

- The UNIX workbench
  - Week 1
    - Command line basics
    - Creation & Inspection
    - Migration & Destruction
  - Week 2
    - Self-help
    - Wildcards
    - Character sets
    - Search: Regular expressions
    - Find
    - Configure
    - Differentiate
    - Pipes
    - Make
  - Week 3
    - Maths
    - Variables
    - Arguments
    - User input - read
    - Logic and If/Else
    - Conditional expressions: `[[ expression ]]`
    - if and else
    - Arrays
    - Braces
    - Loops
    - Writing Functions
    - Getting values from functions
    - Writing Programs
  - Week 4
    - Git and GitHub
    - Important Git Features
    - Nephology
    - Start Building

# The UNIX workbench

---

## Week 1

---

### Command line basics

Notion of absolute/relative path, cd, ~/, / (root level), pwd, ls

## Creation & Inspection

```
wc lines, words, bytes # -n option for displaying line numbers
```

## Migration & Destruction

```
echo, touch, mkdir, cat  
(con**cat**enate, try with 2 arguments)
```

- >, >> to add a line to a (non)empty file and to append at the end, respectively.
- Consider moving files to either ~/.local/share/Trash/ (Linux) or ~/.Trash/ (Mac)

## Week 2

---

### Self-help

```
man <command> #/ to initiate a search, n to move to the next occurrence, Shift +  
n to move to the previous one.
```

```
info
```

```
apropos
```

### Wildcards

\* for zero or more of any character.

### Character sets

```
egrep -v #option for inverted match
```

## Search: Regular expressions

Regular expressions are strings that define patterns.

The real power of regular expressions comes from using metacharacters.

## egrep

- Capture groups
- Escaping-anchors-odds-and-ends

There's a mnemonic that I love for remembering which metacharacter to use for each anchor:

"First you get the power, then you get the money."

Metacharacter	Meaning
.	Any Character
\w	A Word
\W	Not a Word
\d	A Digit
\D	Not a Digit
\s	Whitespace
\S	Not Whitespace
[def]	A Set of Characters
[^def]	Negation of Set
[e-q]	A Range of Characters
^	Beginning of String
\$	End of String
\n	Newline
+	One or More of Previous
*	Zero or More of Previous
?	Zero or One of Previous

	Either the Previous or the Following
{6}	Exactly 6 of Previous
{4, 6}	Between 4 and 6 or Previous
{4, }	More than 4 of Previous

[RegExr](#) is an online tool to learn, build, & test Regular Expressions (RegEx / RegExp).




## Find

```
find # <directory where to look> -method (ie. -name) "filename.ext"
```

### Summary

- grep and egrep can be used along with regular expressions to search for patterns of text in a file.
- Metacharacters are used in regular expressions to describe patterns of characters.
- find can be used to search for the names of files in a directory.

### Exercises

-  Search states.txt and canada.txt for lines that contain the word "New".
-  Make five text files containing the names of states that don't contain one of each of the five vowels.
-  Download the GitHub repository for this book and find out how many .html files it contains.

## Configure

### History

```
history
head -n 5 ~/.bash_history
grep "some_string" ~/.bash_history
```

### Customizing bash

```
alias docs='cd ~/Documents'
alias edbp='nano ~/.bash_profile'
```

```
source ~/.bash_profile
```

history displays what commands we've entered into the console since opening our current terminal.

## Summary

- The ~/.bash\_history file lists commands we've used in the past.
- alias creates a command that can be used as a substitute for a longer command that we use often.
- The ~/.bash\_profile is a text file that is run every time we start a shell, and it's the best place to assign aliases.

## Differentiate

- diff & sdiff

```
diff four.txt six.txt
sdiff four.txt six.txt
```

## Pipes

- The pipe ( | ) takes the output of the program on its left side and directs it to be the input for the program on its right side.

## Exercises

- ☐ Use pipes to figure out how many US states contain the word "New."
- ☐ Examine your ~/.bash\_history to try to figure out how many unique commands you've ever used. (You may need to look up how to use the uniq and sort commands).

