What is data?

Data is a hot topic right now! You may have heard some buzzwords like Big Data or data science in recent years. Companies make millions of dollars selling, analyzing, and using data. Facebook and Apple are in a huge multi-million dollar fight over data right now! Data is *important*. But how did it get to be such an integral part of technology and our daily lives?

We *will* be covering how to read, analyze, and represent data. But before we get into that, we need to have a solid understanding of what data is, what makes good data good, and bad data bad. Because you can be the best data scientist in the world, but if the data is flawed, there goes everything else.
Even the finest chef with the finest recipe can't make a meal out of spoiled ingredients.

So what is data?

A unit of data is a unit of information! Simple enough!

While datum is actually the singular of data, more often than not we hear the word data in reference to more than one unit of information. That's probably because one unit of data is not very useful. But a lot of units of data… we can do something with that!

Data can be collected, stored, and represented in a number of different ways. Some scientists collect data through lab experiments, others collect data by observing subjects, others collect data by asking questions of their subjects. And within those categories theres even more methods and techniques to collect data!

Data can also be stored in a number of ways. The most simple way of storing data has been around since the birth of humanity - our memory. Our brains are hardwired to observe, categorize, analyze, and store pieces of data to make better decisions in our lives.

Think about when a baby is learning new words! The baby might see a dog for the first time and his mother will say "Dog". The baby will store that information, connecting the word "Dog" with the furry creature he saw. The next day , the baby might see another similar furry animal, compare it to the data point he has, and conclude this animal is also a "Dog!". Upon his exclamation, his mother might say "No that's a cat!". Now the baby has another data point to compare new observations to. With time and more data points, that baby will slowly learn to identify dogs, cats, and many other different animals.

That's how data works! We collect points of observations to be able to come up with accurate conclusions!

But this method of data collection has its flaws. It's great to learn how to tell the difference between dogs and cats, but imagine trying to take a census by memory. Our brain is an incredible organ, but a computer (or even a pen and paper) will beat out human memory every single time.

So we invented different ways to relieve our memory and externally store and organize our observations, and even make conclusions! Tables, graphs, visuals, and lists are common ways to store data - and those same concepts were eventually transferred over from paper to the softwares and file types we use in computer science to work with data.

What are the requirements we ask of our data?

1. **A data sample should be representative of the larger group or population it is being used to analyze**

   Imagine you have a big bucket full of blue and red marbles. In this bucket there are 10,000 marbles and someone tells you 60% of the marbles are blue and 40% are red. You need some way to verify this to a certain level of accuracy - but counting 10,000 marbles will take forever!

   Like in this marble example, often times the population that we're trying to analyze is much too big to actually gather individual data points. The next best thing is analyzing a representative sample - or a sample that accurately reflects the data of the total population.

   The easiest way to get a representative sample is a simple random sample. Randomly select some subset of the population and you will get a relatively accurate sample!

   With a simple random sample, bigger is better! The bigger the sample, or the more samples you can gather, the more accurate your data will be to the general population.

   Let's go back to the marble example. What if you pulled out 100 marbles and counted them? 100 is much easier to deal with than 10,000. And in theory, you should pull out 60 blue marbles and 40 red. But if in your sample there are 80 red marbles and 20 blue, it's unlikely that the whole bucket has a 60-40 ratio. And if you pull out ten samples of 100 marbles, there would be deviations from sample to sample, but on average it should get closer and closer to the actual ratio.

2. A data sample should have some source of comparison

You need a control! How can you accurately analyze the data and come up with a conclusion if you have nothing to compare it to? The way we describe our world and data revolves around comparison - big, small, long, short, good, bad, etc. To an ant, a penny is huge. To us, a penny is pretty small. It's all about what you compare it to.

So to make accurate analyses, not only do we need to have a good sample of our data, we need to have a good control group.

A good control group is made up of a sample that is fairly similar to the data sample, but varies on the specific variable(s) that is/are being analyzed or observed.

Let's say you are doing a study on how distance learning affected elementary students' reading level after finishing 1st grade. You got a great representative sample of students who were in 1st grade during the pandemic and tested their reading levels. But now what? Are their reading levels high or low? If you *just* have these test scores you have no way of knowing if distance learning affected their reading level or if they just have a normal reading level for their age.

But if you find a control group to compare it to, then you can make more accurate conclusions. Say now, you find the data for the reading levels of the 1st grade students who were in first grade in 2018-2019. Now you have a control sample that is very similar to the original data sample (same age, same teachers, same schools, etc) but differs on the one variable we're interested in! If the control reading levels are much higher than the samples levels, then we can support our conclusion that distance learning *did* affect the reading level of elementary students.

The Good, the Bad, and the Ugly Data

You might have heard the phrase "lying with statistics". But a lot of times, the statistical analysis is actually fine. They used the correct regression, or found the right p value and rejected the right h0, or whatever. But they were running good analysis on bad data. Because, again, a good chef can't do anything with bad ingredients.

What makes bad data bad?

There's a variety of reasons that a bad data sample is bad. Some have to do with the collection methods of the sample, some with the individuals being sampled, some with the researchers doing the sample, some with the people reading the results, but all of them come down to one thing. Is the data biased towards one conclusion or another?

Types of Bias in Data

Selection Bias:

Finding a good random sample can often be more difficult than it seems. Any bias that results from the selection (or self-selection) process for the sample is known as selection bias. If each member of the relevant population does not have an equal chance of ending up in the sample, the results of that sample will be biased.

Let's say you are taking a poll about the average salary of those employed in Los Angeles. You randomly select 100 random businesses and companies located in Los Angeles and ask them to randomly select and poll 25% of their employees.

At first, this seems like a perfectly good random sample. You randomly picked 100 businesses and found a random sample of employees within that. It seems like anyone employed in LA has the same chance to end up in this sample right?

Except that LA is full of freelancers. About 18% of LA households report self-employment income. Which means this sample automatically excludes almost 1/5 of the target population. That's selection bias!

Selection bias also happens when participating in the collection of data is voluntary. If you ask 100 people in a public place to answer a short survey, and 60 of them say yes, those 60 people are different than the 40 for the mere reason that they said yes. Maybe they're just nicer or have more time to stop, but statistically they are now different than the total population.

Publication Bias:

Just because data is published, peer reviewed, and in a fancy scholarly journal does not mean it's not biased! Positive findings are more likely to be published than negative findings, which skews the results that we can find and see. Negative findings are just not that interesting to a publisher - but that doesn't mean they're not important!

Probability will tell us that more often than not, if the sample is a good random sample, it will reflect the total population. We just went over that with those marbles! And now we're going to say that's not always true?

Well that's the key isn't it! It's true *most* of the time, but sometimes we have a perfectly collected random sample that against all odds, produces bad results! And sometimes, those results lead to a conclusion. And if that conclusion means something, it will get published! Even though all the other studies before it came up with a different conclusion.

This type of bias is hard to catch because the mistake won't be found in any individual study. And, since they're not often published, negative results are hard to find. But in general, studies should be repeated to be credible. If you see a study that comes up with a crazy new conclusion and it seems to check out - make sure another one has been able to prove the same results with a different data set. Otherwise, you might just be seeing the four leaf clover that got published in a field of a bunch of boring regular three leaf clovers.

