and only then will push our changes to the remote repo.

Git supports a variety of ways to connect to a remote repository. Some of the most common are using the HTTP, HTTPS and SSH protocols and their corresponding URLs. HTTP is generally used to allow read only access to a repository. In other words, it lets people clone the contents of your repo without letting them push new contents to it. Conversely HTTPS and SSH, both provide methods of authenticating users so you can control who gets permission to push.

The distributed nature of the work means that there are no limits to how many people can push code into a repository. It's a good idea to control who can push codes to repos and to make sure you give access only to people you trust. Web services like GitHub, offer a bunch of different mechanisms to control access to Repositories. Some of these are available to the general public while others are only available to enterprise users.

**Working with Remotes**

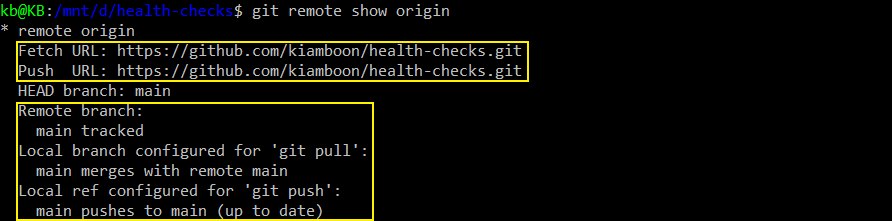
When we call a **git** **clone** to get a local copy of a remote repository, Git sets up that remote repository with the default origin name.



* We can look at the configuration for that remote by running **git remote -v** in the directory of the repo.
* Here we see the URLs associated with the origin remote. There are 2 URLs. One will be used to fetch data from the remote repository, and the other one to push data to that remote repo.
* They'll usually point at the same place. But in some cases, you can have the fetch URL use HTTP for read only access, and the push URL use HTTPS or SSH for access control. This is fine as long as the contents of the repo that you read when fetching are the same that you write to in pushing.

Remote repositories have a name assigned to them, by default, the assigned name is origin. This lets us track more than one remote in the same Git directory.

While this is not the typical usage, it can be useful when collaborating with different teams on projects that are related to each other. We won't look at how to do that here, but we'll include a link for more information in the next reading.



* If we want to get even more information about our remote, we can call **git remote show origin**.
* There's a ton of information here, and we don't need all of it right now. We can see the fetch and push URLs that we saw before, and the local and remote branches too.
* For now we only have a master branch that exists locally and remotely. So the information here seems a bit repetitive.

Once you start having more branches, especially different branches in the local and remote repo, this information starts becoming more complex. So what are these remote branches that we're talking about anyways?

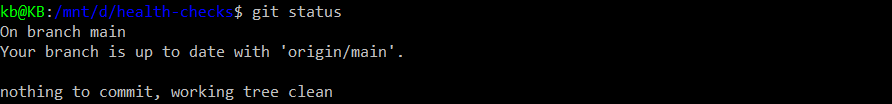


* Whenever we're operating with remotes, Git uses remote branches to keep copies of the data that's stored in the remote repository. We could have a look at the remote branches that our Git repo is currently tracking by running **git branch -r**.
* These branches are read only. We can look at the commit history, like we would with local branches, but we can't make any changes to them directly. To modify their contents, we'll have to go through the workflow we called out before.

First, we pull any new changes to our local branch, then merge them with our changes and push our changes to the repo.

