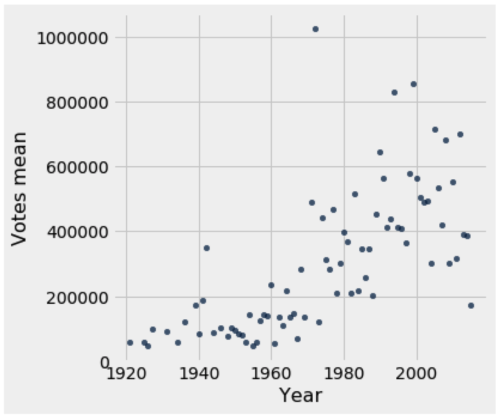
Introduction

Today we're going to write some code! Maybe you know some or maybe you don't, but this class is going to teach you how to use code as a tool to drive statistical analyses. We'll be coding in a language called python, that's relatively easier than other programming languages for beginners and also very prevalent in industry.

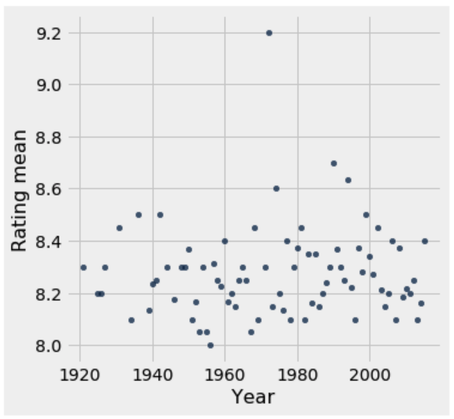
Coding in Jupyter Notebooks

We will be writing our code in Jupyter notebooks. Jupyter notebooks are awesome! It allows us to isolate subsets of our code so it's easier to find errors. Furthermore, Jupyter notebooks let us visualize code blocks very easily like so

```
In [19]: imdb_grouped = imdb.select('Votes', 'Year').group('Year', np.mean).scatter('Year')
```



```
In [21]: imdb_grouped = imdb.select('Rating', 'Year').group('Year', np.mean).scatter('Year')
```



If you're interested, the top graph shows the trend that on average, as time goes on, the more people vote on imdb for movies in general (for many reasons). The bottom graph shows that the average rating each year remains relatively constant as time progresses. More of this to come later in the course.

So, you can type any python code into the cells and once executed, it will display whatever you printed or graphed below it.

Types

Python reads code by identifying different things and knowing what to do with them. These are called types, more specifically primitive types. Here are some of them:

Numbers	`In  [1]: 3.14159`
	`Out [1]: 3.14159`
	
	`In  [2]: 2 + 2 - 1`
	`Out [2]: 3`
	
	`In  [3]: 5 / 2`
	`Out [3]: 2.5`
	
	`In  [4]: 2 ** 3`
	`Out [4]: 8`
		Numbers are like the numbers you know and love. They evaluate to themselves, which means that if you put just a number in a Jupyter cell and run it, it will print out the number itself. You can also do arithmetic with the numbers. Python also converts numbers do decimals when need be. Python covers all the basic operations:
		+, -, *, /, ** (exponent), and follows order of operations as well.

Strings	`In  [1]: “Golden”`
	`Out [1]: ‘Golden’`
	
	`In  [2]: ‘Go Bears!’`
	`Out [2]: ‘Go Bears!’`
	
	`In  [2]: “Berkeley’`
	`Out [2]: Error`
	
	`In  [2]: “Wouldn’t”`
	`Out [2]: “Wouldn’t”`
		Strings are words and you can have any symbol within quotes and python will be able to read it. Just make sure that when you wrap your words in quotes, you use the same quotes, either double or single quotes. Single and double quotes are the same, however if you want to print a single or double quote, then wrap the string in the other one.

