

GCIS-123

Software Development & Problem Solving

*Unit 1.3: More Git, &
Environment Variables*

RIT

**Golisano College of
Computing and
Information Sciences**

Git Log

```
C:\Users\Hermione\SoftDevI> git log > log.txt
C:\Users\Hermione\SoftDevI> cat log.txt
commit 36a2f6daeada7044afcf73fe23475d62b93dca11
Author: Hermione Granger <hjk@hogwarts.edu>
Date: Mon Jun 8 16:50:41 2020 -0400
```

It's Wingardium LevioSAH.

```
commit 89cab2d26b20a68dbd4f54ca362bb5a0e4a81080
Merge: b4dcf70 f7a567c
Author: Ron Weasley <rbw@hogwarts.edu>
Date: Mon Jun 8 16:49:53 2020 -0400
```

Wingardium LeviOHsa!

You can also redirect the output from `git log` to a file, and then use the `cat` command to display the contents of the file in a command prompt. The output will not pause.

- Git not only keeps track of the last version of the files that you committed to your repository, it keeps track of **every** version of **every** file that you have **ever** committed to your repository.
 - This **does not** include files that you have modified in your local workspace but have not committed to your local repository.
 - The remote repository doesn't keep track of the version of any files that have not been pushed.
- The `git log` command will display a history of the versions of your repository.
- Each version in the Git log includes:
 - A unique **commit hash**.
 - The **author's name** and **email**.
 - The **time** and **date** that the version was created.
 - The **comment** entered with the commit. This is one place that good comments can be really useful.
- If the Git log is too long to fit in your command prompt window, it will pause and wait for a key press before continuing.
 - **Press Q to stop**.

1.3.3

Git Log & Cat

A Git Log can be used to see the history of changes that you and your team have made to a repository. The log includes a timestamp, a commit hash, and the comment for every commit to the repository. Try it out now.

- If necessary, launch a command prompt and navigate to your directory for today's activities.
- Run `git log` and examine the results.
- Run `git log` again, but redirect the output to a file named `git_log.txt`.
 - You can redirect output to a file using the `>` operator, e.g. `ls > list.txt` will redirect the output of the `ls` command into a file named "list.txt".
- Use `cat` to display the contents of the `git_log.txt` file and verify that it contains your Git log.
- Use the Git workflow to add the `git_log.txt` file to your repository.

```
C:\Users\Hermione\SoftDevI> git log > log.txt
C:\Users\Hermione\SoftDevI> cat log.txt
commit 36a2f6daeeda7044afcf73fe23475d62b93dca11
Author: Hermione Granger <hjk@hogwarts.edu>
Date: Mon Jun 8 16:50:41 2020 -0400
```

It's Wingardium LevioSAH.

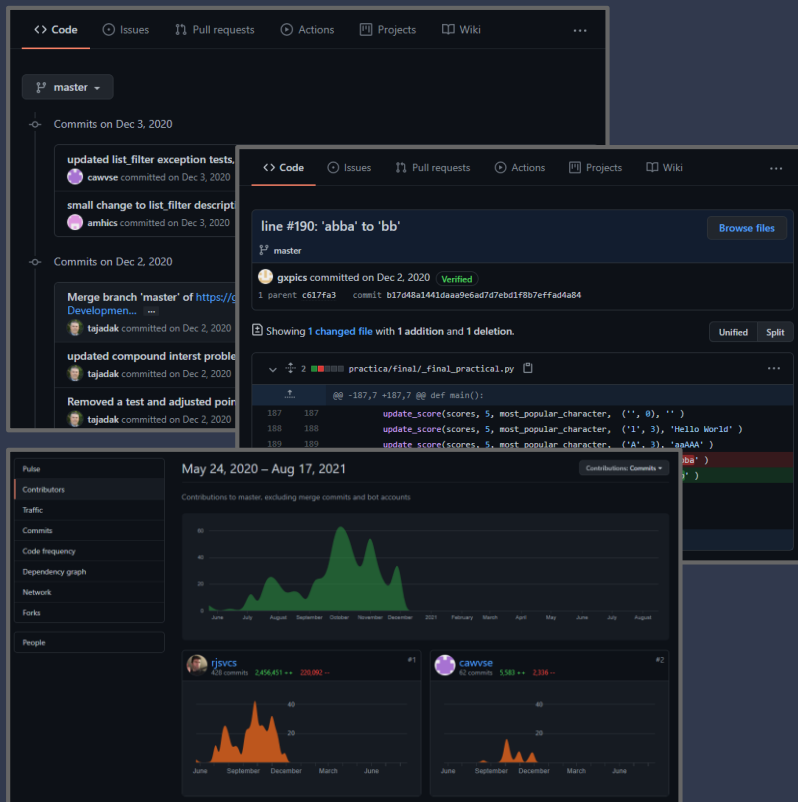
```
commit 89cab2d26b20a68dbd4f54ca362bb5a0e4a81080
Merge: b4dcf70 f7a567c
Author: Ron Weasley <rhw@hogwarts.edu>
Date: Mon Jun 8 16:49:53 2020 -0400
```

Wingardium LeviOHsa!

