Luis Carlos A. Rojas Torres

Google IT Automation with Python Professional Certificate

– Monograph –

September 11, 2021

Springer

Preface

Use the template *preface.tex* together with the Springer document class SVMono (monograph-type books) or SVMult (edited books) to style your preface in the Springer layout.

A preface is a book's preliminary statement, usually written by the *author or editor* of a work, which states its origin, scope, purpose, plan, and intended audience, and which sometimes includes afterthoughts and acknowledgments of assistance.

When written by a person other than the author, it is called a foreword. The preface or foreword is distinct from the introduction, which deals with the subject of the work.

Customarily *acknowledgments* are included as last part of the preface.

Place(s), *Firstname Surname*
month year *Firstname Surname*

Contents

Part I
Introduction to Git and GitHub

Use the template *part.tex* together with the Springer document class SVMono (monograph-type books) or SVMult (edited books) to style your part title page and, if desired, a short introductory text (maximum one page) on its verso page in the Springer layout.

Chapter 1
Week 1

1.1 Course Introduction

1.1.1 Course Introduction

Hi there, and welcome back. You've heard us talk a lot about programming and automation. This course focuses on a slightly different aspect. How to keep track of the different versions of your code and configuration files using version control systems or VCS. These are tools that everyone in IT can benefit from, even if it's not just for programming or automation itself. It will allow us to easily roll back when mistakes happen and also help us collaborate with others. You might have already heard of version control systems in the context of managing configuration files or maintaining the source code of programs and scripts.

In this course, we'll introduce you to a popular VCS called Git, and show you some of the ways you can use it. We'll also go through how to set up an account with the service called GitHub, so that you can create your very own remote repositories to store your code and configuration. By the end of this course, you'll be able to store your codes history in Git, and collaborate with others in GitHub, where you'll also start creating your own portfolio.

Nowadays, lots of employers were asked to see your GitHub portfolio when you're interviewing for IT roles. GitHub portfolios give companies an idea of what projects you've worked on and what kind of code you can write. This course will help you get your setup.

Throughout this course, you'll learn about Git's core functionality, so you can understand how and why it's used in organizations. We'll look at both basic and more advanced features, like branching and merging. We'll demonstrate how having a working knowledge of a VCS like Git can be a lifesaver in emergency situations or when debugging, and we'll explore how to use a VCS to work with others through remote repositories, like the ones provided by GitHub.

To do all this, and so you can follow along with the exercises in these videos, you'll need to install Git on your computer. This will also let you interact with

GitHub, and upload your code there. For the examples in this course, we'll show a bunch of different Python scripts. While you don't need to know any Python to use Git, we do recommend that you have a basic knowledge of the language, so that you can understand the examples and the functionality we'll be demonstrating. If you've done the courses on Python in this program, you're covered. If you haven't, that's okay. But you might need to freshen up your Python skills to follow some of our examples. Also, since all the scripts will use Python 3, you'll need to have Python 3 installed in your computer to run them. For our examples, we're going to use a Linux computer, interacting with the Linux command line through the most common command line tools. Again, if you joined us for the Python courses, you're already familiar with all these concepts. If you're jumping into this program with this course, you might benefit from reviewing some of the most basic Linux commands.

1.2 Before Version Control

1.2.1 Intro to Module 1:

When you work in IT, you manage information across a lot of different files. You write automation scripts that might evolve over time. For example, you might add new features to your script or take into account additional conditions or modify the scope of systems where the script will be executed. You also manage configuration related to your infrastructure like the default settings on an application or the IP addresses assigned to the computers in your fleet. This information changes over time as the security requirements increase. The fleet grows or new versions of software gets deployed.

When trying to manage change in IT, it's super important to have detailed historical information for your organization's configuration files and automation code. This let's the administrators see what was modified and when, which can be critical to troubleshooting. It also provides a documentation trail that will let future IT specialists know why the infrastructure is the way it is, and it provides a mechanism for undoing a change completely. This way, we don't have to undo changes from memory and there's less chance of human error. We'll see this in action when we talk about rollbacks.

Imagine this, your team has added a new feature to a script that checks the health of all the computers that you're responsible for. The new check verifies that the firmware of the computer, also known as the UEFI, is updated to the latest version. When you roll this out, you suddenly realize that half the computers now say they're broken. After some investigation, you discover that the check needs to take into account different computer models. You might be tempted to do a quick code fix, push it to the affected machines right away especially if it seems like an easy fix. But more often than not, quick fixes include their own bugs because we don't take

the time to test a new code properly. So after the first fix, you might end up doing a second or even third emergency push until things are really working correctly.

To avoid these headaches, you can use a version control system to easily roll back your code to the previous version. Since you know that this version was working correctly before the change was made, it would be safe to go back to that one until you had time to fix the code, run some tests, and make sure everything works correctly for all machine models. By releasing code only after properly testing, you avoid having to push quick-fix after quick-fix.

Version control systems let us do this and much more. They are crucial to maintaining a healthy codebase for all kinds of IT resources, and for letting multiple people collaborate on the same coding projects together. We're now going to take our first steps to learning this new tool, which will let us keep track of the changes that we make to our scripts, our configuration files, and any other kind of documents that need to be tracked.

We'll start by looking at what people tend to do when they don't know about version control and then check out some related tools, like diff and patch. Once we have a clear idea of why we need proper version control, we'll jump into our first Git experience. We'll talk about what Git is and how it does what it does. To follow along, you'll need to install Git locally on your machine and learn how to use it from the command line. If this sounds a bit scary, don't panic. We'll guide you along the way and you'll be using it in no time. Once you have Git installed in your computer, we'll do an overview of the basic Git workflow which will let you start keeping track of your scripts. So are you ready to start taking control of your code? Here we go.

1.2.2 Keeping Historical Copies

Have you ever worked on a project that was developing over time? So you occasionally created copies of the work in case you wanted to go back to an earlier version.

Maybe you were working in a team and every day you'd email a part of the work to the rest of the team. And then the other members on your team would add their own work, and send it out to the whole team too.

Or maybe you've worked on a very complex project, that kept changing directions. And you felt that some of the things that got removed one day, might have to be added later on. So anytime you're about to delete a significant part, you made a copy of the whole thing, just in case.

If any of this sounds familiar, you've already worked on **the most primitive form of version control**, keeping historical copies. These copies let you see what the project was like before, and go back to that version if you end up deciding that the latest changes were wrong. They also let you see the progress of the changes over time, and maybe even help you understand why a change was made. We say that this is primitive because it's very manual and not very detailed. First, you need to remember to make the copy. Second, you usually make a copy of the whole thing, even if you're only changing one small part.

And third, even if you're emailing your changes to your colleagues, it might be hard to figure out at the end who did what, and more importantly, why they did it.

But that said, the principle behind version control is the same.It lets us keep track of the changes in our files. These files can be code, images, configuration, or even a video editing project, whatever it is you're working with.

Throughout this course, we'll see the many ways that Git helps us keep track of our changes, and also how we can use it to collaborate with others or avert changes. We'll use a bunch of terms that have special meanings in the world of version control, but don't let those intimidate you. In the end, all we're doing is having better control over our historical copies. So, say you have two copies of the same code made at different points in time. How can you compare them?

1.2.3 Diffing Files

Imagine you had two copies of some code, and you wanted to see what the difference was between them. How would you do it? You could open both files in the editor side by side, look at one then look at the other to spot the differences, but that's super error-prone. We're human and by comparing with our eyes we are bound to miss some differences. Fortunately, there's a better way. You can use some nifty tools that will do this automatically. We can use the `diff` command line tool to take two files or even to directories, and show the differences between them in a few formats. Let's check it out with an example. We have two files `rearrange1.py` and `rearrange2.py` which contain two different versions of the same function. Let's take a look at them using CAT Fig.1.1.

Fig. 1.1 Using cat in linux console

```
user@ubuntu:~$ cat rearrange1.py
#!/usr/bin/env python3

import re

def rearrange_name(name):
    result = re.search(r"^([\w .]*), ([\w .]*)$", name)
    if result == None:
        return result
    return "{} {}".format(result[2], result[1])

user@ubuntu:~$ cat rearrange2.py
#!/usr/bin/env python3

import re

def rearrange_name(name):
    result = re.search(r"^([\w .-]*), ([\w .-]*)$", name)
    if result == None:
        return result
    return "{} {}".format(result[2], result[1])

user@ubuntu:~$
```

Can you spot the difference? Maybe you can but it's not super obvious. Let's use the diff command (Fig.1.2) so that we don't have to strain our eyes trying to spot it.

When we call the `diff` command, we get only the lines that are different between two files. It's much easier to find the difference when we just have two lines.

See the symbols at the beginning of each of those lines? The *less than* << symbol tells us that the first line was removed from the first file, and the *greater than* >> symbol tells us that the second line was added to the second file. In other words, the old line got replaced by the new one. In this example, we had one line that was replaced with a new one. This is a common change when modifying code, but not the only possibility.

Fig. 1.2 Using diff in linux console

```
user@ubuntu:~$ diff rearrange1.py rearrange2.py
6c6
<     result = re.search(r"^([\w .]*), ([\w .]*)$", name)
---
>     result = re.search(r"^([\w .-]*), ([\w .-]*)$", name)
```

Let's check out another example. Here there are more changes going on. We can see that `diff` (Fig.1.3)splits the changes in two separate sections. The section that starts with `5c5,6` shows a line in the first file that was replaced by two different lines in the second file. The number at the beginning of this section indicates the line number in the first and second files. The `c` in between the numbers means that a line was **changed**. The section that starts with `11a13,15` shows three lines that are new in the second file. The `a` stands for, you guessed it, **added**, but that block looks a bit strange doesn't it? It seems like we're adding a return and an if condition but nobody for the if block. What's up with that?

Fig. 1.3 Using `diff` in linux console

```
user@ubuntu:~$ diff validations1.py validations2.py
5c5,6
<     assert (type(username) == str), "username must be a string"
---
>     if type(username) != str:
>         raise TypeError("username must be a string")
11a13,15
>         return False
>     # Usernames can't begin with a number
>     if username[0].isnumeric():
user@ubuntu:~$
```

To understand this better we can use the **dash u** (`-u`) flag to tell `diff` to show the differences in another format. Let's check that out (Fig.1.4).

This unified format is pretty different from the one that we saw before. It shows the change lines together with some context, using the **minus sign** (-) to mark **lines that were removed**, and the **plus sign** (+) to mark **lines that were added**.

The extra context let's us better know what's going on with the change that we're diffing. We can see that the new file actually has a completely new `if` block that's part of a chain of conditionals that looks very similar, and that's why with the `diff` output that we saw before, it was a little confusing which lines had been added. There are a lot of tools out there to compare files. Diff is the most popular one, but not the only one available. For example, `wdiff` **highlights the words that have changed in a file instead of working line by line** like diff does. To help us even more, there are bunch of graphical tools that display files side by side and highlight

Fig. 1.4 Using `diff -u` in linux console

```
user@ubuntu:~$ diff -u validations1.py validations2.py
--- validations1.py     2020-01-05 07:03:46.999900910 -0800
+++ validations2.py     2020-01-05 07:03:46.999900910 -0800
@@ -2,7 +2,8 @@


 def validate_user(username, minlen):
-    assert (type(username) == str), "username must be a string"
+    if type(username) != str:
+        raise TypeError("username must be a string")
     if minlen < 1:
         raise ValueError("minlen must be at least 1")

@@ -10,5 +11,8 @@
         return False
     if not username.isalnum():
         return False
+    # Usernames can't begin with a number
+    if username[0].isnumeric():
+        return False
     return True
```

the differences by using color. Some examples of this include: meld, KDiff3, or vimdiff. We can use these tools to give better contexts to the changes that we see. We've talked about how we can see differences between two files, now how can we use those differences to apply changes? That's coming up in the next video.

1.2.4 Applying Changes

Imagine a colleague sends you a script with a bug and asked you to help fix the issue. Once you understood what was wrong with the script, you could describe to them what they need to change. Something like, "*Well, you can only return values inside functions. I think you meant to use sys.exit instead. Also, you're converting to gigabytes twice, so your script will always fail.*" But this could still be hard for them to understand if the code is complex.

To make the change clear, you could send them a `diff` with the change so that they can see what the modified code looks like. To do this, we typically use a command line `like diff-u old_file new_ file > change.diff`. As a reminder, the **greater than sign redirects the output of the diff command to a file**. So with this command, **we're generating a file called** `change.diff` **with the contents of** `diff-u` **command**. By using the `-u` flag, we include more context which helps the person reading the file understand what's going on with the change. The generated file is usually referred to as a **diff file** or sometimes a **patch file**. It includes all the changes between the old file and the new one, plus the additional context needed to understand the changes and to apply those changes back to the original file.

Now, say you're the one receiving a `diff file` with a change and you want **to apply it to a script you wrote**. You could read the diff file you receive carefully and then manually go through the file that needs to be changed, and apply the modifications. But it sounds like a lot of manual work that could be automated, don't you think? Well, it sure is. There's a command called `patch` to do exactly this. **Patch takes a file generated by diff and applies the changes to the original file**. Let's

check this out in an example. Say we have a small script that checks whether the computer is under too much load, like this one.

This script uses the `psutil` module to check the percentage of the CPU that's currently in use. When the load is above a threshold, in this case 75 percent, it prints a message with an error. When it's under the threshold, it says that everything's okay. Now, we've shared this script with a few colleagues and one of them tells us that the script doesn't work correctly. Even if a computer is completely overloaded, the script will say that everything's okay. Our colleague is so helpful that they sent us a `diff` with the fix for our problem. Let's check that one out (Fig.1.5).

Fig. 1.5 Python file and its `diff` file.

```
user@ubuntu:~$ cat cpu_usage.py
#!/usr/bin/env python3
import psutil

def check_cpu_usage(percent):
    usage = psutil.cpu_percent()
    return usage < percent

if not check_cpu_usage(75):
    print("ERROR! CPU is overloaded")
else:
    print("Everything ok")

user@ubuntu:~$ cat cpu_usage.diff
--- cpu_usage.py        2019-06-23 08:16:04.666457429 -0700
+++ cpu_usage_fixed.py  2019-06-23 08:15:37.534370071 -0700
@@ -2,7 +2,8 @@
 import psutil

 def check_cpu_usage(percent):
-    usage = psutil.cpu_percent()
+    usage = psutil.cpu_percent(1)
+    print("DEBUG: usage: {}".format(usage))
     return usage < percent

 if not check_cpu_usage(75):
```

We can see that our colleague made two changes. They added a one as a parameter to the CPU percent function and they added a debugging line, that prints the value returned by the function. Our colleague explains that by calling the CPU percent function without a parameter, we were not averaging over a period of time, and so the call always returns zero. So we have the `diff file` and we want to apply it to our script. How do we do that? We'll use the `patch` command. We'll pass the name of the file that we want to patch in this case, `cpu_usage.py`, as the first parameter to the command and then we'll provide the `diff` file through standard input. Do you remember how to do that? We will **use the less than symbol to redirect the contents of the file to standard input**. Let's check this out (Fig.1.6).

Fig. 1.6 Patching a file from diff file.

```
user@ubuntu:~$ patch cpu_usage.py < cpu_usage.diff
patching file cpu_usage.py
user@ubuntu:~$
```

So we told patch to apply the changes that come from `cpu_usage.diff` to our `cpu_usage.py` file. We get one single line that says the file was patched, which means that we've successfully applied the changes. Let's verify that by looking at the contents of our script.

Nice. We see that our file was modified with the changes that we got from our colleague. The CPU percent function is being called with a parameter of one and the debugging line is printed. Once we're happy with the script, we could remove the debugging line. But for now, we'll leave it in there. You might be wondering, why go through all this trouble diffing, and patching, and not just send the whole file instead? There are a few reasons for this:

The main reason is that the original code could have changed. In our example, it's possible that the code our colleague was using to prepare the fix wasn't the latest version. By **using a diff instead of the whole file, we can clearly see what they changed**, no matter which version they were using. The **patch command can detect that there were changes made to the file** and will do its best to apply the diff anyways. It won't always succeed but in many cases it will.

Another reason is **structure**. In this case we're patching a single small file. But sometimes, you might be modifying a bunch of large files inside of a huge project. Say you are changing four files in a project tree that contain 100 different files, arranged in different directories according to what they do. If you were to send the whole files, you'd need to specify where those files were supposed to be placed. As we called out, we can diff whole directory structures and in that case **the diff file can specify where each change file should be without us having to do any manual juggling**.

Cool right? Okay, great work. We've now seen how to generate diff files and how to apply their contents with the patch command. In the next video, we'll put all this together to look at a real-world example of how to use diff and patch.

1.2.5 Practical Application of diff and patch

Imagine this, a colleague is asking our help with fixing a script named `disk_usage.py`. The goal of the script is to check how much disk space is currently used, and print an error if it's too little space for normal operation. But the script is currently broken because it has a few bugs. We'll help our colleague fix those bugs to demonstrate how to use `diff` and `patch`. Before we change anything, let's make a couple copies of the script. We'll add `_original` to one copy , which we'll keep unmodified and use for comparison (`cp disk_usage.py disk_usage_original.py`) and `_fixed` to the other copy (`cp disk_usage.py disk_usage_fixed.py`), which we'll use to repair our fix.

Okay, now that we have our copies, we'll edit the `_fixed` version and actually fix it. This file has a bunch of code. Before we try to understand what it does and what's wrong with it, let's execute it and see what we get (`SyntaxError: 'return' outside function`).

The Python interpreter isn't too happy. It's complaining that there's a return outside of function. And if we look at the code, we can clearly see that there's a return that's not inside any function (Fig. 1.7).

Fig. 1.7 return outside function.

```
import shutil

def check_disk_usage(disk, min_absolute, min_percent):
    """Returns True if there is enough free disk space, false otherwise."""
    du = shutil.disk_usage(disk)
    # Calculate the percentage of free space
    percent_free = 100 * du.free / du.total
    # Calculate how many free gigabytes
    gigabytes_free = du.free / 2**30
    if percent_free < min_percent or gigabytes_free < min_absolute:
        return False
    return True

# Check for at least 2 GB and 10% free
if not check_disk_usage("/", 2*2**30, 10):
    print("ERROR: Not enough disk space")
    return 1

print("Everything ok")
return 0
```

You might remember that in Python, we can only use return statements inside functions. So how do we fix this? There's a couple options. We could turn the current code into a function and then call that function from the main part of our script. Or we could use `sys.exit` to make the `return` number of the exit code of our script, which is the code that causes a program to exit with the corresponding exit value.

For now, let's go with the second option (Fig. 1.8).

Fig. 1.8 Changes in `fixed` file.

