Table-1 Data

- New form of data for CS101 -- "table"
- Re-use the code idioms, loops etc. from images
- Tables are a very common way to organize data on the computer

As another example of how data is stored and manipulated in the computer, we'll look at "table data" -- a common a way to organize strings, numbers, dates in rectangular table structure. In particular, we'll start with data from the social security administration baby name site.

Social Security Baby Name Table

- Names for babies born each year in the USA
- Top 1000 boy and girl names, 2000 names total
- Table terminology:

  - **table** the whole rectangle of data

  - **row** data for one name

  - **field** individual items (columns) in a row,
- Each field has a name: name, rank, gender, year

Table of baby-name data
(baby-2010.csv)

name	rank	gender	year
Jacob	1	boy	2010
Isabella	1	girl	2010
Ethan	2	boy	2010
Sophia	2	girl	2010
Michael	3	boy	2010

Field names

One row (4 fields)

2000 rows all told

Tables Are Extremely Common

- Rectangular table format is very common
- "Database" - extension of this basic table idea
- Number of fields is small (categories)
- Number of rows can be millions or billions
- e.g. email inbox: one row = one message, fields: date, subject, from, ...
- e.g. craigslist: one row = one thing for sale: description, price, seller, date, ...
- Demo craigslist search, list output mode

  - Craiglist has a 100 million "rows" , queries out 20 to show us

Much of the information stored on computers uses this table structure. One "thing" we want to store -- a baby name, someone's contact info, a craigslist advertisement -- is one row. The number of fields that

make up a row is fairly small -- essentially the fixed categories of information we think up for that sort of thing. For example one craigslist advertisement (stored in one row) has a few fields: a short description, a long description, a price, a seller, ... plus a few more fields.

The number of fields is small, but the number of rows can be quite large -- thousands or millions. When someone talks about a "database" on the computer, that builds on this basic idea of a table. Also storing data in a spreadsheet typically uses exactly this table structure.

Table Code

We'll start with some CS101 code -- SimpleTable -- which will serve as a foundation for you to write table code. Run the code to see what it does.

- Baby data stored in "baby-2010.csv"
- ".csv" stands for "comma separated values"

  - csv is a simple format to store a table as a text file
- Recall we had: `for (pixel: image) {` *code*
- For tables: `for (row: table) {` *code*
- print(row) prints out the fields of a row on one line

Table Query Logic

- Use if-statement to select certain rows
- A "query" in database terminology
- e.g. select rows where the rank is 6

```
if (row.getField("rank") == 6) { ...
```

- Loop runs for every rows (2000), if picks out some

The above code loops over all the rows, and the if-statement prints just the rows where the test is true -- here testing if the rank field is equal to 6, but really the if-statement could test anything about the row.

- Field names for the baby table: name, rank, gender, year
- `row.getField("field-name")` -- pick field out of row
- `==` are two values equal? (two equal signs)
- (demo) Warning: single equal sign `=` does variable assignment, not comparison. Use `==` inside if-test.
- Other comparisons: < > <= >=
- e.g. select row where the name is "Alice":

```
if (row.getField("name") == "Alice") { ...
```

The row object has a `row.getField("field-name")` function which returns the data for one field out of the row. Each field has a name -- one of "name" "rank" "gender" "year" in this case -- and the string name of the field is passed in to getField() to indicate which field we want, e.g.`row.getField("rank")` to retrieve the rank out of that row.

You can test if two values are equal in JavaScript with two equal signs joined like this: `==`. Using ==, the code to test if the name field is "Alice" is `row.getField("name") == "Alice"`

Note that a **single** equal sign = does variable assignment and not

comparison. It's a common mistake to type in one equal sign for a test, when you mean two equal signs. For this class, the Run button will detect an accidental use of a single = in an if-test and give an error message. The regular less-than/greater-than type tests: < > <= >= work as have seen before.