Part of your grade on each assignment will be based on how well you practice the Git workflow.

Your graders will use `git log` to see how often you commit and push to your repository.

GitHub History

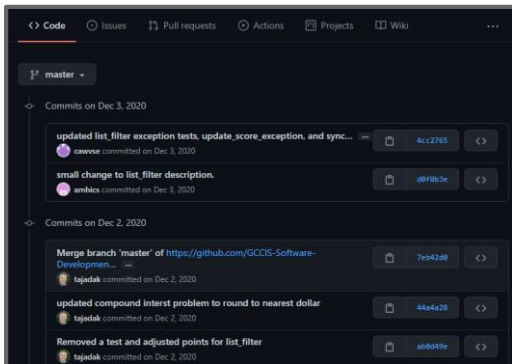


GitHub provides several different views of the history of changes to a repository.

- GitHub provides an alternate mechanism for looking at the history of a repository.
 - In fact, it provides more than one!
- You can see the **number of commits** in the top right corner of the repository.
 - Clicking this will open GitHub's equivalent of the Git log.
 - Clicking on the shortened **commit hash** next to any of the commits will open the commit and show the files that were added, modified, or removed in the commit.
- Alternatively, you can see the history for an individual file by opening the file and clicking the **History** option at the top right.
 - Clicking the **commit hash** next to any version of the file will open that version of the file and show lines that have been added, modified, or removed.
- You can view several different graphs of the activity in the repository as well.
 - Click **Insights** in the menu, then choose one of the options on the left.

GitHub History

Examining the history of changes to a repository can help you to understand how the repository has changed over time. It can also help you find past versions of the repository to view and/or recover the files.



- If you have not done so recently, practice the Git workflow to push the latest versions of your files to your repository.
 - You should be doing this after every activity!
- Open a browser window and navigate to any one of your GitHub repositories.
 - You may choose one of the repositories you used this week, or another repository that you have access to.
 - If you lost the URL, you can always click on the GitHub Classroom Invitation on MyCourses.
- Use the **number of commits** to open the commit history for the repository.
 - Take a screenshot and save it to your Unit 1.3 repository.
- Open an **individual file** and view the **history**.
 - Take a screenshot and save it.
- Use the **Insights** menu to open the **Contributors** and take a screenshot.

- Every entry in the output of git log includes a unique **commit hash**.
 - The commit hash is a 40-character hexadecimal string.
 - e.g. b4dcf708d4cd708f4ce05f26b4a4da3d46d06961
- The commit hash is generated by Git each time the repository is changed (files added, modified, deleted, etc.).
- Because it is **unique** for each version in the repository's version history, it can be used to uniquely identify one specific version.
- One way that the hash can be used is to revert the entire repository or even a specific file to a previous version.
 - The `git checkout` command can be used to do this.
 - e.g. `git checkout <hash> [filename]` where `<hash>` is the hash for the specific version, and `[filename]` is an *optional* specific file to revert.
- After using `git checkout`, you can use the standard Git workflow to continue editing the restored files.

Git Checkout

```
C:\Users\Hermione\SoftDevI> git log
commit 89cab2d26b20a68dbd4f54ca362bb5a0e4a81080
Merge: b4dcf70 f7a567c
Author: Ron Weasley <rbw@hogwarts.edu>
Date: Mon Jun 8 16:49:53 2020 -0400
```

Nuh uh...

```
commit 36a2f6daeeda7044afcf73fe23475d62b93dca11
Author: Hermione Granger <hjk@hogwarts.edu>
Date: Mon Jun 8 16:50:41 2020 -0400
```

It's Wingardium LevioSAH.

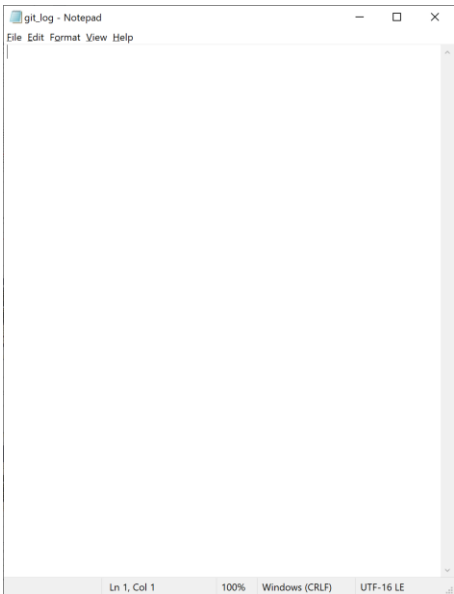
```
C:\Users\Hermione\SoftDevI> git checkout 36a2f6daeeda7044afcf73
fe23475d62b93dca11 spell.txt
```

You will use `git log` to list the version history of your repository and then copy the commit hash for the version to which you would like to revert.

Then you will paste it into your `git checkout`.

Git Checkout

In the event that you make a mistake or just want to undo some recent changes, the `git checkout` command can be used to restore a file to a previous version. Try it out now.