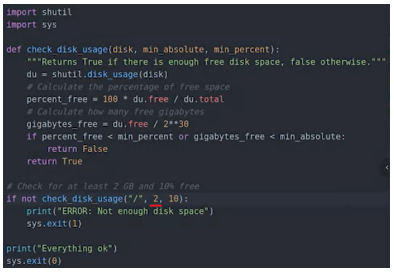
```
import shutil
import sys

def check_disk_usage(disk, min_absolute, min_percent):
    """Returns True if there is enough free disk space, false otherwise."""
    du = shutil.disk_usage(disk)
    # Calculate the percentage of free space
    percent_free = 100 * du.free / du.total
    # Calculate how many free gigabytes
    gigabytes_free = du.free / 2**30
    if percent_free < min_percent or gigabytes_free < min_absolute:
        return False
    return True

# Check for at least 2 GB and 10% free
if not check_disk_usage("/", 2*2**30, 10):
    print("ERROR: Not enough disk space")
    sys.exit(1)

print("Everything ok")
sys.exit(0)
```

Okay, we've made the change. Let's execute this new version of our script.

Darn, we fixed the syntax error, but now the script is telling us we don't have enough space on our disk. But we know that we actually do have some free space, right? What's up with that? If you look closely at the code, you might notice that **the script is converting to gigabytes twice**.

The function call to `check_disk_usage` is passing 2 times 2 double star 30 $2**30$ (blue line). You might remember that the double star operator ($**$) is used to calculate powers. In this case, 2 to the power of 30, which is how many bytes are in a gigabyte. So, this would be 2 gigabytes, but that be if the `check_disk_usage` function was expecting a value in bytes. If we look at the code of the function, we can see that it's already dividing the amount of free bytes by 2 to the power of 30. So in other words, **we're doing the gigabyte conversion twice**. Once when calling the function and once inside the function. We need to get rid of one of them. Let's change how we call the function (Fig. 1.9).

Fig. 1.9 Changes in `fixed` file.



Okay, let's try it out again.

It works now. Okay, now we need to send a fixed to our colleague so that they can fix their script.

To do that, we'll use a technique we just learned to generate a diff file, like this `diff -u disk_usage_original.py disk_usage_fixed.py > disc_usage.diff`. Let's check the contents of the `diff` using the `cat` command...

Awesome. This seems to have what we want. So this is what we need to send to our colleague to have them patch their file. How would they do that? They would run the patch command like this `patch disk_usage.py < disk_usage.diff`.

By calling patch with the diff file, we've applied the changes that were necessary to fix the bugs. Let's check that `disk_usage.py` now executes successfully.

Success. So we've now seen how we can look at differences between files, generate diff files together to gather our changes, and then apply those changes using patch. But this is still a very manual process, where version control systems can really help. But before we jump into that, in the next cheat sheet, you'll find a summary of the commands we just covered. So check that out and then head over to the practice quiz to make sure you've got a grasp on all this.

1.2.6 Reading: diff and patch Cheat Sheet

- `diff` : is used to find differences between two files. On its own, it's a bit hard to use; instead, use it with diff -u to find lines which differ in two files:
- `diff -u` : is used to compare two files, line by line, and have the differing lines compared side-by-side in the same output.

```
~$ cat menu1.txt
Menu1:

Apples
Bananas
Oranges
Pears
```

```
~$ cat menu2.txt
Menu:

Apples
Bananas
Grapes
Strawberries

~$ diff -u menu1.txt menu2.txt
--- menu1.txt   2019-12-16 18:46:13.794879924 +0900
+++ menu2.txt   2019-12-16 18:46:42.090995670 +0900
@@ -1,6 +1,6 @@
-Menu1:
+Menu:

Apples
Bananas
-Oranges
-Pears
+Grapes
+Strawberries
```

- `patch` : is useful for applying file differences. See the below example, which compares two files. The comparison is saved as a `.diff` file, which is then patched to the original file.

```
~$ cat hello_world.txt
Hello World
~$ cat hello_world_long.txt
Hello World

It's a wonderful day!
~$ diff -u hello_world.txt hello_world_long.txt
--- hello_world.txt     2019-12-16 19:24:12.556102821 +0900
+++ hello_world_long.txt        2019-12-16 19:24:38.944207773 +0900
@@ -1 +1,3 @@
Hello World
+
+It's a wonderful day!
~$ diff -u hello_world.txt hello_world_long.txt > hello_world.diff
~$ patch < hello_world.diff
patching file hello_world.txt
~$ cat hello_world.txt
Hello World
```

```
It's a wonderful day!
```

1.3 Version Control Systems

1.3.1 What is version control

We've seen up till now, how we can use existing tools to extract differences between versions of files and apply those changes back to the original files. Those tools are really useful. But most of the time, we won't be using them directly. Instead, we'll use them through a **Version Control System**, or **VCS**. A Version Control System keeps track of the changes that we make to our files. By using a VCS, **we can know when the changes were made and who made them**. It also **lets us easily revert a change** if it turned out not to be a good idea.

It **makes collaboration easier by allowing us to merge changes from lots of different sources**. At first-look, a Version Control System can seem like a complicated, possibly intimidating tool. But if you look closer, you'll see that it's really just a system that stores files. However, unlike a regular file server which only saves the most recent version of a file, a VCS keeps track of all the different versions that we create as we save our changes. There are many different version control systems, each with their own implementation and with their own advantages and disadvantages. But, no matter how the VCS is implemented internally, they always access the history of our files. **Let us retrieve past versions of the file or directory and see who changed which files, how each file was changed and when the file was changed**.

On top of this, we can make edits to multiple files and treat that collection of edits as a single change which is commonly known as a, `commit`. A VCS even **provides a mechanism to allow the author of a commit to record why the change was made, including what bugs, tickets or issues were fixed by the change**. This information can be a lifesaver when trying to understand a complex series of changes, or to debug some obscure issue. So, be sure to record this extra info in your commits to be truly committed to better code. In any organization that produces software, a VCS is a key part of managing the code.

Files are usually organized in **repositories** which contains separate software projects or just group all related code. If there's a lot of people involved in developing software, some developers may have access to only some of the repositories. A single repository can have as little as one person using it. And it can go up to thousands of contributors. And, as we called that earlier, a Version Control System can be used to store much more than just code. We can use it to **store configuration files, documentation, data files, or any other content that we may need to track**. Because of the way tools like `diff` and `patch` work, a VCS is especially useful when tracking text files, which can be compared with diff and modified with patch.

We can also store images, videos or any other complex file formats in a VCS, but, it won't be easy to check the differences between versions when comparing these file formats. It might not be possible to automatically merge changes made to older versions of a file. You now have a basic idea of what a Version Control System is, and how it works You might be asking yourself, do I really need this? Can't I just keep making backups of my code once in awhile? We'll answer that in our next video

1.3.2 Version Control and automation

At first glance, using a VCS might seem like a lot of work for an IT specialist to set up and learn. It might especially seem like **overkill**, if you're the only member of your IT team that writes code or maybe even the only member period. So can a VCS help, even if you don't need to share your scripts or collaborate on them with others? The short answer is yes. A VCS can be invaluable, even in a one-person IT department. A VCS stores your code and configuration. It **also stores the history of that code and configuration**. A version control system can function a lot like a time machine, giving you insights into the decisions of the past. Whenever you write a commit message, after making a change, it's as if the current version of yourself is explaining your decisions to a future you or others who might work on the same scripts and configurations in the future. This can help you avoid finding yourself staring at a piece of code that you or someone else wrote three months ago and puzzling over how it works or even why it exists.

With a VCS, you can v**iew, track and select snapshots** from the history of your project. So nothing you do is lost, and since we can use a VCS to store both code and configuration files, we can make the overall IT systems more scalable and reliable.

For example, let's say you've stored the `DNS zone` file for your company in a VCS and in case you don't remember, a DNS zone file **is a configuration file that specifies the mappings between IP addresses and host names in your network**. When you update the zone information, always use good explanatory commit messages. That way, you'll have access to meta information about your new IP addresses and host names present in the zone file. Like when they were added and for what purpose. If anything breaks after you add a new entry, you can rely on the VCS to tell you what the file looked like before the change. You can then revert to the old version quickly, so you can fix the problem fast and figure out what went wrong later. This functionality enhances the reliability of systems you operate. Because of the audit trail provided by the VCS, you know exactly what version of the zone file to rollback to, which reduces the time it takes to fix the problem.

It's generally better to quickly roll back first and stop errors before spending time figuring out what went wrong. You can curb the fix after the bleeding has stopped. Figuring out the bug might take up valuable time or worse, your first attempt at a solution can have its own bugs.

Let's look at a different example. The configuration for a `DHCP damon` can be replicated in two or more machines, where one acts as a primary server and the

other one acts as standby machine. The standby machines won't do much while the primary is up. But if the primary goes down for any reason, a standby machine can become primary and start responding to `DHCP queries`. For this to work, the configuration files on all machines need to be identical. This is because the `DHCP protocol` doesn't provide a way for standby machines to get an up-to-date version of the configuration files and the way `DNS` does.

To deal with this, we can keep the up-to-date version of the DHCP configuration in a version control system and have the machines download the configuration from the VCS. This means all the machines will have the exact same files. That's already handy enough. But after using it for a while, you're bound to see other benefits. Say you get an urgent alert over the weekend, telling you that your DHCP server isn't responding to any queries. You look at the history of the changes and you find that one of the changes added on Friday evening, included a duplicated entry causing the server to misbehave. By using a VCS, you can easily roll back the change and have the servers back to health in no time.

You might come across a second unexpected benefit, when it's time to replace the server with a new one. By having all the server configuration and a version control system, it's much easier to automate the task of deploying a new server. Are you starting to see how useful version control systems can be? You might even be thinking of some situations when having your files in VCS, would have saved you a few headaches. As we said in the beginning, in this course, we'll use Git, which is one of the most popular version control systems in use today. Up next, we'll learn a bit about the history of Git and what makes it so special.

1.3.3 What is Git

Git is a VCS created in 2005 by Linus Torvalds. The developer who started the Linux kernel.

Git is a free open source system available for installation on Unix based platforms, Windows and macOS. Linus originally created get to help manage the task of developing the Linux kernel. This was difficult because a lot of geographically distributed programmers were collaborating to write a whole bunch of code. Linus had some requirements for the way that the system worked, and its performance that weren't being met by the VCS tools at a time. So he decided to write his own.

Git is now one of the most popular version control systems out there and is used in millions of projects. Unlike some version control systems that are centralized around a single server, Git **has a distributed architecture**. This means that **every person contributing to a repository has full copy of the repository on their own development machines**.

Collaborators can share and pull in changes that others have made as they need. And **because the repositories are all local to the computer being used to create the files, most operations can be done really fast**. If you want to collaborate with others, it usually makes sense to set up a repository on a server to act as a kind of hub

for everyone to interact with. But Git doesn't rely on any kind of centralized server to provide control organizations to its workflow. Git **can work as a standalone program as a server and as a client**. This means that **you can use Git on a single machine without even having a network connection**. Or you can use it **as a server on a machine where you want to host your repository**. And then you can use Git **as a client to access the repository from another machine** or **even the same one**.

Git clients can communicate with Git servers over the network using **HTTP**, **SSH** or Git's own special protocol. If you're curious about diving deeper into Git architecture or communication protocols, we put a link for more information in the next reading. So you can use Git with or without a network connection. You can use it for small projects with like one developer or huge projects with thousands of contributors. You can use it to track private work that you can keep to yourself or you can share your work with others by hosting a code on **public servers like Github, Gitlab** or others. Are you starting to see how powerful Git can be?

When looking for information online you might notice that the official Git website is called `git-scm.com`. And wonder what's the **SCM** at the end for? It's actually another acronym similar to VCS. It stands for **Source Control Management**. While both terms mean the same, we generally prefer VCS, because as we call that already, these systems can actually be used to store much more than just source code. In this course we chose Git for its popularity, multi platform support and robust set of features. As with most things in the IT world, though, there are plenty of other tools that can be used to accomplish the same task. There are other VCS programs like Subversion or Mercurial. Feel free to experiment with alternatives if you think another VCS might better serve your needs. But before we jump into how to use Git, let's go through another quick quiz, to check that everything up until now is making sense.

1.3.4 Installing Git

In this course, we'll cover many different things that you can do with Git. To follow along, you'll need to install Git on your computer. On top of this, we'll show you how to interact with Git through the command line. So if you aren't yet already familiar with the command line, this is your opportunity to brush up. Our examples will be shown on a computer running Linux, but you can use Git on any operating system. There are versions of Git available for all popular operating systems. So before we get any further, let's get Git installed on your computer.

The first step is to check whether you already have it installed. You can do this by running `git --version`. If you're running a version number higher than `2.20`, then you can just use that one. If you get an error message or an older version number, you'll need to install the current version. If you use a package management system like `apt` or yum on Linux, `Chocolatey` on Windows, or `Homebrew` on Mac OS, you can just install Git through that.

If you don't use a package management system, then you can download the latest executable installer from the official website and deploy it on your computer. On Linux, installing and using Git is pretty straightforward. You can install it with the command `apt install git` or `yum install git`, and after that, you'll have Git installed and ready to use. On Mac OS, you can even have it installed when you run `git --version`. If Git isn't installed, this command will ask you if you want to install it and then download it and install it for you. Alternatively, you can also download it from the website and install it by following the prompts. Once it's installed, you'll be able to use it from the command line just like any other tool.

On Windows, after downloading and executing the installer, you'll need to go through a bunch of different configuration options. These options come with preselected defaults that usually makes sense to just keep. Pay attention to the editor question though. You'll probably want to change the editor to one that you feel comfortable with, like *Notepad++* or *Atom*. One interesting thing about the Windows installation is that it comes preloaded with an environment called **MinGW64**. This environment lets us operate on Windows with the same commands and tools available on Linux. So you can practice some Linux command line tools on your Windows machine. After installing Git on your Windows machine, you'll be able to use Git from the Linux command line. If you selected the default option for the path environment question, you'll be able to also run it from the PowerShell command line. If you want to understand more about each option involved in the installation on Windows, check out the optional video where we will talk about the available options and when you might want to select something different from the default. Throughout this course, we'll talk about how to do things from the `command line`. Some integrated developer environments or IDEs let us interact with Git through graphical interfaces. It's fine to use those if you feel more comfortable with them. We'll focus on the command line because it's standard and also because **once you've mastered the command line, you'll definitely be able to use any graphical tool out there**. In the next video, we'll look into the options you can set when installing Git on a Windows machine. This video is optional. If you've already set up Git on your computer, you can jump ahead to the next video.

1.3.5 Installing Git on Windows

In this video, we'll show you how you can install Git on Windows. We'll dive into the different options that you can select and what they all mean. We'll start by downloading the latest version of the software from the gitforwindows.org website.

The software package we get includes a bunch of other tools that might come in real handy. It comes with a Bash emulation environment, where we can run all the Git commands we'll explain in this course. And a bunch of Unix like utilities that we'll show in this and other courses. This bundle also includes a graphical user interface to interact with Git. We won't look at it during this course, but you can

experiment with it on your own. All right, we've downloaded the file, let's get it installed. First, let's start by clicking Keep.

The first window we get is the license of the software. Git is released under the GPL version two license, which is a free software license. This means that if we want, we can look at Git's code to learn how it works, and we can even modify it to do something different.

All right, let's accept this license and continue with the setup.

The first option we get is the installation path, we'll keep it as is. This window lets us select additional components to add to our Git installation. By default, Git integrates with the Windows Explorer to let us run Git Command Line or the Git Graphical Interface in the current folder. This software bundle comes with an extension for improving the support of how large files, like audio or video files, get stored in the version control system. It's a good idea to keep this enabled. The installer will also register the Git configuration files as files that should be opened with a text editor, and the `.sh files` as files that should be executed with Bash. All of this makes sense, so we'll keep these options selected. If you want, you can also enable some of the other options, that will cause the icons to be displayed on the desktop, true type fonts to be used in the console, or software to automatically check for updates. All right, let's keep going.

We get prompted for the name of the folder where we want to create all the shortcuts. is fine as the name for this, so let's continue.

Okay, now's the time to choose the editor. This is the option that you'll most likely want to adjust. You want to select an editor that you're comfortable with. The installer already has a list of possible options that you could choose here, like Notepad++, Visual Studio Code, or Sublime Text. And we can even select any other editor we wanted by entering the path to the editors executable file. For this course, we'll be using Atom in all of our examples. So let's choose Atom for this installation. Heads up, the bundle doesn't ship with any of these graphical editors installed, you'll need to install the one you want separately.

All right, we're now asked to choose how we want to adjust the PATH environment. This option let us decide how we want to execute Git from the command line.

Selecting the first option (`Use Git from Git Bash only`) will make Git only accessible through the embedded command line that is shipped with the bundle. The second option (`Git from the command line and also from 3rd-party software`), the one that's pre-selected, allows us to execute Git from the embedded command line and from the Windows Command Prompt. The third option (`Use Git and optional Unix tools from the Command Prompt`) will add the Unix-like tools shipped together with Git to the Windows Command Prompt. If you choose this option, any commands that have the same name as those in the OS will then come from the bundle and not the basic OS. We'll keep the second option selected, as it's the one that's most convenient for us, while not interfering with the local tools.

This window lets us choose how we want to validate the SSL certificates used for HTTPS connections. We can choose to use the OpenSSL library that shipped

together with Git, or use the native Windows library. You'll want the second option if you need to interact with your company's internal systems. As we'll only be interacting with GitHub, we'll keep the default option selected (`Use the OpenSSL library`).

This window lets us select what we want to do about line endings. The characters used to indicate the end of the line are different between Windows, Linux, and Mac OS. So the Git software bundle lets us decide how we want to handle these differences. The default is to store Windows line endings on the local files, but use Unix line endings in the files stored by Git (`Checkout Windows-style, commit Unix-style line endings`).

This works well when you're using your Windows computer to collaborate with others using a different OS. The second option (`Checkout as-is, commit Unix-style line endings`) is to keep the line endings unchanged when copying the files locally, and use Unix line endings for the files stored in Git. This would work well if you're using a Unix like OS, or if you're only editing through Unix-like editors on Windows. The third option (`Checkout as-is, commit as-is`) is to not do any conversions. This option doesn't work well if you're trying to work with people using a different OS. So it's only recommended if all your collaborators will be running the same OS as you. All right, let's keep the first one selected and move on.

This window lets us configure the terminal emulator. The software bundle ships with its own terminal emulator (`Use MinTTY`) that includes a bunch of nice features, like better Unicode support and a long history of commands that we can scroll. We'll keep that one selected, but if you feel like you're more comfortable with the Windows default console window, you can choose that one (`Use Windows' default console window`).