## Make

How to install programs on UNIX? 1. Download and/or extract into a directory, 2. cd into that directory 3. Run make

Concept of makefile : a file that describes the relationships between different files and programs. In addition to installing programs, make is also useful for creating documents automatically.

```
[target]: [dependencies...]
[commands...]
```

## [Makefile Tutorial by Example](#)

## Summary

- make is a tool for creating relationships between files and programs, so that files that depend on other files can be automatically rebuilt.
- makefiles are text files that contain a list of rules.
- Rules are made up of targets (files to be built), commands (a list of bash commands that build the target), and dependencies (files that the target depends on to be built).

## Week 3

---

### Maths

- The \* (multiplication) operator must be escaped, otherwise bash thinks we're trying to define a RegExp.
- / is for integer division, ie. always rounded down to the nearest integer




```
expr 1 / 3
```

```
bc (**b**ench **c**alculator) is for complex maths
```

### Summary

- Bash programs are executed in order from the first line in a file until the last line.
- Anything written after a pound sign (#) is a comment and is not executed by Bash.
- You can do simple arithmetic with the expr command.
- Perform more complicated arithmetic by piping a string expression into bc using echo.

### Exercises

-  Look at the man pages for bc.
-  Try doing some math in bc interactively.
-  Try writing some equations in a file and then provide that file as an argument to bc.

### Variables

Make sure you follow these rules when you're naming variables:

- Every character should be lowercase.
- The variable name should start with a letter.
- The name should only contain alphanumeric characters and underscores.
- Words in the name should be separated by underscores.

If you follow those rules then you can avoid accidentally overwriting data stored in environmental variables.

```
chapter_number=5
echo $chapter_number

let chapter_number=$chapter_number+1
echo $chapter_number

the_empire_state="New York"
echo $the_empire_state

math_lines=$(cat math.sh | wc -l)
echo $

echo "I went to school in $the_empire_state."
```

## Arguments

```
#!/usr/bin/env bash
#File: vars.sh

echo "Script arguments: $@"
echo "First arg: $1. Second arg: $2."
echo "Number of arguments: $#"
```

## Summary

- Variables can be assigned with the equal sign ( = ) operator.
- Strings or numbers can be assigned to variables.
- The value of a variable can be accessed with the dollar sign ( \$ ) before the variable name.
- You can use the dollar sign and parentheses syntax (command substitution) to execute a command and save the output in a variable.
- You can access command line arguments within your own scripts using the dollar sign followed by the number of the argument.

## Exercises

- ☐ Write a Bash program where you assign two numbers to different variables, and then the program prints the sum of those variables.
- ☐ Write another Bash program where you assign two strings to different variables, and then the program prints both of those strings. Write a version where the strings are printed on the same line, and a version where the strings are printed on different lines.
- ☐ Write a Bash program that prints the number of arguments provided to that program multiplied by the first argument provided to the program.


## User input - read

read stores a string that the user provides in a variable.

```
#!/usr/bin/env bash
#file letsread.sh

echo "Type in a string and then press Enter:"
read response
echo "You entered: $response"
```

### Exercises

-  Write a script that asks the user for an adjective, a noun, and a verb, and then use those words in a sentence (like Mad Libs).

## Logic and If/Else

When writing computer programs it is often useful for your program to be able to make decisions based on inputs like arguments, files, and environmental variables.

```
true
false
```

Notion of exit status:

```
echo $?
```

Logical operators: `&&` ( AND ), `||` ( OR )

In a series of programs joined together by AND operators, any programs to the right of a program that has a non-zero exit status is not executed.

Commands on the right hand side of `||` are only executed if the command on the left hand side fails and therefore has an exit status other than 0.

You can combine AND and OR operators in commands, which are evaluated from left to right.

By combining AND and OR operators you can precisely control the conditions for when certain commands should be executed.

## Conditional expressions: `[[ expression ]]`

Notion of conditional expression



Enabling your Bash script to make decisions is extremely useful.

Conditional execution allows you to control the circumstances where certain programs are executed based on whether those programs succeed or fail, but you can also construct conditional expressions which are logical statements that are either equivalent to true or false.

Conditional expressions either compare two values, or they ask a question about one value. Conditional expressions are always between double brackets (`[[ ]]`), and they either use logical flags or logical operators.

