Intro-to-Python-and-Jupyter-Instructor

April 6, 2022

1 Visualizations in Python

1.1 Computing[MTU] Showcase - Spring 2022

Notebook adapted from UC Berkeley's Data8 and Data6 course materials

1.1.1 Table of Contents

Welcome to Jupyter Notebooks!

1. The Python Programming Language
   a. Expressions and Errors
   b. Names
   c. Functions
2. Tables
   a. Attributes
   b. Transformations
   c. Sorting Tables

1.2 The Jupyter Notebook

Welcome to the Jupyter Notebook! **Notebooks** are documents that can contain text, code, visualizations, and more. Jupyter notebooks support over 40 different programming languages, including Python, R, Julia, Scala, etc.

Notebooks are used for *literate programming*, a programming paradigm introduced by Donald Knuth in 1984, in which a programming language is accompanied with a documentation language, or a natural language. In other words, the computer program has an explanation in a natural language. This approach to programming effectively treats software as works of literature (Knuth, "Literate Programming"). It supports people to have a strong conceptual map of what is happening in the code and also have clarity on the flow and logic of the code/program. which is helpful for both for the writer and the reader.

Jupyter leverages this idea and enables users to create and share documents that combine code, visualizations, narrative text, equations, and rich media. Notebooks are multipurpose and can be used in any discipline. The notebook is like a laboratory notebook, but for computing. Researchers can write code to work with their data while supplementing their methods with explanations, analysis, or hypotheses. Notebooks are also used in education because they enable students to engage with content presented in different forms, , experience computation with no prior experience, and practice programming in a scaffolded way.

In this session, we'll be using Jupyter Notebooks to introduce you to how data scientists work with data, to learn about issues of justice using real-world data sets, and to also learn how to reason about the human choices embedded in the practice of data science and their significance.

1.2.1 Notebook Structure

A notebook is composed of rectangular sections called **cells**. There are 2 kinds of cells: markdown and code. A **markdown cell**, such as this one, contains text. A **code cell** contains code in Python, a programming language that we will be using for the remainder of this module. You can select any cell by clicking it once. After a cell is selected, you can navigate the notebook using the up and down arrow keys.

To run the code in a code cell, first click on that cell to activate it. It'll be highlighted with a little green or blue rectangle. Next, you can either press the | Run button above or press Shift + Return or Shift + Enter. This will run the current cell and select the next one.

If a code cell is running, you will see an asterisk (*) appear in the square brackets to the left of the cell. Once the cell has finished running, a number will replace the asterisk and any output from the code will appear under the cell.

[1]:

```python
# run this cell
print("Hello World!")
```

```
Hello World!
```

You'll notice that many code cells contain lines of blue text that start with a `#`. These are *comments*. Comments often contain helpful information about what the code does or what you are supposed to do in the cell. The leading `#` tells the computer to ignore them.

Editing You can edit a Markdown cell by clicking it twice. Text in Markdown cells is written in **Markdown**, a formatting syntax for plain text, so you may see some funky symbols when you edit a text cell.

Once you've made your changes, you can exit text editing mode by running the cell.

PRACTICE:
Try double-clicking on the text below to edit the cell to fix the misspelling.

Go Michgian Tech Huskies!

Code cells can be edited any time after they are highlighted.

PRACTICE:
Try editing the next code cell to print your name.

```
[2]: # edit the code to print your name
     print("Hello: my name is Laura")
```

```
Hello: my name is Laura
```

Adding Cells You can add a cell by clicking Insert > Insert Cell Below and then choosing the cell type in the drop down menu.

PRACTICE:
Try adding a cell below here and printing today's date (format: mm/dd/yyyy). Do not forget the quotation marks!

You can add cells by pressing the plus sign icon in the menu bar. This will add (by default) a code cell immediately below your current highlighted cell.

To convert a cell to markdown, you can press 'Cell' in the menu bar, select 'Cell Type', and finally pick the desired option. This works the other way around too! (you can also use the drop down menu in the toolbar above)

```
[3]: # added cell
     print("Today is 04/06/2022")
```

```
Today is 04/06/2022
```

Deleting Cells You can delete a cell by clicking the scissors at the top menu or Edit > Cut Cells. Delete the next cell below here.

A common fear is deleting a cell that you needed – but don't worry! This can be undone using 'Edit' > 'Undo Delete Cells'! If you accidentally delete content in a cell, you can use `Ctrl` + `Z` to undo.

