Code Academy

**Multiple Colors**

The text is now displayed in multiple colors! There is a new variable named letterColors with a list of color variables inside brackets ([])separated by commas (,).

letterColors = [red, orange, green, blue, purple];

The displayed text in the browser panel will cycle through the values in letterColors in order when drawName() is called with letterColors.

Learning to code is all about experimentation—trying out new code and observing the result! As you experiment, you may want to reset your code to the starting state. This can be done with the reset button at the bottom of your code editor:

**WELCOME TO CODECADEMY!**

**Change Physics!**

Part of the power of coding is that small changes can create big impacts. Now that you’ve experimented with changing the color and text, it’s time to experiment with the animation itself!

**Instructions**

**1.**

Three variables let you experiment with the animation physics: mouseResponseThreshold, friction, and rotationForce.

mouseResponseThreshold affects how close the mouse pointer needs to be to affect the dots that make up the letters. The larger the number, the more powerful the effect of the mouse interaction. Experiment with changing the mouseResponseThreshold to different numbers and running your code!

Stuck? Get a hint

**2.**

Try experimenting with the friction value! You’ll probably want to keep friction between 0 and 1.

Stuck? Get a hint

**3.**

Finally, try experimenting with the rotationForce value. This variable represents how much each animated dot will try to rotate when interacting with the mouse. Try keeping this value small, maybe around 0.01.

**VARIABLES**

**Introduction to Variables**

In programming, we have a way of storing values so that we can reuse them throughout our program or change them, if necessary. This concept is known as a *variable*.

Maybe you’re familiar with the math term *variable*, or the word *variability*. Programming variables share qualities with each. Similar to a variable in math, a variable is a way of saving a piece of information with a specific name. By giving a value a name, we can easily reuse that value over and over again in our code. Like the idea of variability, a programming variable allows us to easily change a value throughout our code.

Variables are an important first step in coding because with variables we can start using an important programming tool: repetition. Rather than writing out a piece of data every time we need it, we write it out once and the computer remembers it and can repeat that information back.

In this lesson, we’ll explore why variables are an important part of programming, including:

* Using variables to reuse a value
* Using variables to change a value

**Instructions**

Imagine that you’re building a game using tiles with different patterns, representing different [terrains](https://en.wikipedia.org/wiki/Terrain).

The tiles need to be placed in the following order:

First row:

* grass
* rocks
* grass

Second row:

* forest
* rocks
* forest

Third row:

* rocks
* grass
* rocks

In each box, enter the type of terrain that should go in the accompanying tile. Continue until the entire board is filled.

**VARIABLES**

**Reusing Values in Variables**

One reason we use variables is that they allow us to easily *reuse* values in different parts of our code.

When we reuse a value, it will appear in multiple places in our code. Re-typing that value becomes tedious, which leads to errors, and without a variable name, it becomes unclear as to what the value represents. It may also be unclear as to what that value is meant to represent.

Let’s take a look at the following piece of code. Here we have a number that we reuse in order to make some calculations:

847595593392818109495  
847595593392818109495 \* 2  
847595593392818109495 / 4

Rather than writing the same number over and over again, we can save it to a variable named my\_number:

my\_number = 847595593392818109495  
my\_number \* 2  
my\_number / 4

You may be thinking, “But what if my variable name is longer than the value it stores? What’s the point of a variable?”

When we use a value without assigning it to a variable, that’s known as *hardcoding*. While it’s sometimes faster to initially hardcode values in your program, in the long run you’ll run into trouble — especially if you need to change what those values are.

**Instructions**

The last time we built our game, we repeated ourselves a lot. Instead of writing out the pattern for each tile, let’s save the types of terrain we want to use to a set of variables that we can use across the board.

Replicate the previous design by making each variable equal to one of the following:

* grass
* rocks
* forest

As you enter your choices, what do you notice about the board that changes?

How was this process different from the last exercise? Did variables make this process more efficient?

**VARIABLES**

**Changing the Value of a Variable**

The strong selling point of using a variable is that we can easily change their value, making our programs flexible.