Recall Bias:

Human memory is not the most accurate thing ever. And the longer it's been since the event (and the more emotional the event is) the worse our memory is - at least when it comes to the details. The worst part about this is that our brain will actually fill in the blanks to parts of our memory that was lost - so we'll even remember details that aren't even true.

And so if your data relies on asking subjects to remember certain details or events - it will probably have a healthy dose of recall bias. Subjects will misremember details, fill in details that are inaccurate, or simply not be able to remember.

The best way to get around this for these types of studies is a longitudinal study. Let's say you are studying the effects of the use of phones on arthritis. Instead of asking people with arthritis how long they spent on their phones in the last 20 years, it is better to pick a random sample of people, and follow their phone usage and arthritis symptoms over 20 years. It is much easier to report accurate details on the present than on the past.

Survivorship Bias:

Survivorship bias happens when participants leave the study for any reason. The participants that remain will be different than those who didn't, simply because they did stay.

Imagine you are conducting a study where you have participants run a mile every day and then you track their resting heard rate over a period of 6 months. Three months in, 20% of the participants quit because the weather started to get cold and rainy and it was just too hard. The participants that make it to the end are much more likely to have a lower resting heart rate after the experiment. Is that because they ran a mile every day or because they are the type of people to put their health first?

Seems like collecting good data is HARD! What makes good data good?

The reality is that it *is* pretty hard to collect good data. You can't force people to participate in a study and you can't control for everything.

So bad data is a fact of life. Not much we can do about it. But what we can do is take into account the bias that your data might have and try to analyze it with that bias in mind. Make sure to control the important variables and try to randomly select your sample as much as possible. If your data points to an interesting new conclusion, run the study again with a new sample, and see if it still holds up!

And, when you read studies and look at data, think about what bias could have been introduced in that study. Think about how that bias could affect the conclusion. Think about if that bias is significant enough and if that data should be trusted. Look for other sources and data that points to the same things!

Because there's all sorts of data out there! Some of it is good, some of it is bad, and a lot of it is ugly. That's up next - how do we represent data in a way that makes it pretty and easier for people to understand it?

Data Visualization

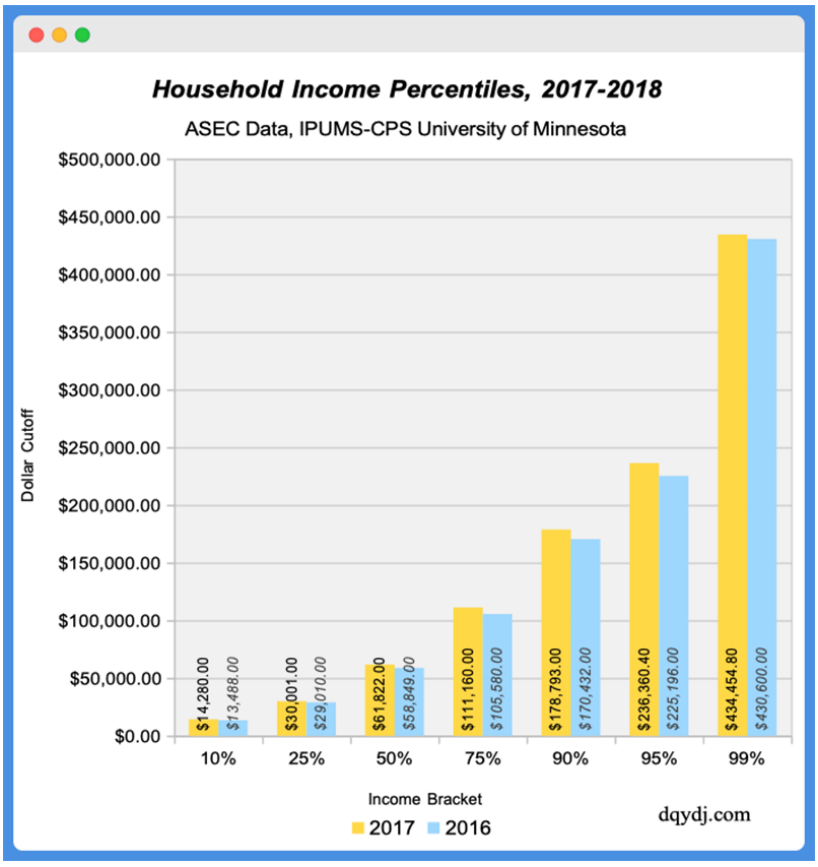
Okay, you have good data. Your methods are good, you've taken your biases into account, your statistical analysis is done correctly - and you have a supported hypothesis! Now, you want to tell everyone about this interesting new finding!

But no one wants to look through excel sheets of data and statistics formulas. So we've come up with a number of ways to represent the data and findings that we've found! You've probably heard of or seen most methods of representing data - charts, tables, figures, graphs, etc.

The biggest benefit of data visualization is being able to communicate your data to a larger audience in a much more digestible manner.

The way we represent data is really important - because that is how we communicate what that data means. And the way you represent it can change the meaning - or at least the conclusions people draw from it.

Take a look at the bar chart below:



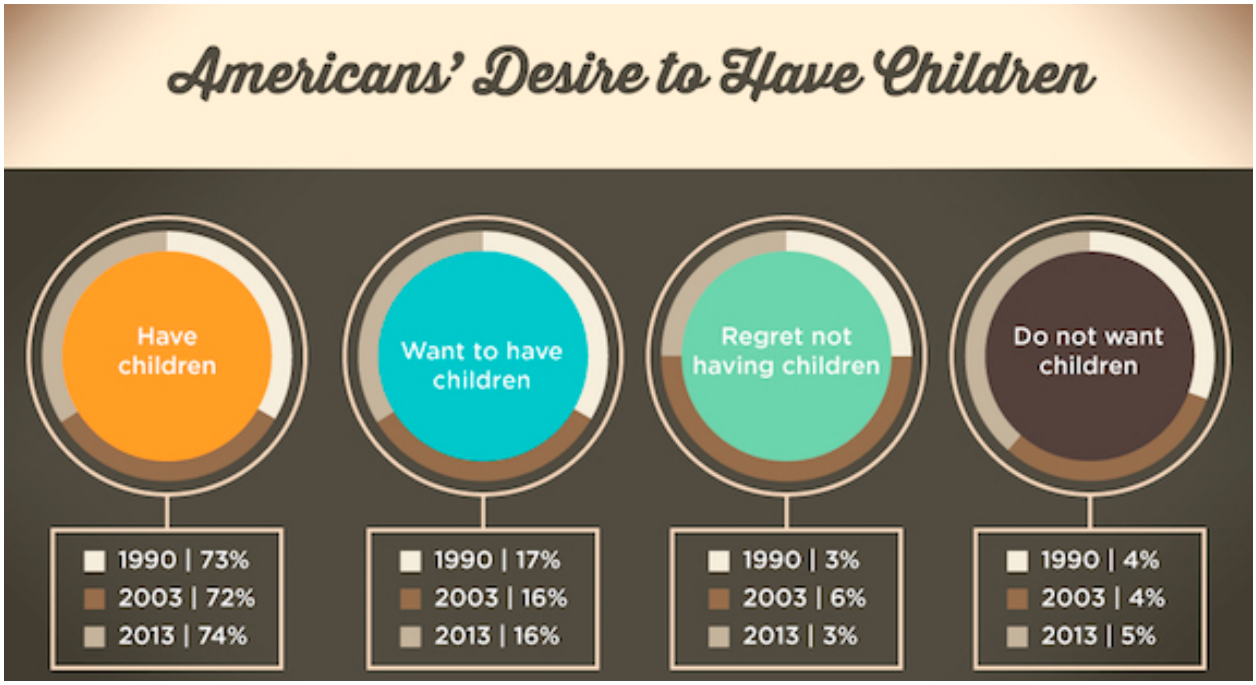
Can you see anything wrong with it?