Example queries, and some you-try-it

- Baby table fields: name, rank, gender, year
- name field is "Robert", "Bob", "Abby", "Abigail" (try each in turn, yes nobody names their child "Bob" .. apparently always using Robert or Bobby)
- rank field is 1
- rank field is < 10
- rank field is <= 10
- rank field is > 990
- gender field is "girl"
- You try it:
- rank field is less than 15
- gender field is "boy"
- What is going on for all these: the loop goes through all 2000 rows and evaluates the if-test for each, printing that row only if the test is true.

Solution code:

Table-2 startsWith endsWith

Previously we did tests with == and <. In this short section, add the startsWith/endsWith functions which test which letters are at the start or end of a string.

- Alternative to ==, very handy for baby names
- Test if the name field in a row starts with "Ab":

```
if (row.getField("name").startsWith("Ab")) { ...
```

- Test if the name field in a row ends with "zy":

```
if (row.getField("name").endsWith("zy")) { ...
```

- Useful string functionality
- Not part of Javascript, but in other languages
- I added them just for this class
- Use these with row.getField("name") for many examples

These tests work very well with the name strings pulled out of the baby data. Here we can look at all the names beginning with "Ab".

- Variants to try above
- name field starts with "Ab", "A", "a" (lower case), "Z", "Za" (each in turn)
- name field ends with "z", "ly", "la" (each in turn)

For our purposes, strings support a `s.startsWith("Ab")` function, here testing if the string in the variable s starts with the "Ab" .. true or false. Likewise, there is `s.endsWith("yz")`, here testing if the string in variable s has "yz" at its very end. (Sadly, these two functions are not part of standard JavaScript; I made them work just for CS101 code because they are so useful. These two functions are common in other computer languages.)

Table-3 Boolean Logic

In this section, we'll extend our code with "boolean logic" ..
using **and or not** to combine multiple true/false tests.

Boolean Logic: && || !

- Want to be able to combine tests, in English like this:

  - Name starts with "A" **and** ends with "y"
- In code "boolean logic"
- **and** `&&` (two ampersands)
- **or** `||` (two vertical bars)
- **not** `!` (exclamation mark, holding off on this one for now)
- Sorry syntax is a bit cryptic -- historical syntax accident

- The && joins a startsWith test and an endsWith test
- The whole test is written on two lines because it is kind of long (optional)
- **Standalone rule:**

  -The tests joined by && || must be syntactically complete tests on their own

  -The tests are then joined with && or ||

- **Incorrect:**

  `row.getField("name).startsWith("A") && endsWith("y")`
- Common error: too few right parenthesis around the test
- (demo) Run tries to detect certain common errors, like omitting the {, or typing & instead of &&, giving an error message

Experiments to demo (then Students try 8 and later)

- For these examples, we'll use one of && || but not both.
- 1. name starts with "Ab" or name starts with "Ac"
- 2. name starts with "Ab" or name starts with "Ac" or name starts with "Al"
- 3. name starts with "O" and name ends with "a"
- 4. name starts with "O" and gender is "girl"
- 5. name ends with "a" and gender is "boy"
- 6. rank is <= 10 and gender is "girl" (translation: "top 10 girl names")
- 7. rank is <= 10 or gender is "girl" (this one doesn't make a ton of sense, but what does this print?)
- 8. name ends with "ia" and gender is "boy" (hah, then try with gender is "girl")
- 9. name ends with "io" and gender is "girl" (then try "boy")
- 10. name ends with "o" and gender is boy and rank is >= 900

Experiment solution code:

Table-3a Inside Outside

Inside/Outside Loop Experiment - Om Nom Nom Nom

Let's look again at the importance of inside vs. outside of loops.

- Loop - power technique to do something a zillion times
- For a line of code ...

  -inside vs. outside the loop is a huge difference.
- Experiment:
- Facts: 2000 names (rows) total, 12 names end with "x"
- Move this line to various spots: `print("nom");`
- How many "nom" for each "Location N" below.
- Enter your guess numbers below
- Then we'll try running it

Table-4 Not

Not !