All right, we have a few extra options that we can choose to enable if we want. We'll keep these as they are, that way we'll get the performance improvements of letting Git do file system caching (`Enable file system caching`), and we'll be able to use Git's credential manager (`Enable Git Credential manager`). We don't need to use symbolic links in our repositories (`Enable symbolic links`), so we'll leave that disabled.

Finally, we get one last prompt before installing, which lets us choose which experimental features we want to select. The features offered here will change with time, as some get adopted and others get dropped. You can decide whether you want to work on the bleeding edge, or you'd rather go with what's already well tested. We aren't feeling too adventurous today, so we won't enable the experimental features. We'll just click install here to get our installation started.

All right, Git is installing on our machine. It'll take a while for it to complete. Once it's done, we'll have all the Git goodness at our fingertips. In case you need a bit more help, you can find links for more information regarding Git installation in our next reading.

1.4 Using Git

1.4.1 First Steps with Git

When starting with Git, there are a bunch of concepts that we need to learn to understand how things are organized and how our files are tracked. Over the next few videos, we'll introduce some of the main Git concepts. If any of these seem confusing at first don't panic, we'll dive into all of them as we expand our Git knowledge.

Let's start by setting some basic configuration. Remember when we said that a VCS tracks who made which changes, for this to work, we need to tell Git who we are. We can do this by using the Git config command and then setting the values of `user.email` and `user.name` to our email and our name like this:

- `git config --global user.email "redtower.soft@gmailc.com"`
- `git config --global user.name "redtower"`

We use the dash dash global flag to state that we want to set this value for all git repositories that we'd use. We could also set different values for different repositories. With that done, there are two ways to start working with a git repository. We can create one from scratch using the `git init` command or we can use the `git clone` command to make a copy of a repository that already exists somewhere else. We'll talk about remote repositories later in the course. For now, let's start by creating a new directory and then a git repository inside that directory (inside a folder call `git init`).

So when we run `git init` we initialize an empty git repository in the current directory. The message that we get mentions a directory called `.git` (`Initialized empty Git reporitory in ...`). We can check that this directory exist using the `ls-la` command which lists files that start with a dot. We can also use the `ls-l .git` command to look inside of it and see the many different things it contains. This is called a **Git directory**. You can think of it as a database for your Git project that **stores the changes and the change history**. We can see it contains a bunch of different files and directories. We won't touch any of these files directly, we'll always interact with them through Git commands. So whenever you clone a repository, this git directory is copied to your computer. Whenever you run `git init` to create a new repository like we just did, a new git directory is initialized.

The area outside the git directory is the **working tree**. The working tree **is the current version of your project**. You can think of it like a workbench or a **sandbox where you perform all the modification you want to your file**. This working tree will contain all the files that are currently tracked by Git and any new files that we haven't yet added to the list of track files.

Right now our working tree is empty. Let's change that by copying the disk usage that py file that we saw in an earlier video into our current directory (`cp ../disk_usage.py`). We now have file and a working tree but it's currently untracked by Git. To make Git track our file, we'll add it to the project using the git add command passing the file that we want as a parameter (`git add disk_usage.py`). With that,

we've added our file to the `staging area`. The area which is also known as the index is a file maintained by Git that **contains all of the information about what files and changes are going to go into your next command**. We can use the `git status` command to get some information about the current working tree and pending changes. Let's check that one out.

We see that our new file is marked to be committed (`Changes to be committed`), this means that our change is currently in the staging area. To get it committed into the `.git`directory, we run the `git commit` command. Let's try that now.

When we run this command, we tell Git that we want to save our changes. It opens a text editor where we can enter a `commit message`. If you want, you can change the editor used to your preferred editor. In our case, this computer has `nano` configured as a default editor. The texts that we get tells us that we need to write a commit message and that the change to be committed is the new file that we've added. We'll deep dive into commit messages later. For now, let's enter a simple description of what we did which was to add this one file and then exit the editor saving our commit message and with that we've created our first git commit. Up next, we'll talk more about the life cycle of each track file in a git repository.

1.4.2 Tracking Files

In our last video, we mentioned that any Git project will consist of three sections. The **Git directory**, the **working tree**, and the **staging area**. The Git directory **contains the history of all the files and changes**. The working tree contains the **current state of the project, including any changes that we've made**. And the staging area contains the **changes that have been marked to be included in the next commit**.

This can still be confusing. So it might be helpful to think about Git as representing your project. Which is the code and associated files and a series of **snapshots**. **Each time you make a commit, Git records a new snapshot of the state of your project at that moment**. It's a picture of exactly how all these files looked at a certain moment in time. Combined, these snapshots make up the history of your project, and it's information that gets stored in the Git directory. Now, let's dive into the details of how we track changes to our files. When we operate with Git, our files can be either **tracked** or **untracked**. **Tracked files are part of the snapshots**, while untracked files aren't a part of snapshots yet. This is the **usual case for new files**. Each track file can be in one of three main states, **modified, staged or committed**. Let's look at what each of these mean.

If a file is in the **modified state**, it means that we've made changes to it that we haven't committed yet. The changes could be **adding, modifying or deleting the contents of the file**. Git notices anytime we modify our files. But won't store any changes until we add them to the staging area.

So, the next step is to stage those changes. When we do this, our modified files become stage files. In other words, the changes to those files are ready to be commit-

ted to the project. All files that are staged will be part of the next snapshot we take. And finally, when a file gets committed, the changes made to it are safely stored in a snapshot in the Git directory. This means that typically a file tracked by Git, will first be modified when we change it in any way. Then it becomes staged when we mark those changes for tracking. And finally it will get committed when we store those changes in the VCS.

Let's see this in action in our example Git repo. First, let's check the contents of the current working tree using `ls -l`. And then the current status of our files using the `git status` command. When we run Git status, Git tells us a bunch of things, including that we're on the master branch (`On branch master`). We'll learn about branches later in the course. For now, notice how it says that there's nothing to commit (`nothing to commit, working tree clean`) and that the working tree is clean. Let's modify a file to change that.

For example, we'll just add periods at the end of the message that our script presents to the user (in case text editor is atom you can write `atom disk_usage.py` and those periods).

So, now that we've made the change, let's call Git status again and see the new output Again, Git tells us a lot of things (`Changes not staged for commit` and `modified: disk_usage.py` in red), including giving us some tips for commands that we might want to use. These tips can come in real handy, especially when we're familiarizing ourselves with Git. See how the file we changed is now marked as modified? And that it's currently not staged for commit?

Let's change that by running the Git add command, passing the `disk_usage.py` file as a parameter (`git add disk_usage.py`).

When we call `git add`, we're telling Git that we want to add the current changes in that file to the list of changes to be committed if we call again `git status` we get `Changes to be committed` and `modified: disk_usage.py` in green. This means that **our file** is currently part of the `staging area`, and it **will be committed** once we run the next Git command, `git commit`. In this case, instead of opening up an editor, let's pass the commit message using the dash m flag (`-m`), stating that we added periods at the end of the sentences `git commit -m 'Add periods to the end of sentences'`.

So, we've now committed our stage changes. This creates a new snapshot in the Git directory. The command shows us some stats for the change made. Let's do one last status check (we get `nothing to commit, working tree clean`).

We see that once again, **we have no changes to commit**. Because the change we made has gone through the full cycle of modified, staged and committed. So to sum up, we work on modified files in our working tree. When they're ready, we staged these files by adding them to the staging area. Finally, we commit the changes sitting in our staging area, which takes a snapshot of those files and stores them **in the database that lives in the Git directory**. If the way Git works is not totally clear yet, don't worry. It will all sink in with a bit more practice. In our next video, we'll put this all together and go over the typical workflow when working with Git

1.4.3 The Basic Git Workflow

In earlier videos, we discussed some of the basic concepts involved in working with Git. We saw that each repository will have a Git directory, a working tree, and a staging area. And we called out that files can be in three different states, modified, staged, and committed. Let's review these concepts one more time by looking at the normal workflow when operating with Git on a day to day basis. First, all the files we want to manage with Git must be a part of a Git repository. We initialize a new repository by running the `git init` command in any file system directory. For example, let's use the `mkdir` command to create a directory called `scripts` (`mkdir scripts`, `cd scripts` and `git init`), and then change into it and initialize an empty Git repository init. Our shiny new Git repository can now be used to track changes to files inside of it. But before jumping into that, let's check out our current configuration by using the `git config -l` command (we get information such as `user.emal`, `user.name` etc).

There's a bunch of info in there, and we won't cover all of it. For now, pay special attention to the `user.email` and the `user.name` lines, which we touched on briefly in an earlier video. This information will appear in public commit logs if you use a shared repository. For privacy reasons, you might want to use different identities when dealing with your private work and when submitting code to public repositories. We'll include more details about changing this information in our next reading. Okay, our repo is ready to work, but it's currently empty. Let's create a file in it, we'll start with a basic skeleton for a Python script, which will help us demonstrate the Git workflow. As with any Python script, we'll start with the `shebang` line (`#!/usr/bin/env python3`). For now, we'll add an empty main function, which we'll fill in later. And at the end, we'll just call this main function.

```
#!/usr/bin/env python3

def main():
pass

main()
```

All right, we've created our file. This is a script that we'll want to execute, so let's make it executable (`chmod +x all_checks.py`). And then let's check the status of our repo using `git status` command.

As we called out before, when **we create a new file in a repository, it starts off as untracked**. We can make all kinds of changes to the file, but until we tell Git to track it, Git won't do anything with an untracked file. Do you remember what command we have to use to make Git track our file? That's right, we need to call the `git add` command (`git add all_checks.py`).

This command will immediately move a new file from untracked to stage status. And as we'll see later, it will also change a file in the modified state to staged state. Remember that when a file is staged, it means it's been added to the staging area and it's ready to be committed to the Git repository. To initiate a commit of staged files,

we issue the `git commit` command. When we do this, **Git will only commit the changes that have been added to the staging area**, untracked files or modified files that weren't staged will be ignored.

Calling `git commit` with **no parameters will launch a text editor**, this will open whatever has been set as your default editor.

If the default editor is not the one you'd like to use, there are a bunch of ways to change it. We'll include more info about changing the default editor in the next reading. For now, let's edit our message with `GNU nano` , which is the current default for this computer. We'll say that our change is creating an empty `all_checks.py` file, then save and exit.

Voila! We've just recorded a snapshot of the code in our project, which is stored in the Git directory. Remember that every time we commit changes, we take another snapshot, which is annotated with a commit message that we can review later.

Okay, that's how we add new files, but usually we'll modify existing ones. So let's add a bit more content to our script to see that in action. We'll add a function called `check_reboot`, that will check if the computer is pending a reboot. To do that, we'll check if the `run/reboot-required` file exists.

```
#!/usr/bin/env python3
import os

def check_reboot():
return os.path.exist("/run/reboot-required")

def main():
pass

main()
```

This is **a file that's created on our computer when some software requires a reboot**. And of course, since we're using `os.path.exists`, we need to add import os to our script (`import os`).

All right, we've added a function to our file. Let's check the current status using `git status` again (we get `Changes not staged for commit`).

Our file's modified, but not staged. To stage our changes, we need to call git add once again (`git add all_checks.py`).

Okay, our changes our now staged. What do we need to do next? You got it, we have to call `git commit` to store those changes to the Git directory. This time, we'll use the other way of setting the commit message. We'll call `git commit -m`, and then pass the commit message that we want to use. So in this case, we'll say that we've added the `check_reboot` function (`git commit -m 'Add check_reboot function'`).

With that, we've demonstrated the basic Git workflow. We make changes to our files, stage them with `git add`, and commit them with git commit. Are you starting to feel more comfortable with this process, and see how it fits within the rest of your tasks?

If there's anything that's not totally clear yet, remember, that the only way to get familiar with these concepts is practice. Feel free to try these examples out on your computer as we go along, until you get comfortable with these commands.

Up next we'll talk more about how to write useful commit messages.

1.4.4 Anatomy of a Commit Message

In earlier videos, we saw how we can commit snapshots of changes to the Git repository. Let's now talk a little bit more about what makes a good commit message. Writing a clear informative commit message is important when you use a VCS, future you or other developers or IT specialists who might read the commit message later on will really appreciate the contextual information as they try and figure out some of the parts of the code or configuration. So what makes a good commit message? It can be helpful to keep your audience in mind when you write commit messages. What would someone reading a message weeks or months from now want to know about the changes you've made? What might be especially important or tricky to understand about them? Is there extra information that might help the reader out, like links to design documents or tickets in your ticketing system? Similarly to how style guides exist for writing code, your company might have specific rules for you to follow when you write commit messages. Even if they don't, it's good to use a few general guidelines to make sure your commit messages are as clear and useful as possible. A commit message is generally broken up into a few sections. The first line is a short summary of the commit followed by a blank line. This is followed by a full description of the changes which details why they're necessary and anything that might be especially interesting about them or difficult to understand. When you run the git commit command, Git will open up a text editor of your choice so you can write your commit message. A good commit message might look something like this.

```
Provide a good commit message example

The purpose of this commit is to provide an example of a hand-crafted,
artisanal commit message. The first line is a short, approximately 50 characte
summary, followed by an empty line. The subsequent paragraphs are jam-packed
with descriptive information about the change, but each line is kept under 72
characters in length.

If even more information is needed to explain the change, more paragraphs can
be added after black lines, with links to issues, tickets, or bugs. Remember
that future you will thank current you for your thoughtfulness and foresight!

#Please enter.....
#...
```

So the first line is usually kept to about 50 characters or less. The line contains a short description of what the commit changes are about. After the first line, comes an empty line, and the rest of the text is usually kept under 72 characters. This text is intended to provide a detailed explanation of what's going on with the change. It can reference bugs or issues that will be fixed with the change. It can also include links to more information when relevant. The line limits can be annoying but they help in making the commit message be more digestible for the reader. There's a git command used to display these commit messages called `git log`. This command will do any line wrapping for us. Which means that if we don't stick to the recommended line wrapping, long commit messages will run off the edge of the screen and be difficult to read.

Now, take a look at the lines in the commit message that start with the pound symbol #. Just like in Python, this symbol indicates that these lines are comments and won't get included in the commit message. Git shows them to us whenever we're writing a commit message as a reminder of what files were about to commit.

Sometimes it can be tempting to just write something short like update, change or fix as the description of our commit messages. Don't do it. It's super frustrating to go back to repositories history and discover that there's not enough context to understand what was changed and why. **It takes only a few more seconds to write a better description. This can be invaluable down the line**. Following these guidelines can help make your commit message really useful, and the investment of work now will really pay off later. If you're interested in learning more about git commit style, there are plenty of resources out there to read including the Linux kernel documentation itself along with impassioned opinions from other developers. We'll include links to all of them in the next reading. We said that we can check the history of the commits of our project using the git log command. Let's go back to our example scripts directory where we performed two commits and check out what git log has to say about those two commits.

Fig. 1.10 Using `git log`

```
user@ubuntu:~/scripts$ git log
commit 4a713b8df5cc6dcde7a7435e103952fe03866cc2 (HEAD -> master)
Author: My name <me@example.com>
Date:   Sun Jan 5 15:34:16 2020 -0800

    Add a check_reboot function

commit 46a2a6fcb685471c9539e7025700eae4927fec3e
Author: My name <me@example.com>
Date:   Sun Jan 5 15:26:16 2020 -0800

    Create an empty all_checks.py
user@ubuntu:~/scripts$
```

Take a look at what git tracks as part of the log. It's packing a lot of information in just a few lines. The first thing listed for each commit is its **identifier** (`commit 4a713b...` and `commit 46a2a6...`), which is a long string of letters and numbers that uniquely identify each commit. The first commit in the list also says that the head indicator is pointing to the master branch (`HEAD -> master`). If this is gibberish to you, don't worry. We'll talk more about what a head and master

means in later videos. For each commit, we see the name and the email of the person who made the commit which is indicated as the author. Then we get the date and time the commit was made. Finally the commit message is displayed. Our commit messages are very brief as we're just starting to work on our repository. **As the work we do becomes more complex, we'll probably write longer descriptions with a lot more detail**. Coming up, we've got a cheat sheet for you that lists all the git commands we've seen up until now along with links to additional information that you might find useful. After that, head over to the next practice quiz to help make sure you've wrapped your head around all these new concepts.

Chapter 2
Week 2

2.1 Advanced Git Interaction

2.1.1 Intro to Module 2: Using Git Locally

Welcome back. Up till now, we've talked about what version control is, why it's necessary, and how we might benefit from it in diverse context. We also started learning some basic Git commands, and procedures, nice job. Over the course of the next videos, we'll go into much more detail about what we can do with Git. These are Git's greatest hits.

We'll start by learning some handy shortcuts and looking into how we can get more info out of our version control system. Then we'll experience the true power of Git by seeing how we can undo some of our changes. The ability to revert previous changes is one of the most useful aspects of version control systems. Depending on what needs to be undone, there's a bunch of different techniques that we can use in Git. We can discard the changes made to a file, fix a commit that was incorrect and even roll back our project to an older snapshot. We'll look into all these techniques and dive into when to use each of them.

Finally, we'll check out yet another important concept, Branches. We can use branches to work on an experimental feature without affecting the main code of our project. Support separate versions of a program that can't be merged together and much more. We'll dive into **what branches are, when and how to use them and how to deal with merge conflicts**.

Admittedly, some of these concepts can get pretty tricky, so we really recommend that you follow along in your computer, experiment with the commands that we show, and try things on your own so you feel comfortable with these techniques. Remember that you can always go back and review any earlier videos if something isn't totally clear. If after all this you still feel stuck, be sure to use the discussion forums to ask for help as well. So you're ready to learn more about advanced features in Git, let's get to it.

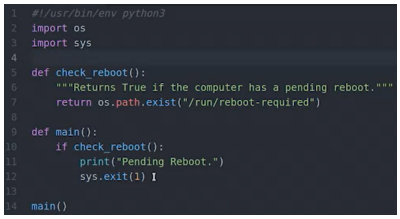
2.1.2 Skipping the Staging Area

When we covered the basic Git workflow, we called out that the process is usually to make changes, stage them, and then commit them. The separate step between staging and committing allows us to stage several changes in one commit. But if we already know that the current changes are the ones that we want to commit, we can skip the staging step and go directly to the commit. No dress rehearsals. We do this by using the dash a-flag to the git commit command (`git commit -a`). This flag automatically stages every file that's tracked and modified before doing the commit letting it skip the `git add` step.

At first, you might think that `git commit -a` is just a shortcut for `git add` followed by `git commit` but **that's not exactly true**. Actually `git commit -a` **doesn't work on new files because those are untracked**. Instead, `git commit -a` is a shortcut to stage any changes to tracked files and commit them in one step. **If the modified file has never been committed to the repo, we'll still need to use git add to track it first**.

So let's make a change to our example script from an earlier video and try out this new flag. We'll now modify our main function and make it call the check reboot function that we wrote before. If a reboot is pending, we'll print a message and then exit our program with an exit status of one. Since we're using the sys module, we'll need to import it (Fig. 2.1).

Fig. 2.1 Using `Function modified.`



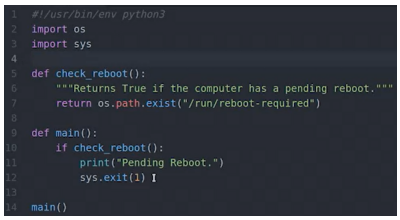
All right. Now that we've made the change, we're ready to try out the new `-a` flag. We'll also use the `-m` flag to add the commit message directly. This time, we'll say that we're calling check underscore reboot and exiting with one on the error condition (`git commit -a -m` Call `check_reboot` from main, exit with 1 on error). Success. These shortcuts are useful when making small changes that we know we'll want to commit directly without keeping them in the staging area and having to write long and complex descriptions.

Keep in mind that **when you use the** `-m` **shortcut, you can only write short messages** and can't use the best practices regarding commit descriptions that we talked about earlier. **So it's best reserved for truly small changes** that don't require extra context or explanation, short and sweet.

Heads up, when you use the `-a` shortcut, you skip the staging area. Meaning, you can't add any other changes before creating the commit. So you need to be sure that

you've already included everything you want to include in that commit. In the end, using a shortcut like -a is just like using the regular commit workflow. The commit will show up in the `log` along with the message just as usual. Let's check that out.

Fig. 2.2 Using `git log`.



See how our latest commit was added to the top of the list of commits and notice how the `Head` indicator has now moved to the latest commit (red line). You might be wondering, what is this `Head` and where is it heading? We'll keep coming across it. So let's clarify.

Git uses **the head alias to represent the currently checked out snapshot of your project**. This lets you know what the contents of your working directory should be. **In this case, the current snapshot is the latest commit in the project**. We'll soon learn about branches. In that case, (if we consider branches) **head can be a commit in a different branch of the project**. We can even use git to go back in time and have head representing old commit from before the latest changes were applied. In all cases, head is used to indicate what the currently checked out snapshot is. This is how git marks your place in the project. **Think about it as a bookmark that you can use to keep track of where you are**. Even if you have multiple books to read, the bookmark allows you to pick up right where you left off.

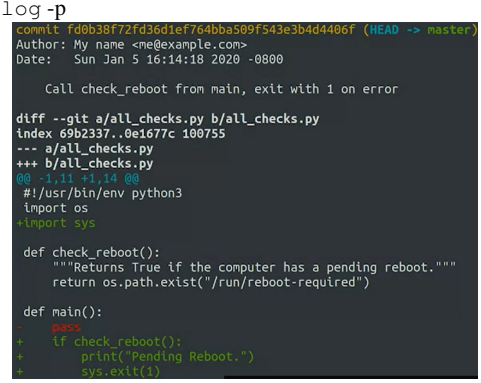
When you run git commands like diff, branch, or status, git will use the head bookmark as a basis for whatever operation it's performing. We'll see `Head` used when we learn how to undo things and perform rollbacks. We'll talk more about branches in later videos. As a shortcut, **it's generally easy to think of head as a pointer to the current branch**, although it can be more powerful than that. Next, we'll dig into how we can get more information about our changes, both before and after we commit them.

2.1.3 Getting More Information About Our Changes

We've seen how git log shows us the list of commits made in the current Git repository. By default, it prints the commit message, the author, and the date of the change. This is useful, but if we're combing through a history of changes in a repo to try and find what caused the latest outage, we'll probably also need to look at the actual lines that changed in each commit. To do this with `git log`, we can use the `-p`

flag. The `p` comes from `patch`, because using this flag gives us the patch that was created. Let's try it out (Fig. 2.3).

Fig. 2.3 Using `git log` -p



The format is equivalent to the `diff -u` output that we saw on an earlier video.

It shows added lines (green color) with plusses and remove lines (red color) with dashes. Because the amount of text is now longer than what fits on your screen, Git automatically uses a paging tool that allows us to scroll using page up, page down, and the arrow keys.

We still have one commit below the other, but now each commit takes up a different amount of space, depending on how many lines were added or removed in that commit.

Using this option, we can quickly see what changes were made to the files in our repository. This can be especially useful if we're trying to track down a change that recently broke our tools. If we don't want to scroll down until we find the commit that we're actually interested in, another option is to use the `git show` command. This command takes a `commit ID` **as a parameter**, and will display the information about the commit and the associated patch. We'll talk more about `commit IDs` in a later video. But for now, remember that this is an identifier that we see next to the word commit in the log.

Let's check this out by first listing the current commits in the repo and then calling `git show` (Fig. 2.5) for the second commit in the list. First, I'm going to exit out by pressing `q`.

We've shown how we can use `git log` for listing commits , and `git log -p` for showing the associated patches. Another interesting flag for `git log` is the `--stat` flag. This will cause `git log` to **show some stats about the changes in the commit**, like which files were changed and how many lines were added or removed. Let's try it with our repo (`git log --stat`).

There are a bunch of other options to git log, so we won't cover them all. You can always use the reference documentation or the manual pages to find out more.

Fig. 2.4 Using `git log`

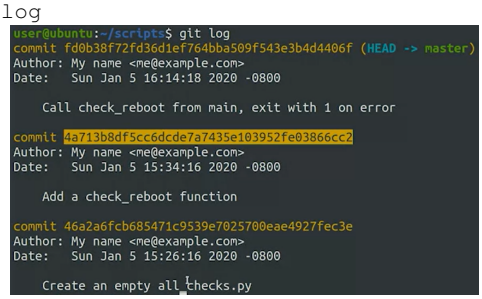
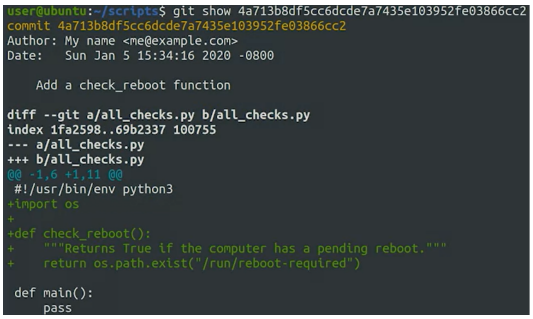


Fig. 2.5 Using `git show`



And as we called up before, you don't need to memorize any of this, you'll learn the different commands and flags by using them.

The important thing to remember is that all the information is stored in the repository and you have it at your fingertips when you need it. You're welcome.

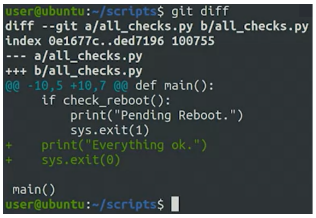
Now, what about changes that haven't been committed yet? Until now, whenever we've made changes to our files, we've either added them to the staging area with `git add` and committed them with `git commit`, or committed them directly using `git commit -a`. This works fine, but it means we have to know exactly which changes we've made. Sometimes it can take a while until we're ready to commit. We call these commitment issues. Just kidding. But imagine you've been working on adding a new complex feature to a script and it requires thorough testing.

Before committing it, you need to make sure that it works correctly. Check that all the test cases are covered and so on and so on. So while doing this you find bugs in your code that you need to fix. It's only natural that by the time you get to the commit step you don't really remember everything you changed. To help us keep track git gives us the **git diff** command.

Let's make a new change to our script and then try this command out. We'll add another message to the user to say that everything is okay when the check is successful and then exit with 0 instead of 1.

Okay, we've made the change. Let's now save it and check out what **git diff** shows us (Fig. 2.6).

Fig. 2.6 Using `git diff`



Again, this format is equivalent to the `diff -u` output that we saw in an earlier video.

In this case, we see that the only change is the extra lines that we've added (green color). If our change was bigger and included several files, we could pass a file by parameter to see the differences relevant to that specific file instead of all the files at the same time. Something else we can do to review changes before adding them is to use the `-p` flag with the git add command.

When we use this flag, **git will show us the change being added and ask us if we want to stage it or not** (we will get `Stage this hunk [y,n,q,a,d,e,?]?`).

This way we can detect if there's any changes that we don't want to commit. Let's try that one out.

We've staged our change and it's **now ready to be committed**.

If we call `git diff` again, **it won't show any differences, since git diff shows only unstaged changes by default**. Instead, we can call `git diff --staged` to see the **changes that are staged but not committed**. With this command, we can see the actual stage changes before we call git commit.

Let's commit these changes now so that they aren't pending anymore. We'll say that we've added a message when everything's okay (`git commit -m` 'Add a message when everything is ok').

Nice, and with that, we've learned a bunch of different ways to get more information about our changes. We've covered a lot of ground, so take the time to review and practice these commands if they don't make sense yet. Up next, we'll look into what happens when we want to delete or rename files in our repository. It's time to do a little housekeeping.

2.1.4 Deleting and Renaming Files

Let's say that you've decided to clean up some old scripts and want to remove them from your repository.

Or you've done some refactoring, which makes that particular file, obsolete. You can remove files from your repository with the `git rm` command, which will stop the file from being tracked by git and remove it from the git directory.

File removals go through the same general workflow that we've seen. So you'll need to **write a commit message as to why you've deleted them**. Let's try this out in our checks repository that contains a file we decided we actually don't want.

Well first look the contents of the directory with `ls`, then delete the file with `git rm`, then check the contents with `ls` again, and finally check the status with `git status` (First `ls -l`, then `git rm process.py` , `ls -l` and finally `git status`).

So, we see that by calling `git rm`, **the file was deleted from the directory, and the change was also staged to be committed in our next commit**. Let's do that now, by calling `git commit` and sending a message indicating that we've deleted the unneeded file (`git commit -m 'Delete unneeded processes file'`).

As usual, we get a bunch of stats when we do the commit . Check out all the deletions that reported.

These are all lines in the file that are no longer there (`1 file changed, 23 deletions (-)`). And it states the file itself was deleted.

What if you have a file that isn't accurately named? This can happen. For example, if you start writing a script that you thought would only do one thing, and then expands to cover more use cases. Or conversely, if you named your script thinking that it would be very generic, but it ends up being more specific. You can use the `git mv` command to **rename files** in the repository. Let's rename our existing script to `check_free_space.py` and check what `git status` has to say about that (`git mv disk_usage.py check_free_space.py` and then `git status`).

The status shows us that **the file was renamed** (in green color `renamed: disk_usage.py -> check_free_space.py`) and clearly displays the old and new names. As with the previous example, **the change is staged, but not committed**. Let's commit it by calling `git commit` once again (`git commit -m 'New name for disk_usage.py'`).

The `git mv` command works in a similar way to the `mv` command on Linux and so can be used for both moving and renaming. If our repository included more directories in it, we can use the same git mv command to move files between directories.

As you can probably tell from our examples, the output of `git status` is a super useful tool to help us know what's up with our files. It shows us which files have tracked or untracked changes, and **which files were added, modified, deleted or renamed**. It's important that the output of these commands stays relevant to what we're doing. If we have a long list of untracked files, we might lose an important change in the noise. If there are files that get automatically generated by our scripts, or our operating system generates artifacts that we don't want in our repo, we'll want to ignore them so that they don't add noise to the output of git status. To do this, we can use the `git ignore` file.

Inside this file, we'll specify rules to tell git which files to skip for the current repo.

For example, if we're working on an OSX computer, we'll probably want to ignore the dot DS store file, which is automatically generated by the operating system.

To do this, we'll create a `.gitignore` file containing the name of this file (`echo .DS_STORE > .gitignore`).

Remember that the **dot prefix in a Unix-like file system indicates that the file or directory is hidden** and won't show up when you do the normal directory listing. That's why we have to use `ls-la` to see all files.

We've added a git ignore file to our repo but we haven't committed it yet. This file needs to get tracked just like the rest of the files in the repo. Let's add it now (`git add .gitignore`).

Looks great. We've learned a lot of new commands and techniques related to git in these past videos. If anything isn't clear at this point, now's a good time to experiment on your own and share any challenges in the discussion forums. Coming up, we put together a cheat sheet listing all the new commands and options. After that, a quiz let you practice these concepts. Now, go on, get.

2.1.5 Reading: Advanced Git Cheat Sheet

- `git commit -a` : Stages files automatically.
- `git log -p` : Produces a patch text.
- `git show` : Show various objects.
- `git diff` : (Similar to Linux `diff` command) Show the differences in various commits.
- `git diff --staged` : An alias to `--cached`, this will show all staged files compared to the named commit.
- `git add -p` : Allows a user to interactively review patches to add to the current commit.
- `git mv` : (Similar to Linux `mv` command) moves a file.
- `git rm` : (Similar to Linux `rm` command) deletes or removes a file.

2.2 Undoing Things

2.2.1 Undoing Changes Before Committing

Being able to revert our changes is one of the most powerful features offered by version control systems. There's a bunch of different techniques available depending on which changes we need to undo. In this video and the next few coming up, we'll talk about the most common ways to revert changes in Git and when to use each approach. For example, you might find yourself in a situation where you've made a bunch of changes to a file but decide that you don't want to keep them. You can change a file back to its earlier committed state by using the `git checkout` command followed by the name of the file you want to revert (`git checkout`

`filename`). Speaking of, let's try this out using our scripts repository. We'll edit our `all_checks.py` script and remove the `check_reboot` function, then save and go back to the command line.

Cool. We've made our change. Let's try our script and see what happens.

By deleting that function, we've actually broke the script (it does not run). Let's see what git status has to say about this. As expected, we see that **our file is modified** and the **changes aren't staged yet**. Check out how git gives us a couple helpful tips on what to do now. We can run `git add` to **stage** our changes or we can run `git checkout` to **discard** them. If you need help remembering what this command does, think of it this way, you're checking out the original file from the latest storage snapshot. Let's do that now. We'll check out at the original file and then take a look at what git status has to say about it and finally retry our script (`git checkout all_checks.py`).

Looks like we have a typo. Let's go back and fix it.

Done and done. With that, we've demonstrated how we can use `git checkout` **to revert changes to modify files before they get staged**. This command will restore the file to the latest storage snapshot, which can be either committed or staged. So **if you've made additional changes to a file after you've staged it, you can restore the file to the earlier stage version**. If you need to check out individual changes instead of the whole file, you can do that using the dash p (`-p`) flag. This will ask you change by change if you want to go back to the previous snapshot or not. That's it for undoing unstaged changes. What if you added the changes to the staging area already? Don't stress.

If we realize we've added something to the staging area that we didn't actually want to commit, we can unstage our changes by using the `git reset` command. Staging changes that we don't actually intend to commit happens all the time. Especially if we use a command like git add star (`git add *`), where the star is a file glob pattern used in Bash that expands to all files. This command will end up adding any change done in the working tree to the staging area.

While sometimes that might be what we want, it can also lead to some surprises. Let's try it out with an example. First, we'll pretend we're trying to debug a problem in our script. For that, we create a temporary file with the output of our script (`./all_checks.py > output.txt`). Then, we'll add all unstaged changes in our working tree using `git add *`. Finally, check the status using `git status`. We can see that this output file, which was supposed to be a temporary file for debugging, has now been staged in our repo (`output.txt`) but **we didn't want to commit it**. Conveniently, the **git status** command tells us **how to unstage the file right there in the output**. The example output mentions the `head` alias. Remember what that means? That's right. It's the current checked out snapshot. So by running the suggested command, we're resetting our changes to whatever's in the current snapshot. Let's try it out (`git reset HEAD output.txt`).

The file `output.txt` is once again untracked in our working tree and no longer staged. You can think of reset as the counterpart to add. With add, you can well add changes to the staging area. With reset, you remove changes from the staging area.

You can use `git reset -p` to get git to ask you which specific changes you want to reset. Get it?. But wait, let's remember to commit our typo fix.

With that, we've seen how we can revert unstaged and stage changes. But **what if you've already created a commit with the changes that you want to undo?** Great question. That's coming up in the next video.

2.2.2 Amending Commits

In general, we try to make sure our commits include all the right changes and descriptions. But we're all human and we make mistakes. It's not uncommon for developers and IT specialists to realize that there is an error in a recent commit, which is why it's important to know how to take action and fix it. Let's say you just finished committing your latest batch of work, but **you've forgotten to add a file that belongs to the same change**. You'll **want to update the commit to include that change**. Or maybe the files were correct, but you realize that **your commit message just wasn't descriptive enough**. So you want to fix the description to add a link to the bug that you're solving with that commit. What can you do? We can solve problems like these using the `--amend` option of the `git commit` command. When we run `git commit --amend`, git will take whatever is currently in our staging area and run the `git commit` workflow to overwrite the previous commit. Let's see this in an example. We'll go to our scripts directory and create two new files using the `touch` command. Then list the contents of the directory using `ls` at our Python script and commit it saying that we've added two files.

- `cd scripts/`
- `touch auto-update.py`
- `touch gather-information.sh`
- `ls -l`
- `git add auto-update.py`
- `git commit -m 'Add two new scripts'`

As you can see, the message printed by git says that only one file was added `1 file changed, 0 insertions(+), 0 deletions(-)`. Our **commit message said that we added two files, but we forgot to add one** of them. Ouch. Don't panic. We can fix it. We'll start by adding the missing file and then amending our commit.

- `git add gather-information.sh`
- `git commit --amend`

We call `git commit --amend` and an editor opened up showing the commit message and the stats about the commit that we're working with. The list of added files for this commit now includes both files that we wanted to add. Yay. Now that the files have been added, we can also improve our initial commit message which

was a bit too short. We'll keep the existing description as the first sentence of our commit, and then add a line of description about the intended purpose of each file.

```
Add two new scripts.

gather-information.sh will be used to collect information in case of errors.
auto-update.py will be run daily to update computers automatically.
```

With that, our commit is ready to be amended. Let's save the new description as usual (`ctrl+o`, then `ENTER` and `ctrl+x`).

We've amended our previous commit to include both files and a better message. You could also just update the message of the previous commit by running the `git commit --amend` command with no changes in the staging area. An important heads up. While `git --amend` is okay for fixing up local commits, you **shouldn't use it on public commits**. Meaning, those that have been pushed to a public or shared repository. This is because using `--amend` **rewrites the git history removing the previous commit and replacing it with the amended one**. This can lead to some confusing situations when working with other people and should definitely be avoided. So remember, fixing up a local commit with amend is great and you can push it to a shared repository after you fixed it. But **you should avoid amending commits that have already been made public**. If this sounds confusing now, don't worry. We'll mention it again when we talk about collaborating with others through shared repositories. We've covered how to fix staged and unstaged changes, and how to fix a commit that was incomplete. Up next, we'll talk about what to do if you come across a bad commit that needs to be completely reverted.

2.2.3 Rollbacks

Fixing your work before you commit is good. But what happens if it's already been snapshotted by Git? Let's say you host to Git repository on a company server that contains all kinds of useful automation scripts that you and your coworkers use. One morning before coffee, you make a few changes to one of these scripts and commit the updated files. A few hours later, you start to receive tickets from users indicating some part of the script is broken. From the errors they describe, it sounds like the problem is related to your recent changes. Oh, you could look at the code you updated to see if you can spot the bug. But more tickets are pouring in and you want to fix the problem as fast as possible. You decided it's time for a rollback. There are a few ways to rollback commits in Git. For now, we'll focus on using the `git revert` command. Git revert doesn't just mean undo. Instead, **it creates a commit that contains the inverse of all the changes made in the bad commit in order to cancel them out**.

For example, if a particular line was added in the bad commit, then in the reverted commit, the same line will be deleted. This way **you get the effect of having undone the changes, but the history of the commits in the project remains con-**

sistent leaving a record of exactly what happened. So `git revert` **will create a new commit**, that is the opposite of everything in the given commit. We can revert the latest commit by using the `Head` alias that we mentioned before. Since we can think of `Head` as a pointer to the snapshot of your current commit, when we pass `Head` to the `revert` command we tell Git to rewind that current commit, makes sense? To check this out, we'll first add a faulty commit to our example repo.

```
cd scripts
atom all_checks.py  #Something was edited
git commit -a -m 'Add call to disk_full function'
```

We've added some code to our script. Let's save and commit this. So now, our code is committed. We didn't even test it which is a bad idea if you're doing this for real. You might have already spotted the problem with our code. This is where users start filing tickets and saying that things are broken, and so we run our script to see what happens.

Oops, we use the function that we forgot to define. Okay. It's rollback time. Let's get rid of this faulty code by typing `git revert head`. So once we issue that git revert command, we're presented with the text editor commit interface that we've all seen before.

```
Revert "Add call to disk_full function"

This reverts commit 5aab0fd....
```

In this case, we can see that git has automatically added some text to the command indicating it's a rollback. The first-line mentions that it's reverting the commit we just did called `Add call to disk full function`. The extra description even includes the identifier of the commit that got reverted. While we could use this description as is, it's usually a good idea to add an explanation of why we're doing the rollback. Remember that **the goal of these descriptions is to help our future selves understand why things happen**. In this case, we'll explain that the reason for the rollback is that the code was calling a function that wasn't defined.

```
Revert "Add call to disk_full function"
Readon for rollback: ...
This reverts commit 5aab0fd....
```

Once we're done entering the description, we can exit and save as usual. You'll notice the output that we get from the `git revert` command looks like the output of the `git commit` command. This is because `git revert` **creates a commit for us**. Since a revert is a normal commit, **we can see both the commit and the reverted commit in the log**.

Let's look at the last two entries in the log using dash P and dash two as parameters. As demonstrated before, the dash P parameter lets us see the patch created by the commit while the dash two perimeter limits the output to the last two entries. So in this log, we can see that when we called revert, git created a new commit that's

the inverse of the previous one. This removes the lines that we added in the previous commit. We can see that the original commit shows the lines we added by preceding them with a plus sign. The same line shows up with a minus sign in the newer commit message indicating that they were removed. Just like that, the bad commit is reverted and the error stopped. In this example, we reverted the latest commit in our tree. But what if we had to revert a commit that was done before that? Rev up your time machines because in the next video, we're turning back the clock big-time.

2.2.4 Identifying a Commit

So far we've used the head alias to specify the most recently checked out commit in our Git history. In our bad snapshot example, the error also happened to be in the most recently created commit, but errors can sometimes take a while to be detected. And so, we might need to revert other commits farther back in time. **We can target a specific commit by using its commit ID**. We've seen commit IDs a few times already. They show up when we're running the `git log` command, and we also saw the commit ID of the reverted commit in our last example. **Commit IDs are those complicated looking strings** that appear after the word commit in the log messages. Let's have a look at the latest log entry in our checks repo.

The commit ID is the 40 character long string after the word commit, you really can't miss it. This long jumble of letters and numbers is actually something called a `hash`, which is calculated using an algorithm called SHA1. Essentially, what this algorithm does is take a bunch of data as input and produce a 40 character string from the data as the output. In the case of Git, the input is all information related to the commit, and the 40 character string is the commit ID. Cryptographic algorithms like `SHA1` can be really complex, so we won't go too deep into what this means. If you're interested, you'll find links to more information in the next reading. Still you might be wondering, why on earth would you use a long jumble of letters as an ID for commit, instead of incrementing an integer, like 123, etc? To answer that, let's take a quick look at the reason why Git uses a hash instead of a counter, and how that hash is computed. Although SHA1 is a part of the class of cryptographic hash functions, Git doesn't really use these hashes for security. Instead, they're used to guarantee the consistency of our repository.

Having consistent data means that we get exactly what we expect. To quote Git's creator, Linus Torvalds, *you can verify the data you get back out is the exact same data you put in*. This is really useful in distributed systems like Git because everyone has their own repository and is transmitting their own pieces of data. Computing the hash keeps data consistent because it's calculated from all the information that makes up a commit. The commit message, date, author, and the snapshot taken of the working tree. The chance of two different commits producing the same hash, commonly referred to as a **collision**, is extremely small. So small, it wouldn't happen by chance. It'd take a lot of processing power to cause this to happen on purpose.

If you use a hash to guarantee consistency, you can't change anything in the Git commit without the SHA1 hash changing too.

Remember our discussion about fixing commits with the dash dash amend (`--amend`) command? Each time we amend a commit, the commit ID will change. This is why it's important not to use dash dash amend on commits that have been made public.

The data integrity offered by the commit ID means that **if a bad disk or network link corrupt some data in your repository, or worse, if someone intentionally corrupt some data, Git can use the hash to spot that corruptio**n. Aha, it will say, the data you've got isn't the data you expected, something went wrong.

Thank you, Detective Git, you've saved the day once again.

Okay, enough backstory. How can you use commit IDs to specify a particular commit to work with, like during a rollback? Let's look at the last two entries in our repo using the `git log -2` command.

Say we realized that we actually liked the previous name of our script, and so we want to revert this commit where we renamed it. First, let's look at that specific commit using `git show`, which we mentioned in an earlier video (`git show 30e70712....` 40 characters).

We've copied and pasted the commit ID that we wanted to display, and that works. Alternatively, we could provide just the first few characters identifying the commit to the command, and Git will be smart enough to guess which commit ID starts with those characters, as long as there's only one matching possibility. Let's try this out.

Two characters is not enough, but usually four to eight characters will be plenty.

Okay, now that we've seen how we can identify the commit that we want to revert, let's call the `git revert` (`git revert 30e70712`) command with this identifier. As usual, this will open an editor where we should **add a reason for the rollback**. In this case, we'll say that the previous name was actually better. Hooray for flip-flopping.

As we called out before, when we generate the rollback, **Git automatically includes the ID of the commit that we're reverting**. This is useful when looking at a repo with a complicated history that includes a lot of commits. Now, once w e save and exit the commit message, Git will actually perform the rollback and generate a new commit with its own ID. See how before the name of our commit the revert command already shows the first eight characters of the commit ID? Let's use git show to look at it.

```
$ git revert 30e70712
[master 7d1de19] Revert "Title of the reverted commit"
```

All right, we've managed to revert a commit that wasn't the most recent one. Well done, time travelers. Over the past several videos we've covered a bunch of ways to undo things in Git. Whether for unstaged changes, staged changes, amending commits, or rolling back changes. If anything still seems unclear, now's a great time to practice these commands on your local computer, try things out, and come up

with more examples of use cases you want to test. Up next, we've spun up a handy cheat sheet summarizing all the content we just covered. When you're ready, move on to the next quiz and put all this new knowledge into practice. We've been learning a lot of complex things over the past few videos, so once you're done with that quiz you should reward yourself for getting through this technical detail with a break. You earned it. Grab a coffee, tea, or snack and let the concepts settle for a bit. I'll meet you over the next video.

2.2.5 Reading: Git Revert Cheat Sheet

`git checkout` is effectively used to switch branches.

`git reset` basically resets the repo, throwing away some changes. It's somewhat difficult to understand, so reading the examples in the documentation may be a bit more useful.

There are some other useful articles online, which discuss more aggressive approaches to `resetting the repo`.

`git commit --amend` is used to make changes to commits after-the-fact, which can be useful for making notes about a given commit .

`git revert` makes a new commit which effectively rolls back a previous commit. It's a bit like an undo command.

There are a few ways you can rollback commits in Git.

There are some interesting considerations about how git object data is stored, such as the usage of sha-1.

Feel free to read more here:

- https://en.wikipedia.org/wiki/SHA-1
- https://github.blog/2017-03-20-sha-1-collision-detection-on-github-com/

2.3 Branching and Merging

2.3.1 What is a branch

Up until now, we've only briefly mentioned branches. You might have seen the text on branch master and commit messages, or you might remember that we talked about branches in the context of the head pointer. Branches are an important part of the Git work flow. So we'll branch out and explore them more thoroughly in the coming videos.

So what is a branch? What is it used for? In Git, a branch at the most basic level is just **a pointer to a particular commit**. But more importantly, it r**epresents an independent line of development in a project**. Of which the commit it points to is the latest link in a chain of developing history.

The default branch that Git creates for you when a new repository initialized is called `master`. All of our examples and development have taken place on this branch so far. The **master branch is commonly used to represent the known good state of a project**.

When **you want to develop a feature or try something new in your project, you can create a separate branch** to do your work without worrying about messing up this current working state.

If this seems confusing, maybe an analogy will help. You can think of a Git project as an assignment your teacher gives you in a class. You do all your work on the assignment in a set of notebooks, each notebook representing a different branch. You use some notebooks to jot down rough drafts in experiments, but you keep one notebook the master branch, in a tidy state and you copy the polish versions of these drafts into it. No doodling in the master note book, please. **Branches make it really easy to experiment with new ideas or strategies and projects**. When you want to add a feature or fix something, you can create a new branch and do your development there. **You can merge back into the master branch, when you've got something you like, or discard your changes without negative impact if they don't work out.**

In Git, branches are used all the time, as a part of the normal development workflow. As an example, think back to the problematic commit we fixed in an earlier video. We added a call to the `disk_full` function, but forgot to actually define the function. So we had to roll it back because our users we're seeing errors. Knowing what we know now, we could have done that work on a separate branch, maybe called something like `add_disk _full`. In that case, we could have iterated on our code there until it was working correctly, without it effecting the master branch. Only after the code is ready to be deployed, we would merge those changes back into the master branch.

In the next few videos, we'll look into how to create new branches and merge their content into the master branch. We'll also go over what to do if you run into any scary merge conflicts. Heads up, this is about to get pretty complicated. So make sure that you follow all of our exercises along in your computer and keep practicing coming up with new ways of using branches and merging, until you're comfortable with each of the steps we show.

2.3.2 Creating New Branches

As branches are essential to how work is done in git, there's tons of different ways to work with them. We can use the `git branch` command to **list, create, delete, and manipulate branches**. Running `git branch` by itself will show you a list of all the branches in your repository. Let's try it out in our checks repo.

```
$ cd checks
$ git branch
```

```
* master
```

Our list is looking pretty empty right now, but don't worry. Creating a branch is a snap. We do it by calling `git branch` with the name of the new branch Let's create a `new-feature` branch and then list the branches again with git branch.

```
$ git branch new-feature
$ git branch

* master
  new-feature
```

Our new branch was created based on the value of head. Remember that this might not necessarily be the master branch. The list we get shows that we're still on the master branch. We can tell because **the current branch is indicated in the command's output with an asterisk in a different color** (`* master`). We'll look into other things that git branch lets us do with branches later on, but right now we want to switch to a new branch. To do that, we'll need to use the `git checkout` command. We saw earlier how we can use git checkout to restore a modified file back to the latest commit. Checking out branches is similar in that, **the working tree is updated to match the selected branch including both the files and the git history**. If this seems a bit confusing at first, you're not alone. I also found it hard to wrap my head around it first. But rest assured that it will become clear after we use these commands for awhile. It might help to remember that **we use git checkout to check out the latest snapshot for both files and for branches**. All right. Let's switch to our new feature branch by calling git checkout new feature, and then listing our branches once again.

```
$ git branch new-feature
$ git branch

  master
* new-feature
```

Before we were working on the master branch but now that we've changed to our new branch, the star has moved to new feature. Creating a branch and switching to it immediately is a pretty common task. So common that git gives us a useful shortcut to create a new branch and to switch to it in a single command. We can use the `git checkout -b new branch` to do this. Take a look.

```
$ git checkout -b even-better-feature
Switched to a new branch 'even-better-feature'
```

See how the message says that we've switched to a new branch? Git created the new branch and switched to it in just one command. Super efficient. Now that we have our shiny new branch, let's create a new file in. We'll create a new Python3 file, that will include the usual shebang line and empty main function and a call to that function.

```
$ atom free_memory.py
(in GNU nano)
#!/usr/bin/env python3

def main():
pass

main()
```

This file is empty because it's only the beginning of our work. As it's in a separate branch, it's okay for it to not be finished yet. Let's save our file and commit it to the current branch now.

```
$ git add free_memory.py
$ git commit -m 'Add an empty free_memory.py'
```

We've added a commit in this branch and it's looking better. Let's check the last two entries in the log.

```
$ git log -2
commit 43618801... (HEAD -> even-beter-feature)
Author: ...
Date: ...

commit 7d1de193... (new-feature, master)
Author: ...
Date: ...
```

We see the last two commits in this branch. Notice how next to the latest commit ID, git shows that **this is where head is pointing to and that the branch is called** `even-better-feature`. Next to the previous commit, git shows that both the master and the new feature branches are pointing to that snapshot of the project. In this way, we can see that the `even-better-feature` branch is ahead of the master branch. With that, we've seen how we can create new branches and commit changes to them. You might say your knowledge of branches has grown. Up next, we'll learn even more things we can do to operate with branches. So stick around.

2.3.3 Working with Branches

In our last video, we created a new branch different than the `master` branch and added a `commit` to it. Let's check out the current status of our repo by calling `git status` and `ls -l`.

```
$ cd checks
```

```
$ git status
On branch even-better-feature
nothing to commit, working tree clean

$ ls -l
total 8
-rwxr-xr-x 1 user user 658 Jan  6 09:15 disk_usage.py
-rw-r--r-- 1 user user  57 Jan  6 09:46 free_memory.py
```

So we see that we're on a clean working tree in the `even-better-feature` branch, and that a new `free_memory.py` file is in our working tree. Let's now change back to the `master` branch using `git checkout master` and then lists the latest two commits there.

```
$ git log -2
commit 7d1de193... (HEAD -> master, new-feature)
Author: ...
Date: ...

commit bb9bd782...
Author: ...
Date: ...
```

When we switch to a different branch using `git checkout`, under the hood, **git changes where head is pointing**. Thanks to this checkout, **head went from pointing to the latest commit in the** `even_better_feature` **branch to the most recent commit of the master branch**.

The commit from `even_better_feature` doesn't show up at all, and the latest snapshot is the second entry we've seen before. Remember that **when we switch branches, git will also change files in our working directory** or working tree to whatever snapshot head is currently pointing at. Let's look at the current contents of our directory.

```
$ ls -l
total 4
-rwxr-xr-x 1 user user 658 Jan  6 09:15 disk_usage.py
```

The file `free_memory.py` isn't there. This demonstrates that **when we switch branches in git, the working directory and commit history will be changed to reflect the snapshot of our project in that branch**. When we check out a new branch and commit on it, those changes will be added to the history of that branch. Since `free_memory.py` was committed on another branch, it doesn't show up in the history or working directory of the master branch.

One thing to note after all this back and forth, is that **each branch is just a pointer to a specific commit in a series of snapshots**. It's very easy to create new

branches because there isn't any data that needs to be copied around. **When we switch to another branch, we check out a different commit and git updates both head and the contents of our working directory**. Head floats around with us. It's like a free spirit. What a head trip. Okay. We've now seen how to create and switch between branches.

So what if we **want to delete a branch that we don't need anymore**? We can do that by using git branch dash d (`git branch -d`). Let's first list the current branches in our repo and then get rid of the `new-feature` branch by calling `git branc -d new-feature`. Just like that, our branch was trimmed. We can check with another call to `git branch` that is not there anymore.

```
$ git branch
  even-better-feature
* master
  new-feature

  $ git branch -d new-feature
Deleted branch new-feature (was 7d1de19)

  $ git branch
     even-better-feature
  * master
```

If there are changes in the branch we want to delete that haven't been merged back into the master branch, git will let us know with an error.

Hopefully, git also gives us the command to run if we were sure that we wanted to delete the branch, even if it has unmerged changes. But we won't do that just yet. We actually want to merge those changes back into the repo first. How do we do that? It's all coming up in our next video

2.3.4 Merging

A typical workflow for managing branches in Git, is to create a separate branch for developing any new features or changes. Once the new feature's in good shape, we merge the separate branch back into the main trunk of code. Merging is the term that Git uses for **combining branch data and history together**. We'll use the `git merge` command, which lets us **take the independent snapshots and history of one Git branch, and tangle them into another**.

Let's try this out with our example branch from the last video. First, will **check that we're in master branch**, and then we'll call `git merge even-better-feature` to merge the `even-better-feature` branch into the master branch.

```
$ git branch
even-better-feature
```

```
* master

$ git merge even-better-feature
Updating
Fast-forward
free_memory.py
1 file changed, 6 insertions(+)
create mode 100644 free_memory.py
```

Now we've brought the master branch up to speed, which we can see by looking at the git log.

```
$ git log

commit 43618801... (HEAD -> master, even-better-feature)
Author: ...
Date: ...

commit 7d1de193...
Author: ...
Date: ...

commit bb9bd782...
Author: ...
Date: ...
```

As we're on the master branch, `HEAD` **points at** `master`. We can see the `even-better-feature` and `master` branches **are now both pointing at the same commit**.

Git **uses two different algorithms to perform a merge**, `fast-forward` and `three-way` merge. The merge we just performed is an example of a `fast-forward` merge. This kind of merge **occurs when all the commits in the checked out branch are also in the branch that's being merged** (Fig. 2.7).

Fig. 2.7 Using `Fast-forward merge`.