Question: What year were the income percentile cutoffs higher? 2016 or 2017?

Five of the Most Common Data Visualization Mistakes

1. Wrong Type of Chart

For example, a line graph will give the impression that something is increasing or decreasing, so this isn't the best type of visualization to use if you're not charting over time and/or dates. In addition, you might create a pie chart based on assignee to see how much work each assignee has. However, what you really want to know is how many tickets they have in a specific status, or for a specific version, in which case a stacked bar chart or grouped bar chart might be better.

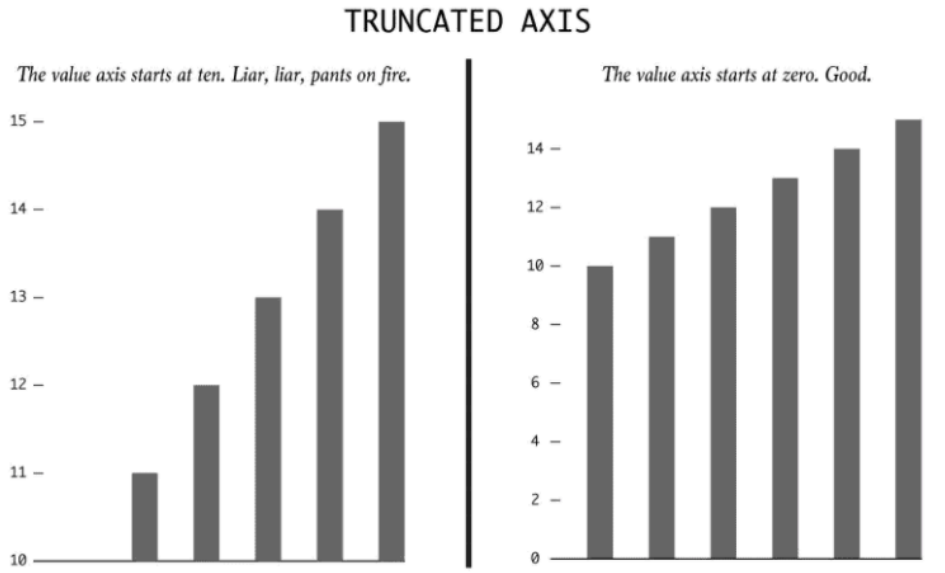


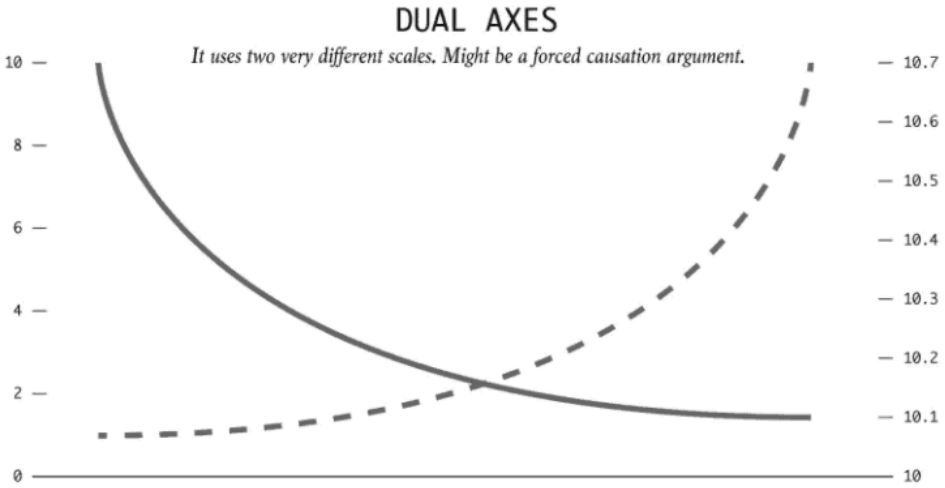
2. Too Many Variables

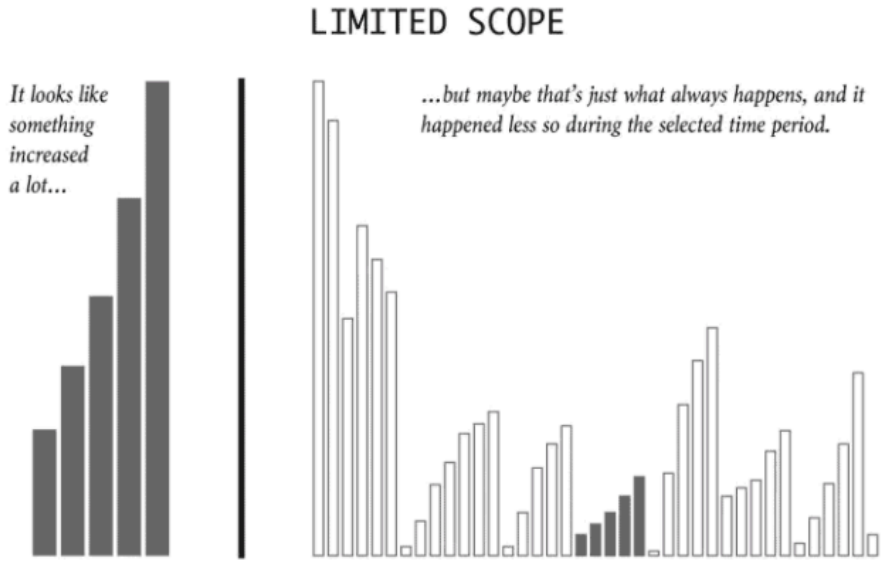
Data visualization is about telling a story. And no one likes when someone tells a story with too many or irrelevant details. It's confusing, hard to follow, and boring. So, identify only what you need to convey and exclude everything else. Then pick the format that's going to convey the data in the clearest manner possible.

3. Improper Scaling

You (and your user) should understand what scale you are setting your data to, be consistent, and label it correctly. This includes things like: having axes with irregular intervals, starting an axes at a number other than 0, confusing dual axes, limiting the scope, comparative charts displaying data on different scales, etc.