- The boolean "not" operation inverts true and false
- We'll look at just two forms of not:
- Form (1):
- ! (exclamation mark) can go in front of an s.startsWith()

  s.endsWith() expression
- !row.getField("name").startsWith("A")
- translation: names not starting with "A"
- i.e. starting with any letter other than "A"
- Form (2):
- != a variant of ==, meaning "not equal to"
- e.g. row.getField("name") != "Alice"
- translation: names which are not equal to "Alice"
- i.e. any name other than "Alice"
- There are other ways that ! can be used, but the syntax gets a bit ugly, so we will stick to just these two forms.

Experiments to try:

- 1. girl names starting with "A" (no "nots" in this one)
- 2. girl names not starting with "A"
- 3. names starting with "A" and not ending with "y"
- 4. names starting with "A" and ending with "y" and not equal to "Abbey"

Table-5 Counting

Counting

- Thus far, we print all matching rows
- More useful to **count** the number of matching rows
- Make a "report"
- Requires some new variable manipulation code

Thus far we've used an if/print structure inside a loop to select certain rows to print or not. In this short section, we'll use a variable with a little code to **count** how many times an if-test is true. Below is a loop as we have seen before that detects all the names that start with "A". After printing the rows, the code prints a summary of how many names started with "A".

Code To Count

Count by making 3 additions to the standard loop/if/print structure we've used up to now:

- Three things to do counting:
- 1. Create a **count** variable and set it to 0 before the loop

  ```
  count = 0;
  ```

- 2. Inside the if-statement, increase count by 1

  ```
  count = count + 1;
  ```

  -Above line increases the value in **count** by 1

  -Evaluates the right hand side, then stores into variable (=)

- 3. Print the final value stored in count after the loop
  `print("count:", count);`
- **Pattern** three parts, the same every time (init, increment, print)
- Just know that `x = x + 1;` increments the value stored in a variable

Inside the if-statement, `count = count + 1;` increases whatever number is stored in count by 1 when the if-test is true. This is a weird use of the equal sign = vs. how it works it mathematics. First the line evaluates the right hand side. Then it assigns that value back into the count variable, so in effect it increases the number stored in count by 1.

Experiments:

- 1. Try commenting out or removing the print(row); line inside the { .. } then-code. What is the output now?
- 2. How many names start with "X"? Then change to count starting with "Y"?
- 3. How many girl names begin with "A"? Then change to count how many boy names begin with "A"?

Table-6 Counting Multiple Things

Now that we have table counting, the natural thing to want to do is count multiple things to compare them.

- Count multiple things in the loop
- Have multiple counters:

```
count1 = 0; // boy counter
count2 = 0; // girl counter
```

- Series of if-statements inside the loop (our official form)

  `x = x + 1;` -- within if-statement, correct variable

  Note the if-statements are not nested (more complex)
- After the loop, print both counters
- Alternative: could use more mnemonic variable names,

  like countBoy and countGirl

Do more boy or girl names end with "y"? We want a program which by the end prints "girl count:nn" and "boy count:nn", counting whatever characteristic we are looking for. This is beginning to look like an actual, practical use of computers to sift through data.

One approach is to use two (or more) counters, one for each case we want to count. Here we'll have one boy counter (count1) and one girl counter (count2). Initialize both counters to 0 before the loop. Then in the loop, have two if statements, one for each case we want to count. Inside each if-statement, increment the appropriate counter. At the end of the loop, print the counters.

It's possible to write the above code in different ways, but we will use that simple pattern: one if-statement for each case we want to count,

and the if-statements are all in the loop, one after the other (not one if inside another).