*If we want to make a change to a remote branch, what must we do?*

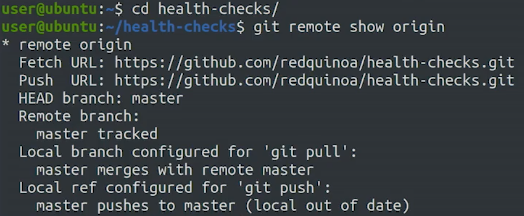
* *Pull the remote branch, merge it with the local branch, then push it back to its origin. We still have to go through the normal workflow to change remote branches.*



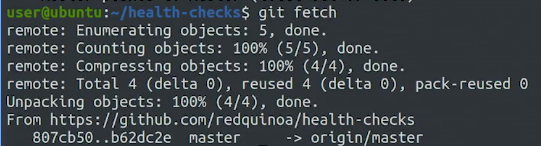
* Remember how we've been using git status to check the status of our changes? We can also use git status to check the status of our changes in remote branches as well.
* Now that we're working with a remote repository, git status gives us additional information. It tells us that our branch is up to date with the origin/master branch, which means that the master branch in the remote repository called origin, has the same commits as our local master branch.

**Fetching New Changes**

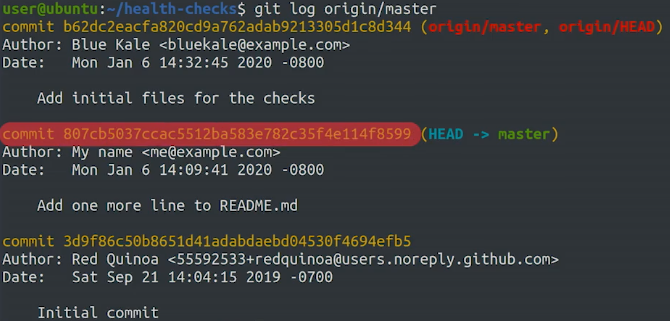
While we were learning about remotes, our colleague Blue Kale added some files to our repo. (Note: Tutorial didn’t show us how the 3rd party – Blue Kale added file to our repo). We could always use the GitHub website to browse the changes that were submitted. But we want to learn how to do it by interacting through the command line because you might need to do it this way at your job, and it'll work the same no matter which platform you use to interact with Git.



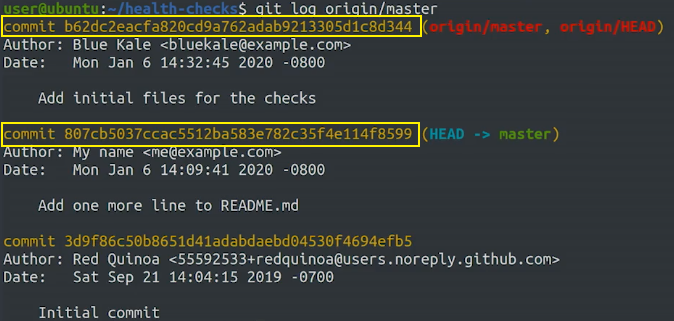
* So first, let's look at the output of the Git remote show origin command.
* Check out how it says that the local branches out of date. This happens when there were commits done to the repo that aren't yet reflected locally. Git doesn't keep remote and local branches in sync automatically, it waits until we execute commands to move data around when we're ready.



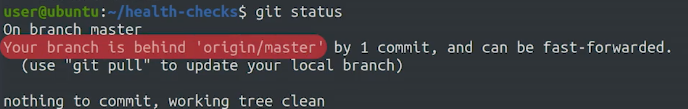
* To sync the data, we use the **git** **fetch** command. This command copies the commits done in the remote repository to the remote branches, so we can see what other people have committed.
* Fetched content is downloaded to the remote branches on our repository. So it's not automatically mirrored to our local branches.



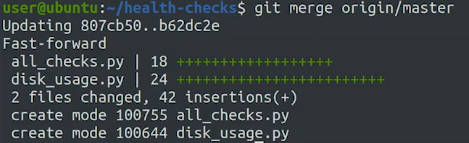
We can run **git** **checkout** on these branches to see the working tree, and we can run **git** **log** to see the commit history.