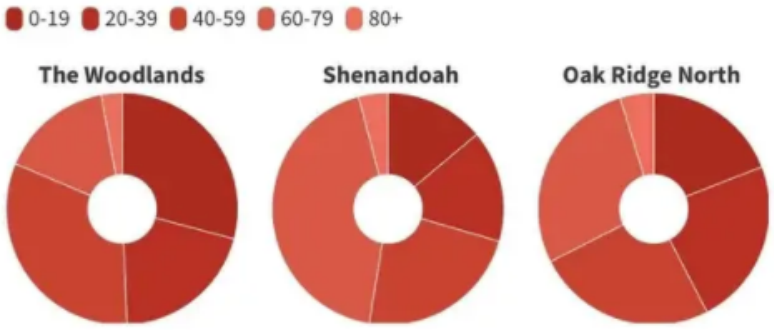




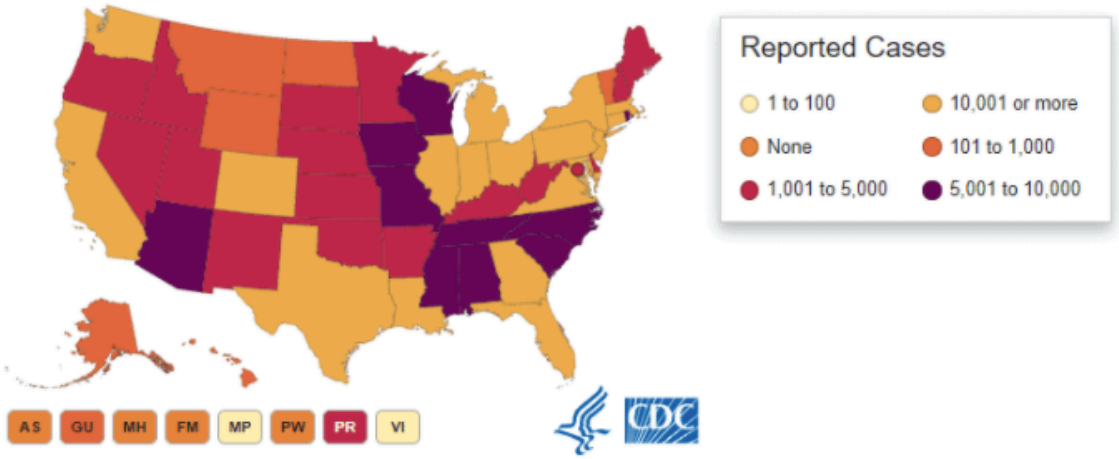


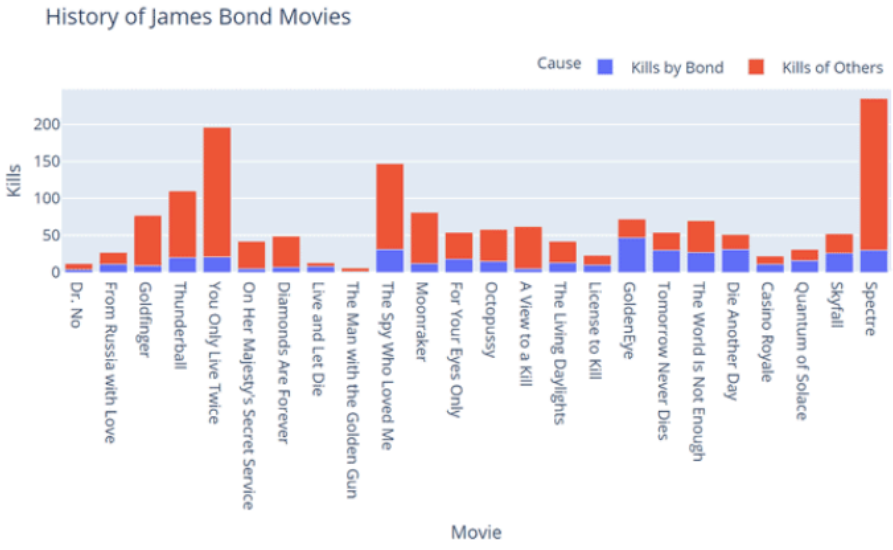
4. Design Choices: Bad & Good User Experience Design (UX)

Good UX design principles apply here too! Don't use colors that are too similar to each other to compare different things, don't use colors that have implied meanings in opposing ways (e.g. making a temperature graph where red is cooler and blue is hotter), and don't make the graph hard or uncomfortable to read!



Source: U.S. Census Bureau

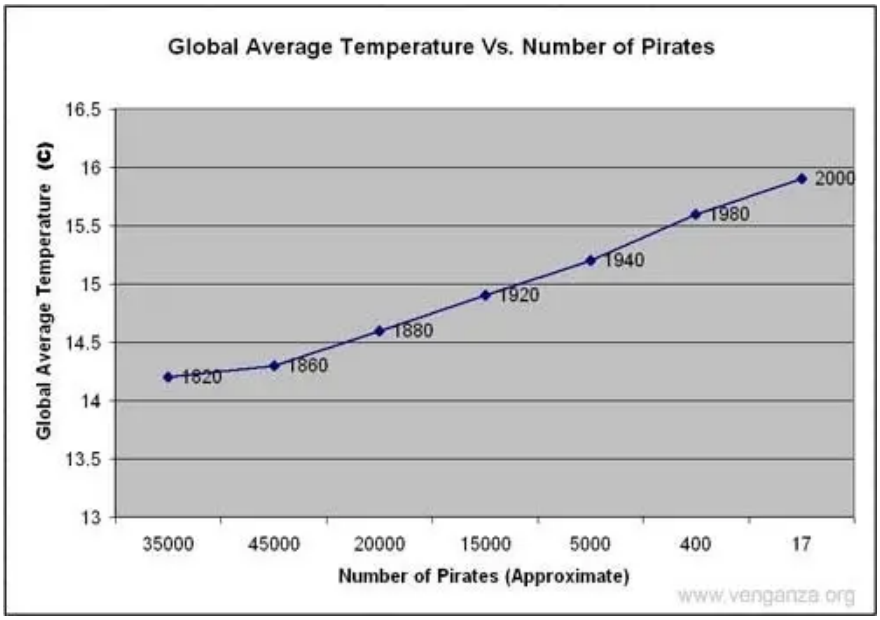




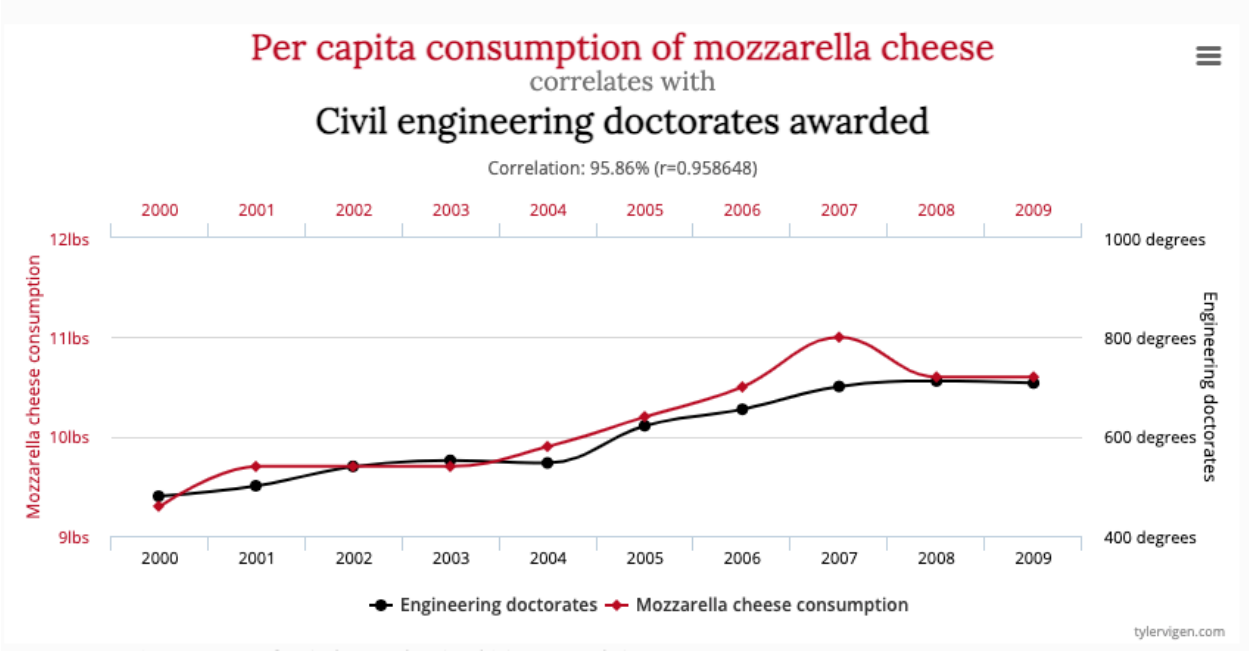
5. Implying Correlation = Causation

Correlation DOES NOT imply causation! Now say it three times fast!