Class Survey

As another example of a table, we have the class survey of data from the live class at Stanford with people's favorite movies and what have you. The questions for this survey were:

- Gender: "male" or "female"
- What is your favorite color?
- What is your favorite current TV show?
- What is your favorite current movie?
- What is your favorite sport to play?
- What is your favorite current book?
- What is your favorite canned soda to drink?
- Field names for the survey table: gender, color, tv, movie, sport, book, soda

The survey answers are automatically translated to a google spreadsheet which can be exported in csv table. This data is available in the file "survey-2015.csv". This also illustrates that .csv format's role as an interchange format between systems.

Some data cleanup to make the answers consistent: changed "coca-cola" to "coke", "Navy" to "blue", "Dr. pepper" spelled with a period. Print the raw rows to see what the data looks like

The convertToLowerCase() function of the table changes all the text the table contains to lower case. This simplifies our logic, so we don't have to worry if someone typed in "Blue" or "blue" .. in the table it will always be the lowercase "blue" form. Therefore our query code should always look for the lowercase "blue" form. I cleaned up the data a bit for consistency, changing "Dark Blue" to just "Blue" and the many spellings of "Coca-Cola" to just "Coke", and things like that.

Survey Code - Example Problems

These can all be written using our "loop containing series of if-statements" form.

1. Write code to just print the soda field of each row, so we can get a feel for what people typed in. Note the effect of the convertToLowerCase() operation. Look at "color" and "sport" fields too.

2. Count 2 soda favorites: coke vs. sprite

3. Variant on (2) above, look only at people who's favorite color is blue

4. Variant on (2), use || to lump together for counting "coke" with "diet coke"

5. (**You Try It**) Count sports: soccer and volleyball

6. (**You Try It**) Variant of (5), count only gender female rows, then change to count male

Table Spreadsheet

Math Paradigms

- A "spreadsheet" is an easy way to do simple computations
- Everyone should be able to make a basic spreadsheet
- "paradigm" .. fancy word, but it applies here
- Numbers and formulas on paper is a paradigm
- Numbers and variables in computer code is a paradigm - e.g. `x = x + 1;`
- Spreadsheet paradigm is rows and columns of numbers (visual)
- Spreadsheets enable math without programming, a great invention
- History: Visicalc (1979), then Lotus, then Excel
- Spreadsheets energized the "personal computer" revolution

Spreadsheet software - many options: Google docs has a free spreadsheet in the browser, and now Microsoft has a free browser one too with skydive (yay competition!). There's also the free open source Libre Office application. And the famous Microsoft Excel spreadsheet products which work great and are kind of expensive. Any of these will work for all our examples and homeworks.

Aside: Who Makes Great Software?

- The creators of the spreadsheet knew finance math and paper spreadsheets
- They had some computer knowledge
- Theory: knowing the **problem domain** creates great software more than knowing CS
- -What problem to solve
- -How users look at the problem
- -The user's priorities
- e.g. Perhaps a working biologist will think up the next great biology program, not a CS person dabbling in biology
- Hidden agenda: everyone should know a little CS

Monster Example

Monster example spreadsheet in google docs. Below we'll use this as a running first example.

To edit above (or any of our spreadsheets): either (a) In google docs: File > Make Copy to edit. or (b) File > Download As > .xlsx file, and then edit using any program.

For references here is the monster spreadsheet in completed form

1. Spreadsheet Cells and Naming

- A spreadsheet is a rectangle of individual cells
- Each cell can contain number, date, text, .. whatever

- Addressing: columns are named: A, B, C, D, ...
- Addressing: rows are numbered: 1, 2, 3, 4, 5, ...
- So one cell can be identified like: B3, C12, A1, ..