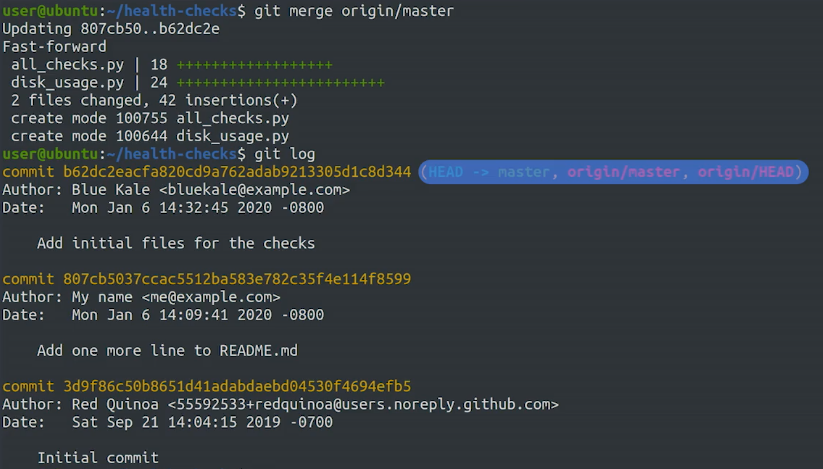
* Let's look at the current commits in the remote repo by running **git log origin/master**.
* Looking at this output, we can see that the remote origin/branch is pointing to the latest commit.
* While the local master branch is pointing to the previous commit we made earlier on.



* Let's see what happens if we run git status now.
* Git status helpfully tells us that there's a commit that we don't have in our branch. It does this by letting us know our branches behind their remote origin/master branch.



* If we want to integrate the branches into our master branch, we can perform a merge operation, which merges the origin/master branch into our local master branch. To do that, we'll call **git merge origin/master**.
* We've merged the changes of the master branch of the remote repository into our local branch. See how Git tells us that the code was integrated using fast-forward? It also shows that two files were added, all\_checks and disk\_usage.py.



* If we look at the log output on our branch now, we should see the new commit. We see that now our master branch is up to date with the remote origin/master branch. With that, we've updated our branch to the latest changes.

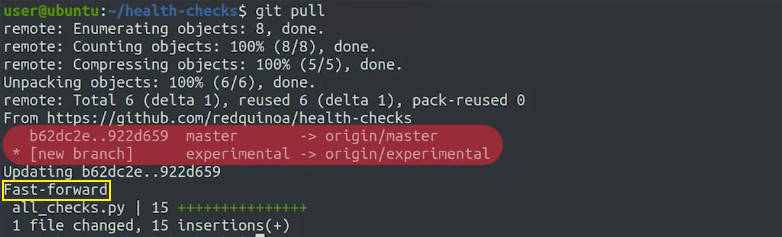
*What’s the main difference between git fetch and git pull?*

* *git fetch fetches remote updates but doesn't merge; git pull fetches remote updates and merges. git pull instantly merges while git fetch only retrieves remote updates.*

We can use **git fetch** like this to review the changes that happen in the remote repository. If we're happy with them, we can **use git merge** to integrate them into the local branch. Fetching commits from a remote repository and merging them into your local repository is such a common operation in Git that there's a handy command to let us do it all in one action.

**Updating the Local Repository**

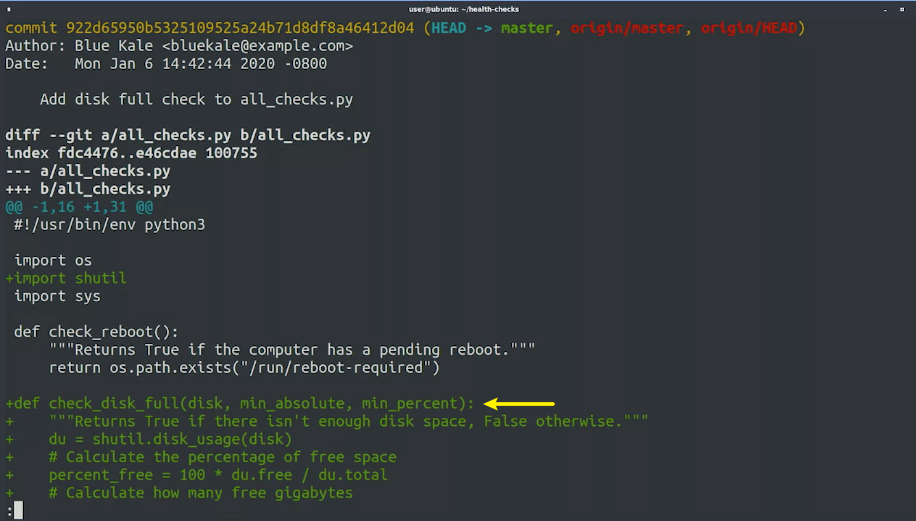
Earlier, we took a look at the basic workflow for working with remotes when we want to fetch the changes manually, merge if necessary, and only then push any changes of our own. Since fetching and merging are so common, Git gives us the **git pull** command that does both for us. Running **git pull** will fetch the remote copy of the current branch and automatically try to merge it into the current local branch.