- If necessary, run a command prompt and navigate to your repository.
- Use Notepad to open your `git_log.txt` file and delete the contents.
- Use `cat` to verify that the file is now empty.
- Use the Git workflow to push the empty file to your repository.
- Use `git log` to show the version history for your repository and copy the penultimate commit hash.
- Use `git checkout` to check out the `git_log.txt` corresponding with that version, for example:
 - `git checkout ade14e00980ec3235a9608cdf4974b5ef74e6399 git_log.txt`
- Use `cat` to verify that the file has been restored.

A Detached HEAD?!



- If you make a mistake when performing a checkout, you may end up in a “detached HEAD” state. But what does this mean?
- First, **HEAD is just a shortcut for the most recent version of your repository.**
 - `git checkout HEAD` will always revert a file to the last version that you committed.
- Git includes support for **branches**. A **branch** is a virtual copy of your repository where you can make independent changes without affecting the files in the other branches.
 - The default branch is called **master**.
 - We won't be doing anything with branches this semester.
- A “detached HEAD” state means that your local repository is disconnected from its branch on the remote repository.

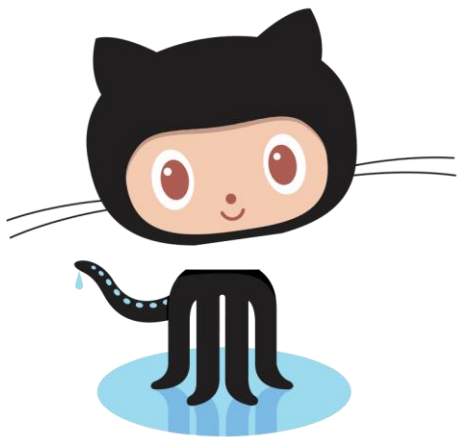
A “detached HEAD” state can happen if you use `git checkout` with a hash but without specifying a filename.

```
C:\Users\Hermione\SoftDevI> git checkout ade14e0
Note: switching to 'ade14e0980ec3235a9608cdf4974b5ef74e6399'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

```
C:\Users\Hermione\SoftDevI> git checkout master
Previous HEAD position was ade14e0 updated spell.txt file
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
C:\Users\Hermione\SoftDevI> _
```

The fix is to check out the master branch. This will reattach your local repository.



Detach and Reattach Your HEAD

Using `git checkout` without a specific filename will cause your repository to be in a "detached HEAD" state. Thankfully, it is easy to fix by checking out the master branch. Force it to happen now, and then fix it.

- If necessary, launch a new command prompt and navigate to your repository.
- Run a `git log` to see the version history.
 - Note that the top entry in the log is labeled `HEAD -> master`, meaning that it is the most recent version of the `master` branch.
- Use `git checkout` with one of the previous versions of your repository. **Do not** specify a file to check out.
 - You should see the "detached HEAD" message.
- Use `git checkout master` to reconnect your local repository with the master branch.

Git Restore

A `git restore` can be used to revert back to the most recently committed version of a file, and even restore files that have been deleted.

```
C:\Users\Hermione\SoftDevI> rm primes.txt
C:\Users\Hermione\SoftDevI> cat primes.txt
cat : Cannot find path 'primes.txt' because it does not exist.
C:\Users\Hermione\SoftDevI> git restore primes.txt
C:\Users\Hermione\SoftDevI> cat primes.txt
2 3 5 7 11 13 17 19 23 29
C:\Users\Hermione\SoftDevI> _
```

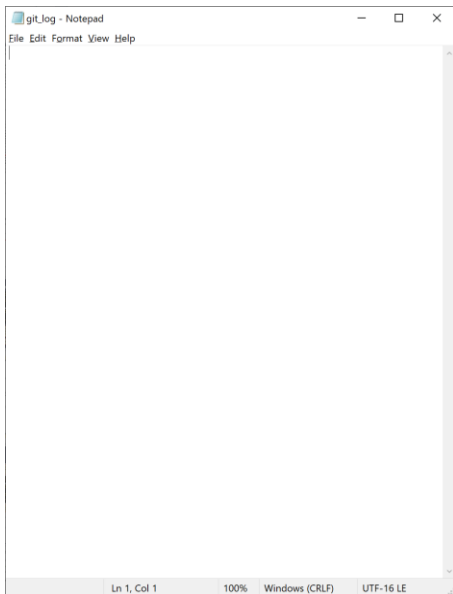
It should be used with caution, though! Once a file is restored any changes are permanently lost! Make sure that you really want to restore a file before you do it!

- Sometimes you would like to discard local changes that you made to a file.
 - You break some code that was working.
 - You accidentally delete something important.
 - You just want to start over.
 - etc.
- The `git restore <FILENAME>` command will replace the working copy of a file with the last version that was committed to the local repository.
 - Instead of a specific filename you may also use wildcards.
 - For example, you could use `*.py` to restore all of your Python files.
- Using `git restore` is essentially identical to using `git checkout HEAD <FILENAME>`.
 - In both cases the HEAD version of the file replaces the working copy.
 - Arguably, `git restore` is a little easier to use.

Git Restore

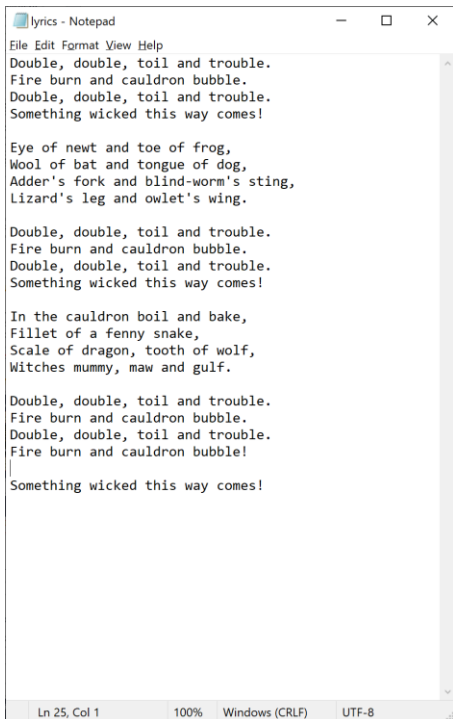
The `git restore` command can be used to revert a modified (or even deleted!) file to the version that was most recently committed to the local repository. This is a convenient way to undo any changes you made since the last commit. Try it now.

- If necessary, run a command prompt and navigate to your repository.
- Use Notepad to open your `git_log.txt` file and delete the contents.
- Use `cat` to verify that the file is now empty.
- Use `git restore` to restore the `git_log.txt` to the HEAD version.
 - e.g. `git restore git_log.txt`
- Use `cat` to verify that the file has been restored.



Creating a Conflict

Clone another copy of your repository and then make conflicting edits to the same text file in two different places.