Here are some examples of where implying a connection between two things is a very slippery slope!



Source: Buzzfeed.com



Characteristics of Good Data Visualizations

1. Uses Good Data

Good data is the backbone of good research! It doesn't matter how amazing and clear and beautiful your chart is - if the data is bad, the audience will come to wrong conclusions.

When searching for good data, make sure your study is peer-reviewed, look over the methods and make sure there's nothing fishy, and make sure to note any biases that may have occurred! It is also best if you can find multiple other studies with multiple datasets that also support the same conclusion!

2. Provides Context

Labels, labels, labels! The audience should know what scale the data is set to - which means labeling axes and units of measurements, providing a key for more information, adding titles, making sure your scales are not misleading, etc. Think about the information someone needs to accurately understand this data and make sure to include it somehow!

If designing something like an infographic, it is also helpful to add written explanations on the background of the data, the study/studies where it is gathered, the methods, the limitations, the conclusions, etc.

3. Simple and Easy to Read Charts

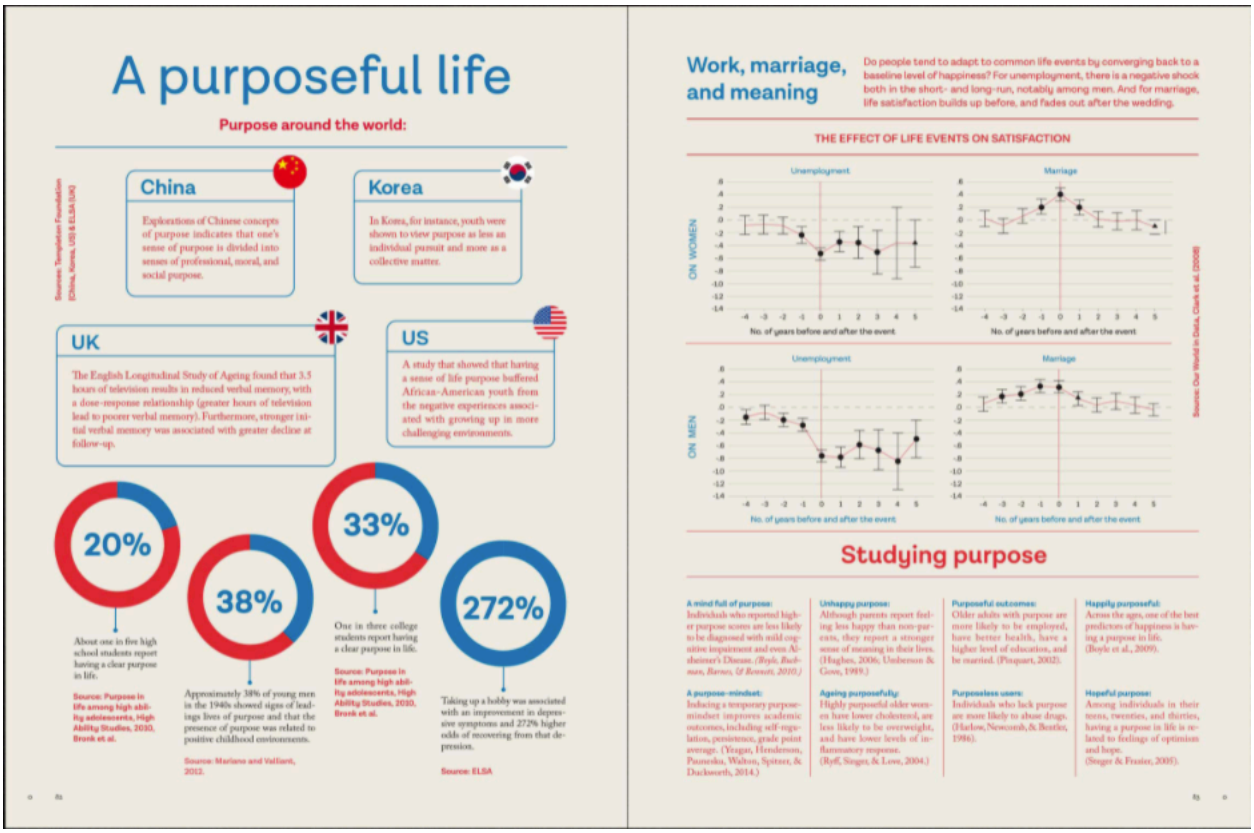
Keep in mind some of those design mistakes that we just covered! Don't make people read things sideways, or squint to be able to read tiny text, or use colors that are too similar, etc. Make sure it's easy to read!

4. Interesting and Aesthetically Pleasing

If it's interesting and aesthetically pleasing, more people will look at it. If it's boring and ugly, less people will.

Don't forget to use colors, consistent fonts, pictures and drawings, and adequate white space! Like we said, good design principles still hold true!

A Good Example of Data Visualization in an Infographic



Here's the link!
A Purposeful Life, by Studio Pattern

Javascript Objects

Whats an object again?

First, let's think about what an object is in general?

What is a ball? What is a dog? What is a book? What is a student? What is a TV remote? How do we define objects?

Kinda an abstract concept! But if you really break it down, objects are just ways to classify items that have similar states and behaviors. Remember those words?

STATES

properties that define the object
what it looks like, features it
has, current state, etc.

BEHAVIOR

functionality of the object
what it can do and what can be
done to it

OBJECT	STATES	BEHAVIORS
DOG	breed, size, age, weight, hunger level, etc.	bark, run, eat, smell, jump, be pet, etc.
BOOK	author, title, length, publisher, audience, cover, etc.	open, close, be read, be bought, sit on bookshelf, etc.
STUDENT	name, school, grade, ID, schedule, age, favorite class, etc	go to class, change classes, move to a new grade, do homework, etc
TV REMOTE try it on your own!		

Implementing JavaScript Objects

A JavaScript object is just a way to represent a type of variable that has a certain set of states and behaviors. The cool part is we get to program in these states and behaviors to be whatever! You can create a flying dog if you want!

There are two methods to implement objects in Javascript.

Object Constructors & Instances

To create an object using a constructor, you first need to define the class. This is the template or "cookie cutter" for your objects. This defines which properties and methods the object will hold, but, unlike object literals, it does NOT define the actual values.

```
class Name {

      constructor(constructorParameters) {
            this.propertyName = valueParameter;
            this.propertyName = valueParameter;
      }

      methodName = function (methodParameters) {
            function body;
      }
};
```

But now all we've done is build the cookie cutter. How do we actually cut out some cookies?