As we saw in the last exercise, we can save a number to a variable and reuse it throughout a program:

my\_number = 847595593392818109495  
my\_number \* 2  
my\_number / 4

We could easily switch out the value of my\_number, without having to change that number in multiple places in our code.

my\_number = 1  
my\_number \* 2  
my\_number / 4

We could also change the value of my\_number part way through our program:

my\_number = 1  
my\_number \* 2  
   
my\_number = 3  
my\_number / 4

**Instructions**

You’d like there to be options for an expansion pack, and that means — more terrains!

Use the drop-down menus to explore the different options for the board and try creating new game designs by swapping out the different terrains.

**VARIABLES**

**Putting it All Together**

Now that you know a bit about variables, let’s code some ourselves!

We create or *declare* a variable by giving it a name and setting it equal to a value.

terrain = 'lake'

In the code editor, we’ve written out an example of a variable declaration. We’ve included one variable in the workspace, named one, and it’s currently set to 'grass'.

**Instructions**

Below variable ‘one’, declare a variable called two. Set variable ‘two’ to equal 'rocks'. Remember to pay attention to details, like punctuation.

Next, create a variable three and set it equal to 'forest'.

Run the code! How does the board game change?

Now, try replacing the values with other ones that you’ve seen in this lesson! Here’s the full list:

Group 1:

* grass
* rocks
* forest

Group 2:

* lake
* beach
* town

Group 3:

* glacier
* desert
* moon

**VARIABLES**

**Review - Variables**

Congrats! Now you’ve learned about variables and why they’re useful in programming. In this lesson, we learned that:

* *Variables* allow us to store information
* Variables allow us to quickly *reuse* a value in our program.
* Variables let us easily *change* a value in our program.

In the next lesson, we’ll look at different kinds of information that we can use in a program and save to variables.

**DATA TYPES**

**Data Types Review**

Congrats! Now you know the building blocks of most programming languages and should start to have an idea about the types of simple programs that you could write.

In this lesson, we learned:

* Categorizing information using *data types* is a way for a computer to distinguish different types of input.
* *Primitives* are the simplest data types and are shared across many programming languages. They include:
  + *Numbers* — values that allow us to do calculations and keep count.
  + *Strings* — a sequence of characters or symbols often used to denote text.
  + *Booleans* — logical values that represent the idea of true or false.

**OPERATORS**

**Introduction**

In computing, we work with lots of different forms of data. But the real fun comes when we can *do* something with this data.

In this lesson, we’ll look at how we can write short computer programs using *operators*.

Operators are different symbols that represent an *operation*, such as the plus sign (+) as a symbol for addition. Operations enable us to *process* our data, to transform it into something else.

You may already be familiar with operators and operations from other contexts. This lesson looks at different ways we can use operators in programming, including:

* Making calculations using *arithmetic* operators.
* Comparing information using *comparison* operators.
* Creating logical expressions using *logical* (aka Boolean) operators.

**Instructions**

We can think of writing a program as similar to following a recipe.

In this lesson, we’ll look at how we can use operators to make a fruit salad.

To make the salad, we’ll use operators to:

* Calculate the fruit needed
* Compare the different fruits
* Determine if we have the right ingredients

Right now, we can only add or take away data points by manually including them or deleting them from our code, sort of like adding and subtracting.

When you’re ready, move to the next exercise.

**Using Operators to Make Comparisons**

When writing a program, we often need to check if a value is correct or compare two values.

*Comparison operators* allow us to compare values and evaluate their relationship. Rather than evaluating to an integer, they evaluate to true or false, AKA boolean values. Expressions that evaluate to boolean values are known as *boolean expressions*.

Comparison operators include:

* Less than < — value to the left is **less than** the value to the right: 2 < 6
* Greater than > — value to the left is **more than** the value to the right: 14 > 5
* Equals == — value to the left is **equal to** the value to the right: 3 == 3

Note: we use a double-equal sign to show that we’re checking a value, rather than setting it equal to something, like we would with a variable. Some languages even use a triple-equal sign === to super-triple-check!