```
lyrics - Notepad
File Edit Format View Help
Double, double, toil and trouble.
Fire burn and cauldron bubble.
Double, double, toil and trouble.
Something wicked this way comes!

Eye of newt and toe of frog,
Wool of bat and tongue of dog,
Adder's fork and blind-worm's sting,
Lizard's leg and owl's wing.

Double, double, toil and trouble.
Fire burn and cauldron bubble.
Double, double, toil and trouble.
Something wicked this way comes!

In the cauldron boil and bake,
Fillet of a fenny snake,
Scale of dragon, tooth of wolf,
Witches mummy, maw and gulf.

Double, double, toil and trouble.
Fire burn and cauldron bubble.
Double, double, toil and trouble.
Fire burn and cauldron bubble!

Something wicked this way comes!
```

Ln 25, Col 1 100% Windows (CRLF) UTF-8

- If necessary, start a command prompt and navigate to your repository.
- Use Notepad to create a file called `lyrics.txt` that contains the lyrics from one of your favorite songs (feel free to use Google).
- Use the Git workflow to push the new file to your repository.
- Create a new directory under **Unit01** named **temp** and change into it.
- Clone your repository into this directory.
 - e.g. `git clone https://www.github.com/SoftDevI/assignment-03-hjg`
- Edit the `lyrics.txt` file to add your name to the first line of the file, and use the Git workflow to push it to your remote repository.
- Change back into the other copy of your repository. **Do not pull.**
- Edit the `lyrics.txt` file to add today's date to the first line of the file, and use the Git workflow to push it to your remote repository.
 - What happens?
 - What do you see when you run a `git status` check?

- Throughout this semester you may do some work on a computer in the classroom, and then clone your repository to continue working on a different computer, e.g. your laptop or a desktop in your dorm.
 - In fact, you may move back and forth several times.
- This means that you will have multiple copies of the same GitHub project in more than one location.
 - If you **push** a change from one, you will need to **pull** the change to the other.
 - In fact, if your local repository is not up to date with the remote repository Git will **require** you to synchronize with a **git pull** before you can push.
- Having more than one copy of your repository in different places means that you may end up making different changes to the same file.
- Most of the time Git transparently **merges** changes together to create a version of the file with all of the changes.
- A **merge conflict** occurs when multiple, overlapping changes are made to the same line(s) in the file.
 - If this happens, the changes must be merged **manually**.

Merge Conflicts

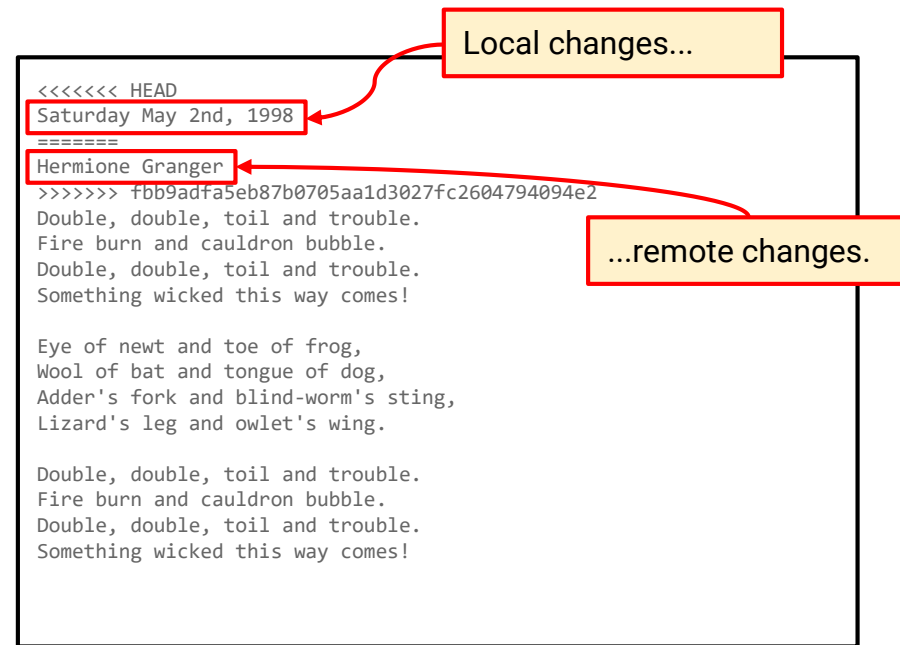
When you execute a **git pull**, Git will notify you if it can't automatically merge the local copy of a file with the remote version.