Booleans	`In  [1]: True`
	`Out [1]: True`
	
	`In  [1]: False`
	`Out [1]: False`
	
	`In  [1]: 2 == 2`
	`Out [1]: True`
	
	`In  [1]: 2 == 3`
	`Out [1]: False`
		Booleans are either True or False. It’s binary. The keywords True and False evaluate to True and False, but you can also do more with it. For instance, the double equal signs “==” checks to see if two numbers are the same. So that is either True or False. Many other examples of this will pop up later in the course.

Names

Names are a way for python to store the values we just talked about above. Say we had some complicated expression like this:

```
2 + 7 + 3 / 5 + 10.38483 - 5
```

And we needed that number over and over again in our code. We wouldn't want to type that in every time we needed it. Instead, we can "bind" it to a name like so

```
In  [1]: random_number = 2 + 7 + 3 / 5 + 10.38483 - 5
         random_number
Out [1]: 14.984829999999999
```

In the example above, we "binded" that long expression to the name random_number. So now whenever you see random_number, that's the same as that super long math expression. As the course goes on, we will be storing things like large tables and numbers that change in names. For now just focus on the idea that we can "bind" or "store" the types we talked about above in names that are like words you use everyday.

The key is the equal sign, the left hand side is the name, then comes the "=" sign, then the expression on the right hand side. This is the syntax to bind a type to a name.

Also keep in mind that names cannot start with numbers. You should explore why that is the case and why python developers chose to make that rule

Comments

Comments are a way to document your code. The lines are preceded by a '#' sign which tells python "ignore everything after the '#' sign on this line". Here's an example to illustrate this

```
In  [1]: # This is a comment
         # print(5)
         print('Go Bears!')
Out [1]: Go Bears!
```

You can see that the comments were ignored and the only line that was executed was the third line. Even on the second line we said print(5), but since it was commented out, python ignored the line and didn't print the 5.

Functions

Functions are a big part of programming. For now, let's focus on the simpler kinds of functions. These functions take in numbers and spit out other numbers. This is best seen through examples

```
In  [1]: # Consider the function called abs that takes in a number
         # and spits out the absolute values of that number
         abs(-5)
Out [1]: 5
```

This is an example of a call expression. We take the function (abs) and put -5 within the parentheses next to the name abs. This signals to python, use the abs function and spit out whatever that function does, in this case the absolute value of a number. Let's see another example

```
In  [1]: # Consider the function called max that takes in numbers
         # and spits out the maximum of those numbers
         max(-5, 100, 35)
Out [1]: 100
```

Terminology:

- spitting out is the same as the word **returning**. So, the function max **returned** 100, which means it took in a bunch of numbers and what it spits out was 100.
- In `max(-5, 100, 35)`, max is the **operator** and -5, 100, and 35 are called **operands**

Nesting Call Expressions

You may come across a time where you have to do something like this:

```
min(abs(324-23423), add(max(2343, 2049), 2304))
```

This may seem intimidating at first, but if you follow some formulaic steps, it can be broken down into easier problems that you already know.

1. Look at each operand from left to right
2. Evaluate that operand until you get a certain value
3. Once all operands are completed, call the big expression on the value we found in step 2.

So let's use these steps to solve the problem above. First, we look at the first operand:

```
abs(324-23423)
```

Now, we evaluate it until we get a certain value. Since this is a call expression, we have to look at the operands from left to right. Since there's only one operand, we just look at that one:

```
324-23423
```

And this evaluates to -23, 099. Great, so now we go back and call the bigger call expression which is now simplified:

```
abs(-23099)
```

This becomes 23, 099. Now we can plug that into the super big call expression we had before:

```
min(23099, add(max(2343, 2049), 2304))
```

Now we have to look at the next operator:

```
add(max(2343, 2049), 2304)
```

Now, that I've done one sequence of the steps, I'll let you do the rest, but I'll show each step the whole way:

1. `add(2343, 2304)`
2. `min(23099, add(2343, 2304))`
3. `min(23099, 4647)`
4. `4647`

Hopefully you got 4647!

Next discuss more about types and functions as well as learn about importing functions. We'll also learn about the backbone of the course, arrays, the main way to store data.