There are two main instances where we use comparison operators:

**If we have an unknown quantity.** What if we knew that we needed a half pound of strawberries, but we didn’t know the weight of each strawberry? We could weigh the strawberries and see if the total weight equals a half pound.

**OPERATORS**

**Putting it All Together**

Now that we know a bit about different operators, let’s try writing our own expressions!

As we’ve seen, we can use operators to change something’s value, add other things to our program, to take them away, or to perform calculations. We can also use operators to compare statements or determine whether they are true or false.

One important thing about operators when you program is that they may look different from language to language. That’s because different programming languages have different *syntax*, or rules for how they are written.

**Instructions**

Let’s use the operators that we’ve learned to add different fruits to the screen and determine their relationships.

* Create a variable named orange and set it equal to 4.
* **Decrease** the number of oranges from 4 to 2. Save the result to the orange variable again.
* Run the code and see how many oranges appear on the screen.

**OPERATORS**

**Review**

Congrats! Now you know about different operators and how they can be used to build simple programs.

In this lesson, we learned:

* *Operators* are symbols that represent different ways of modifying, comparing, and evaluating information.
* *Arithmetic operators* are used to make calculations.
* *Comparison operators* determine the relationship between two values, which results in a boolean.
* *Logical operators* determine the logical state of multiple boolean values or expressions, which results in another boolean.

Now that you know the building blocks of operators, how can you further combine them to write programs?

**FUNCTIONS**

**Functions**

This lesson requires you to know the basics of hamburger-making:

* Place the bread down
* Add the burger patty
* Add the pickles
* Place the bread on top

What if you had to say each step every time you ordered a hamburger? It’s tedious. It takes a long time. (How do you fit that on a menu?) And it risks making mistakes.

In this lesson you will learn a solution to that problem: functions. You’ll learn how and why they are used in programming and be able to communicate their benefits to other programmers!

Functions are used throughout programming — in fact, some styles of programming [rely completely on functions](https://en.wikipedia.org/wiki/Functional_programming). Knowing this information will be useful for anywhere your programming path takes you.

**Instructions**

Build hamburgers without a function: make three hamburgers by clicking on each instruction.

Hint: Add bread – Add burger patty – Add pickles – Add bread.

**Concept Review**

Want to quickly review some of the concepts you’ve been learning? Take a look at this material's [cheatsheet](https://www.codecademy.com/learn/paths/code-foundations/tracks/learn-how-to-code/modules/bop-ii/cheatsheet" \t "_blank)!

**Functions for Flexibility**

Not everyone wants to eat hamburgers.

We could write a new function for each new sandwich type, but that takes a lot of work and risks making mistakes.

Instead, we’ll generalize the hamburger function to a sandwich function. This new sandwich function will still make a bread-topping-topping-bread combination, but the toppings may change based on inputs to the function:

function makeSandwich(topping1, topping2) {

Add bread

Add topping1

Add topping2

Add bread

}

We’ve renamed the function makeSandwich() and given it two inputs, or *parameters*. Each time we call the function, we’ll give actual values for each input, called *arguments*.

For example, we make a ham-and-cheese sandwich with makeSandwich("ham", "cheese"). We call the function with the arguments “ham” and “cheese”. Those will be the values for the topping1 and topping2 parameters.

Instead of writing a different function for each type of sandwich, we have one function that can make them all!

**Instructions**

Call the makeSandwich() function with the arguments "ham" and "cheese".

Notice how the instructions change with different inputs.

**FUNCTIONS**

**Functions for organization**

A sandwich wouldn’t be complete without fries and dessert!

Here are the instructions to make the complete meal:

* Add bread, Add burger patty, Add fried potatoes, Add pickles, Add bread, Add salt, Add chocolate scoop, Add ketchup, Add vanilla scoop, Add strawberry scoop

Oof. All of the instructions are there, but they’re confusing and hard to edit. If you decide to ask for a fruit salad instead of fries, you would have to find all of the fries-related instructions and replace every line.