PRACTICE:
Delete the code cell below.

```
[4]: ## DELETE THIS CELL
     print("delete me")
```

```
delete me
```

The cell above was deleted, do not delete any of the following cells.

Saving and Loading Your notebook can record all of your text and code edits, as well as any graphs you generate or calculations you make.

You are accessing the Jupyter notebook using "Binder", which allows hosted notebooks (from a url or github repository) to be shared as an interactive environment.

Therefore, if you close this page and relaunch the notebook from the website you will lose your changes.

Alternatively, if you access the notebook from a local computer, you can save the notebook as you make changes by clicking Control-S, clicking the floppy disc icon in the toolbar at the top of the page, or by going to the File menu and selecting "Save and Checkpoint".

Note: after loading a notebook you will see all the outputs (graphs, computations, etc) from your last session, but you won't be able to use any variables you assigned or functions you defined. You can get the functions and variables back by re-running the cells where they were defined- the easiest way is to highlight the cell where you left off work, then go to the Cell menu at the top of the screen and click "Run all above". You can also use this menu to run all cells in the notebook by clicking "Run all".

Downloading as PDF You can download this notebook as a pdf by clicking File > Download as > PDF via LaTeX. Try doing this now.

Downloading the Notebook You can also download the notebook, .ipynb extension, to save a copy with your changes to your local machine with File > Download as > Notebook (.ipynb)

Completing the Notebooks As you navigate the notebooks, you'll see cells with bold, all-capitalized headings that need to be filled in to complete the notebook.

Before we begin, we'll need a few extra tools to conduct our analysis. Run the next cell to load some code packages that we'll use later.

Note: this cell MUST be run in order for most of the rest of the notebook to work.

[5]:

```
# dependencies: THIS CELL MUST BE RUN
from datascience import *
import numpy as np
import math
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
import ipywidgets as widgets
%matplotlib inline
```

2 1. Python

Python is programming language- a way for us to communicate with the computer and give it instructions.

Just like any language, Python has a *vocabulary* made up of words it can understand, and a *syntax* giving the rules for how to structure communication.

Errors Python is a language, and like natural human languages, it has rules. It differs from natural language in two important ways: 1. The rules are *simple*. You can learn most of them in a few weeks and gain reasonable proficiency with the language in a semester. 2. The rules are *rigid*. If you're proficient in a natural language, you can understand a non-proficient speaker, glossing over small mistakes. A computer running Python code is not smart enough to do that.

Whenever you write code, you will often accientally break some of these rules. When you run a code cell that doesn't follow every rule exactly, Python will produce an **error message**.

Errors are *normal*; experienced programmers make many errors every day. Errors are also *not dangerous*; you will not break your computer by making an error (in fact, errors are a big part of how you learn a coding language). An error is nothing more than a message from the computer saying it doesn't understand you and asking you to rewrite your command.

We have made an error in the next cell. Run it and see what happens.

[ ]:

```
print("This line is not missing something."
```

```
  File "<ipython-input-6-1a13539de791>", line 1
    print("This line is not missing something."
                                               ^
SyntaxError: unexpected EOF while parsing
```

You should see something like this (minus our annotations):

The last line of the error output attempts to tell you what went wrong. The *syntax* of a language is its structure, and this `SyntaxError` tells you that you have created an illegal structure. "`EOF`" means "end of file," so the message is saying Python expected you to write something more (in this case, a right parenthesis) before finishing the cell.

There's a lot of terminology in programming languages, but you don't need to know it all in order to program effectively. If you see a cryptic message like this, you can often get by without deciphering it. (Of course, if you're frustrated, you can usually find out by searching for the error message online or posting on the Piazza.)

2.0.1 1a. Data

Data is information- the "stuff" we manipulate to make and test hypotheses.

Almost all data you will work with broadly falls into two types: numbers and text. *Numerical data* shows up green in code cells and can be positive, negative, or include a decimal.

[ ]:

```
# Numerical data

4

87623000983

```

```
-667

3.14159
```

[ ]: 3.14159

Text data (also called *strings*) shows up red in code cells. Strings are enclosed in double or single quotes. Note that numbers can appear in strings.

[ ]:

```
# Strings
"a"

"Hi there!"

"We hold these truths to be self-evident that all men are created equal."

# this is a string, NOT numerical data
"3.14159"
```

[ ]: '3.14159'

2.0.2 1a. Expressions

A bit of communication in Python is called an **expression**. It tells the computer what to do with the data we give it.