Experiment: click on a cell, note its "address" B1 or whatever, type in a word or number

2. Columns of Numbers

- Very common to have a few columns of numbers
- e.g. the Red Castle and Blue Castle numbers here
- These are just raw numbers without computation

3. Add Computation: sum()

- Compute the total number of monsters in the blue castle
- Click on the B8 cell, a couple rows below the last blue castle number
- Type in the following "formula" (with the equal sign):**=sum(B1:B6)**
- The equal sign **=** at the start means this cell is computed from other cells
- The **sum()** adds up all the numbers in a range of cells

- The **B1:B6** means the whole vertical group of cells from B1 down through B6 (lowercase letters like b1:b6 work too)
- Type in "Total Count" in the cell to the left (A8) to serve as a label
- Famous Reinhart/Rogoff bug - wrong cells in formula

Spreadsheet Editing Tricks

- When you change a number up above, the sum is automatically updated
- Once you type in the =sum(...) in the cell, it is replaced with the computed sum number (28 in this case)
- Click the cell, edit up above
- OR double click the cell to edit
- Color shows cell-dependency (vs. wrong-cells bug)
- Type in "b1:b6" vs. click-drag
- Hit the esc key to cancel out of editing, a life saver -somehow I always find myself editing a cell I did not want to edit
- Using =sum() to add up a bunch of numbers is super common

From the headlines: Reinhart and Rogoff had a popular economics paper supporting austerity, but it had a significant spreadsheet bug. Essentially they wrote something like sum(a1:a8) when they intended (a1:a11), so they left out some numbers. This bug was significant in the paper's results. The subsequent history of the great recession has shown austerity to be a bad idea. Note when you double-click a cell, it shows you what it depends on to help avoid this sort of bug. It's amusing that such high level research can have ordinary bugs just like the rest of us, although of course this should be no surprise. Bugs are a common part of software.

4. Add Computation: + - * /

- Suppose every monster pays $100 per night and we want to compute the $ income per night, i.e. count*100
- We can write an arithmetic formula like **=B1 * B2** in a cell to compute a number based on the values of other cells
- Click the B9 cell just below the sum
- Type in the formula (with the equal sign): **=B8 * 100**
- Probably the easiest way to edit an existing formula such as in B8 and B9 is double clicking the cell
- Trick: while typing in the formula, instead of typing "B8", just click the cell you mean
- Type in "Total $/night" as a label to the left
- This is similar to the earlier sum() computation, but with basic + - * / type arithmetic