To make it easier to read, we’ll separate and organize our instructions:

* Add bread, Add burger patty, Add pickles, Add bread
* Add fried potatoes, Add salt, Add ketchup
* Add chocolate scoop, Add vanilla scoop, Add strawberry scoop

Better… Now let’s group these instructions into three functions:

* makeSandwich()
* makeSomeFries()
* makeIceCream()

We can easily substitute one line, makeSomeFries(), for a different function, like makeFruitSalad(). And if there’s an error, we know where to look for the recipe for each part of the meal.

The new instructions are starting to look like a program! By using functions, we made it easier to read, reusable, and *modular:* each set of related instructions (sandwich, fries, dessert) is grouped into its own function, which we can easily add, remove, and swap to make a diversity of meals.

**Instructions**

*“I’d like a sandwich, fries, and triple ice cream scoop, please!”*

1. Make this lunch WITHOUT functions
2. Make this lunch again WITH functions

Questions: Which approach was easier? Which approach was less prone to error?

*“I’d like 3 sandwiches, 2 fries, and 2 triple ice cream scoop, please!”*

1. Click the reset icon
2. Make this lunch WITHOUT the functions
3. Make this lunch WITH functions

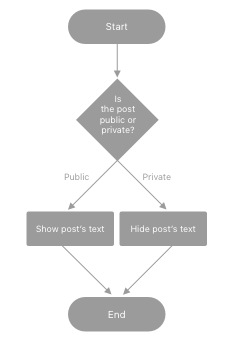
Question: Which approach was easier this time?

**FUNCTIONS**

**Review**

Well done! Functions are a fundamental concept in programming, and learning the basics will serve you wherever your path leads, regardless of language or domain. To review:

* A *function* is a sequence of instructions that performs a specific task, packaged as a unit.
* When we *define* a function, we specify the instructions, inputs, and name of the function.
* When we *call* a function, all of its instructions are executed.
* Functions can be executed many times, making its instructions *reusable*.
* Functions can have *parameters*, which accept input values, making its instructions *flexible*.
* Functions organize a program into distinct units, making interchanging and editing them easier. This makes your entire program organized and *modular*.



The control flow shown above is written as a *conditional* control structure in **main.js**. It looks something like this:

if (condition) {  
  DO SOMETHING  
} else {  
  DO SOMETHING DIFFERENT  
}

If the condition evaluates to true, the computer will execute the body of that statement. In this case, it’s DO SOMETHING.

If the condition is false, the computer will execute the body of the else statement. In this case, it’s DO SOMETHING DIFFERENT.

Let’s adapt this general template to our example:

1. The condition is true when the privacy mode is set to 'public'. This is already written for you in **main.js**.
2. You’ll replace DO SOMETHING with a JavaScript function: showDetails().
3. You’ll replace DO SOMETHING DIFFERENT with another function: hideDetails().

The instructions below will show you how to do this in code!

**Instructions**

Complete the control structure by calling the functions below. If the mode is 'public', call:

showDetails();

Else, call:

hideDetails();

Run the code and test the website.

If done correctly, the content of the post should be visible when you choose “Make Profile Public” in the dropdown, and the content of the post should be hidden when you choose “Make Profile Private”.

**LISTS**

**Introduction to Lists**

When we start writing more complex programs, we’ll start working with more pieces of data. But data can get messy real fast if we’re not careful.

To keep our data tidy, we’ll want to use *data structures*. Data structures are formats that we can use to keep track of our data in an organized fashion.

*Lists* are one very basic data structure. Programmers use lists as a container to store multiple pieces of information that relate to each other in some way. Like a list of the presidents of the US, types of cheeses in alphabetical order, and the finishing positions of runners in a race.

What makes lists special is that they order our data in a specific, linear sequence.

Since our values are kept in order, it allows us to easily find the information we’re looking for; otherwise, we’d have a huge jumbled mess of data!

In this lesson, we’re going to cover:

* Accessing an item from a list
* Adding an item to a list
* Removing an item from a list

**Instructions**

Comic strips function in a similar way to lists. We can think of the strip as the list and each frame as a separate item in the list. The narrative that makes up a comic demonstrates that the items follow a specific order.