Here's an example of an expression. In general when you are writing expressions, you want to have a space between the number and operator. If it is a call expression (these use parentheses), then you do not have a space before the left end of the parentheses

[ ]:

```
# an expression
14 + 20
```

[ ]: 34

When you run the cell, the computer **evaluates** the expression and prints the result. Note that only the last line in a code cell will be printed, unless you explicitly tell the computer you want to print the result.

[ ]:

```
# more expressions. what gets printed and what doesn't?
100 / 10

print(4.3 + 10.98)

33 - 9 * (40000 + 1)

884
```

```
15.280000000000001
```

```
[ ]: 884
```

Many basic arithmetic operations are built in to Python, like `*` (multiplication), `+` (addition), `-` (subtraction), and `/` (division). There are many others, which you can find information about here. *Links to open source textbook used in an Introductory Data Science course*

The computer evaluates arithmetic according to the PEMDAS order of operations (just like you probably learned in a math class): anything in parentheses is done first, followed by exponents, then multiplication and division, and finally addition and subtraction.

```
[ ]: # before you run this cell, can you say what it should print?
     4 - 2 * (1 + 6 / 3)
```

```
[ ]: -2.0
```

PRACTICE: If you're new to python and coding, one of the best ways to get comfortable is to practice. Try writing and running different expressions in the cell below using numbers and the arithmetic operators `*` (multiplication), `+` (addition), `-` (subtraction), and `/` (division). See if you can generate different error messages and figure out what they mean.

```
[ ]: # Optional: try out different arithmetic operations
     # EXAMPLE SOLUTION
     2 * ((404 / 2)**0.5)**2 * 5 + 1
```

```
[ ]: 2021.0
```

2.0.3 1b. Names

Sometimes, the values you work with can get cumbersome- maybe the expression that gives the value is very complicated, or maybe the value itself is long. In these cases it's useful to give the value a **name**.

We can name values using what's called an *assignment* statement.

```
[ ]: # assigns 442 to x
     x = 442
```

The assignment statement has three parts. On the left is the *name* (`x`). On the right is the *value* (442). The *equals sign* in the middle tells the computer to assign the value to the name.

You'll notice that when you run the cell with the assignment, it doesn't print anything. But, if we try to access `x` again in the future, it will have the value we assigned it.

```
[ ]: # print the value of x
     x
```

```
[ ]: 442
```

You can also assign names to expressions. The computer will compute the expression and assign the name to the result of the computation.

[ ]:
```
y = 50 * 2 + 1
y
```

[ ]:
```
101
```

We can then use these name as if they were numbers.

[ ]:
```
x - 42
```

[ ]:
```
400
```

[ ]:
```
x + y
```

[ ]:
```
543
```

PRACTICE: Try rewriting the problem below, so the names make more sense. Do we expect an error below?

[ ]:
```
# Experiment with assigning names and doing arithmetic operations with named variables

one = 10
two = 300

sixty = one * two
sixty
```

[ ]:
```
3000
```

2.0.4 1c. Functions

We've seen that values can have names (often called **variables**), but operations may also have names. A named operation is called a **function**. Python has some functions built into it.

[ ]:
```
# a built-in function
round
```

[ ]:
```
<function round(number, ndigits=None)>
```

Functions get used in *call expressions*, where a function is named and given values to operate on inside a set of parentheses. The `round` function returns the number it was given, rounded to the nearest whole number.

[ ]:
```
# a call expression using round
round(1988.74699)
```

```
[ ]: 1989
```

A function may also be called on more than one value (called *arguments*). For instance, the `min` function takes however many arguments you'd like and returns the smallest. Multiple arguments are separated by commas.

```
[ ]: min(9, -34, 0, 99)
```

```
[ ]: -34
```

PRACTICE

The `abs` function takes one argument (just like `round`)

The `max` function takes one or more arguments (just like `min`)

Try calling `abs` and `max` in the cell below. What does each function do?