To create new objects using the class template, you need to define an object instance. Use the new keyword to construct a new object with specific states and set it to a variable to be able to use it later!

```
var objectInstanceName = new Name(constructorParameters);
```

Unless you're one of the rare programmers that totally understands objects, you might be a little confused right now. Don't worry - objects can be a little tricky so let's do an example!

```
class Dog {

      constructor(breedInput, nameInput, ageInput) {
            this.breed = breedInput;
            this.name = nameInput;
            this.age = ageInput;
      }

      calcHumanAge = function () {
            return this.age * 7;
      }
};

var myDog = new Dog("Border Collie", "Luna", 5);
```

myDog is now a Dog object! So I can use myDog to access states or methods!

```
console.log("My dogs name is: " + myDog.name);
console.log("She is " + myDog.age + " years old which is " +
myDog.calcHumanAge() + "in human years!")
```

This will print:

My dog's name is Luna.
She is 5 years old which is 35 in human years!

Now I can make another object instance with the exact same class!

```
var myPuppy = new Dog("German Shepard", "Daisy", 1);

console.log("My puppy's name is: " + myPuppy.name);
console.log("She is " + myPuppy.age + " years old which is " +
myPuppy.calcHumanAge() + "in human years!")
```

This will print:

My puppy's name is Daisy.
She is 1 years old which is 7 in human years!

But what if you want to create a type of object where each object has the slightly different states and behaviors? Instead of making a new class for every variation, you can quickly make individual objects with individually defined states and behaviors.

Think of creating objects like cutting cookies. You can either cut each cookie shape manually with a knife or use a cookie cutter. The benefits of using a knife is that you can make a bunch of different shapes, even though it will take longer. But if you're only looking for one shape of cookie, a cookie cutter will save you a lot of time and effort.

Creating objects using constructors is like using a cookie cutter! Next we will learn to cut individual cookies with object literals.

Object Literals

An object literal is a simple way to create a singular object. This method is really similar to creating a variable. The only difference is that an object can hold multiple properties (states) and can have methods (behaviors).

```
const objName = {

propertyName: value,
propertyName: value,
propertyName: value,

methodName: function () {
      function body;
      }
};
```

Then I can access these properties and methods using dot notation!

objName.propertyName;
objName.methodName(methodParams)

For example:

```
const myDog = {

breed:"Border Collie",
name:"Luna",
age: 5,

calcHumanAge: function(){
      return this.age * 7;
      }
};

console.log(myDog.age)
```

```
const myPuppy = {

adoption date:"08/01/2021",
name:"Daisy",
age: 1,

calcHumanAge: function(){
      return this.age * 7;
      }
};

console.log(myPuppy.calcHumanAge())
```

Notice the difference between object literals and constructors! When using literals, we were able to build these exact same two objects, but we had to define the states and behaviors for each. When using a constructor, we defined the states and behaviors once, then were able to build multiple objects using the same definitions!

Using Objects

JavaScript Objects are super useful when it comes to storing, accessing, and manipulating data. Soon, we'll talk about JSON data - which stands for JavaScript Object Notation. It's exactly what it sounds like - a bunch of objects that store a bunch of data.

Sounds simple enough? It's actually not too bad. But before we can effectively use JSON data, we need to understand how objects are normally presented and used.

Objects and Arrays

Objects can hold a lot of data! So if you add in a lot of objects with a lot of data… it can get tricky fast. One of the best ways to hold and keep track of a lot of objects is by keeping them in an array! That way you can loop through, access specific objects at a time, and access specific properties of specific objects!

Let's make a group of objects to hold the animals to be adopted!

```
const Maisy {
     animal: "cat",
     breed:"shorthair"
     age: 2
};
```

```
const Spot {
     animal: "dog",
     breed:"yorkie"
     age: 4
};
```

```
const Carrot {
     animal: "bunny",
     breed: "english"
     age: 1
};
```

```
let  animals = [Maisy, Spot, Carrot];
```

This seems easy, right? If we want to access Maisy, we can just index the animals array at animals[0]. Spot is at animals[1]. Carrot is at animals[2]. Let's say we want to find the average age of all the animals at my shelter. How could we do that?

```
var avgAge = (animals[0].age + animals[1].age + animals[2].age) / 3;
```

That would work!! Cool! But… what if our shelter has 1000 animals. Is there an easier way than indexing each one 1 by 1?? That would take forever! The phrase 1 by 1 might have triggered a light bulb moment - what about… a loop!

```
var sum = 0;

for (var i = 0; i < animals.length; i++) {
     sum = sum + animals[i].age;
}

var avgAge = sum/animals.length
```

Now, no matter how big the array of animals gets, this code block will find the average age!

Nested Objects

To hold more specific organized data, you can also place objects inside of objects! Just like you set an object literal to a variable, you can set an object literal to a parent's object property.

```
const student {

      name: "Joe Brown"
      id: "1234567"
      school: "Pasadena High School"
      schedule {
            period1: "Math"
            period2: "English"
            period3: "Biology"
            period4: "Physical Education"
            period5: "App Academy"
            period6: "History"
      }

}
```

So how do we access the inner object? Same as before! Just use dot notation!

```
student.schedule.period1 -> "Math"
student.schedule.period2 -> "English"
student.schedule.period3 -> "Biology"
student.schedule.period4 -> "Physical Education"
student.schedule.period5 -> "App Academy"
student.schedule.period6 -> "History"
```

In JavaScript, more often than not, you will see the object literal syntax, not the object constructor syntax. And you will often see data sets of large arrays of nested objects on nested objects. That's essentially what JSON files are! So it's important to be able to understand how to loop through arrays, access specific objects, and parse data accordingly.

Let's Do an Example

```
Let  augustExpenses = [
        {expense: "groceries", category: "necessities", cost: 300}
        {expense: "utilities", category: "necessities", cost: 150}
        {expense: "car repairs", category: "necessities", cost: 100}
        {expense: "dining out", category: "luxury", cost: 150 }
        {expense: "entertainment", category: "luxury", cost: 200 }
        {expense: "birthday gift", category: "gift", cost: 100}
]
```

Problem 1:

Calculate the total expenses in the three categories: necessities, luxury, gift.

Problem 2:

Find the highest expense and the lowest expense.

Solutions on Next Page!

Problem 1 Solution:

```
var nTotal = 0;
var lTotal = 0;
var gTotal = 0;

for (var i = 0; i < augustExpenses.length; i++) {
     if (augustExpenses[i].category == "necessities") {
          nTotal = nTotal + augustExpenses[i].cost;

     } else if (augustExpenses[i].category == "luxury") {
          lTotal = lTotal + augustExpenses[i].cost;

     } else if (augustExpenses[i].category == "gift") {
          gTotal = gTotal + augustExpenses[i].cost;
     }
}

console.log("The total for necessities is " + nTotal);
console.log("The total for luxuries is " + lTotal);
console.log("The total for gifts is " + gTotal);
```

Problem 2 Solution:

```
var lowest = augustExpenses[0].cost;

for (var i = 0; i < augustExpenses.length; i++) {
     if (augustExpenses[i].cost < lowest) {
          lowest = augustExpenses[i].cost;
     }
}

var highest = augustExpenses[0].cost;

for (var i = 0; i < augustExpenses.length; i++) {
     if (augustExpenses[i].cost > highest) {
          highest = augustExpenses[i].cost;
     }
}

console.log("The lowest expense is " + lowest);
console.log("The highest for luxuries is " + highest);
```

Application Programming Interfaces
aka an API

Maybe you've heard the term API before! Someone said I'm doing this project where I call the Google API… or you've come across it on a reference… or maybe you've used one before! APIs are a huge part of software development, so you're bound to come across one in your coding journey!