* Let's check if our friend Blue Kale has made any new changes to the repo. We'll run **git pull** and see what changes we get.
* If we look closely at this output, we'll see that it includes the output of the fetch and merge commands that we saw earlier.
* First, Git fetched the updated contents from the remote repository, including a new branch.
* And then it did a fast forward merge to the local master branch. We'll see that the all\_checks file was updated as well.

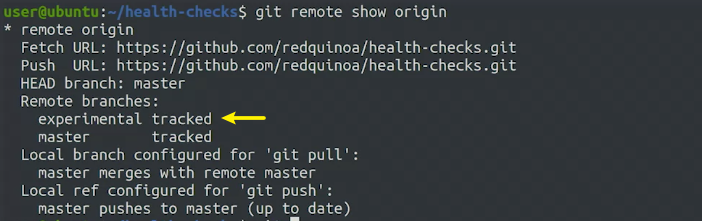


We can look at the changes by using **git log -p -1**.



* We see that our colleague added a check\_disk\_full function that includes the code from the other disk\_usage py file that we saw earlier.
* So now I'll exit the editor with q.

When we called **git pull**, we saw that there was also a new remote branch called experimental.



* Our friend Blue Kale told us that they've started working on a new feature in that branch. Let's check out the output of **git remote show origin** and see what it says about that new branch.
* We see that there's a new remote branch called experimental, which we don't have a local branch for yet.



* To create a local branch for it, we can run **git checkout experimental**.
* When we checked out the experimental branch, Git automatically copied the contents of the remote branch into the local branch. The working tree has been updated to the contents of the experimental branch. Now we're all set to work on the experimental feature together with our colleague.

*Assuming no merge conflicts, which type of merge does git pull perform automatically?*

* *Fast-forward merge. As long as there are no conflicts, Git will move the current branch tip up to the target branch tip and combine histories of both commits.*

In this last example, we got the contents of the experimental bunch together with those of the master branch when we called **git pull**, which also merged new changes onto the master branch. If we want to get the contents of remote branches without automatically merging any contents into the local branches, we can call **git remote update**. This will fetch the contents of all remote branches, so that we can just call checkout or merge as needed. We've now seen a bunch of different ways that we can use to interact with remote repositories. We've seen how to check their status, how to push and pull changes into repositories, and even how to get new branches out of them.

There's still more to come, but you're probably starting to see how useful this can be for collaborating with others. In our past examples, we've only looked at what happens with changes when they can be solved through fast forward. In upcoming exercise, we'll look at what happens when we try to push changes, especially when our changes generate conflicts.

**Git Remotes Cheat-Sheet**

|  |  |
| --- | --- |
| **Command** | **Explanation & Links** |
| git remote | Lists remote repos |
| git remote -v | List remote repos verbosely |
| git remote show <name> | Describes a single remote repo |
| git remote update | Fetches the most up-to-date objects |
| git fetch | Downloads specific objects |
| git branch -r | Lists remote branches; can be combined with other branch arguments to manage remote branches |