We create lists by adding items to an empty list. Right now we have a bunch of comic frames, but they’re not in any order and our comic strip is empty. Complete the comic strip by placing the frames in the correct order. Here’s how the story should go:

* Codey plants a seed
* Codey waters the seed
* Codey waits
* A sprout grows

**LOOPS**

**Introduction to Loops**

How do we use code to tell a computer this: “Create a variable and call a function 15 times”?

We could write it out 15 times:

create a variable  
call a function  
create a variable  
call a function  
create a variable  
call a function....

…We’ll spare you the rest. This approach takes a long time and it can easily lead to mistakes. Instead, let’s give the instructions once and tell the computer how many times to repeat them:

Repeat this 15 times:  
  create a variable  
  call a function

This structure is called a *loop*, and you’ll learn all about them in this lesson. These special structures in programming will give you the ability to repeat instructions multiple times without writing the instructions out multiple times.

This lesson will cover:

* for loops
* while loops
* for each loops

By the way, we assume that you’re familiar with creating variables, using operators, calling functions, and working with lists.

**Instructions**

Create a pattern WITHOUT loops: In the text box, repeat these instructions 10 times, then run it:

1. Place a pink tile
2. Place a orange tile
3. Place a mint tile

If you did it correctly, you’ll see all 30 tiles placed on the board!

**What are Algorithms?**

Algorithms have three main characteristics:

* They are **sequential**.
  + Algorithms are performed step by step from start to finish. Kenny’s algorithm had to first build one pair of students, then another pair, then another pair, until all the students were paired up.
* They are **conditional**.
  + Algorithms perform certain steps based on conditions of the system. Kenny’s algorithm needs to check the grade of each student as it builds the pairs to make sure that the highest scoring student is paired with the lowest scoring student and so forth.
* They are **generalizable**.
  + Algorithms are applicable to many different problems that are of a similar type. For example, Kenny used his sorting algorithm to pair students in each of his different classes and then, later in the day, he also used it to build a tournament bracket for the school’s chess club. These are two different applications, but the generalizable nature of the sorting algorithm allowed it to be used in both situations.

What would have been a pretty annoying task to do by hand became an easy job for the computer. This is the power of algorithms! Nearly every piece of technology in the world today uses some form of algorithms to sort, store, and access data efficiently and accurately.

Studying computer science gives you the tools you need to apply algorithmic thinking to real-world problems and quickly complete tasks and optimize processes that would have otherwise been a burden to tackle. Kenny doesn’t know it yet, but algorithms are going to help him a lot in all sorts of situations!

**Functions**

Kenny got tired of the classroom and wanted to see more of the outdoors, so he took a job as a farmhand. Kenny, being the capable young man he is, quickly moved up the ranks of the farmhand hierarchy and was tasked with managing a team of new farmhands to harvest the strawberry fields.

“How do I impart my knowledge of harvesting strawberries to these apprentice pickers?” thinks Kenny to himself. Well, whenever Kenny has a task that needs to be done many times over, like harvesting a strawberry, he thinks back to his computer science days and realizes he needs to write a *function*.

A **function** is a specific set of repeatable instructions that takes an input, like a strawberry plant, and produces an output, like freshly picked strawberries. If this sounds like the definition of an algorithm, you’re right! An algorithm is conceptual, and a function can put an algorithm into practice.

Kenny finds it helpful to consider functions as he writes instructions on how to harvest strawberries for all of the new pickers. “I need to have explicit, step by step instructions or my pickers will not consistently produce the correct output,” thinks Kenny. By thinking of these instructions as a function he is giving to the harvesters, Kenny ensures that harvesting strawberries is a repeatable, dependable task.

**OVERVIEW OF THE INTERNET**

**Review**

Congratulations! You should now have a general understanding of how the internet works, including:

* The growth of the internet as a network
* The difference between the internet and the world wide web
* The relationship between browsers and servers
* HTTP status codes, like 404 Not Found
* Big trends in web development, from static websites to Web 2.0 and the rise of mobile

If you work with engineers, this information will help you talk about websites and web applications at a more technical level. Or if you’re interested in becoming a web developer yourself, you now have the important context to start building your own website or web application!