But what exactly IS an API? And what do we use it for?

An API is a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service.

So you totally get it now, right? The definition that google gave us is super helpful and not at all dry and boring? Okay I'm done here… go do your project :)

Just kidding, let's break down this definition!

An API is basically the set of rules a system defines so that other systems can talk to it and get the information they need! It's like a mini translator!

Think of using an API as going out to eat at a restaurant. When you go to a restaurant you don't just barge into the kitchen and make your food yourself. There are rules here! We're civilized!

First you sit down and the waiter brings you a menu. In general, at a restaurant you can't just order anything you please! They give you documentation (aka a menu) of what kinds of items you can receive and what to say so that you can communicate what you want to order!

Then you use that menu to communicate with the waiter! And that menu makes it really easy! Let's say that you go to the Mexican restaurant above. You don't need to tell the waiter, "Can you take a large tortilla and put carne asada, fries, shredded cheese, creamy salsa blanca, onion, and cilantro on it then wrap it up like you're swaddling a baby and plate it with a side of salsa verde." You can just say, "Can I have a California Burrito?" - and the waiter automatically knows that you want all that! It's almost like a very tasty code!

The waiter will then take your order to the kitchen and get the meal that you asked for!

The steps were as follows:

1) Read and understand the documentation to be able to order (aka the menu)
2) Use that documentation to communicate the meal you want to your waiter
3) The waiter takes that information to the kitchen
4) In the kitchen, the waiter works with the rest of the staff to put together the meal you asked for
5) The waiter brings it back out to you to enjoy!

Easy and efficient, right?

An API is essentially just a waiter! It gives a system the documentation to communicate with another system and acts as the point person, taking the order, grabbing the data the system asked for, and then bringing it back!

The internet is basically just a big smorgasbord of restaurants :)

Okay great… we get it! An API is a waiter! But… how do we use an API?

Using an API

There's a bunch of APIs out there that we can use that can do all sorts of things! Take a look at this list of fun free APIS!

Oh wait, here's another list!

And another one :-)

In other words… there's a lot of APIs out there! So how do we use them?

Using an API has three moving parts:

User: the person or system making the request
Client: the computer that sends the request to the server
Server: the computer that respond to the request

As a programmer, we will need to program our client to make the correct API call based on what the user/system needs! Basically we need to tell our waiter the order!

First, we need to read the menu! Each API has it's own documentation that you need to use to understand how to call your API and what to "order".

There are four types of requests you can make to an API:

GET: requests data from a server — can be status or specifics (like last_name)

POST: sends changes/data from the client to the server

PUT: revises or adds to existing information

DELETE: deletes existing information

But remember! Each API has a different menu! So the way you make these requests is going to look slightly different every time!

To use an API:

Step 1: Figure out what you are trying to do and what you want from your API! Are you trying to get data? Post data? Revise data? Delete data? What kind of data?
Step 2: Read the documentation to figure out how to ask that specific API for what you want!
Step 3: Make the request!

Seems easy? It is! Let's do some examples :)

Working with JSON data

aka
JavaScript Object Notation

Okay now we know how to load and access some data! But what is that data going to look like? How can we parse through it and actually use it for what I want?

Let's do a quick review on what we covered with objects and data!

Let's say we want to hold user's login information - like their name, email, username, password, etc.

Let's start with just one person! We could make an object!

```
user1 {

        firstName: "Joe",
        lastName: "Biden",
        email: "joe@gmail.com",
        user: "mrPresident",
        password: "12345",

}
```

Okay great! But what if I need to work with multiple users? What if you have 100 users? How could I group/access them all together?

What about an array! We could make a bunch of users and store them all in a big list!! You can even create the object inside the array!

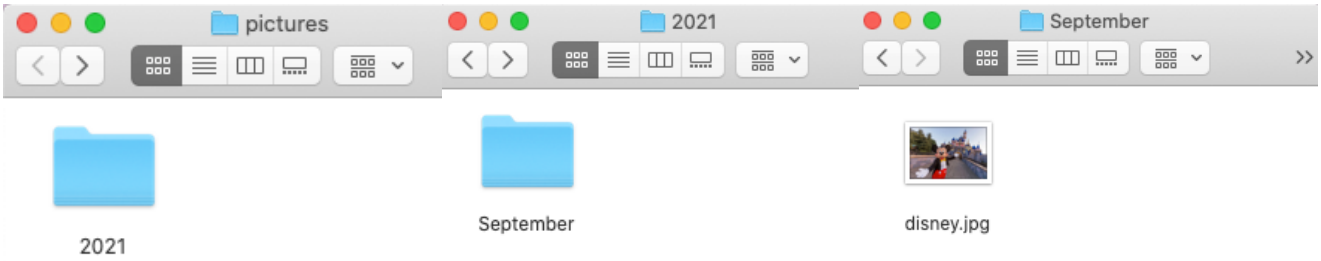
```
let users = [
    user1 { firstName: "Joe", lastName: "Biden", email: "joe@gmail.com", user:
    "mrPresident", password: "12345"},

    user2 { firstName: "Kamala", lastName: "Harris",email: "kamala@gmail.com", user:
    "VicePres", password: "54321"},

    user3 { firstName: "Mike", lastName: "Wazowski", email: "wazowski@gmail.com",
    user: "mikewazowSKI", password: "iloveboo"}

]
```

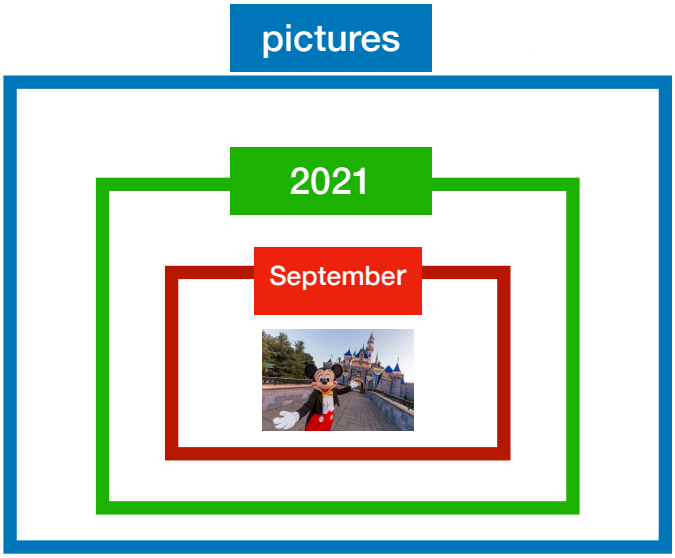
Now the users array can store as many users as we need - and it makes it really easy to add, delete, insert, or edit the list of users! Nice!

Now how can I access this data? What if I need to access Kamala Harris' username?