Also try calling each function *incorrectly*, such as with the wrong number of arguments. What kinds of error messages do you see?

```
[ ]: # replace the ... with calls to abs and max
     abs(-100)
```

```
[ ]: 100
```

```
[ ]: max(10, -100, 0)
```

```
[ ]: 10
```

Dot Notation Python has a lot of built-in functions (that is, functions that are already named and defined in Python), but even more functions are stored in collections called *modules*. Earlier, we imported the `math` module so we could use it later. Once a module is imported, you can use its functions by typing the name of the module, then the name of the function you want from it, separated with a `.`.

```
[ ]: # a call expression with the factorial function from the math module
     math.factorial(5)
```

```
[ ]: 120
```

Practice: `math` also has a function called `sqrt` that takes one argument and returns the square root. Call `sqrt` on 16 in the next cell.

```
[ ]: # use math.sqrt to get the square root of 16
     math.sqrt(16)
```

```
[ ]: 4.0
```

3 2. Tables

Note, this section makes use of the `datascience` *package that provides simple methods for dealing with data tables. For those without much Python experience, the functions provided are a natural way to start handing data tables. For those with more experience, you could rewrite the following code to make use of the* `pandas` *library.*

Tables are fundamental ways of organizing and displaying data. Run the next cell to load the data. **Run the cell below to load a dataset.**

[ ]:

```
# Below we see an assignment statement.
# We are telling the computer to create a Table and read in some data.

ratings = Table.read_table("data/imdb_ratings.csv")
ratings
```

[ ]:

```
Votes  | Rank | Title                          | Year | Decade
88355  | 8.4  | M (1931)                       | 1931 | 1930
132823 | 8.3  | Singin' in the Rain (1952)     | 1952 | 1950
74178  | 8.3  | All About Eve (1950)           | 1950 | 1950
635139 | 8.6  | Léon (1994)                    | 1994 | 1990
145514 | 8.2  | The Elephant Man (1980)        | 1980 | 1980
425461 | 8.3  | Full Metal Jacket (1987)       | 1987 | 1980
441174 | 8.1  | Gone Girl (2014)               | 2014 | 2010
850601 | 8.3  | Batman Begins (2005)           | 2005 | 2000
37664  | 8.2  | Judgment at Nuremberg (1961)   | 1961 | 1960
46987  | 8    | Relatos salvajes (2014)        | 2014 | 2010
… (240 rows omitted)
```

This table is organized into **columns**: one for each *category* of information collected:

You can also think about the table in terms of its **rows**. Each row represents all the information collected about a particular instance, which can be a person, location, action, or other unit.

PRACTICE: What do the rows in this table represent? By default only the first ten rows are shown. Can you see how many rows there are in total?

3.1 2a. Table Attributes

Every table has **attributes** that give information about the table, like the number of rows and the number of columns. Table attributes are accessed using the dot method. But, since an attribute doesn't perform an operation on the table, there are no parentheses (like there would be in a call expression).

Attributes you'll use frequently include `num_rows` and `num_columns`, which give the number of rows and columns in the table, respectively.

[ ]:

```
# get the number of columns
ratings.num_columns
```

```
[ ]: 5
```

PRACTICE: Use `num_rows` to get the number of rows in our table.

```
[ ]: # get the number of rows in the table
```

Note on Tables In this notebook, we worked with the datascience library to work with tables. In practice and in industry, you'll most likely use pandas to manipulate tabular data. Using the datascience library is more friendly syntax-wise, but pandas is more powerful overall.

3.2 2b. Table Transformation

Not all of our columns are relevant to every question we want to ask. We can save computational resources and avoid confusion by *transforming* our table before we start work.

Subsetting columns with `select` and `drop`

3.2.1 `select`

The `select` function is used to get a table containing only particular columns. `select` is called on a table using dot notation and takes one or more arguments: the name or names of the column or columns you want. Not this does not change the original table. To save your changes, you must assign your change a name.

```
[ ]: # make a new table with only selected columns
     ratings.select("Votes", "Title")
```

```
[ ]: Votes  | Title
     88355  | M (1931)
     132823 | Singin' in the Rain (1952)
     74178  | All About Eve (1950)
     635139 | Léon (1994)
     145514 | The Elephant Man (1980)
     425461 | Full Metal Jacket (1987)
     441174 | Gone Girl (2014)
     850601 | Batman Begins (2005)
     37664  | Judgment at Nuremberg (1961)
     46987  | Relatos salvajes (2014)
     … (240 rows omitted)
```

```
[ ]: # No changes made to the original table
     ratings
```

```
[ ]: Votes  | Rank | Title                          | Year | Decade
     88355  | 8.4  | M (1931)                       | 1931 | 1930
```

```
132823 | 8.3  | Singin' in the Rain (1952)   | 1952 | 1950
74178  | 8.3  | All About Eve (1950)         | 1950 | 1950
635139 | 8.6  | Léon (1994)                  | 1994 | 1990
145514 | 8.2  | The Elephant Man (1980)      | 1980 | 1980
425461 | 8.3  | Full Metal Jacket (1987)     | 1987 | 1980
441174 | 8.1  | Gone Girl (2014)             | 2014 | 2010
850601 | 8.3  | Batman Begins (2005)         | 2005 | 2000
37664  | 8.2  | Judgment at Nuremberg (1961) | 1961 | 1960
46987  | 8    | Relatos salvajes (2014)      | 2014 | 2010
… (240 rows omitted)
```