If this is the case, we can say that **the commit history of both branches doesn't diverge**. In these cases, all **Git has to do is update the pointers of the branches to the same commit, and no actual merging needs to take place**.

On the other hand, a `three-way` merge is performed when the history of the merging branches has diverged in some way, and **there isn't a nice linear path to combine** them via fast-forwarding. **This happens when a commit is made on one branch after the point when both branches split**.

Fig. 2.8 Using `Three-way merge.`

In our case, this could have happened if we made a commit on the master branch after creating the other branches.

When this occurs, Git will **tie the branch histories together with a new commit. And merge the snapshots at the two branch tips with the most recent common ancestor**, the commit before the divergence. To do this successfully, Git tries to figure out how to combine both snapshots. If the changes were made in different files, or in different parts of the same file, Git will take both changes and put them together in the result. **If instead the changes are made on the same part of the same file**, Git won't know how to merge those changes, and **the attempt will result in a merge conflict**. This sounds scary, but don't panic. Git doesn't quit, we'll solve those conflicts in our next video.

2.3.5 Merge Conflicts

From time to time, we might find that both the branches we're trying to merge have edits to the same part of the same file. This will result in something called a `merge conflict`. Normally, Git can automatically merge files for us. But when we have a `merge conflict`, it will need a little help to figure out what to do. To see how this would look, let's edit the `free_memory.py` file in the `master` branch and replace the pass statement with a comment about what the main function should do.

```
#Old code

def main():
pass

main()

#New code

def main():
```

```
"""Checks if there is enough free memory in the computer"""

main()
```

Cool, we made the change so let's save it and commit it back to our `master` branch ().

```
git commit -a -m 'Add comment to main()'
[master fe2fc5b] Add comment to main()
 1 file changed, 2 insertions(+), 2 deletions(-)
```

Next, Let's **check out** the `even-better-feature` branch and **make a change in the same place**. In this case, we will **replace the call to pass with a call to print**, saying that everything is okay. Now, we'll save this other change and commit it to this branch.

```
>>$ git checkout even-better-feature
Switched to branch 'even-better-feature'

>>  $ atom free_memory.py
#Old code

def main():
pass

main()

#New code

def main():
print("Everything is OK")

main()

>>  $ git commit -a -m 'Print everything ok'
[even-better-feature ca6de99] Print everything ok'
 1 file changed, 2 insertions(+), 2 deletions(-)
```

We are primed for chaos with our file all setup for a merge conflict. Let's **check out the** `master` **branch** again and **try to merge the** `even-better-feature` back into it.

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

>>  $ git merge even-better-feature
Auto-merging free_memory.py
```

```
CONFLICT (content): Merge conflickt in free_memory.py
Automatic merge failed; fix conflicts and then commit the result.
```

Git tells us it **tried to automatically merge the two versions** of the `free_memory` file, but **it didn't know how to do it**. We can use Git's status to get more information about what's going on.

```
>>$ git status
On branch master
You have unmerged paths
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>... to mark resolution)

both modified: free_memory.py
no changes added to commit (use "git add" and/or "git commit -a")
```

As usual, git status gives us a lot of additional information. It tells us that **we have files that are currently unmerged**, and that **we need to fix the conflicts or abort the merge if we decide it was a mistake**. It also tells us that we **need to run** `Git add` **on each unmerged file to mark that the conflicts have been resolved**.

Let's get to work. To fix the conflict, let's open up `free_memory.py` in our text editor.

```
def main():

<<<<<<< HEAD
"""Checks if there's enough free memory in the computer."""
=======
print("Everything is ok.)
>>>>>>> even-better-feature

main()
```

Thankfully, Git has added some information to our files to tell us which parts of the code are conflicting. The **unmerged content of the file at head**, remember, in this case, **head points to master**, is the **docstring stating what the main function should do**. The unmerged content of the file **in** the `even-better-feature` branch **is the call to the print function**. It's up to us to decide which one to keep or if we should change the contents of the file altogether. In this case, **we'll keep both statements and delete the merger markers**.

```
def main():
```

```
    """Checks if there's enough free memory in the computer."""
    print("Everything is ok.)

    main()
```

Now that we've fixed the conflict, we'll mark it as resolved by running `git add` on the file, and then call the `git status` to see how our merge is doing.

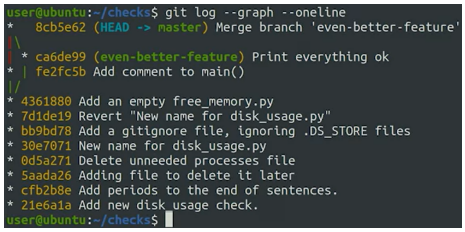
```
>> $ git add free_memory.py
>> $ git status
On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:
modifies: free_memory.py
```

See how Git now tells us that all conflicts have been resolved. Woo-hoo, we just need to call git commit to wrap up the merge (`git commit`).

The comments that git commit shows us look different than other commits. That's because this is a merge and Git tells us so. It also tells us which file had conflicts which have now been resolved. The commit already has a description saying that it's merging the other branch. This description was automatically created when we called the git merge command. But we can add onto this description if we want. For example, we can say that we're keeping the lines from both branches, and then just save and exit as usual. The merge conflict is resolved.

To see what the commit history looks like now, we'll use a couple of handy options to the `git log command`; `--graph` for seeing the commits as a graph, and `--oneline` to only see one line per commit.

Fig. 2.9 Commit Graph



This format helps us better understand the history of our commits and how merges have occurred. We can see the new commit (`8cb5e62`) that was added and also the two separate commits that we merged. One coming from the `master` branch (`fe2fc5b`) and the other coming from the `even-better-feature` branch (`ca6de99`).

We can also see that `master` **is pointing to the merge commit** but `even-better-feature` **is still pointing to the previous one**.

In our example, resolving the conflict was straightforward and easy. But in the real world, this won't always be the case. Merge conflicts can sometimes be tricky, complicated, and spread across multiple files. If you want to throw the merge away and start over, you can use the `git merge --abort` command as an escape hatch. This will **stop the merge and reset the files in your working tree back to the previous commit before the merge ever happened**. So by now you know how to create, delete, and switch between branches in Git. You've also seen that **each branch represents a pointer to a commit in a sequence of independent snapshots**. You know how to merge these commits back into the main trunk by using the git merge command. Amazing work. Seriously, this isn't easy stuff. Up next, you'll find a cheat sheet summarizing all of these branching techniques followed by a quiz to consolidate these concepts.

2.3.6 Reading: Git Branches and Merging Cheat Sheet

- `git branch` : Used to manage branches.
- `git branch <name>` : Creates the branch.
- `git branch -d <name>` : Deletes the branch.
- `git branch -D <name>` : Forcibly deletes the branch.
- `git checkout <branch>` : Switches to a branch.
- `git checkout -b <branch>` : Creates a new branch and switches to it.
- `git merge <branch>` : Merge joins branches together.
- `git merge --abort` :If there are merge conflicts (meaning files are incompatible), `--abort` can be used to abort the merge action..
- `git log --graph --oneline` : This shows a summarized view of the commit history for a repo.

Chapter 3
Week 3

3.1 Introduction to GitHub

3.1.1 Intro to Module3: Working with Remotes

Welcome back. Up until now we've learned a lot of interesting and complex techniques on how to use Git locally. We started with the very basic Git workflow for modifying files staging the changes and then committing them. We then looked at other things we can do with Git, like deleting or renaming files, getting more information about commits or skipping the staging area altogether. Next, we saw how to undo staged, unstaged or committed changes in our Git repositories. Finally, we learned a bit about how to work with different branches, how to merge changes from separate branches. That's a lot of complex stuff and it's natural that it might seem scary. But it can also be very satisfying once you start to understand what's going on and just think of how valuable these new skills will be as you advance your IT career. So remember that you can review the material as many times as needed and that the best way to master it is to keep practicing on your own, and if something is still unclear after that, don't forget you can use the discussion forums to ask for help.

In this module, we'll learn a load of new things related to GitHub and remote repositories. We'll first talk about what GitHub is and why it matters, and then we'll dive into how to work with GitHub and other remote repositories. Being able to use remote repositories allows us to effectively collaborate with others. Our collaborators can be sitting in the same office as we are or they can be thousands of miles away on a different continent working at a different time of day. Using a version control system like Git lets us incorporate the work of different people no matter where they are or when they're working. I've personally had a bunch of interesting experiences collaborating through a version control system. One that stands out was a debugging session a friend and I had while at work. I was working on some tricky code and I couldn't figure out the best way to implement a function. I was stuck and ready to give up completely. My colleague suggested that I work on another

piece of the file further down while he figured out the optimal solution for the code that I was having trouble with. This was a super seamless experience and was only possible because of a VCS, and the superpower of teamwork of course. By the end of the module you'll be able to collaborate with friends and colleagues just like I did. Let's get to it.

3.1.2 What is GitHub

Git is a *distributed* VCS (version control system). Distributed means that each developer has a copy of the whole repository on their local machine.

Each copy is a peer of the others. But we can *host* one of these copies on a server and then use it as a *remote repository* for the other copies. This lets us synchronize work between copies through this server. Any of us can create a Git server like this one, and many companies have similar internal services. But if you don't want to set up a Git server yourself and host your repositories, you can use an online service like *GitHub*.

GitHub is a web-based Git repository hosting service. On top of the version control functionality of Git, GitHub includes extra features like *bug tracking*, *wikis*, and *task management*. GitHub lets us share and access repositories on the web and copy or clone them to our local computer, so we can work on them. GitHub is a popular choice with a robust feature set, but it's not the only one. Other services that provide similar functionality are *BitBucket*, and *GitLab*.

For the rest of this course, we'll be using GitHub for our examples. But feel free to use the tool that best fits your needs. GitHub provides free access to a Git server for public and private repositories. It *limits the number of contributors for the free private repositories*, and offers an unlimited private repository service for a monthly fee. We'll be using a free repository for our examples, which is fine for educational use, small personal projects, or open source development.

A word of caution on how you can manage these repos though. If hackers get hold of information about your organization's IT infrastructure, they can use it to try and break into your network. So make sure you treat this information as confidential. **For real configuration and development work, you should use a secure and private Git server, and limit the people authorized to work on it**.

To use GitHub, *the first thing you need to do is create an account* if you don't have one already. Signing up online is free and relatively simple. Once you've done that, you can either create your own repos or contribute to repos from other projects. If you don't have a GitHub account yet, now is a good time to create one. Visit `github.com` to sign up for their service. Once you've done that, meet me over in the next video, where we'll go over some basic interactions with GitHub.

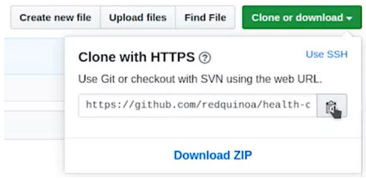
3.1.3 Basic Interaction with GitHub

As we called out, GitHub is an online service. To use it, you first need to create an account on the site. Once you have your account, you're ready to create your brand new repository on GitHub. Let's do this now. Going step-by-step.

We'll start by clicking the `Create a repository` link on the left. This will take us to the repo creation wizard. The wizard is pretty straightforward. The first thing we need to do is give a name for our repo. We'll call this repo *health checks*. After that comes a description of what the repo will be used for. We'll say that'll be used for `scripts that check the health of our computers`. Then we need to select whether we want the repo to be `public` or `private`. We'll go with private for now. Finally, the wizard can help us get started with some few initialization files like a `README`, a `gitignore`, or license file. We'll go with just the `README` for now, and then create the repo. Using the wizard, we created the repo and have a fresh remote repository ready to go. Just like magic.

Let's get to work. First step is to create a local copy of the repository. We'll do that by using the `git clone` command followed by the `URL` of the repo. GitHub conveniently lets us copy the URL from our repo from the interface so that we don't have to type it (Fig. 3.1).

Fig. 3.1 Clone with HTTPS window.



We're now ready to clone the repo into our computer. We'll do that by calling git clone and paste in the URL we copied. To do this, GitHub will ask for our username and password (Fig. 3.2).

Fig. 3.2 Console `git clone URL`.

```
user@ubuntu:~$ git clone https://github.com/redquinoa/health-checks.git
Cloning into 'health-checks'...
Username for 'https://github.com': redquinoa
Password for 'https://redquinoa@github.com':
```

Just like that, we've downloaded a copy of the remote repository from GitHub onto the local machine. This means that we can perform all the git actions that we've learned up till now. Since the repo is called `health-checks`, **a directory with that name was automatically created for us and now has the working tree of the Repository in it**. So let's change that directory and look at the contents using

`cd health-checks/` and then `ls -l`. Our repo is basically empty. It only has the `README` file that GitHub created for us. This file is in a special format called `markdown`. Let's add a bit more content to it writing `This repo will be populated with lots of fancy checks.`

We've changed this file. What do we need to do now? We need to stage the change and committed. We've seen a couple of different ways to do that. Let's use our shortcuts to do this in just one command:

`git commit -a -m "Add one more line to README.md"`.

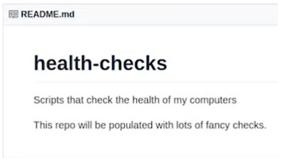
Okay. We've modified our `README` file. But we've seen all this before. We got to remote repository set up on GitHub. So let's use it. We can send our changes to that remote repository by using the `git push` command which will gather all the snapshots we've taken and send them to the remote repository. In this case, we've only taken one snapshot. We'll talk more about what's going on behind the scenes with git push and remote repositories in later videos. But the mechanics are pretty simple. To push our modified `README` up to GitHub, we'll just call git push (Fig. 3.3).

Fig. 3.3 git push.

```
user@ubuntu:~/health-checks$ git push
Username for 'https://github.com': redquinoa
Password for 'https://redquinoa@github.com':
```

Once again, we're asked for our password. After that, we see a bunch of messages from git related to the push. When we access our project, we see the contents of the `README` file. So if we check our repository on GitHub, we should see the updated message (Fig. 3.4).

Fig. 3.4 Updated README file.



Pretty cool, right? We've taken the local changes on our computer and pushed them out to a remote repository hosted on GitHub.

You've probably noticed that we had to enter our password both when retrieving the repo and when pushing changes to the repo. There are a couple ways to avoid having to do this. One way is to create an *SSH key pair* and store the public key in our profile so that GitHub recognizes our computer. Another option is to *use a credential helper* which caches our credentials for a time window so that we don't need to enter our password with every interaction. Git already comes with a credential helper baked in. We just need to enable it. We do that by calling:

```
git config --global credential.helper cache
```

Now that we've enabled the credential helper, we'll need to enter our credentials once more. After that, they'll be cached for 15 minutes. To check this, we can try another git command, `git pull` which is the command we use to *retrieve new changes from the repository*. We'll enter our credentials on the first call to the command and they'll be cached, so we won't need to enter them again.

```
$ git pull
Already up to date.
```

With that, we've seen how to create repositories on GitHub, *clone our remote repository, push changes to it, and pull changes from it*. There's a lot going on behind the scenes here and we'll dive deeper into the details and more collaboration techniques in later videos. For now, check out the next reading for a summary of this basic workflow. Then there's a quick quiz to make sure this is making sense.

3.1.4 Reading: Basic Interaction with GitHub

Table 3.1 Cheat-sheet

Command	Explanation
git clone URL	used to clone a remote repository into a local workspace
git push	used to push commits from your local repo to a remote repo
git pull	used to fetch the newest updates from a remote repository

This can be useful for keeping your local workspace up to date.

- https://help.github.com/en/articles/caching-your-github-password-in-git
- https://help.github.com/en/articles/generating-an-ssh-key

3.2 Using a Remote Repository

3.2.1 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*. If all this protocol talk is making your head spin you might want to review the video on the subjects made by my colleague Gian. You'll find the link to this in the next reading.

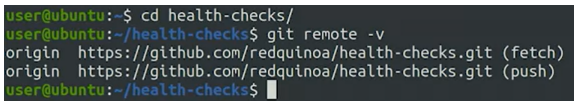
The distributed nature of the work means that *there are no limits to how many people can push code into a repository*. It's a good idea to control who can push codes to repos and to make sure you give access only to people you trust. Web services like GitHub, offer a bunch of different mechanisms to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

Repositories. Some of these are available to the general public while others are only available to enterprise users. By now you have an idea of what a remote repository is and how it interacts with local Git repositories. Up next, we'll dive into some of the commands that let us interact with remotes.

3.2.2 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 two URLs (3.5).

Fig. 3.5 git remote -v.



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` (Fig. 3.6).

Fig. 3.6 `git branch -r`.

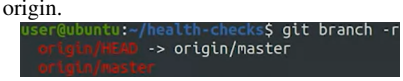
There's a ton of information here, and we don't need all of it right now. We can see the `fetch` and `push URL`s that we saw before, and the `local` and `remote` branches too.

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` (Fig. 3.7).

Fig. 3.7 git remote show origin.



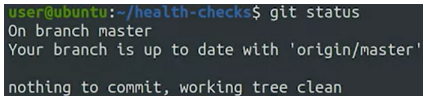
These branches are read only. We can look at the commit history, like we would with local branches, but we can't make any changes to them directly.

To modify their contents, we'll have to go through the workflow we called out before. First, we pull (`git pull`) any new changes to our local branch, then merge them with our changes and push (`git push`) our changes to the repo. 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.

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

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* (Fig. 3.8), which means that the `master` **branch in the remote repository called** `origin`, **has the same commits as our** `local master` **branch**. But *what if it wasn't up to date?*.

Fig. 3.8 git status.



3.2.3 Fetching New Changes

While we were learning about remotes, our colleague *Blue Kale added some files 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` (Fig. 3.9).

Fig. 3.9 git remote show origin.

```
user@ubuntu:~$ cd health-checks/
user@ubuntu:~/health-checks$ git remote show origin
* remote origin
  Fetch URL: https://github.com/redquinoa/health-checks.git
  Push  URL: https://github.com/redquinoa/health-checks.git
  HEAD branch: master
  Remote branch:
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (local out of date)
```

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. Let's call it now and see what happens.

Fig. 3.10 git fetch.