Tip to quickly look at the resulting value of a logical expression:

```
[[ 4 -gt 3 ]] && echo t || echo f
[[ 3 -gt 4 ]] && echo t || echo f
```

## Binary logical expression vs. Unary logical expressions

- Logical flags

Logical Flag	Meaning	Usage
<code>-gt</code>	Greater Than	<code>[[ \$planets -gt 8 ]]</code>
<code>-ge</code>	Greater Than or Equal To	<code>[[ \$votes -ge 270 ]]</code>
<code>-eq</code>	Equal	<code>[[ \$fingers -eq 10 ]]</code>
<code>-ne</code>	Not Equal	<code>[[ \$pages -ne 0 ]]</code>
<code>-le</code>	Less Than or Equal To	<code>[[ \$candles -le 9 ]]</code>
<code>-lt</code>	Less Than	<code>[[ \$wives -lt 2 ]]</code>
<code>-e</code>	A File Exists	<code>[[ -e \$taxes_2016 ]]</code>
<code>-d</code>	A Directory Exists	<code>[[ -d \$photos ]]</code>
<code>-z</code>	Length of String is Zero	<code>[[ -z \$name ]]</code>
<code>-n</code>	Length of String is Non-Zero	<code>[[ -n \$name ]]</code>

- Logical operators

One of the most useful logical operators is the regex match operator `=~`

Logical Operator	Meaning	Usage

<code>=~</code>	Matches Regular Expression	<code>[[ \$consonants =~ [aeiou] ]]</code>
<code>=</code>	String Equal To	<code>[[ \$password = "pegasus" ]]</code>
<code>!=</code>	String Not Equal To	<code>[[ \$fruit != "banana" ]]</code>
<code>!</code>	Not	<code>[[ ! "apple" =~ ^b ]]</code>

## if and else

```
#!/usr/bin/env bash
# File: simpleif.sh

echo "Start program"

if [[ $1 -eq 4 ]]
then
    echo "You entered $1"
fi

echo "End program"
```

You should also know that you can combine conditional execution, conditional expressions, and IF/ELIF/ELSE statements. The conditional execution operators AND ( `&&` ) and OR ( `||` ) can be used in an IF or ELIF statement.

```
#!/usr/bin/env bash
# File: condexif.sh

if [[ $1 -gt 3 ]] && [[ $1 -lt 7 ]]
then
    echo "$1 is between 3 and 7"
elif [[ $1 =~ "Jeff" ]] || [[ $1 =~ "Roger" ]] || [[ $1 =~ "Brian" ]]
then
    echo "$1 works in the Data Science Lab"
else
    echo "You entered: $1, not what I was looking for."
fi
```

The conditional execution operators work just like they would on the command line. If the entire conditional expression evaluates to the equivalent of true then the code within the IF statement is executed, otherwise it is skipped.

Finally we should note that IF/ELIF/ELSE statements can be nested inside of other IF statements.

```
#!/usr/bin/env bash
# File: nested.sh

if [[ $1 -gt 3 ]] && [[ $1 -lt 7 ]]
then
    if [[ $1 -eq 4 ]]
    then
        echo "four"
```

```

elif [[ $1 -eq 5 ]]
then
echo "five"
else
echo "six"
fi
else
echo "You entered: $1, not what I was looking for."
fi

```

In order to get to the inner IF statement, the conditions for the outer IF statement must be met first (the first argument for the script must be between 3 and 7). As you can see combining variables, arguments, conditional expressions, and IF statements allow you to write more powerful Bash programs.

## Summary

- All Bash programs have an exit status. true has an exit status of 0 and false has an exit status of 1.
- Conditional execution uses two operators: AND ( && ) and OR ( || ) which you can use to control what command get executed based on their exit status.
- Conditional expressions are always in double brackets ([[ ]]). They have exit an exit status of 0 if they contain a true assertion or 1 if they contain a false assertion.
- IF statements evaluate conditional expressions. If an expression is true then the code within an IF statement is executed, otherwise it is skipped.
- ELIF and ELSE statements also help control the flow of a Bash program, and IF statements can be nested within other IF statements.

## Exercises

- ☐ Write a Bash script that takes a string as an argument and prints “how proper” if the string starts with a capital letter.
- ☐ Write a Bash script that takes one argument and prints “even” if the first argument is an even number or “odd” if the first argument is an odd number.
- ☐ Write a Bash script that takes two arguments. If both arguments are numbers, print their sum, otherwise just print both arguments.
- ☐ Write a Bash script that prints “Thank Moses it’s Friday” if today is Friday. (Hint: take a look at the date program).