**Review**

Alejandra now has a fully functional web application for her small business!

In building out the features that she needed for her business’s application, she learned about the four languages that form the core of the World Wide Web today:

* **HTML** — structures website content
* **CSS** — applies styling to websites
* **JavaScript** — adds interactivity to websites
* **SQL** — allows your web application to store and retrieve data

While these languages are each essential to web development, many of them also have applications in other fields. For example, JavaScript was initially just a language for interacting with HTML, but JavaScript has expanded to be a general-purpose programming language that can be run outside of the browser. You can now build web applications, browser games, desktop applications, and even VR/AR experiences in JavaScript.

Learning web development has given Alejandra the skills that she needed for her business and also opened the door for her to take on more and more technical projects in a wide variety of fields.

**Review**

Review! You now know the basic parts that make up the field of data science.

* **Data Science**—the field of taking data and transforming it into meaningful information that can help us make decisions
* **Descriptive Statistics**—statistics that describe the data in objective terms
* **Inferential Statistics**—inferences for the overall population based on data
* **Probability**—the likelihood that an event will happen
* **Programming**—the act of giving the computer instructions to perform a task
* **Domain Expertise**—the particular set of knowledge that someone cultivates and brings with them in order to understand their data

# **Determine the Necessary Data**

In [94]:

*# Import modules:***import** **pandas** **as** **pdimport** **numpy** **as** **np***# Import the data:*user\_data = pd.read\_csv("user\_data.csv")*# Create age variable and find population mean:*population\_mean = np.mean(user\_data["age"])*# Select increasingly larger samples:*extra\_small\_sample = user\_data["age"][:10]small\_sample = user\_data["age"][:50]medium\_sample = user\_data["age"][:100]large\_sample = user\_data["age"][:200]*# Calculate the mean of those samples:*extra\_small\_sample\_mean = np.mean(extra\_small\_sample)small\_sample\_mean = np.mean(small\_sample)medium\_sample\_mean = np.mean(medium\_sample)large\_sample\_mean = np.mean(large\_sample)*# Print them all out!*print ("Extra Small Sample Mean: " + str(extra\_small\_sample\_mean))print ("Small Sample Mean: " + str(small\_sample\_mean))print ("Medium Sample Mean: " + str(medium\_sample\_mean))print ("Large Sample Mean: " + str(large\_sample\_mean))print ("**\n**Population Mean: "+ str(population\_mean))

**Recap**

As we can see, the process of data science involves several steps. While we’ve presented it in a linear manner, it’s important to remember that each step can happen multiple times and steps may be repeated.

To recap:

* Come up with a question you want to answer
* Form a hypothesis and determine the data you need
* Collect the data
* Clean and organize your data
* Explore your data
* Build models to analyze the data
* Communicate your findings
* Ensure that your process and findings are reproducible

Now that you know what it takes to do data science, let’s look more closely at the applications of data science and the types of projects you can make.

# **Data Science Applications**

**Read about the different projects you can do with data science!**

### **Introduction**

Now that we know a bit about how to do data science, what exactly can we do with it?

In this article, we’ll look at how we can apply data science thinking to different problems. We’ll explore several common applications, including:

* Reports
* Recommender Systems
* Dynamic Pricing
* Natural Language Processing

Similar to how data science is made up of different disciplines, the applications of data science are far ranging. But in all of these situations, you will apply data science to find patterns, draw meaningful conclusions, and make decisions.

Don’t worry if these terms sound intimidating! Throughout the article, we’ll break down each of them so that you have a solid understanding of what you can build with data science.

#### **Reports**

Reports are the most fundamental application of data science. A report is a document in which you present your process and your findings. Reports are important because they enable those who work with data to translate numbers and calculations into accessible insights and recommendations for team members.

A report could take the form of a publication that circulated within your company, or an article that’s published on the internet, or as part of a conference presentation.