```
C:\Users\Hermione\SoftDevI> git pull
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 4 (delta 3), reused 4 (delta 3), pack-reused 0
Unpacking objects: 100% (4/4), 460 bytes | 51.00 KiB/s, done.
From https://github.com/SoftDevI/assignment-03-hjg
   59a676c..fbb9adf  master    -> origin/master
Auto-merging lyrics.txt
CONFLICT (content): Merge conflict in lyrics.txt
Automatic merge failed; fix conflicts and then commit the
result.
```

This will require you to solve the merge conflict manually, and then commit and push the merged file to the repository.

Resolving Merge Conflicts Manually

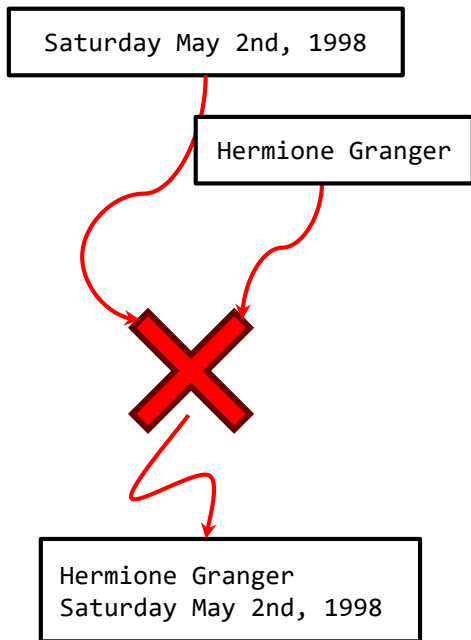
- After executing a `git pull`, Git will notify you if there are any changes that cannot be merged automatically.
- A version of the file will have been created with **all** of the changes separated by special characters.
 - `<<<<<<< HEAD` indicates the beginning of the changes in the HEAD version in the local repository.
 - `=====` is the delimiter between the remote and local changes, which start on the next line.
 - `>>>>>>> <HASH>` indicates the end of the changes in version of the file in the remote repository.
 - There may be **multiple conflicts!**
- For each conflict you will need to:
 - Choose which changes to **keep** (maybe both!)
 - **Delete** all of the other unwanted text (including the special characters).
 - Use the Git workflow to **push** the merged file to the remote repository.



Resolve a Merge Conflict

Resolve the merge conflict in your file and then push the merged file to your repository.

- If necessary, launch a command prompt and navigate to your repository.
 - **Not** the temporary repository that you created to cause the conflict.
- Synchronize your local repository with a **git pull**.
 - You should see that the lyrics.txt file could not be merged.
- Use Notepad to open the file.
- Edit the file so that **both** changes are included (your name *and* today's date).
 - Keep both sets of changes.
 - Remove any unwanted text, e.g. special characters.
- Use the Git workflow to push the merged file to your repository.



Environment Variables

- Environment variables are important values that affect how the programs and operating system on a computer behave.
- Every environment variable has a unique **name** and **value**. Some important variables include:
 - PATH - determines where your operating system will search for executable applications or scripts.
 - PATHEXT - determines which file extensions are considered executable.
 - HOMEPATH - the path to the user directory.

Environment variables are stored on a virtual drive named `env` that is accessible from the command prompt.

You may display the environment variables on a computer by listing the contents of the virtual drive, e.g. `ls env:`

You may display an individual environment variable using `$env:<NAME>` where `<NAME>` is the name of the variable, e.g. `$env:PATH`

You may also redirect output to a file using the `>` operator after **any** command. For example, if you wanted to store the list of files in C:\ in a file named "files.txt": `ls C:\ > files.txt`

Displaying Environment Variables I

Listing the environment variables on your computer can be a quick way to see the variables that have been set and/or confirm that a variable has been set to the right value. Try it now.

- If necessary, launch a command prompt and navigate to your Day02 directory.
- Recall that the **gdr** command can be used to list the drives on a computer. Use it and find the virtual drive named “**env**” in the list.
- List the contents of the drive to display all of the environment variables on your computer.
- List the variables again, but this time redirect the output to a file named “**envvars.txt**”.
- Use the Git workflow to upload the new file to your remote repository.