## Arrays

```

plagues=(blood frogs lice flies sickness boils hail locusts darkness death)

echo ${plagues[0]}
echo ${plagues[3]}

```



```
echo ${plagues[*]} # get all array elements

echo ${plagues[*]}
plagues[4]=disease # change an individual elements in the array by specifying
their index with square brackets
echo ${plagues[*]}
```

## Summary

- Arrays are a linear data structure with ordered elements which can be stored in variables.
- The each element of an array has an index and the first index is 0.
- Individual elements of an array can be accessed using their index.

## Exercises

-  Write a bash script where you define an array inside of the script, and the first argument for the script indicates the index of the array element that is printed to the console when the script is run.
-  Write a bash script where you define two arrays inside of the script, and the sum of the lengths of the arrays are printed to the console when the script is run.

## Braces

Brace expansion uses the curly brackets and two periods ({ .. }) to create a sequence of letters or numbers:

```
echo {0..9}
echo {a..e}
echo {W..Z}
```

You can put strings on either side of the curly brackets and they'll be "pasted" onto the corresponding end of the sequence:

```
echo a{0..4}
echo b{1..3}c
```

You can also combine sequences so that two or more sequences are pasted together:

```
echo {1..3}{A..C}
```

If you want to use variables in order to define a sequence you need to use the eval command in order to create the sequence:

```
start=4
end=9
```

```
echo {$start..$end}
eval echo {$start..$end}
```


In fact you can do this with any number of strings:

```
echo {Who, What, Why, When, How}?
```

## Summary

- Braces allow you create string sequences and expansions.
- To use variables with braces you need to use the eval command.

## Exercises

-  Create 100 text files using brace expansion.

## Loops

Loops allow you to repeat lines of code based on logical conditions or by following a sequence.

### FOR loops

Valid sequences include brace expansions, explicit lists of strings, arrays, and command substitutions, see `manyloops.sh`

### WHILE loops

The WHILE loop combines parts of the FOR loop and the IF statement, see `whileloop.sh`

```
#!/usr/bin/env bash
# File: whileloop.sh
```

The WHILE loop begins first with the while keyword followed by a conditional expression. As long as the conditional expression is equivalent to true when an iteration of the loop begins, then the code within the WHILE loop will continue to be executed.

If the logical expression is never equivalent to false then we've created an infinite loop.

By changing the value of the variable in the logical expression inside of the loop we're able to ensure that the logical expression will eventually be equivalent to false, and therefore the loop will eventually end.

Control + C breaks out of an infinite loop, or any running program.

## Nesting

Just like IF statements for and while loops can be nested within each other.



Besides nesting loops within each other you can also nest loops within IF statements and IF statements within loops, see ifloop.sh

There are endless combinations for nesting IF statements and loops, but one good rule of thumb you should remember is that your nesting should never go more than two or possibly three layers deep. If you find yourself writing code with lots of nesting, you should consider restructuring your program. Deeply nested code is difficult to read and even more difficult to debug if your program contains mistakes.

## Summary

- Loops allows you repeat sections of your program.
- FOR loops iterate through a sequence so that a variable that you assign takes on the value of every element of the sequence in every iteration of the loop.
- WHILE loops check a conditional statement at the beginning of every iteration. If the condition is equivalent to true then one iteration of the loop is executed and then the conditional statement is checked again. Otherwise the loop ends.
- IF statements and loops can be nested in order to make more powerful programming structures.

## Exercises

-  Write several programs with three levels of nesting and include FOR loops, WHILE loops, and IF statements. Before you run your program try to predict what your program is going to print. If the result is different from your prediction try to figure out why.
-  Enter the yes command into the console, then stop the program from running. Take a look at the man page for yes to learn more about the program.

## Writing Functions

A function is a small piece of code that has a name. Writing functions allows us to re-use the same code multiple times across programs.

```
#!/usr/bin/env bash
# File: ntmy.sh

function ntmy {
echo "Nice to meet you $1"
}
```

The source command, which allows us to use function definitions in bash scripts as command line commands.

```
source ntmy.sh
ntmy Benjo
> Nice to meet you Benjo
```

Once you close your current shell you'll lose access to the ntmy command, but in the next section we'll discuss how to

set up your own commands so that you always have access to them.

Example: create a function that allows to add up a sequence of numbers of arbitrary length.