3.2.2 drop

If instead you need all columns except a few, the **drop** function can get rid of specified columns. **drop** works very similarly to **select**: call it on the table using dot notation, then give it the name or names of what you want to drop.

```
[ ]: # drop a column
     ratings.drop("Decade")
```

```
[ ]: Votes  | Rank | Title                        | Year
     88355  | 8.4  | M (1931)                     | 1931
     132823 | 8.3  | Singin' in the Rain (1952)   | 1952
     74178  | 8.3  | All About Eve (1950)         | 1950
     635139 | 8.6  | Léon (1994)                  | 1994
     145514 | 8.2  | The Elephant Man (1980)      | 1980
     425461 | 8.3  | Full Metal Jacket (1987)     | 1987
     441174 | 8.1  | Gone Girl (2014)             | 2014
     850601 | 8.3  | Batman Begins (2005)         | 2005
     37664  | 8.2  | Judgment at Nuremberg (1961) | 1961
     46987  | 8    | Relatos salvajes (2014)      | 2014
     … (240 rows omitted)
```

PRACTICE: Pick two columns from our table. Create a new table containing only those two columns two different ways: once using **select** and once using **drop**.

```
[ ]: # use select
     ratingsOne = ratings.select("Votes", "Title")
     ratingsOne
```

```
[ ]: Votes  | Title
     88355  | M (1931)
     132823 | Singin' in the Rain (1952)
     74178  | All About Eve (1950)
     635139 | Léon (1994)
     145514 | The Elephant Man (1980)
     425461 | Full Metal Jacket (1987)
```

```
441174 | Gone Girl (2014)
850601 | Batman Begins (2005)
37664  | Judgment at Nuremberg (1961)
46987  | Relatos salvajes (2014)
… (240 rows omitted)
```

```
[ ]: # use drop
     ratings2 = ratings.drop("Rank", "Year", "Decade")
     ratings2
```

```
[ ]: Votes  | Title
     88355  | M (1931)
     132823 | Singin' in the Rain (1952)
     74178  | All About Eve (1950)
     635139 | Léon (1994)
     145514 | The Elephant Man (1980)
     425461 | Full Metal Jacket (1987)
     441174 | Gone Girl (2014)
     850601 | Batman Begins (2005)
     37664  | Judgment at Nuremberg (1961)
     46987  | Relatos salvajes (2014)
     … (240 rows omitted)
```

3.3 2c. Sorting Tables

The last section covered select and drop. In this next section, we'll test our knowledge and use what we have learned and understand how we can sort and manipulate data that are placed in tables.

3.3.1 `show`

In a table, we can display a specific amount of rows using the `show` operation. The `show` operations allows you to enter the amount of rows you want displayed from a table.

```
[ ]: # use show to display 20 rows
     ratings.show(20)
```

```
<IPython.core.display.HTML object>
```

3.3.2 `sort`

Some details about sort from the UC Berkeley's Data 8 lab 2 (Spring 2020):

1. The first argument to `sort` is the name of a column to sort by.
2. If the column has text in it, `sort` will sort alphabetically; if the column has numbers, it will sort numerically.

3. The `descending=False` bit is called an *optional argument*. It has a default value of `False`, so when you explicitly tell the function `descending=True`, then the function will sort in descending order.
4. The `distinct=True` bit is also an *optional argument*. When the function `distinct=True` is used, the function will delete any duplicate values based on the column_or_label value that was also passed in the `sort` function.
5. Rows always stick together when a table is sorted. It wouldn't make sense to sort just one column and leave the other columns alone. For example, in this case, if we sorted just the `Year` column, the movies would all end up with the wrong year.

The format for sorting code is written as….