```
user@ubuntu:~/health-checks$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 4 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
From https://github.com/redquinoa/health-checks
   807cb50..b62dc2e  master     -> origin/master
```

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** (`commit b62d...`). While the **local master branch is pointing to the previous commit** (`commit 807c...`) we made earlier on. Let's see what happens if we run `git status` now.

Fig. 3.11 git log origin/master.

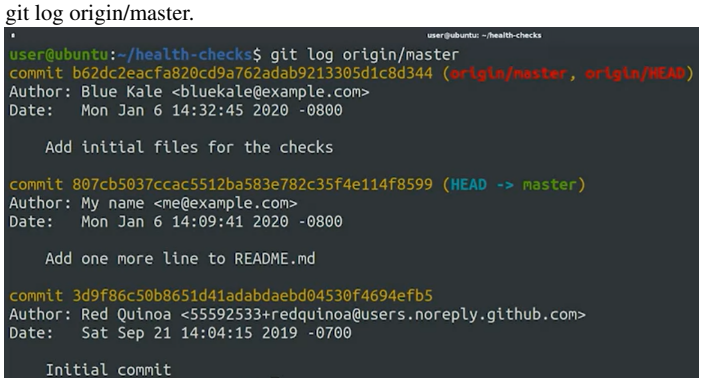
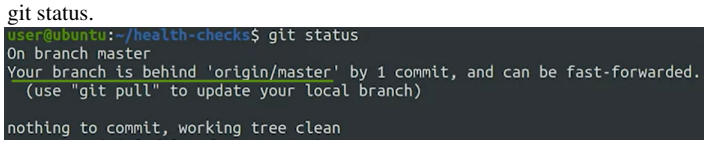
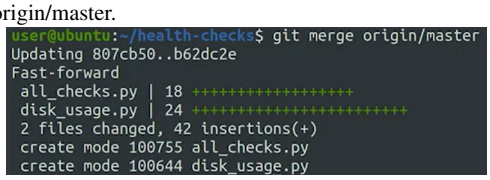


Fig. 3.12 git status.



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`.

Fig. 3.13 git merge origin/master.

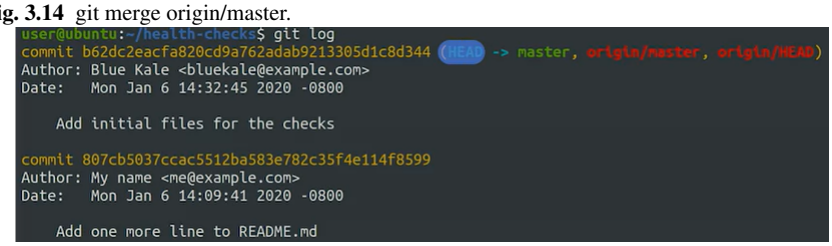


Great. We've *merged the changes of the master branch of the remote repository into our local branch*. See how Git tells us that the code was integrated using `fast-forward`? It also shows that *two files were added*, `all_checks` and `disk_usage.py`.

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*.

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

Fig. 3.14 git merge origin/master.



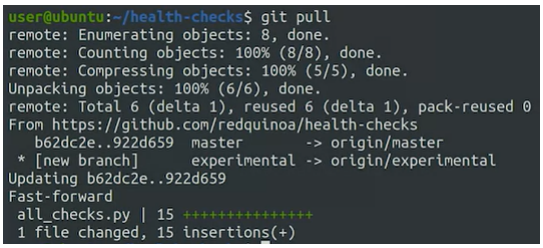
and merging them into your local repository is such a common operation in Git that there's a handy command to let us do it all in one action `git pull`. We'll check that out in our next video.

3.2.4 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.

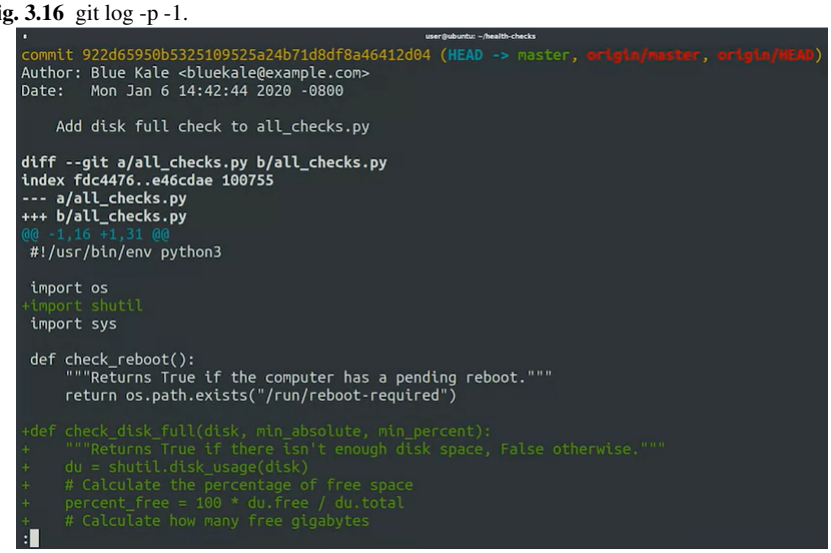
Fig. 3.15 git pull.



If you look closely at this output, you'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` called `experimental`. 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.

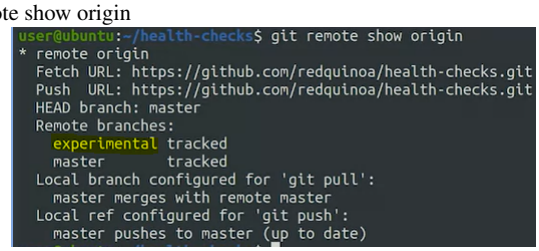
We can look at the changes by using `git log -p -1` (Fig. 3.16).

Fig. 3.16 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` (Fig. 3.17).

Fig. 3.17 git remote show origin



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.

Fig. 3.18 git checkout experimental

```
user@ubuntu:~/health-checks$ git checkout experimental
Branch 'experimental' set up to track remote branch 'experimental' from 'origin'.
Switched to a new branch 'experimental'
```

In this last example, we got the contents of the `experimental branch` 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 videos, we'll look at *what happens when we try to push changes, especially when our changes generate conflicts*. But before we do that, check out the reading for the list of all the commands involved, and then take the quiz to put this knowledge into practice.

3.2.5 Reading: Git Remotes Cheat-Sheet

Table 3.2 Cheat-sheet

Command	Explanation
`git remote`	List 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

3.3 Solving Conflicts

3.3.1 The Pull-Merge-Push Workflow

We've now looked at the details of fetching and pulling data from a remote repositories without any local changes. We saw earlier how we can use the `git push` command to send our changes to the remote repo. But *what if when we go to push our changes, there are new changes to the remote repo*? To find out, let's start by making a change to our `all_checks.py` script.

Remember way back to the beginning of the course, when we fixed the bug in the function that checks the disk space? The one that was doing gigabyte conversion twice?

Part of the reason why our code was so buggy, was that *we were passing numbers around without saying what those numbers were for*. We could have made our code clearer by renaming our `min_absolute` parameter to `min_GB`. So that it's obvious that the function expects gigabytes (Fig. 3.19).

Fig. 3.19 min_absolute

```
11  def check_disk_full(disk, min_absolute, min_percent):
12      """Returns True if there isn't enough disk space, False otherwise."""
13      du = shutil.disk_usage(disk)
14      # Calculate the percentage of free space
15      percent_free = 100 * du.free / du.total
16      # Calculate how many free gigabytes
17      gigabytes_free = du.free / 2**30
18      if percent_free < min_percent or gigabytes_free < min_absolute:
19          return True
20      return False
```

With that, we've clarified the code of the function (Fig. 3.20).

Another way we can make the code invocation clearer, we can use the name of the parameters in the call to the function, like this (Fig. 3.21).

By using the names of the parameters, our invocation is clear, and we can even alter the order of the values and our code would still work (Fig. 3.22).

All right, we've made the change. Let's stage it and commit it as usual. We'll first use `git add -p` to look at the changes we made and accept them (Fig. 3.23).

Then we'll create a commit message to show that we've renamed `min_absolute` to `min_GB`, and that we're using parameter names for the invocation. We've made our change, staged it, and committed it. **We should be ready to push into the remote repo, except now we have a collaborator also making changes**.

Let's see what happens when we try running `git push`. And it failed. Can you work out what went wrong here? (Fig. 3.24).

Fig. 3.20 min_gb

```
11  def check_disk_full(disk, min_gb, min_percent):
12      """Returns True if there isn't enough disk space, False otherwise."""
13      du = shutil.disk_usage(disk)
14      # Calculate the percentage of free space
15      percent_free = 100 * du.free / du.total
16      # Calculate how many free gigabytes
17      gigabytes_free = du.free / 2**30
18      if percent_free < min_percent or gigabytes_free < min_gb:
19          return True
20      return False
```

Fig. 3.21 min_gb

```
def main():
    if check_reboot():
        print("Pending Reboot.")
        sys.exit(1)
    if check_disk_full("/", 2, 10):
        print("Disk full.")
        sys.exit(1)

    print("Everything ok.")
    sys.exit(0)
```

Fig. 3.22 min_gb

```
def main():
    if check_reboot():
        print("Pending Reboot.")
        sys.exit(1)
    if check_disk_full(disk="/",min_gb=2, min_percent=10)
        print("Disk full.")
        sys.exit(1)

    print("Everything ok.")
    sys.exit(0)
```

Fig. 3.23 min_gb

```
user@ubuntu:~/health-checks$ git add -p
diff --git a/all_checks.py b/all_checks.py
index e46cdae..a40047c 100755
--- a/all_checks.py
+++ b/all_checks.py
@@ -8,14 +8,14 @@ def check_reboot():
     """Returns True if the computer has a pending reboot."""
     return os.path.exists("/run/reboot-required")

-def check_disk_full(disk, min_absolute, min_percent):
+def check_disk_full(disk, min_gb, min_percent):
     """Returns True if there isn't enough disk space, False otherwise."""
     du = shutil.disk_usage(disk)
     # Calculate the percentage of free space
     percent_free = 100 * du.free / du.total
     # Calculate how many free gigabytes
     gigabytes_free = du.free / 2**30
-    if percent_free < min_percent or gigabytes_free < min_absolute:
+    if percent_free < min_percent or gigabytes_free < min_gb:
         return True
     return False
```

Fig. 3.24 git push rejected

```
user@ubuntu:~/health-checks$ git push
Username for 'https://github.com': redquinoa
Password for 'https://redquinoa@github.com':
To https://github.com/redquinoa/health-checks.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/redquinoa/health-checks.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
user@ubuntu:~/health-checks$
```

There are a few hints. When we tried to push, Git **rejected** our change, that's because **the remote repository contains changes that we don't have in our local branch that Git can't fast-forward**. Maybe you remember when we talked about Git's merging algorithms? As usual, Git gives us some helpful information along with the error message, especially the part about integrating remote changes with `git pull`.

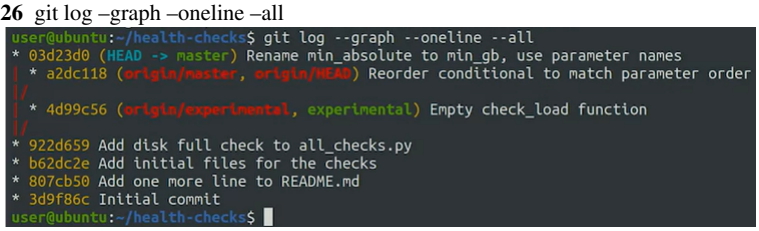
This means **we need to sync our local remote branch with the remote repository before we can push**. We learned earlier that we can do this with `git pull`. Let's do this now. Git tried to automatically merge the local and remote changes to `all_checks.py`, but found a conflict (Fig. 3.25).

Fig. 3.25 git pull conflict

```
user@ubuntu:~/health-checks$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/redquinoa/health-checks
   922d659..a2dc118  master     -> origin/master
Auto-merging all_checks.py
CONFLICT (content): Merge conflict in all_checks.py
Automatic merge failed; fix conflicts and then commit the result.
user@ubuntu:~/health-checks$
```

Let's first look at the tree of commits on all branches as represented by: `git log --graph --oneline --all` (Fig. 3.26).

Fig. 3.26 git log –graph –oneline –all

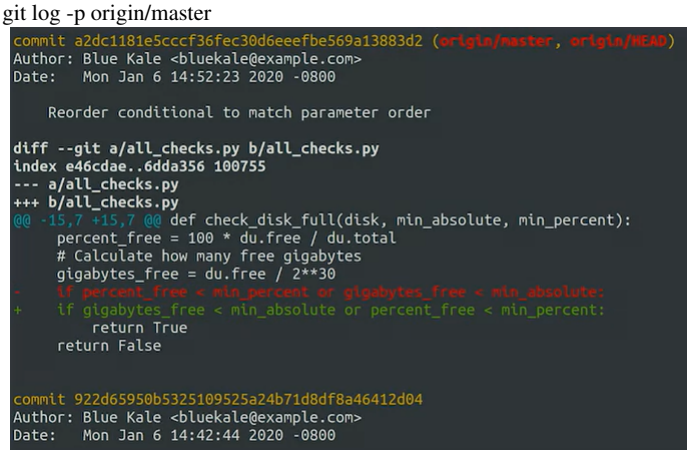


This graph shows us the different commits and positions in the tree. We can see the master branch, the `origin/master branch`, and the experimental branch.

The graph indicates that our current commit and the commit in the `origin/master branch` share a common ancestor (`commit 992d659...`), but they don't follow one another. This means that we'll need to do a `three-way merge`. To do this, let's look at the actual changes in that commit by running `git log -p origin/master`.

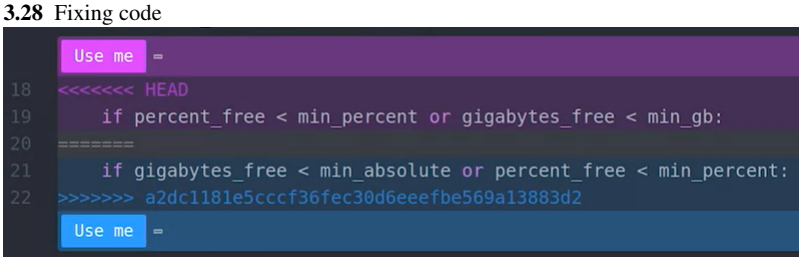
So our colleague decide to reorder the conditional clauses in the function to match the order that the parameters are passed to the function. They happen to change in the same line (- and + lines) (Fig. 3.27) that we changed when we renamed the `min_gb` variable, which caused the conflict that Git couldn't resolve. Let's **fix it by editing the file to remove the conflict**. So first, let me exit with `q` .

Fig. 3.27 git log -p origin/master



We see that the problem occurred in the conditional (Fig. 3.28). On the first line, we see our change, where `min_absolute` was renamed to `min_gb`. In the second line, we see the old variable names, with the checks done in a different order.

Fig. 3.28 Fixing code



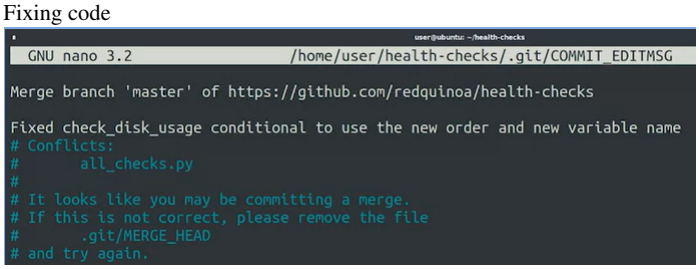
We need to decide what to do to this. For example, we can keep the new order, but use `min_gb`. One thing to notice is that *Git will try to do all possible automatic merges* and only leave manual conflicts for us to resolve when the automatic merge fails. In this case, we can see that the other changes we made were merged successfully without intervention.**Only the change that happened in the same line of the file needed our input**.

We fixed the conflict here, and the file is short enough that we can very quickly check that there are no other conflicts. *For larger files, it might make sense to search for the conflict markers*, `>>>` (greater than, greater, greater than), in the whole file. This lets us check that there are no unresolved conflicts left. Nice, now that we fixed the conflict, you can finish the merge.Do you remember how to do it?

We need to `add` the `all_checks.py` file, and then call `git commit` to finish the merge. But first, we're going to save and close.

The editor message shows that it's performing a merge of the remote branch with the local branch. We can add extra information to this message. For example, we can say that we fixed the conditional in the `check_disk_usage` function to use the new variable name and the new order (Fig. 3.29).

Fig. 3.29 Fixing code



Our merge is finally ready, we can try pushing to the remote again. Yes, after fixing the conflict, we were able to push our work to the remote repo (Fig. 3.30).

Let's look at the commit history of the master branch now, by calling:
`git log --graph --oneline` (Fig. 3.31).

Fig. 3.30 git push fixed

```
user@ubuntu:~/health-checks$ git push
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 4 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 876 bytes | 876.00 KiB/s, done.
Total 6 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/redquinoa/health-checks.git
   a2dc118..58351ff  master -> master
```

Fig. 3.31 git push fixed