It's important to break down a larger goal into a series of individual components before writing a program, that way we more easily can identify which features and tools will be required.

```
#!/usr/bin/env bash
# File: addseq.sh

function addseq {
sum=0

for element in $@
do
let sum=sum+$element
done

echo $sum
}

source addseq.sh

addseq 10 90 3
addseq 0 1 1 2 3 5 8 13
```

## Getting values from functions

Functions are used for two primary purposes: computing values and side effects.

A side effect occurs whenever a function (or a command) creates or changes files on our computer. These commands don't print any value if they succeed.

When you create variables in functions those variables become globally accessible, meaning that even after the program is finished that variable retains its value in your shell.

This is an example of one strategy we can use to retrieve values that a function has calculated. Unfortunately this approach is problematic because it changes the values of variables that we might be using in our shell.

In order to avoid this problem it's important that we use the `local` keyword when assigning variables within a function. The `local` keyword ensures that variables outside of our function are not overwritten by our function.






### Summary

- Functions start with the `function` keyword followed by the name of the function and curly brackets ( `{ }` ).
- Functions are small, reusable pieces of code that behave just like commands.
- You can use variables like `$1`, `$2`, and `$@` in order to provide arguments to functions, just like a Bash script.
- Use the `source` command in order to read in a Bash script with function definitions so that you can use your functions in your shell.
- Use the `local` keyword to prevent your function from creating or modifying global variables.

- Be sure to echo the results of your function (if there are any) so that they can be captured with command substitution.

## Exercises

Below this list of exercises you can find examples of how these programs should work when used on the command line.

-  Write a function called `plier` which multiplies together a sequence of numbers.
-  Write a function called `isiteven` that prints 1 if a number is even or 0 a number is not even.
-  Write a function called `nevens` which prints the number of even numbers when provided with a sequence of numbers. Use `isiteven` when writing this function.
-  Write a function called `howodd` which prints the percentage of odd numbers in a sequence of numbers. Use `nevens` when writing this function.
-  Write a function called `fib` which prints the number of fibonacci numbers specified.

## Writing Programs

### The Unix Philosophy

Unix tools were designed along a set of guidelines which are best summarized by Ken Thompson's idea that each Unix program should do one thing well. Following this rule when writing functions and programs accomplished several goals:

- Limiting a program to only doing one thing reduces the length of the program, and the shorter a program is the easier it is to fix if it contains bugs or if it needs to be revised.
- Writing short programs also helps the users of your code understand what's going on in your code in the event that they need to read your code. Reading a poem induces a different cognitive load compared to reading a novel.
- Folks who don't read the source code of your program (most users won't - they shouldn't have to) will be able to understand the inputs, outputs, and side effects of your program more easily.
- Using small programs to write a new program will increase the likelihood that the new program will also be small. Composability is the concept of stringing small programs together to create a new program.

The concept of composability in Unix is best illustrated by the use of the pipe operator ( `|` ) for creating pipelines of programs. When you're considering what inputs your program is going to have and what your program is going to print to the console you should consider whether or not your program might be used in a pipeline, and you should organize your program accordingly.

In the previous section we discussed the difference between functions that compute values and functions that produce side effects. You should notice that the side effect functions like `mv` and `cp` do not print any text to the console if they are successful.

The concept of quietness is another important part of the Unix philosophy. Quietness in this case means that a function should not print to the console unless it is necessary, either to inform the user of a value (`pwd`), to display the



result of a computation (bc), or to warn the user that an error has occurred.

## Making programs executable - Permissions

The `chmod` command takes two arguments. The first argument is a string which specifies how we're going to change permissions for a file, and the second argument is the path to the file. The first argument has to be composed in a very specific way. First we can specify which set of users we're going to change permissions for:

```
chmod u+x some_file.text
```

We then need to specify whether we're going to add, remove, or set the permission:

Character	Meaning
u	The owner of the file
g	The group that the file belongs to
o	Everyone else
a	Everyone above

We then need to specify whether we're going to add, remove, or set the permission:

Character	Meaning
+	Add permission
-	Remove permission
=	Set permission

Finally we specify what permission we're changing:

Character	Meaning
r	Read a file
w	Write to or edit a file
x	Execute a file

To run an executable file we need to specify the path to the file, even if the path is in the current directory, meaning we

need to prepend `./` to a program.

Symbolic:	r--	-w-	--x		421
Binary:	100	010	001		-----
Decimal:	4	2	1		000 = 0
					001 = 1
Symbolic:	rwX	r-X	r-X		010 = 2
Binary:	111	101	101		011 = 3
Decimal:	7	5	5		100 = 4
	/	/	/		101 = 5
Owner	---/	/	/		110 = 6
Group	-----/	/	/		111 = 7
Others	-----/	/	/		Binary to Octal chart

## Environmental variables

An environmental variable is a variable that Bash creates where data about your current computing environment is stored. Environmental variable names use all capitalized letters.





```
echo $HOME
echo $PWD
echo $PATH
```

## Summary

- According to the Unix Philosophy you should keep your programs short, simple, and quiet.
- Use `chmod` to make your programs executable.
- You can modify your `~/.bash_profile` in order to make scripts and functions available to use on the command line.
- Use `export` to change an environmental variable.

## Exercises

Below this list of exercises you can find examples of how the programs described here should work when used on the command line.

-  Make a script executable.
-  Put that script in a directory that you create and make that directory part of your `PATH`.
-  Write a program called `range` that takes one number as an argument and prints all of the numbers between that number and 0.
-  Write a program called `extremes` which prints the maximum and minimum values of a sequence of numbers.

## Week 4

# Git and GitHub

## Git and GitHub Basics

```
git --version
git config --list

git config --global user.name "myUserName"
git config --global user.email "myUserName@myHost.net"
```

First we need to create a directory:

```
cd
mkdir my-first-repository
cd my-first-repository

git init
echo "Welcome to My First Repo" > readme.txt
git status

git add readme.txt

git status

git rm --cached readme.txt

git add readme.txt

git commit -m "added readme.txt"

git status
touch file1.txt
touch fil2.txt

echo "Learning Git is going well so far." >> readme.txt

git status
```

If we want to track all of the changes to all of the files in our directory we should use the command `git add -A`.

```
git add -A
git status
git commit -m "added two files"
```

We made a typo! Let's fix it.

```
git reset --soft HEAD~
git status

mv fil2.txt file2.txt
git status
```

```
git add -A
git status

git commit -m "added two files"
```

## Summary

- Git tracks changes to plain text files (code files and text documents).
- A directory where changes to files are tracked by Git is called a Git repository.
- Change your working directory, then run git init to start a repository.
- You can track changes to a file using git add [names of files].
- You can create a milestone about the state of your files using git commit -m "message about changes since the last commit".
- To examine the state of files in your repository use git status.

## Exercises

- ☐ Start a repository in a new directory.
- ☐ Create a new file in your new Git repository.
- ☐ Make sure Git is tracking the file and then create a new commit. Make changes to the file, and then commit these changes.
- ☐ Add two new files to your repository, but only commit one of them.
- ☐ What is the status of your repository after the commit? Undo the last commit, add the untracked file, and redo the commit.

## Important Git Features

### Gitting Help, Logs and Diffs

```
git help status

git log

echo "The third line." >> readme.txt
git diff readme.txt
```

A plus sign shows up next to the added line

```
nano readme.txt
# Delete the second line
```

```
git diff readme.txt
```

A minus sign appears next to the line we deleted.

```
git status  
git checkout
```

The changes made to readme.txt have been undone.




## Ignoring files

```
touch toby.jpg  
git status  
  
echo "*.jpg" > .gitignore  
git status  
  
git add -A  
git commit -m "added gitignore"  
  
touch bernie.jpg  
git status
```

## Summary

- `git help` allows you to read the man pages for specific Git commands.
- `git log` will show you your commit history.
- `git diff` displays what has changed between the last commit and your current untracked changes.
- You can specify a `.gitignore` file in order to tell Git not to track certain files.

## Exercises

-  Look at the help pages for `git log` and `git diff`.
-  Add to the `.gitignore` you already started to include a specific file name, then add that file to your repository.
-  Create a file that contains the Git log for this repository. Use `grep` to see which day of the week most of the commits occurred on.

## Branching

Creating different Git branches allows you to work on a particular feature or set of files independently from other “copies” of a repository.

You can list all of the available branches with the command `git branch`:

```
git branch
```

The star ( \* ) indicates which branch you're currently on. The default branch that is created is always called master. Usually people use this branch as the working version of the software that they are writing, while they develop new and potentially unstable features on other branches.

To add a branch we'll also use the git branch command, followed the name of the branch we want to create:

```
git branch my_new_feature
git branch
```

We can make my-new-feature the current branch using git checkout with the name of the branch:

```
git status
##On branch my-new-feature
##nothing to commit, working tree clean
git branch
```

If we look at git status we can also see that it will tell us which branch we're on:

```
git status
```

We can switch back to the master branch using git checkout:

```
git checkout master
git branch
```

Now we can delete a branch by using the -d flag with git branch and the name of the branch we want to delete

```
git branch -d my-new-feature
git branch
```

We can create a new branch and switch to that branch at the same time using the command git checkout -b and the name of the new branch we want to create:

```
git checkout -b update-readme
```

Now that we've created and switched to a new branch, let's make some changes to a file:

```
echo "I added this line in the update-readme branch." >> readme.txt
cat readme.txt
```

Now that we've added a new line let's commit these changes:

```
git add -A
git commit -m 'added a thrid line to readme.txt'
```

Now that we've made a commit on the update-readme branch, let's switch back to the master branch:

```
git checkout master
cat readme.txt
```

The third line that we added is gone! We committed the change to this file while we were on the update-readme branch, so the updated file is safely in that branch:

```
git checkout update-readme
cat readme.txt
```

And the third line is back! Let's add and commit yet another line while we're on this branch:

```
echo "It's sunny outside today." >> readme.txt
git add -A
git commit -m "added weather info"
```

## Merging

Now that we've made a couple of changes to readme.txt, let's combine those changes with what we have in the master branch. This is made possible by a Git merge.

Git incorporates other branches into the current branch by default. When you're merging, the current branch is also called the base branch. Let's switch to the master branch so we can merge in the changes from the update-readme branch:

```
git checkout master

git merge update-readme
cat readme.txt
```

What if there are two commits in two separate branches that make different edits to the same line of text? When this occurs it is called a conflict. Let's create a conflict so we can learn how they can be resolved.

```
git checkout update-readme
nano readme.txt
cat readme.txt
```

```
git add -A
git commit -m "changed sunny to cloudy"

git checkout master
nano readme.txt
cat readme.txt
git add -A
git commit -m "changed sunny to windy"
```

We've now created two commits that directly conflict with each other. On the update-readme branch the last line says It's cloudy outside today., while on the master branch the last line says It's windy outside today.. Let's see what happens when we try to merge update-readme into master.

```
git merge update-readme
```

Uh-oh, there's a conflict! Let's check the status of the repo right now:

```
git status
cat readme.txt
```

In Git terminology the HEAD represents the most recent commit on the branch which is currently checked out (which is master in this case).

```
nano readme.txt
cat readme.txt



git add -A
git commit -m "resolved conflict"
```

[Git Book \(open source\)](#)


## Summary

- Git branching allows you and others to work on the same code base together.
- You can create a branch with the command `git branch [name of branch]`.
- To switch to a branch use `git checkout [name of branch]`.
- You can combine a branch with your current branch by using `git merge`.

## Exercises

-  Start a new branch.
-  Switch to that branch and add commits to it. Switch to an older branch and then merge the new branch into your current branch.



-  Purposefully create and resolve a merge conflict.

## GitHub

```
git remote  
git remote add origin https://github.com/user/repo.git
```

We'll need to use the `-u` flag in order to set origin as the default remote repository so we don't have to provide its name every time we want to interact with it.

```
git push -u origin master
```

## Markdown

Markdown is a powerful markup language because it's small, intuitive, and readable when it's written as plain text. GitHub transforms Markdown files (which end in the file extension `.md`) into simple HTML web pages in your repository. If there is a file called `README.md` in any folder in your repository, then that file is rendered to HTML and displayed on GitHub.

[Mastering Markdown](#) [Markdown Editor](#)

## Pull requests and Forking

A pull request allows you to interactively compare two different branches before you merge them so you can either go ahead with the merge or provide feedback to whoever opened the pull request.

Essentially a pull request allows a person to ask another person if they're willing to incorporate changes on one branch into another branch.

This social coding transaction may involve you and a collaborator, you and a stranger, or you might open a pull request on your own repository just as a method of staying organized.

When working on a remote GitHub repository with many other folks these pull requests and merges can happen without you being involved at all if the commits effect parts of the code that you're not working on. Still it's important to keep your local repository up to date with the latest changes in the remote repository.

```
git checkout master  
git pull
```

## GitHub Pages

[GitHub Pages](#) allows you to create and host a website on GitHub using only Git and Markdown.