A good report should have the following characteristics:

* Simple: non-technical team members will need to be able to quickly grasp findings
* Clear: language should be to the point
* Engaging Presentation: charts and presentations should be well designed

Five-Thirty-Eight is a popular statistics website that publishes reports on a range of topics, such as politics and sports. Take a look at [this piece that they did on the Bechdel Test](https://projects.fivethirtyeight.com/next-bechdel/), along with a range of other new tests that examine the presence of gender and race in film.

#### **Recommendation Engines**

One of the more well-known applications of data science is using data and machine learning to build a *recommender system*, also known as a recommendation engine.

A recommender system is a type of content filtration system that seeks to predict what a user would be interested in consuming. These suggestions could come from the user’s preferences that they’ve shared with the platform, like how Amazon suggests things you might want to buy based off of previous purchases. Other recommender systems work by looking at the preferences of people in your network, or those of people with similar demographics.

Recommender systems work for all types of information, from Spotify using it to recommend new artists to Netflix predicting what will be your next binge fest.

#### **Dynamic Pricing**

Another data science application is dynamic pricing. Ever go to buy an airplane ticket but then 5 minutes later the pricing changes? You have dynamic pricing to thank for that.

Dynamic pricing, also known as surge pricing, is the practice of setting prices for products or services based on market demand. Companies that use dynamic pricing build algorithms that take into account competition, supply and demand, as well as other factors related to the specifics of the industry. Dynamic pricing exists across several industries, including transportation, entertainment, amusement parks, and professional sports.

The most common example is airline tickets. Airlines started to use computers to determine flight prices as early as the 1950s, taking into account the season, day of the week, and time of day when setting ticket prices. However, airlines have recently come under scrutiny for utilizing more robust dynamic pricing techniques. Several are now determining fares based on the buyer.

While dynamic pricing is increasingly common, its a tactic that is often seen to benefit the company more than the consumer. At one point, the app MoviePass implemented surge pricing as part of their ticket buying experience. Previously, customers paid a flat monthly rate to see an unlimited number of movies. After this change, tickets for certain movies or popular theaters incurred additional fees.

#### **Natural Language Processing**

While data science typically makes you think of numbers, it can also be helpful in recognizing trends and patterns in language.

Natural Language Processing (often referred to as NLP) is the application of programming and artificial intelligence to process and analyze text. It can be used for research purposes, to understand, for example, the grammatical structure of a text, to more creative pursuits such as generative poetry.

In NLP, your datasets are made up of examples of language usage, known as a corpus. After training on this dataset, a machine can then perform different functions, such as part-of-speech tagging, language translation, and sentiment analysis.

A common application of NLP is chatbots. While many chatbots follow pre-determined scripts, more advanced ones can use NLP to enable a dynamic discourse. NLP enables a bot to continue learning as it talks, making it better at handling different and unexpected situations.

### **Summary**

As you can see, there are many different applications and kinds of projects that you can do once you know a bit of data science!

* Reports - a way of presenting your process, insights, and recommendations
* Recommender Systems - a process that uses data about users and items to predict interest
* Dynamic Pricing - a strategy that takes into account factors such as demand to increase and decrease prices to drive profit
* Natural Language Processing - ways of analyzing text to gain insights as well as support applications, such as chatbots

Of course, there are many applications beyond just the ones that we’ve covered here. So no matter your interest or professional industry, data science thinking can create impact.

**Review**

Take a minute to review what you’ve learned about bash scripting.

* Any command that can be run in the terminal can be run in a bash script.
* Variables are assigned using an equals sign with no space (greeting="hello").
* Variables are accessed using a dollar sign (echo $greeting).
* Conditionals use if, then, else, fi syntax.
* Three types of loops can be used: for, while, and until.
* Bash scripts use a unique set of comparison operators:
  + Equal: -eq
  + Not equal: -ne
  + Less than or equal: -le
  + Less than: -lt
  + Greater than or equal: -ge
  + Greater than: -gt
  + Is null: -z
* Input arguments can be passed to a bash script after the script name, separated by spaces (myScript.sh “hello” “how are you”).
* Input can be requested from the script user with the read keyword.
* Aliases can be created in the .bashrc or .bash\_profile using the alias keyword.