```
PS C:\Users\ron> ls env:
```

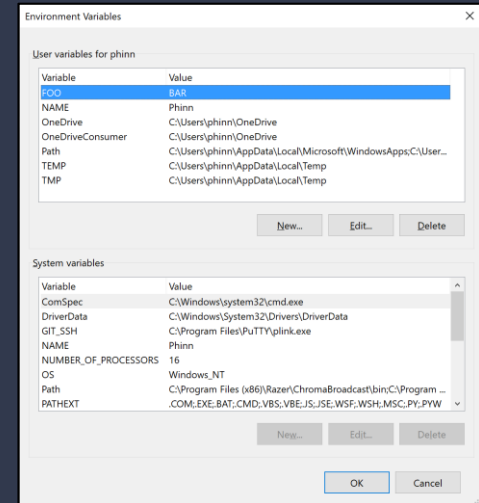
Name	Value
ALLUSERSPROFILE	C:\ProgramData
APPDATA	C:\Users\ron\AppData\Roaming
CommonProgramFiles	C:\Program Files\Common Files
CommonProgramFiles(x86)	C:\Program Files (x86)\Common Files
CommonProgramW6432	C:\Program Files\Common Files
COMPUTERNAME	THEBURROW
ComSpec	C:\Windows\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
GIT_SSH	C:\Program Files\Putty\plink.exe
HOMEDRIVE	C:
HOMEPATH	\Users\ron
LOCALAPPDATA	C:\Users\ron\AppData\Local
LOGONSERVER	\\THEBURROW
NAME	ron
NUMBER_OF_PROCESSORS	16
OneDrive	C:\Users\ron\OneDrive
OneDriveConsumer	C:\Users\ron\OneDrive
OS	Windows_NT
Path	C:\Program Files (x86)\Razer\ChromaBroadcast\... COM; .EXE; .BAT; .CMD; .VBS; .VBE; .JS; .JSE; .WSF; ...
PATHEXT	AMD64
PROCESSOR_ARCHITECTURE	Intel64 Family 6 Model 158 Stepping 13, Genui...
PROCESSOR_IDENTIFIER	6
PROCESSOR_LEVEL	968d
PROCESSOR_REVISION	968d
ProgramData	C:\ProgramData
ProgramFiles	C:\Program Files
ProgramFiles(x86)	C:\Program Files (x86)
ProgramW6432	C:\Program Files
PSModulePath	C:\Users\ron\OneDrive\Documents\WindowsPowe...
PUBLIC	C:\Users\Public
SESSIONNAME	Console
SystemDrive	C:
SystemRoot	C:\Windows
TEMP	C:\Users\ron\AppData\Local\Temp
TMP	C:\Users\ron\AppData\Local\Temp
USERDOMAIN	THEBURROW
USERDOMAIN_ROAMINGPROFILE	THEBURROW
USERNAME	ron
USERPROFILE	C:\Users\ron
windir	C:\Windows

```
PS C:\Users\ron> _
```

- **status, add, commit, push.**

Editing Environment Variables

- While it is possible to permanently set environment variables from the command line, it is fairly convoluted.
- It is far easier to use the **environment variables editor** provided by Windows to add or change environment variables.
- There are two different categories of environment variables.
 - **User variables** are environment variables specific to your user account.
 - **System variables** are system-wide and apply to any account logged into the computer.
- Depending on your permissions, you may only be able to change the user variables.
- Launching the editor is not very intuitive:
 - Press the Windows key.
 - Type “environment” into the search field.
 - Use the arrows keys to choose **Edit environment variables for your account**.
- From here you can **create, edit, or delete** environment variables.



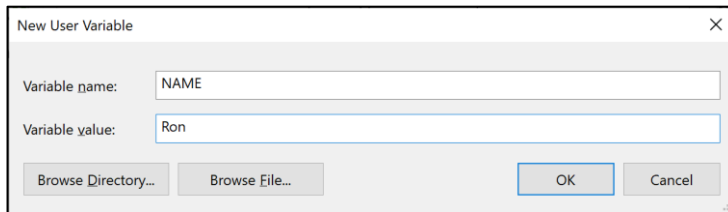
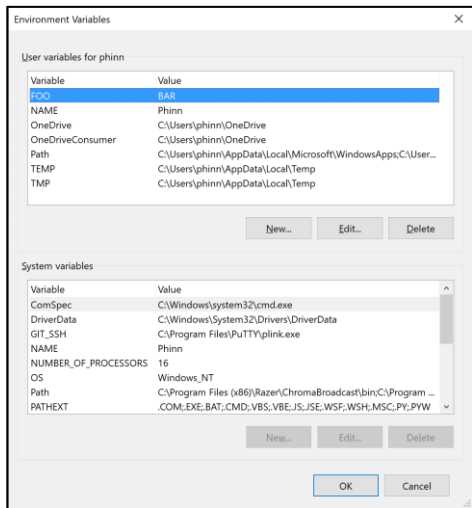
Changes made to environment variables through the editor are “sticky” and system-wide.

They will not apply to any currently opened command prompts; you will need to restart PowerShell to see the changes.

Add a New Environment Variable II

You may need to view, create, or modify the environment variables on your computer to change the way programs behave.

- Launch the Environment Variables editor.
- In the User variables section, click the **New...** button.
- Create a new environment variable named **NAME** with your first name as the value.



- If necessary, close any open command prompts and then launch a new prompt.
- Display the environment variables. You should see your new environment variable in the list.

Creating New Environment Variables

Trying to display an environment variable that doesn't exist will no produce any output.