## Forking

Forking a GitHub repository copies somebody else's GitHub repository into your GitHub account. You can then modify this copy of their software however you like.




The original source repository (the repository you forked) is often called the upstream repository.

```
git clone https://github.com/[your-github-username]/the-unix-workbench.git
git remote -v
```

## Summary

- You can use GitHub you create and host remote Git repositories. A remote Git repository is a Git repository that is always connected to the internet.
- List remote repositories with `git remote`.
- Add remote repositories with `git remote add [name-of-remote] https://github.com/\[username\]/\[repo-name\].git`
- Add commits to your remote repository with `git push [name-of-remote] [name-of-branch]` or just `git push` if you've set up a default remote and branch.
- To merge commits on a remote repository into your local repository use `git pull [name-of-remote] [name-of-branch]` or just `git pull` if you've set up a default remote and branch.
- A pull request allows you to interactively compare two different branches before you merge them.
- [GitHub Pages](#) allows you to host websites written in Markdown for free!
- Forking a repository allows you to make changes to a copy of a public repository. You can then open a pull request if you think your changes should be merged into the upstream repository!

## Exercises

-  Create a new repository on GitHub. Clone your repository and add a README.md file. Push this file to GitHub and create a GitHub Pages website for this repository.
-  Fork an existing repository (try one of mine [seankross](#)) and try to identify something valuable you could contribute. Make changes or additions to that repository, then open a pull request.
-  Read through GitHub's Guides.

## Nephology

"The cloud is just someone else's computer"

### Nephology Introduction

[Digital Ocean 2 months free trial](#)

```
ssh [username]@[IP address]
```

```
logout
```

## Cloud Computing Basics

- Moving Files In and Out

Copy a file and an entire repository from remote machine to local one:

```
scp [username]@[IP address]:path/to/file/on/server path/on/my/computer  
scp -r [username]@[IP address]:path/to/folder/on/server folder/on/my/computer
```

Copy from local machine to a remote one:

```
scp path/on/my/computer [username]@[IP address]:path/to/file/on/server
```

- Talking to other servers

To download a file with curl, we simply need to provide the -O flag and the URL of the file:

```
curl -O http://website.org/textfile.txt
```

The curl command is also commonly used for communicating with APIs.

```
curl https://api.github.com/repos/seankross/the-unix-workbench/languages
```

HTTP requests & verbs:

```
curl http://httpbin.org/get
```

In the general case we can provide arguments to an HTTP API by putting a question mark (?) after the API's URL.

```
curl http://httpbin.org/get?Baltimore
```

We can specify an argument's name with the template [argument name]=[argument value].

```
curl http://httpbin.org/get?city=Baltimore
```

We can add more named arguments by separating them with an ampersand ( & )

```
curl "http://httpbin.org/get?city=Baltimore&state=Maryland"
```

- Automating tasks

One of the most commonly used programs for executing other programs with a regular frequency is called cron. First, let's see if cron is running. We can get a list of all running programs with the `ps` command while using the `-A` flag:

```
ps -A
#or better
ps -A | grep "cron"
```

In order to assign programs to be executed with cron we need to edit a special text file called the cron table. Before we edit the cron table we need to select the default text editor.

```
select-editor
...
```




Now that we've chosen a text editor we can edit the cron table using the command `crontab -e` (cron table edit) which will automatically open nano with the appropriate file.

```
crontab -e

# Example of job definition:
# .----- minute (0 - 59)
# | .----- hour (0 - 23)
# | | .----- day of month (1 - 31)
# | | | .----- month (1 - 12) OR jan,feb,mar,apr ...
# | | | | .----- day of week (0 - 6) (Sunday=0 or 7) OR
# | | | | | sun,mon,tue,wed,thu,fri,sat
# | | | | |
# * * * * * user command to be executed]
```

Star (\*) which represents all of the possible values in a column.

## Cloud Computing Exercises

-  Write a bash script that takes a file path as an argument and copies that file to a designated folder on your server.
-  Find a file online that changes periodically, then write a program to download that file every time it changes.
-  Try creating your own Twitter or GitHub bot with the Twitter API or the GitHub API.

## Shutting Down a Server

## Start Building