**Generalizations**

You have now been introduced to the fundamental Git workflow. You learned a lot! Let’s take a moment to generalize:

* Git is the industry-standard version control system for web developers
* Use Git commands to help keep track of changes made to a project:
  + git init creates a new Git repository
  + git status inspects the contents of the working directory and staging area
  + git add adds files from the working directory to the staging area
  + git diff shows the difference between the working directory and the staging area
  + git commit permanently stores file changes from the staging area in the repository
  + git log shows a list of all previous commits

### Instructions

Click Next to complete the lesson!

### Concept Review

Want to quickly review some of the concepts you’ve been learning? Take a look at this material's [cheatsheet](https://www.codecademy.com/learn/learn-git/modules/learn-git-git-workflow-u/cheatsheet" \t "_blank)!

**Generalizations**

Congratulations! You’ve learned three different ways to backtrack in Git. You can use these skills to undo changes made to your Git project.

Let’s take a moment to review the new commands:

* git checkout HEAD filename: Discards changes in the working directory.
* git reset HEAD filename: Unstages file changes in the staging area.
* git reset commit\_SHA: Resets to a previous commit in your commit history.

Additionally, you learned a way to add multiple files to the staging area with a single command:

git add filename\_1 filename\_2

**Generalizations**

Congratulations! You’ve learned three different ways to backtrack in Git. You can use these skills to undo changes made to your Git project.

Let’s take a moment to review the new commands:

* git checkout HEAD filename: Discards changes in the working directory.
* git reset HEAD filename: Unstages file changes in the staging area.
* git reset commit\_SHA: Resets to a previous commit in your commit history.

Additionally, you learned a way to add multiple files to the staging area with a single command:

git add filename\_1 filename\_2

**Generalizations**

You have now been introduced to the fundamental Git workflow. You learned a lot! Let’s take a moment to generalize:

* Git is the industry-standard version control system for web developers
* Use Git commands to help keep track of changes made to a project:
  + git init creates a new Git repository
  + git status inspects the contents of the working directory and staging area
  + git add adds files from the working directory to the staging area
  + git diff shows the difference between the working directory and the staging area
  + git commit permanently stores file changes from the staging area in the repository
  + git log shows a list of all previous commits

**Generalizations**

Congratulations! You’ve learned three different ways to backtrack in Git. You can use these skills to undo changes made to your Git project.

Let’s take a moment to review the new commands:

* git checkout HEAD filename: Discards changes in the working directory.
* git reset HEAD filename: Unstages file changes in the staging area.
* git reset commit\_SHA: Resets to a previous commit in your commit history.

Additionally, you learned a way to add multiple files to the staging area with a single command:

**GIT BRANCHING**

**generalizations**

Let’s take a moment to review the main concepts and commands from the lesson before moving on.

* Git *branching* allows users to experiment with different versions of a project by checking out separate *branches* to work on.

The following commands are useful in the Git branch workflow.

* git branch: Lists all a Git project’s branches.
* git branch branch\_name: Creates a new branch.
* git checkout branch\_name: Used to switch from one branch to another.
* git merge branch\_name: Used to join file changes from one branch to another.
* git branch -d branch\_name: Deletes the branch specified

**GIT TEAMWORK**

**Overview**

So far, we’ve learned how to work on Git as a single user. Git also offers a suite of collaboration tools to make working with others on a project easier.

Imagine that you’re a science teacher, developing some quizzes with Sally, another teacher in the school. You are using Git to manage the project.

In order to collaborate, you and Sally need:

* A complete replica of the project on your own computers
* A way to keep track of and review each other’s work
* Access to a definitive project version

You can accomplish all of this by using *remotes*. A remote is a shared Git repository that allows multiple collaborators to work on the same Git project from different locations. Collaborators work on the project independently, and merge changes together when they are ready to do so.