```
user@ubuntu:~/health-checks$ git log --graph --oneline
*   58351ff (HEAD -> master, origin/master, origin/HEAD) Merge branch 'master' of https://github.com/redq
uinoa/health-checks
|\
| * a2dc118 Reorder conditional to match parameter order
* | 03d23d0 Rename min_absolute to min_gb, use parameter names
|/
* 922d659 Add disk full check to all_checks.py
* b62dc2e Add initial files for the checks
* 807cb50 Add one more line to README.md
* 3d9f86c Initial commit
user@ubuntu:~/health-checks$
```

We see that the latest commit is the merge, followed by the two commits that caused the merge conflict, which are on split paths in our graph. As we called out before, when Git needs to do a `three-way merge`, **we end up with a separate commit for merging the branches back into the main tree**. Now we know how to successfully complete a pull, merge, and push cycle, even when it means doing some manual merges.

This was a complex exercise, and it's okay if some things still seem a bit scary. We all felt panic the first time we encountered a merge conflict. But don't worry, it gets easier with practice. To practice dealing with merge conflicts, you want to have two copies of your repository in separate directories, then try editing the same lines of the same files. You can follow along with the examples shown here, or come up with your own.

Up next, we'll talk about using branches with a remote repositories.

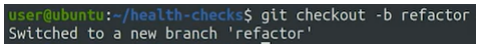
3.3.2 Pushing Remote Branches

As we called up before, *when using Git to work on a new feature or a big refactor of some kind, it's recommended best practice to create separate branches*. There are many advantages to doing this. For example, it might take you a while to finish a new feature and in the meantime, there could be a critical bug that needs fixing in the main branch of the code. By having separate branches, you can fix the bug in the main branch, release a new version and then go back to working on your feature without having to integrate your code before it's ready. Another advantage of

working in separate branches is that *you could even release two or more versions out of the same tree*. One being **the stable version** and the other being the **beta version**. That way, *any disruptive changes can be tested on a few users or computers before they're fully released*.

So **let's start a new branch to work** on a small refactor of our code. Do you remember how to do that? You could create the branch first, and then check it out or *we can just create it and check it out with* `git checkout-b <branchName>` and the new branch name, in this case will br `git checkout-b refactor`.

Fig. 3.32 git checkout -b refactor

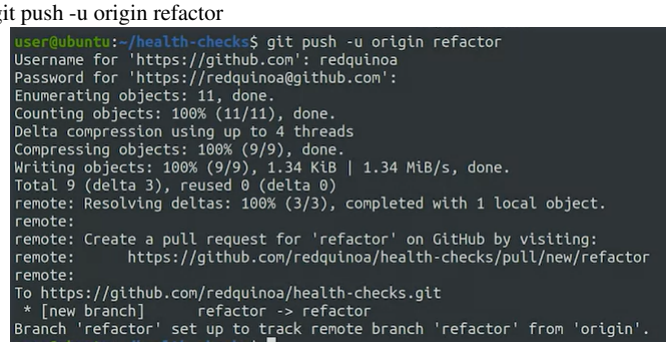


We're ready to start working on our refactor. Let's open up the file, and have a look at it. After some modifications and commits ...

Before we merge any of this into the `master branch`, we want to `push` this into the `remote` repo, **so that our collaborators can view the code, test it, and let us know if it's ready for merging**.

The first time we push a branch to a remote repo, **we need to add a few more parameters** to the `git push` command. We'll need to add the `-u` flag **to create the branch upstream**, which is another way of referring to remote repositories. We'll also have to say that **we want to push this to the origin repo**, and that we're pushing the `refactor branch` with `git push -u origin refactor`.

Fig. 3.33 git push -u origin refactor



Whoa, that's a lot of information that Git's giving us. It's telling us if we want, we can create a pull request. We'll talk more about pull requests later on. For now, we're happy to see that new refactor branch has been created in the remote repo, which is what we wanted. This was a super complex example that incorporated a lot of concepts that we've learned about in this course, and also carried out some interesting Python concepts. If anything is still unclear, feel free to re-watch this video and follow along in your computer until you're comfortable with these steps.

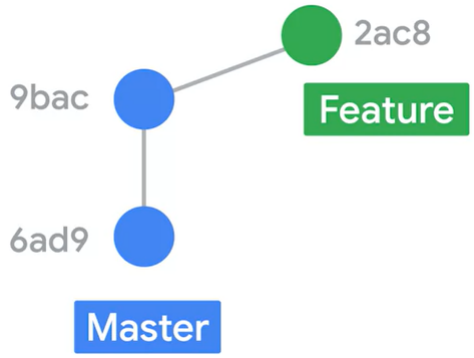
So now that our branch is pushed to the remote repo, it can be reviewed by our collaborators. Assuming they say it's okay, **how should this branch get merged back into the master branch**? We'll talk about that in our next video.

3.3.3 Rebasing Your Changes

In our last video, we mentioned that once our branch has been properly reviewed and tested, it can get merged back into the master branch. This can be done by us or by someone else. One option is to use the `git merge` command that we discussed earlier. Another option is to use the `git rebase` command. Rebasing means **changing the base commit that's used for our branch**. To understand what this means, let's quickly recap what we've learned about merges up till now.

As we've seen in a lot of our earlier examples, when we create a branch at a certain point in the repo's history, **Git knows the latest commit that was submitted on both branches**.

Fig. 3.34 Master and feature branches



If *only one of the branches has new changes when we try to merge them, Git will be able to fast forward and apply the changes*. But if both branches have new changes when we try to merge, **Git will create a new merge commit** for the `three way merge`.

The problem with `three way merges` **is that because of the split history, it's hard for us to debug when an issue is found in our code, and we need to understand where the problem was introduced**. By **changing the base where our commits split from the branch history, we can replay the new commits on top of the new base**. This **allows Git to do a fast forward merge and keep history linear**.

So how do we do it? We run the command `git rebase`, followed by the **branch that we want to set as the new base**. When we do this, Git will *try to*

Fig. 3.35 Master and feature branches with commits

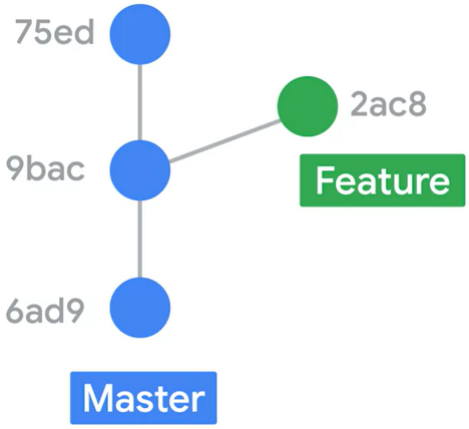
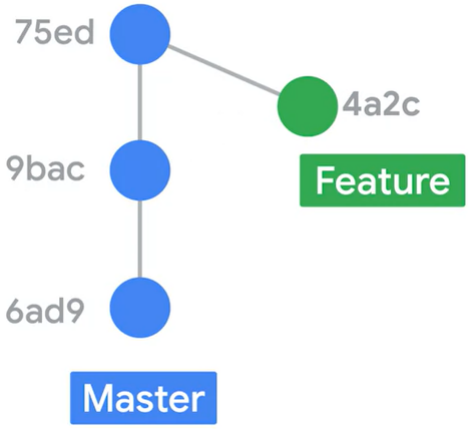


Fig. 3.36 git rebase



replay our commits after the latest commit in that branch. This will work automatically if the changes are made in different parts of the files, but **will require manual intervention** if the changes were made in other files. Let's check out this process by **rebasing our refactor branch onto the master branch**.

First, we'll `check out` the master branch and `pull` the latest changes in the remote repo.

Git tells us that *it's updated the master branch with some changes that our colleague had made*. At this point, *the changes that we have in the* `refactor branch` *can no longer be merged through fast forwarding into the* `master branch`. That's **because there's now an extra commit in the master that's not present in the refactor**.

Fig. 3.37 git rebase

```
user@ubuntu:~/health-checks$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
user@ubuntu:~/health-checks$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/redquinoa/health-checks
   58351ff..0789f64  master     -> origin/master
Updating 58351ff..0789f64
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
```

Let's see how this looks by asking the `git log --graph --oneline --all` command to show us the current graph of all branches (Fig. 3.38).

Fig. 3.38 git log –graph –oneline –all

```
user@ubuntu:~/health-checks$ git log --graph --oneline --all
* 0789f64 (HEAD -> master, origin/master, origin/HEAD) Add reference to all_checks.py to README
| * cbee3f7 (origin/refactor, refactor) Allow printing more than one error message
| * 75bdd43 Iterate over a list of checks and messages to avoid code duplication
| * e914aee Create wrapper function for check_disk_full
|/
*   58351ff Merge branch 'master' of https://github.com/redquinoa/health-checks
|\
| * a2dc118 Reorder conditional to match parameter order
* | 03d23d0 Rename min_absolute to min_gb, use parameter names
|/
| * 4d99c56 (origin/experimental, experimental) Empty check_load function
|/
* 922d659 Add disk full check to all_checks.py
* b62dc2e Add initial files for the checks
* 807cb50 Add one more line to README.md
* 3d9f86c Initial commit
```

It might take a bit to follow everything that's going on with this graph. But it can be really useful to understand complex history trees. As you can see, **the refactor branch has three commits before the common ancestor** (`commit 58351ff`), with the current commit that's at the head of the master branch (`commit 0789f64`).

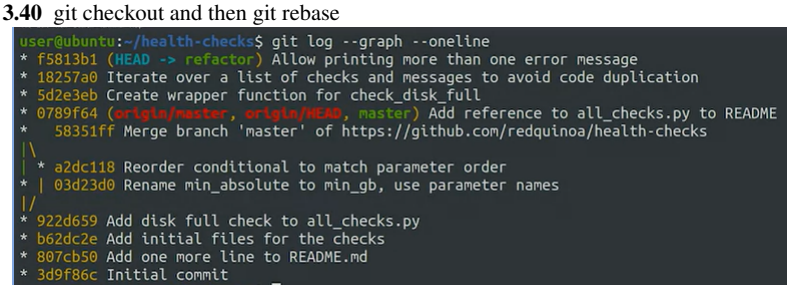
If we merged our branch now, it would cause a `three way merge`. But we want to **keep our history linear**. We'll do this **with a rebase of the refactor against master** (Fig. 3.39).

Fig. 3.39 git checkout and then git rebase

```
user@ubuntu:~/health-checks$ git checkout refactor
Switched to branch 'refactor'
Your branch is up to date with 'origin/refactor'.
user@ubuntu:~/health-checks$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Create wrapper function for check_disk_full
Applying: Iterate over a list of checks and messages to avoid code duplication
Applying: Allow printing more than one error message
```

The line `git rebase nameOfBranch` moves the current branch on top of the `nameOfBranch` branch. As usual, Git gives us a bunch of helpful information. It says that it rewound head and replayed our work on top of it. And luckily, everything succeeded. Let's look at the output of `git log --graph --oneline` for our branch right now (Fig. 3.40).

Fig. 3.40 git checkout and then git rebase



The previous `HEAD -> master commit 0789f64` was converted into a *base* for the commits in `refactor branch` (`commits e914aee, 75bdd43` and —`cbee3f7`), these commits were applied to tha *base* becoming the `commits 5d2e3eb, 18257a0` and `f5813b1`. Now we can **see the master branch and linear history with our list of commits**. We're ready to merge our commits back onto the main trunk of our repo and have this fast forwarded. To do that, we'll check out the master branch (`git checkout master`) and merge the refactor branch (`git merge refactor`).

Awesome, **we were able to merge our branch through a fast forward merge and keep our history linear**. We're now done with our refactor and can **get rid of that branch, both remotely and locally**. To **remove the remote branch**, we'll call `git push --delete origin refactor`. To **remove the local branch**, we'll call `git branch -d refactor`.

Yes, we're done with our refactor. We can now push changes back into the remote repo (`git push`).

All right, we've just gone through an example using the git rebase command. We had a feature branch created against an older commit from master. So we rebased our feature branch against the latest commit from master and then merged the feature branch back into master.

That was a complicated exercise. So if you're still confused about what's going on, take your time to review, and maybe come up with your own examples when you'd use a rebase.

Up next, we'll look at another example of how to use rebase.

3.3.4 Another Rebasing Example

In our last video, we looked at an example use case for git rebase. Where we used it to rebase a feature branch so that it could be cleanly integrated. There are many other possible uses of rebase. One common example is to rebase the changes in the master branch when someone else also made changes and we want to keep history linear. This is a pretty common occurrence when you're working on a change that's small enough not to need a separate branch and your collaborators just happened to commit something at the same time. Let's check out how this would work in practice. First, we'll make a change to our script.

Now that we've made it easy to add new checks, will add a check to warn when there's no working network. There's a ton of things to check for this but for now we'll keep it simple and just check whether we can resolve the google.com URL. To do this, we'll use the socket module. We'll add a new function called check no network that will return true if it fails to resolve the URL and false if it succeeds. This socket.gethostbyname function raises an exception on failure. So we'll use it try except block to wrap the call to the function and return false when the call succeeds or true when it fails.

With this new function defined, we can now add the check to our list of checks. We'll just add the name of the function and the message will be, "No working network."

We've made the change, let's save it and commit it.

Once more we'll use the git commit -a shortcut and pass a message saying that we've added a simple network connectivity check.

We want to check if one of our teammates also made a change in the master branch while we were working on our change. In an earlier video, we showed how to do that by running git pull which will automatically create a three-way merge if necessary. In this example, we want to look at a different approach to keep our project history linear. So we'll start by calling git fetch which you might remember we'll put the latest changes into the origin slash master branch but we won't apply them to our local master branch.

We see that we fetched some new changes. This means that if we tried to merge our changes, we end up with a three-way merge. Instead, we'll now run git rebase against our origin/master to rebase our changes against those made by our colleague and keep history linear.

We've got a conflict and we'll need to fix it. Git is giving us a lot of info on what it tried to do including what worked, what didn't work and what we can do about it. Since we asked it to rebase, it tried to rewind our changes and apply them on top of what was in the `origin/master branch`. The first commit made by our colleague, renamed `all_checks.py` to `health_checks.py`. Git detected this and automatically merged our changes into the new file name. But when trying to merge our changes with the changes made by our colleague in the file, there was a merge conflict. The output gives us a bunch of instructions on how to solve this. We could fix the conflict, skip the conflicting commit or even abort the rebased

completely. In this example, we want to fix the conflict. So let's do that. We'll start by looking at the current state of the `health_checks.py` file.

We see that while we were adding the connectivity check, our colleague was adding a check for the CPU being constrained. We want both functions and the end result. So let's remove the conflict markers, cleaning up our file.

This looks good. Let's save and test our script out.

So close it looks like our colleague forgot to import the psutil module. Let's do that now.

Let's hope this works.

We fix the conflict and our script is working again. We now need to add the changes made to the `health_checks.py` file and continue with the rebase.

Now, the rebase has finished successfully let's check out the output of

`git log --graph --oneline` to see what the history looks like at this point. We see that we've applied our change on top of the other changes without needing a `three-way merge`. What we did just now to resolve the conflict is very similar to what we did earlier to merge our changes. The difference is, that the commit history ended up being linear instead of branching out. We're now ready to push our new check to the remote repo.

In this example, we've seen how we can use the fetch rebase push workflow to merge our changes with our collaborators changes while keeping the history of our changes linear. As we called out, keeping history linear helps with debugging especially when we're trying to identify which commit first introduced a problem in our project. We've now seen two examples of how to use the git rebase command. One for merging feature branches back into the main trunk of our code and one for making sure that our commits made in the master branch apply cleanly on top of the current state of the master branch and it doesn't stop there. We can also use git rebase to change the order of the commits or even squash two commits into one. This is a very powerful tool but don't worry you don't need to memorize all of its possible uses you'll learn them as you need them. Up next, we'll do a round up of some of the best practices for operating with git when collaborating with others.

3.3.5 Best Practices for Collaboration

Over the past few videos, we've looked at a lot of things we can do with Git and remote repositories. It's worth spending some time talking about best practices for collaborating with others. It's a good idea to **always synchronize your branches before starting any work on your own**.

That way, whenever you start changing code, you know that you're starting from the most recent version and you minimize the chances of conflicts or the need for rebasing. Another common practice is to try and **avoid having very large changes that modify a lot of different things**. Instead, try to make changes as small as possible as long as they're self-contained. For example, if you are renaming a variable for clarity reasons, you don't want to have code that adds new functionality in the

same commit. It's better if you split it into different commit. This **makes it easier to understand what's going on with each commit**.

On top of that, **if you remember to push your changes often and pull before doing any work, you reduce the chances of getting conflict**. We called out already that **when working on a big change, it makes sense to have a separate feature branch**.

This lets you work on new changes while still enabling you to fix bugs in the other branch. To make the final merge of the feature branch easier, it makes sense to **regularly merge changes made on the master branch back onto the feature branch**.

This way, we won't end up with a huge number of merge conflicts when the final merge time comes around.

If you need to maintain more than one version of a project at the same time, it's common practice to **have the latest version of the project in the master branch and a stable version of the project on a separate branch**.

You'll merge your changes into the separate branch whenever you declare a stable release.

When using these two branches, some bug fixes for the stable version may be done directly on the stable branch if they aren't relevant to the latest version anymore. In the last couple of videos, we looked at how we can use `rebase` to make sure our history is linear. Rebasing can help a lot with identifying bugs, but use it with caution. `Whenever we do a rebase, we're rewriting the history of our branch`. The old commits get replaced with new commits, so they'll be based on different snapshots than the ones we had before and they'll have completely different hash sums. This works fine for local changes, but can cause a lot of trouble for changes that have been published and downloaded by other collaborators. So as a general rule, **you shouldn't rebase changes that have been pushed to remote repos**.

The Git server will automatically reject pushes that attempt to rewrite the history of the branch. It's possible to force Git to accept the change, but it's not a great idea unless you really know what the implications will be. In our `feature` branch example, we rebased the branch. Merged it to the master and then deleted the old one. That way, we didn't push the rebase changes to the refactor branch, only to the master branch that hadn't seen those changes before.

Early in our Git journey, we mentioned that **having good commit messages is important**. It's already important when you're working alone since good commit messages help the future you understand what's going on, but it's even more important when you're collaborating with others since it gives your collaborators more context on why you made the change and can help them understand how to solve conflicts when necessary. So commit to being a good collaborator and remember to add those commit messages.

Whenever we collaborate with others, there's bound to be some merge conflicts and they can sure be a pain.

I've definitely been frustrated when encountering complex merge conflicts and trying to debug the results. If I'm dealing with this type of merge conflict, my first

step is to work backward and disable everything I've done and then see if the source still works, then I slowly add pieces of code until I hit the problem. That usually gets me through the tough times and has definitely highlighted some weird occurrences. Up next, we have a reading that puts together all the commands related to solving conflicts and then a quick practice quiz.

3.3.6 Reading: Conflict Resolution Cheat Sheet

Merge conflicts are not uncommon when working in a team of developers, or on Open Source Software. Fortunately, GitHub has some good documentation on how to handle them when they happen:

- https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-merge-conflicts
- https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/resolving-a-merge-conflict-using-the-command-line

You can also use git rebase branch name to change the base of the current branch to be branchname

The git rebase command is a lot more powerful. Check out this link for more information.

Chapter 4
Week 4

Appendix A
Chapter Heading

All's well that ends well

Use the template *appendix.tex* together with the Springer document class SVMono (monograph-type books) or SVMult (edited books) to style appendix of your book in the Springer layout.

A.1 Section Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the LaTeX automatism for all your cross-references and citations.

A.1.1 Subsection Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the LaTeX automatism for all your cross-references and citations as has already been described in Sect. A.1.

For multiline equations we recommend to use the `eqnarray` environment.

$$\begin{aligned} \mathbf{a} \times \mathbf{b} &= \mathbf{c} \\ \mathbf{a} \times \mathbf{b} &= \mathbf{c} \end{aligned} \tag{A.1}$$

A.1.1.1 Subsubsection Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the

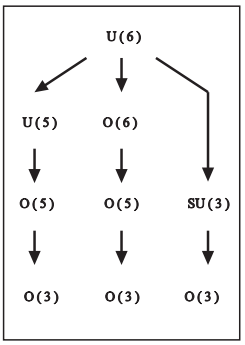


Fig. A.1 Please write your figure caption here

LaTeX automatism for all your cross-references and citations as has already been described in Sect. A.1.1.

Please note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

Table A.1 Please write your table caption here

Classes	Subclass	Length	Action Mechanism
Translation	mRNA[a]	22 (19–25)	Translation repression, mRNA cleavage
Translation	mRNA cleavage	21	mRNA cleavage
Translation	mRNA	21–22	mRNA cleavage
Translation	mRNA	24–26	Histone and DNA Modification

[a] Table foot note (with superscript)