```
PS C:\users\ron> $env:PET
PS C:\users\ron> _
```

A new environment variable can be created using \$env to set the value (in quotes).

```
PS C:\users\ron> $env:PET = 'SCABBERS'
PS C:\users\ron> $env:PET
SCABBERS
PS C:\users\ron> _
```

Existing environment variables can also be changed using \$env.

```
PS C:\users\ron> $env:HOMEPAH = 'C:\temp'
PS C:\users\ron> _
```

- It is possible to create new environment variables (or change existing ones) from the command prompt using the command
`$env:<NAME> = '<VALUE>'`
 - <NAME> is the name of the variable to create or change.
 - '<VALUE>' is the value. The value must be enclosed in either single (') or double (") quotes.
- The **scope** of environment variables created this way is restricted to the command prompt through which they are created.
 - If another command prompt is launched, the changes will not be present.
 - **Once the command prompt is closed, the changes are lost.**
- Creating environment variables in this way can be useful for a number of reasons:
 - Testing changes to important variables like PATH.
 - A temporary change is needed, e.g. while running a script.
 - Creating/changing variables based on user input.
 - etc.

Add a New Environment Variable I

Use the command prompt to create a new environment variable.

Create a new environment variable and display it.

- If necessary, launch a command prompt.
- Create a new environment variable NAME and assign it a value of your first name.
- Display the new environment variable.
- Close the command prompt and launch a new one. Display all of the environment variables. What do you notice?

```
PS C:\Users\røn> ls env:

Name                           Value
----                           -
ALLUSERSPROFILE                C:\ProgramData
APPDATA                        C:\Users\røn\AppData\Roaming
CommonProgramFiles             C:\Program Files\Common Files
CommonProgramFiles(x86)        C:\Program Files (x86)\Common Files
CommonProgramW6432             C:\Program Files\Common Files
COMPUTERNAME                   THEBURROW
ComSpec                        C:\Windows\system32\cmd.exe
DriverData                     C:\Windows\System32\Drivers\DriverData
GIT_SSH                        C:\Program Files\Putty\plink.exe
HOMEDRIVE                      C:
HOMEPATH                      \Users\røn
LOCALAPPDATA                   C:\Users\røn\AppData\Local
LOGONSERVER                    \THEBURROW
NAME                           røn
NUMBER_OF_PROCESSORS           16
OneDrive                       C:\Users\røn\OneDrive
OneDriveConsumer               C:\Users\røn\OneDrive
OS                             Windows_NT
Path                           C:\Program Files (x86)\Razer\ChromaBroadcast\...
PATHEXT                        .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;...
PROCESSOR_ARCHITECTURE         AMD64
PROCESSOR_IDENTIFIER           Intel64 Family 6 Model 158 Stepping 13, Genuine...
PROCESSOR_LEVEL                 6
PROCESSOR_REVISION             9e8d
ProgramData                    C:\ProgramData
ProgramFiles                   C:\Program Files
ProgramFiles(x86)              C:\Program Files (x86)
ProgramW6432                   C:\Program Files
PSModulePath                   C:\Users\røn\OneDrive\Documents\WindowsPowe...
PUBLIC                          C:\Users\Public
SESSIONNAME                    Console
SystemDrive                    C:
SystemRoot                     C:\Windows
TEMP                           C:\Users\røn\AppData\Local\Temp
TMP                             C:\Users\røn\AppData\Local\Temp
USERDOMAIN                     THEBURROW
USERDOMAIN_ROAMINGPROFILE      THEBURROW
USERNAME                       røn
USERPROFILE                    C:\Users\røn
windir                         C:\Windows
```