5. Magic: Fill Right

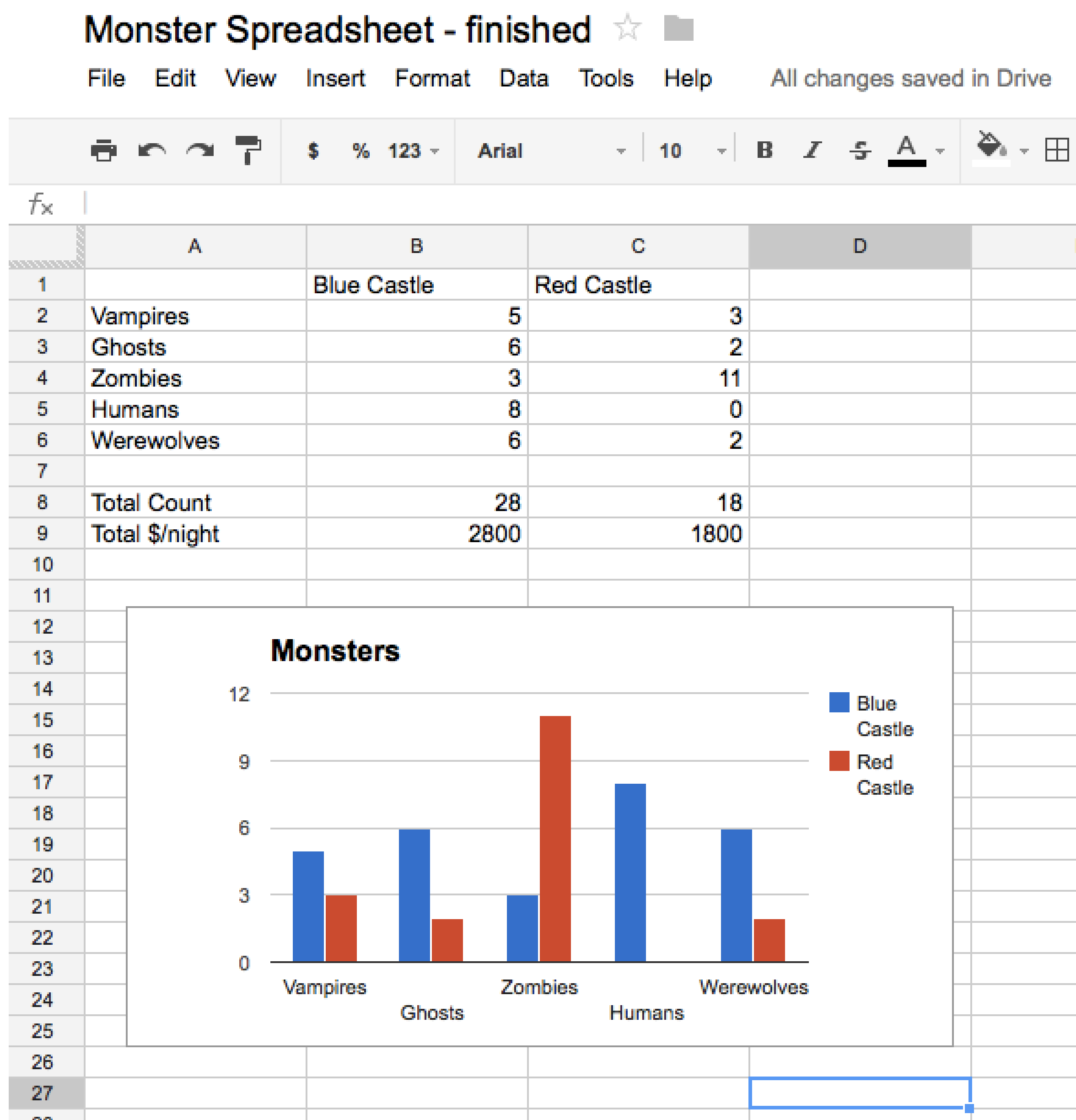
- Once you have the B8 and B9 formulas working the way you want, how to replicate them for the Red Castle?
- Easy!
- Click on B8 and drag right to highlight C8
- Type ctrl-R, the Fill Right command .. this is extreme magic
- Fill Right duplicates the formula over to the right
- Filling from column B to C
- Formula in column C updated to use C numbers
- Click B and C computed cells to check this
- Do Fill Right for the Total $/night formula as well

6. Chart Magic

- Finally we'll add a chart
- Click on A1 (the upper left of the data) and drag down to the lower right of the data (C6)
- Here just using the column titles and data
- Not the coputed cells
- Select Insert Chart
- There are many types of chart available

- Experiment with bar vs. line chart, or maybe add a title, resize it a bit
- Position the chart below all the numbers
- Notice: changing a number updates the chart
- Making pretty charts with your data is pretty easy

Here's a picture of it in done form:



Compute average with average(a1:a10)

- Above sum(a1:a10) computes the total sum of range of of numbers
- Similarly, average(a1:a10) compute the mean average of range of numbers
- sum() and average() are probably the two most commonly used functions

2. Cell Phone Example - You Try It

- Say we are studying how many times each person check's their cell phone per day
- Here it is in google docs:[Cell Phone Example]
- To edit above: either (a) File > Make Copy to edit in google docs. or (b) File > Download As > .xlsx file, and then edit using any system
- At the bottom of the numbers
- 1. Compute in separate cells the sum and average for each person (use sum() and average() and fill-right)
- 2. Off to the right, compute a single grand-total number of all the sums. You can just use = and + to make the grand total.

- 3. Make a line-graph chart of the raw data. Are the graphs in agreement with the computed averages?
- -Select upper-left (emily), drag down to the lower right (550). Click Insert > Chart