Think about this as finding the path to a file on your computer!

Let's say you have a picture of you at Disneyland in September, and you put that picture in a folder named September, and you put that folder in another folder called 2021, and then put that folder into a big folder called pictures.





So the path to this specific picture would be something like:

pictures/2021/September/disney.jpg

Arrays, objects, and data works similarly! You just need to know the path to access the data - and instead of slashes, we use dot notation!

To access Kamala Harris' email: users[1].user;

Let's make it a tiny bit more complicated:

```
let activities = [


        clubs = [
                club1 {name: "Creative Labs", day: "Wednesdays", Room: "E208"} ,
                club2 {name: "Fashion Club", day: "Tuesdays", Room: "E208"} ,
                club3 {name: "28%", day: "1st Monday of the Month", Room: "E208"}
        ],

        sports[
                sport1 {name: "volleyball", teams = ["FROSH", "JV", "Varsity"], season =
                "fall"},
                sport2 {name: "baseball", teams = ["FROSH", "JV", "Varsity"], season =
                "spring"}
        ]

]
```

So here I have an array **activities**, that holds two smaller arrays **clubs** and **sports** which each holds a series of objects!

Let's find the path to displaying "Varsity" for the volleyball team!

```
activities[1].sports[0].teams[2]
```

Guess what! This IS (basically) JSON!

JSON is just a format to store and access data using objects! JSON stands for JavaScript Object Notation! It basically is exactly what we just did… with a few syntax differences.

JSON Syntax and Format

https://www.youtube.com/watch?v=_NFkzw6oFtQ

JSON data is written in key/value pairs similar to objects! There is a *key** difference though - key values in JSON are surrounded by quotes!

```
{"name":"John"}
```

In **JSON**, *values* must be one of the following data types:

- a string
- a number
- an object
- an array
- a boolean
- null

An example of a simple JSON file!

```
{
    "name":"Katherine Johnson",
    "age":101,
    "orbital_mechanics": ["trajectories","launch windows","emergency return paths"],
    "mathmatician": true,
    "last_location": null
}
```

JSON best practices

Always enclose the key, value pair within double quotes. Most JSON parsers don't like to parse JSON objects with single quotes.

```
{ "name": "Katherine Johnson" }
```

Never use hyphens in your key fields. Use underscores (_), all lower case, or camel case.

```
{ "first_name":"Katherine", "last_name":"Johnson" }
```

Use a JSON linter to confirm valid JSON. Install a command line linter or use an online tool like JSONLint. If you copy this next example into a JSON linter, you should get a parse error for the pesky single quotes around the value for last_name.

```
{ "first_name":"Katherine", "last_name":'Johnson' }
```

* hehe - get it? KEY? :-)

Client - Server Architecture

This week we are going to be working with a javascript framework called Node.js to run our own servers, store and work with data, and start to build the foundation to work on multiplayer applications. But before we get into all of that, we need to understand how to client-server architecture works!

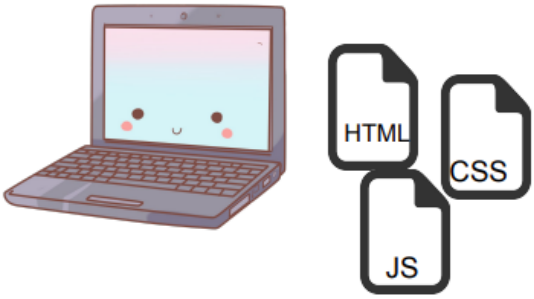
We're heard some of these words before when talking about APIs! Before we get into how it all works (and how to build our own server!), let's quickly review some definitions!

A **client** is the computer that is making the request.
A **server** is the computer fulfilling the request.

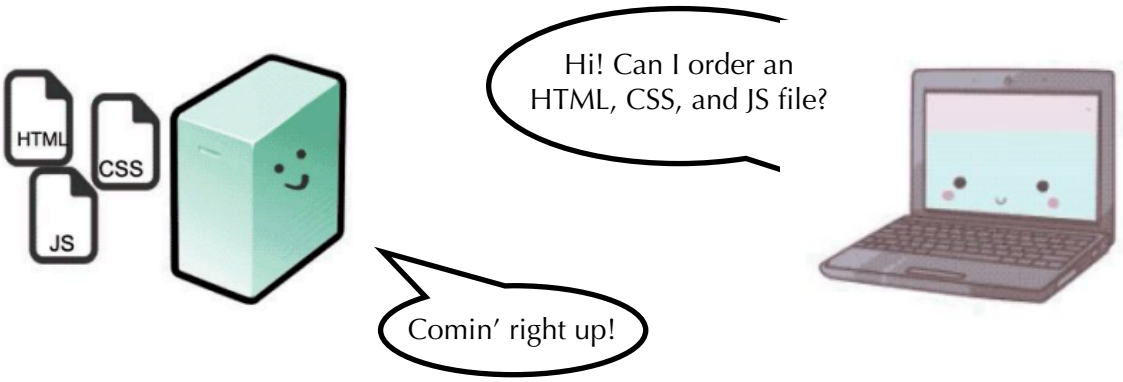
Easy enough? It is :) But how does it *work?* Let's see an example!

Let's say that you make a website! The website uses HTML, CSS, and JavaScript (and some images). They are stored in separate files. These files are stored on your computer. You can view and interact with this website by opening up the index.html file. It will open in your browser, and you can interact with the website.

So, now your website lives on your laptop. You can view and interact with the website on your laptop. What if you want to show your website to someone else? They would have to come over to your laptop to view it. What if we want any person, from any computer, to be able to view your website?



Let's put those website files on a server, instead of your personal computer. Now, when anyone wants to view the website, they ask the server for those files. Now, if I want to view my website, I ask the server for the files that make up the website, and once I get them, I render them in my browser!



Great! We've established how the first part of the client / server model works: we put the files for our website on the server, and so any client can make a request to the server for the website. But this example still only covers client-side only or static websites. What does that mean?

All of these clients are seeing the same website. They can interact with the website, but nothing that any one client does will change what another client sees.

What if one of these clients wants to leave a comment or a review on the website, that everyone can see? What if two of these clients want to chat with each other?

That would mean that a client would need to be able to interact with the site and send data to the server that other clients could also see --they would actually be able to change information being displayed on the site. The server would act as a sort of hub for the clients. How can our server handle all that?

Let's get into the server-side :)

To build up a server that can handle something other than just a static website, you need some server-side code! This code is what will allow us to handle communications between clients, store data, and change data!

Already seems like too much work… when do we NEED server side code?

You're playing a game, and want to store your high scores in a leaderboard. Anyone who visits the site can see all of the high scores from all of the users that have played that game! ***You need server side code!***

You're viewing a product on a website, and you want to leave a review for that product. Your review will be stored and displayed for any visitor to that product page to view! ***You need server side code!***

You want to play a game against someone else. Your player moves have to show up on their device, and vice versa! ***You need server side code!***

You're in a chatroom, and you want to be able to see what others are saying, and have your messages appear to the other users in the chatroom! ***You need server side code!***

So yeah, server side code is ***important!*** But how do we write it?

There are a lot of different languages and frameworks that you can use to write server side code. We're going to use **JavaScript**, specifically a framework called **Node.js**. This is nice because it means we can use JavaScript on the frontend (the client side) AND the backend (the server side)! That's up next!