```
PS C:\Users\røn> _
```

The Path

The PATH is actually divided into two parts: the **system path**, which is global to all users on the computer, and the **user path**, which is specific to the currently logged in user.

Depending on your level of access, you may not be able to change the system path (e.g. on lab computers).

The full path is created by concatenating the two together - the system path always comes **first**, and therefore will be searched first.

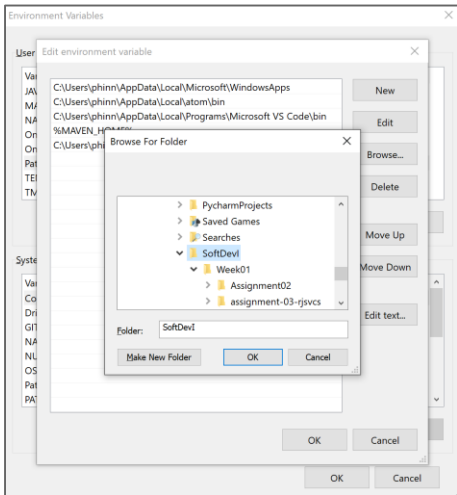
This is just one reason why it is not a good idea to name your executables the same as existing system commands.

- Without doing anything special, running programs requires you to type the path to the program.
 - From the current directory: `.\hello.exe`
 - Two directories up: `..\..\hello.exe`
 - etc.
- But what if you wanted to run a program from anywhere without having to type the path?
- The **path** is an environment variable that is essentially a list of directories in your file system.
 - Usually, each directory contains at least one executable program.
- Whenever you try to execute a command, Windows will search for an executable file with the same name in each of the directories in your path, **in order**.
 - It will attempt to execute the **first** matching file that it finds.
 - This means if you have two programs with the same name, it will execute the one that appears **earlier** in your path.
 - This can happen if you have two different versions of a program installed, e.g. Python versions 2 and 3.
- The **PATHEXT** environment variable determines which extensions are considered executable.

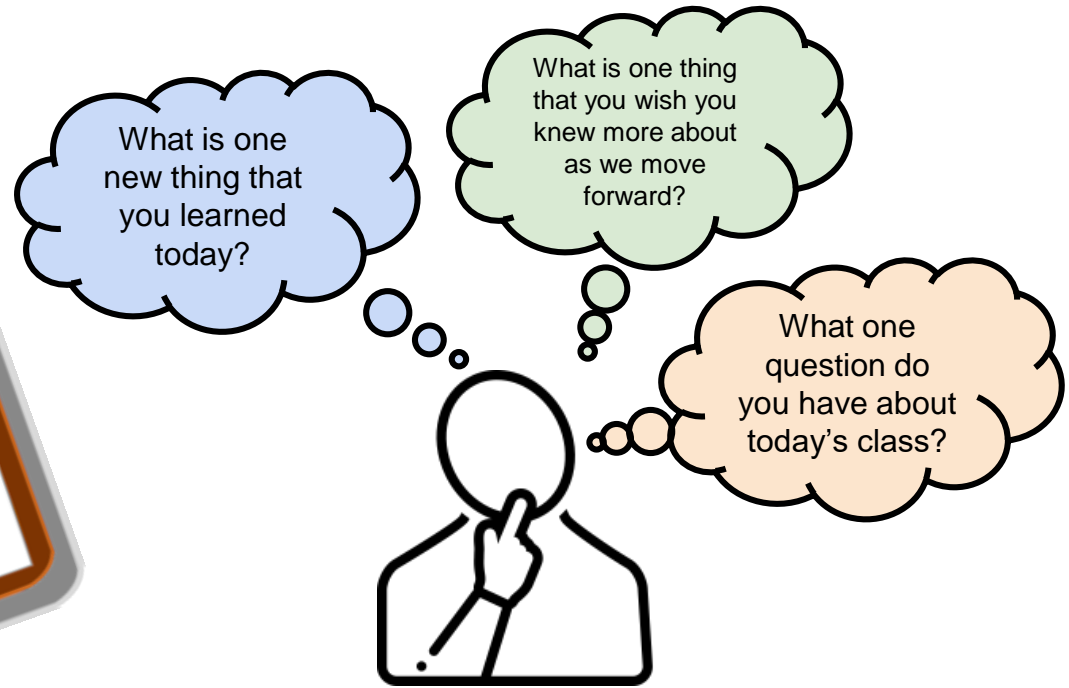
Modify the User Path

You will sometimes need to make changes to your `PATH` so that the correct program executes when you run it from the command line.

- Close any currently opened command prompts.
- Open the **environment variables editor**.
 - Press the Windows-key and search the start menu for “environment.”
 - Make sure to pick “Edit environment variables for your account.”
- Edit the **Path** user variable.
 - In the dialog, click the **New** button at the top right. This will add a new entry to the bottom of the list.
 - With the new entry selected, click the **Browse...** button.
 - In the dialog, find your `SoftDev1` directory, and click **OK**.
- Launch a new command prompt.
 - Use the `$env:PATH` command to show your path.
 - Is your `SoftDev1` directory in the path?



Summary & Reflection



Please answer the questions above in your notes for today.

Homework Assignment 1.3

- Software Development & Problem Solving is a fast paced course that can be challenging, especially for students new to computing.
- The homework assignments are designed to give you an opportunity to practice between lectures.
- Each is designed to take about 90-120 minutes.
- The assignments will also help you to identify topics with which you need more help. Ask questions!

You can find the full instructions for this and any other assignment on MyCourses under Content.

- If you are using a computer other than the one that you used in class today, you will need to clone your repository to the new computer.
- You will then continue to practice with the (slightly) more advanced Git features that we used in class today:
 - Git Log
 - Checkout/Restore
- You will also:
 - Take some screen shots
 - Get more practice with the command line
- You will need to make sure that all of your work is pushed to your GitHub repository before the start of the next class!
 - There are no extensions for late assignments!