`Table.sort(column_or_label, descending=False, distinct=False)`

```
[ ]: # sort a column
     ratings.sort("Year")
```

```
[ ]: Votes  | Rank | Title                                | Year | Decade
     55784  | 8.3  | The Kid (1921)                       | 1921 | 1920
     58506  | 8.2  | The Gold Rush (1925)                 | 1925 | 1920
     46332  | 8.2  | The General (1926)                   | 1926 | 1920
     98794  | 8.3  | Metropolis (1927)                    | 1927 | 1920
     88355  | 8.4  | M (1931)                             | 1931 | 1930
     92375  | 8.5  | City Lights (1931)                   | 1931 | 1930
     56842  | 8.1  | It Happened One Night (1934)         | 1934 | 1930
     121668 | 8.5  | Modern Times (1936)                  | 1936 | 1930
     69510  | 8.2  | Mr. Smith Goes to Washington (1939)  | 1939 | 1930
     259235 | 8.1  | The Wizard of Oz (1939)              | 1939 | 1930
     … (240 rows omitted)
```

```
[ ]: # Sorted from most recent year to oldest year
     ratings.sort("Year", descending=True)
```

```
[ ]: Votes  | Rank | Title                               | Year | Decade
     79615  | 8.5  | Inside Out (2015/I)                 | 2015 | 2010
     262425 | 8.3  | Mad Max: Fury Road (2015)           | 2015 | 2010
     441174 | 8.1  | Gone Girl (2014)                    | 2014 | 2010
     46987  | 8    | Relatos salvajes (2014)             | 2014 | 2010
     427099 | 8    | X-Men: Days of Future Past (2014)   | 2014 | 2010
     321834 | 8    | The Imitation Game (2014)           | 2014 | 2010
     689541 | 8.6  | Interstellar (2014)                 | 2014 | 2010
     527349 | 8    | Guardians of the Galaxy (2014)      | 2014 | 2010
     369141 | 8.1  | The Grand Budapest Hotel (2014)     | 2014 | 2010
     264333 | 8.5  | Whiplash (2014)                     | 2014 | 2010
     … (240 rows omitted)
```

PRACTICE: Sort the table from highest ranked movie to lowest ranked.

[ ]:

```
# Sorted from highest ranked movie to lowest ranked
ratings.sort("Rank", descending=True)
```

[ ]:

```
Votes   | Rank | Title                                            | Year |
Decade
1027398 | 9.2  | The Godfather (1972)                             | 1972 |
1970
1498733 | 9.2  | The Shawshank Redemption (1994)                  | 1994 |
1990
692753  | 9    | The Godfather: Part II (1974)                    | 1974 |
1970
447875  | 8.9  | Il buono, il brutto, il cattivo (1966)           | 1966 |
1960
1473049 | 8.9  | The Dark Knight (2008)                           | 2008 |
2000
384187  | 8.9  | 12 Angry Men (1957)                              | 1957 |
1950
1074146 | 8.9  | The Lord of the Rings: The Return of the King (2003) | 2003 |
2000
761224  | 8.9  | Schindler's List (1993)                          | 1993 |
1990
1166532 | 8.9  | Pulp Fiction (1994)                              | 1994 |
1990
1177098 | 8.8  | Fight Club (1999)                                | 1999 |
1990
… (240 rows omitted)
```

PRACTICE: Sort the table by year in ascending order, making sure each year is distinct.

[ ]:

```
ratings.sort("Year", distinct=True)
```

[ ]:

```
Votes  | Rank | Title                                 | Year | Decade
55784  | 8.3  | The Kid (1921)                        | 1921 | 1920
58506  | 8.2  | The Gold Rush (1925)                  | 1925 | 1920
46332  | 8.2  | The General (1926)                    | 1926 | 1920
98794  | 8.3  | Metropolis (1927)                     | 1927 | 1920
88355  | 8.4  | M (1931)                              | 1931 | 1930
56842  | 8.1  | It Happened One Night (1934)          | 1934 | 1930
121668 | 8.5  | Modern Times (1936)                   | 1936 | 1930
69510  | 8.2  | Mr. Smith Goes to Washington (1939)   | 1939 | 1930
55793  | 8.1  | The Grapes of Wrath (1940)            | 1940 | 1940
101754 | 8.1  | The Maltese Falcon (1941)             | 1941 | 1940
… (70 rows omitted)
```

[ ]:

```
```

References

- Sections of “Intro to Jupyter”, “Table Transformation” adapted from materials by Kelly Chen and Ashley Chien in UC Berkeley Data Science Modules core resources
- “A Note on Errors” subsection and “error” image adapted from materials by Chris Hench and Mariah Rogers for the Medieval Studies 250: Text Analysis for Graduate Medievalists data science module.
- Rocket Fuel data and discussion questions adapted from materials by Zsolt Katona and Brian Bell, BerkeleyHaas Case Series

Initially authored by Keeley Takimoto, adapated by BUDS team, further adapted by Laura Brown.

[ ]: