# Lab 4: Functions and Visualizations

Welcome to Lab 4! This week, we've learned about functions, table methods such as `apply`, and how to generate visualizations! We will expand upon all of those in Lab today.

Recommended Reading:

- Applying a Function to a Column
- Visualizations

First, set up the notebook by running the cell below.

```
import numpy as np
from datascience import *

# These lines set up graphing capabilities.
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
import warnings
warnings.simplefilter('ignore', FutureWarning)

from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

## 1. Defining functions

Let's write a very simple function that converts a proportion to a percentage by multiplying it by 100. For example, the value of `to_percentage(.5)` should be the number 50 (no percent sign).

A function definition has a few parts.

`def`

It always starts with `def` (short for **def**ine):

```
def
```

Name

Next comes the name of the function. Like other names we've defined, it can't start with a number or contain spaces. Let's call our function `to_percentage`:

```
def to_percentage
```

Signature

Next comes something called the *signature* of the function. This tells Python how many arguments your function should have, and what names you'll use to refer to those arguments in the function's code. A function can have any number of arguments (including 0!).

`to_percentage` should take one argument, and we'll call that argument `proportion` since it should be a proportion.

```
def to_percentage(proportion)
```

If we want our function to take more than one argument, we add a comma between each argument name. Note that if we had zero arguments, we'd still place the parentheses () after than name.

We put a colon after the signature to tell Python it's over. If you're getting a syntax error after defining a function, check to make sure you remembered the colon!

```
def to_percentage(proportion):
```

Documentation

Functions can do complicated things, so you should write an explanation of what your function does. For small functions, this is less important, but it's a good habit to learn from the start. Conventionally, Python functions are documented by writing an **indented** triple-quoted string:

```
def to_percentage(proportion):
    """Converts a proportion to a percentage."""
```

Body

Now we start writing code that runs when the function is called. This is called the *body* of the function and every line **must be indented with a tab**. Any lines that are *not* indented and left-aligned with the def statement is considered outside the function.

Some notes about the body of the function:

- We can write code that we would write anywhere else.
- We use the arguments defined in the function signature. We can do this because we assume that when we call the function, values are already assigned to those arguments.
- We generally avoid referencing variables defined *outside* the function. If you would like to reference variables outside of the function, pass them through as arguments!

Now, let's give a name to the number we multiply a proportion by to get a percentage:

```
def to_percentage(proportion):
    """Converts a proportion to a percentage."""
    factor = 100
```

`return`

The special instruction `return` is part of the function's body and tells Python to make the value of the function call equal to whatever comes right after `return`. We want the value of `to_percentage(.5)` to be the proportion .5 times the factor 100, so we write:

```
def to_percentage(proportion):
    """Converts a proportion to a percentage."""
    factor = 100
    return proportion * factor
```

`return` only makes sense in the context of a function, and **can never be used outside of a function**. `return` is always the last line of the function because Python stops executing the body of a function once it hits a `return` statement.

*Note:* `return` inside a function tells Python what value the function evaluates to. However, there are other functions, like `print`, that have no `return` value. For example, `print` simply prints a certain value out to the console.

`return` and `print` are **very** different.

**Question 1.1.** Define `to_percentage` in the cell below. Call your function to convert the proportion .2 to a percentage. Name that percentage `twenty_percent`.

```
def to_percentage(proportion):
    ''' Converts a proportion to a percentage'''
    factor = 100
    return proportion * factor

twenty_percent = to_percentage(.2)
twenty_percent
```

```
20.0
```

Like you've done with built-in functions in previous labs (max, abs, etc.), you can pass in named values as arguments to your function.

**Question 1.2.** Use `to_percentage` again to convert the proportion named `a_proportion` (defined below) to a percentage called `a_percentage`.

*Note:* You don't need to define `to_percentage` again! Like other named values, functions stick around after you define them.

```
a_proportion = 2**(.5) / 2
a_percentage = to_percentage(a_proportion)
a_percentage
```

```
70.71067811865476
```

Here's something important about functions: the names assigned *within* a function body are only accessible within the function body. Once the function has returned, those names are gone. So even if you created a variable called `factor` and defined `factor = 100` inside of the body of the `to_percentage` function and then called `to_percentage`, `factor` would not have a value assigned to it outside of the body of `to_percentage`:

```
# You should see an error when you run this.  (If you don't, you might
# have defined factor somewhere above.)
factor
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-a219be0dab32> in <cell line: 3>()
      1 # You should see an error when you run this.  (If you don't, you might
      2 # have defined factor somewhere above.)
----> 3 factor

NameError: name 'factor' is not defined
```

As we've seen with built-in functions, functions can also take strings (or arrays, or tables) as arguments, and they can return those things, too.

**Question 1.3.** Define a function called `disemvowel`. It should take a single string as its argument. (You can call that argument whatever you want.) It should return a copy of that string, but with all the characters that are vowels removed. (In English, the vowels are the characters "a", "e", "i", "o", and "u".) You can use as many lines inside of the function to do this as you'd like.

*Hint:* To remove all the "a"s from a string, you can use `that_string.replace("a", "")`. The `.replace` method for strings returns a new string, so you can call `replace` multiple times, one after the other.

```
def disemvowel(a_string):
    return a_string.replace("a", "").replace("o", "").replace("e", "").replace("i", "").replace("u", "")
    ...

# An example call to your function.  (It's often helpful to run
# an example call from time to time while you're writing a function,
# to see how it currently works.)
disemvowel("Can you read this without vowels?")
```

```
'Cn y rd ths wtht vwls?'
```

## Calls on calls on calls

Just as you write a series of lines to build up a complex computation, it's useful to define a series of small functions that build on each other. Since you can write any code inside a function's body, you can call other functions you've written.

If a function is a like a recipe, defining a function in terms of other functions is like having a recipe for cake telling you to follow another recipe to make the frosting, and another to make the jam filling. This makes the cake recipe shorter and clearer, and it avoids having a bunch of duplicated frosting recipes. It's a foundation of productive programming.

For example, suppose you want to count the number of characters *that aren't vowels* in a piece of text. One way to do that is this to remove all the vowels and count the size of the remaining string.

**Question 1.4.** Write a function called `num_non_vowels`. It should take a string as its argument and return a number. That number should be the number of characters in the argument string that aren't vowels. You should use the `disemvowel` function you wrote above inside of the `num_non_vowels` function.

*Hint:* The function `len` takes a string as its argument and returns the number of characters in it.

```
def num_non_vowels(a_string):
    """The number of characters in a string, minus the vowels."""
    return len(disemvowel(a_string))

# Try calling your function yourself to make sure the output is what
# you expect. You can also use the interact function in the next cell if you'd like.
```

Functions can also encapsulate code that *displays output* instead of computing a value. For example, if you call `print` inside a function, and then call that function, something will get printed.

The `movies_by_year` dataset in the textbook has information about movie sales in recent years. Suppose you'd like to display the year with the 5th-highest total gross movie sales, printed in a human-readable way. You might do this:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
movies_by_year = Table.read_table("/content/drive/MyDrive/movies_by_year.csv")
rank = 5
fifth_from_top_movie_year = movies_by_year.sort("Total Gross", descending=True).column("Year").item(rank-1)
print("Year number", rank, "for total gross movie sales was:", fifth_from_top_movie_year)
```

```
Year number 5 for total gross movie sales was: 2010
```

After writing this, you realize you also wanted to print out the 2nd and 3rd-highest years. Instead of copying your code, you decide to put it in a function. Since the rank varies, you make that an argument to your function.

**Question 1.5.** Write a function called `print_kth_top_movie_year`. It should take a single argument, the rank of the year (like 2, 3, or 5 in the above examples). It should print out a message like the one above.

*Note:* Your function shouldn't have a `return` statement.

```
def print_kth_top_movie_year(k):

    kth_from_top_movie_year = movies_by_year.sort("Total Gross", descending=True).column("Year").item(k-1)
    print("Year number", k, "for total gross movie sales was:", kth_from_top_movie_year)

# Example calls to your function:
print_kth_top_movie_year(2)
print_kth_top_movie_year(3)
```

```
Year number 2 for total gross movie sales was: 2013
Year number 3 for total gross movie sales was: 2012
```

```
# interact also allows you to pass in an array for a function argument. It will
# then present a dropdown menu of options.
_ = interact(print_kth_top_movie_year, k=np.arange(1, 10))
```

```
k 1
Year number 1 for total gross movie sales was: 2015
```

## `print` is not the same as `return`

The `print_kth_top_movie_year(k)` function prints the total gross movie sales for the year that was provided! However, since we did not return any value in this function, we can not use it after we call it. Let's look at an example of another function that prints a value but does not return it.

```
def print_number_five():
    print(5)
```

```
print_number_five()
```

```
5
```

However, if we try to use the output of `print_number_five()`, we see that the value `5` is printed but we get a TypeError when we try to add the number 2 to it!

```
print_number_five_output = print_number_five()
print_number_five_output + 2
```

```
5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-8bfda6b3fa47> in <cell line: 2>()
      1 print_number_five_output = print_number_five()
----> 2 print_number_five_output + 2

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

It may seem that `print_number_five()` is returning a value, 5. In reality, it just displays the number 5 to you without giving you the actual value! If your function prints out a value without returning it and you try to use that value, you will run into errors, so be careful!

Explain to your neighbor how you might add a line of code to the `print_number_five` function (after `print(5)`) so that the code `print_number_five_output + 5` would result in the value `10`, rather than an error.

## 2. Functions and CEO Incomes

In this question, we'll look at the 2015 compensation of CEOs at the 100 largest companies in California. The data was compiled from a Los Angeles Times analysis, and ultimately came from filings mandated by the SEC from all publicly-traded companies. Two companies have two CEOs, so there are 102 CEOs in the dataset.

We've copied the raw data from the LA Times page into a file called `raw_compensation.csv`. (The page notes that all dollar amounts are in **millions of dollars**.)

```
raw_compensation = Table.read_table('/content/drive/MyDrive/raw_compensation.csv')
raw_compensation
```

| Rank | Name | Company (Headquarters) | Total Pay | % Change | Cash Pay | Equity Pay | Other Pay | Ratio of CEO pay to average industry worker pay |
|---|---|---|---|---|---|---|---|---|
| 1 | Mark V. Hurd* | Oracle (Redwood City) | $53.25 | (No previous year) | $0.95 | $52.27 | $0.02 | 362 |
| 2 | Safra A. Catz* | Oracle (Redwood City) | $53.24 | (No previous year) | $0.95 | $52.27 | $0.02 | 362 |
| 3 | Robert A. Iger | Walt Disney (Burbank) | $44.91 | -3% | $24.89 | $17.28 | $2.74 | 477 |
| 4 | Marissa A. Mayer | Yahoo! (Sunnyvale) | $35.98 | -15% | $1.00 | $34.43 | $0.55 | 342 |
| 5 | Marc Benioff | salesforce.com (San Francisco) | $33.36 | -16% | $4.65 | $27.26 | $1.45 | 338 |

We want to compute the average of the CEOs' pay. Try running the cell below.

```
np.average(raw_compensation.column("Total Pay"))
```

```
---------------------------------------------------------------------------
UFuncTypeError                            Traceback (most recent call last)
<ipython-input-19-f97fab5a8083> in <cell line: 1>()
----> 1 np.average(raw_compensation.column("Total Pay"))

1 frames
/usr/local/lib/python3.10/dist-packages/numpy/core/_methods.py in _mean(a, axis, dtype,
out, keepdims, where)
    116             is_float16_result = True
    117 
--> 118     ret = umr_sum(arr, axis, dtype, out, keepdims, where=where)
    119     if isinstance(ret, mu.ndarray):
    120         with _no_nep50_warning():

UFuncTypeError: ufunc 'add' did not contain a loop with signature matching types
(dtype('<U7'), dtype('<U7')) -> None
```

You should see a TypeError. Let's examine why this error occurred by looking at the values in the `Total Pay` column.

**Question 2.1.** Use the `type` function and set `total_pay_type` to the type of the first value in the "Total Pay" column.

```
total_pay_type = type(raw_compensation.column("Total Pay")[0])
total_pay_type
```

```
numpy.str_
```

**Question 2.2.** You should have found that the values in the `Total Pay` column are strings. It doesn't make sense to take the average of string values, so we need to convert them to numbers if we want to do this. Extract the first value in `Total Pay`. It's Mark Hurd's pay in 2015, in *millions* of dollars. Call it `mark_hurd_pay_string`.

```
mark_hurd_pay_string = raw_compensation.column("Total Pay")[0]
mark_hurd_pay_string
```

**Question 2.3.** Convert `mark_hurd_pay_string` to a number of *dollars*.

Some hints, as this question requires multiple steps:

- The string method `strip` will be useful for removing the dollar sign; it removes a specified character from the start or end of a string. For example, the value of `"100%".strip("%")` is the string `"100"`.
- You'll also need the function `float`, which converts a string that looks like a number to an actual number.
- Finally, remember that the answer should be in dollars, not millions of dollars.

```
mark_hurd_pay = float(raw_compensation.column("Total Pay")[0].strip("$"))*1000000
mark_hurd_pay
```

```
53250000.0
```

To compute the average pay, we need to do this for every CEO. But that looks like it would involve copying this code 102 times.

This is where functions come in. First, we'll define a new function, giving a name to the expression that converts "total pay" strings to numeric values. Later in this lab, we'll see the payoff: we can call that function on every pay string in the dataset at once.

The next section of this lab explains how to define a function For now, just fill in the ellipses in the cell below.

**Question 2.4.** Copy the expression you used to compute `mark_hurd_pay`, and use it as the return expression of the function below. But make sure you replace the specific `mark_hurd_pay_string` with the generic `pay_string` name specified in the first line in the `def` statement.

*Hint*: When dealing with functions, you should generally not be referencing any variable outside of the function. Usually, you want to be working with the arguments that are passed into it, such as `pay_string` for this function. If you're using `mark_hurd_pay_string` within your function, you're referencing an outside variable!

```
def convert_pay_string_to_number(pay_string):
    """Converts a pay string like '$100' (in millions) to a number of dollars."""
    return float(pay_string.strip("$"))*1000000
```

Running that cell doesn't convert any particular pay string. Instead, it creates a function called `convert_pay_string_to_number` that can convert *any* string with the right format to a number representing millions of dollars.

We can call our function just like we call the built-in functions we've seen. It takes one argument -- a string -- and it returns a float.

```
convert_pay_string_to_number('$42')
```

```
42000000.0
```

```
convert_pay_string_to_number(mark_hurd_pay_string)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-27-93d94b960822> in <cell line: 1>()
----> 1 convert_pay_string_to_number(mark_hurd_pay_string)

NameError: name 'mark_hurd_pay_string' is not defined
```

```
# We can also compute Safra Catz's pay in the same way:
convert_pay_string_to_number(raw_compensation.where("Name", are.containing("Safra")).column("Total Pay").item(0))
```

```
53240000.0
```

So, what have we gained by defining the `convert_pay_string_to_number` function? Well, without it, we'd have to copy the code `10**6 * float(some_pay_string.strip("$"))` each time we wanted to convert a pay string. Now we just call a function whose name says exactly what it's doing.

## 3. `apply` ing functions

Defining a function is a lot like giving a name to a value with `=`. In fact, a function is a value just like the number 1 or the text "data"!

For example, we can make a new name for the built-in function `max` if we want:

```
our_name_for_max = max
our_name_for_max(2, 6)
```

```
6
```

The old name for `max` is still around:

```
max(2, 6)
```

```
6
```

Try just writing `max` or `our_name_for_max` (or the name of any other function) in a cell, and run that cell. Python will print out a (very brief) description of the function.

```
max
```

```
<function max>
```

Now try writing `?max` or `?our_name_for_max` (or the name of any other function) in a cell, and run that cell. A information box should show up at the bottom of your screen a longer description of the function

*Note: You can also press Shift+Tab after clicking on a name to see similar information!*

```
?our_name_for_max
```

Let's look at what happens when we set `max` to a non-function value. You'll notice that a TypeError will occur when you try calling `max`. Things like integers and strings are not callable. Look out for any functions that might have been renamed when you encounter this type of error

```
max = 6
max(2, 6)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-e18bead4130c> in <cell line: 2>()
      1 max = 6
----> 2 max(2, 6)

TypeError: 'int' object is not callable
```

```
# This cell resets max to the built-in function. Just run this cell, don't change its contents
import builtins
max = builtins.max
```

Why is this useful? Since functions are just values, it's possible to pass them as arguments to other functions. Here's a simple but not-so-practical example: we can make an array of functions.

```
make_array(max, np.average, are.equal_to)
```

```
array([<built-in function max>, <function average at 0x7c5c1b19fb30>,
       <function are.equal_to at 0x7c5bd4427010>], dtype=object)
```

**Question 3.1.** Make an array containing any 3 other functions you've seen. Call it `some_functions`.

```
some_functions = make_array(np.average, len, min)
some_functions
```

```
array([<function average at 0x7c5c1b19fb30>, <built-in function len>,
       <built-in function min>], dtype=object)
```

Working with functions as values can lead to some funny-looking code. For example, see if you can figure out why the following code works. Check your explanation with a neighbor.

```
make_array(max, np.average, are.equal_to).item(0)(4, -2, 7)
```

```
7
```

A more useful example of passing functions to other functions as arguments is the table method `apply`.

`apply` calls a function many times, once on *each* element in a column of a table. It produces an *array* of the results. Here we use `apply` to convert every CEO's pay to a number, using the function you defined:

```
raw_compensation.apply(convert_pay_string_to_number, "Total Pay")
```

```
array([  5.32500000e+07,   5.32400000e+07,   4.49100000e+07,
         3.59800000e+07,   3.33600000e+07,   2.48400000e+07,
         2.20400000e+07,   1.98600000e+07,   1.96200000e+07,
         1.93200000e+07,   1.87600000e+07,   1.86100000e+07,
         1.83600000e+07,   1.80900000e+07,   1.71000000e+07,
         1.66300000e+07,   1.63300000e+07,   1.61400000e+07,
         1.61000000e+07,   1.60200000e+07,   1.51000000e+07,
         1.49800000e+07,   1.46300000e+07,   1.45100000e+07,
         1.44400000e+07,   1.43600000e+07,   1.43100000e+07,
         1.40900000e+07,   1.40000000e+07,   1.36700000e+07,
         1.23400000e+07,   1.22000000e+07,   1.21800000e+07,
         1.21300000e+07,   1.20500000e+07,   1.18400000e+07,
         1.17100000e+07,   1.16300000e+07,   1.11600000e+07,
         1.11100000e+07,   1.11100000e+07,   1.07300000e+07,
         1.05000000e+07,   1.04300000e+07,   1.03700000e+07,
         1.02800000e+07,   1.02700000e+07,   1.01800000e+07,
         1.01600000e+07,   9.97000000e+06,   9.96000000e+06,
         9.86000000e+06,   9.74000000e+06,   9.42000000e+06,
         9.39000000e+06,   9.22000000e+06,   9.06000000e+06,
         9.03000000e+06,   8.86000000e+06,   8.76000000e+06,
         8.57000000e+06,   8.38000000e+06,   8.36000000e+06,
         8.35000000e+06,   8.23000000e+06,   7.86000000e+06,
         7.70000000e+06,   7.58000000e+06,   7.51000000e+06,
         7.23000000e+06,   7.21000000e+06,   7.12000000e+06,
         6.88000000e+06,   6.77000000e+06,   6.64000000e+06,
         6.56000000e+06,   6.14000000e+06,   5.92000000e+06,
         5.90000000e+06,   5.89000000e+06,   5.73000000e+06,
         5.42000000e+06,   5.04000000e+06,   4.92000000e+06,
         4.92000000e+06,   4.47000000e+06,   4.25000000e+06,
         4.08000000e+06,   3.93000000e+06,   3.72000000e+06,
         2.88000000e+06,   2.83000000e+06,   2.82000000e+06,
         2.45000000e+06,   1.79000000e+06,   1.68000000e+06,
         1.53000000e+06,   9.40000000e+05,   8.10000000e+05,
         7.00000000e+04,   4.00000000e+04,   0.00000000e+00])
```

Note that we didn't write `raw_compensation.apply(convert_pay_string_to_number(), “Total Pay”)` or `raw_compensation.apply(convert_pay_string_to_number(“Total Pay”))`. We just passed the name of the function, with no parentheses, to `apply`, because all we want to do is let `apply` know the name of the function we'd like to use and the name of the column we'd like to use it on. `apply` will then call the function `convert_pay_string_to_number` on each value in the column for us!

**Question 3.2.** Using `apply`, make a table that's a copy of `raw_compensation` with one additional column called `Total Pay ($)`. That column should contain the result of applying `convert_pay_string_to_number` to the `Total Pay` column (as we did above). Call the new table `compensation`.

```
compensation = raw_compensation.with_column(
    "Total Pay ($)",
    raw_compensation.apply(convert_pay_string_to_number, "Total Pay")
    )
compensation
```

| Rank | Name | Company (Headquarters) | Total Pay | % Change | Cash Pay | Equity Pay | Other Pay | Ratio of CEO pay to average industry worker pay | Total Pay ($) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Mark V. Hurd* | Oracle (Redwood City) | $53.25 | (No previous year) | $0.95 | $52.27 | $0.02 | 362 | 5.325e+07 |
| 2 | Safra A. Catz* | Oracle (Redwood City) | $53.24 | (No previous year) | $0.95 | $52.27 | $0.02 | 362 | 5.324e+07 |
| 3 | Robert A. Iger | Walt Disney (Burbank) | $44.91 | -3% | $24.89 | $17.28 | $2.74 | 477 | 4.491e+07 |
| 4 | Marissa A. Mayer | Yahoo! (Sunnyvale) | $35.98 | -15% | $1.00 | $34.43 | $0.55 | 342 | 3.598e+07 |
| 5 | Marc Benioff | salesforce.com (San Francisco) | $33.36 | -16% | $4.65 | $27.26 | $1.45 | 338 | 3.336e+07 |
| 6 | John H. Hammergren | McKesson (San Francisco) | $24.84 | -4% | $12.10 | $12.37 | $0.37 | 222 | 2.484e+07 |
| 7 | John S. Watson | Chevron (San Ramon) | $22.04 | -15% | $4.31 | $14.68 | $3.05 | 183 | 2.204e+07 |
| 8 | Jeffrey Weiner | LinkedIn (Mountain View) | $19.86 | 27% | $2.47 | $17.26 | $0.13 | 182 | 1.986e+07 |

Now that we have all the pays as numbers, we can learn more about them through computation.

**Question 3.3.** Compute the average total pay of the CEOs in the dataset.

```
average_total_pay = np.average(compensation.column(9))
average_total_pay
```

```
11445294.117647059
```

**Question 3.4.** Companies pay executives in a variety of ways: in cash, by granting stock or other equity in the company, or with ancillary benefits (like private jets). Compute the proportion of each CEO's pay that was cash. (Your answer should be an array of numbers, one for each CEO in the dataset.)

*Note:* When you answer this question, you'll encounter a red box appearing below your code cell that says something like `RuntimeWarning: invalid value encountered in true_divide`. Don't worry too much about the message. Warnings are raised by Python when it encounters an unusual condition in your code, but the condition is not severe enough to warrant throwing an error.

The warning below is Python's cryptic way of telling you that you're dividing a number by zero. If you extract the values in `Total Pay ($)` as an array, you'll see that the last element is 0.

```
cash_proportion = to_percentage(compensation.apply(convert_pay_string_to_number, "Cash Pay") / compensation.column("Total Pay ($)"))
cash_proportion
```

```
<ipython-input-49-de69fd97024d>:1: RuntimeWarning: invalid value encountered in divide
  cash_proportion = to_percentage(compensation.apply(convert_pay_string_to_number, "Cash Pay") / compensation.column("Total Pay ($)"))
array([  1.78403756,   1.78437265,  55.42195502,   2.77932185,
        13.93884892,  48.71175523,  19.5553539 ,  12.43705942,
        25.99388379,  35.19668737,  30.75692964,  22.13863514,
        13.12636166,  17.08126036,  23.0994152 ,   6.7348166 ,
        13.04347826,  28.00495663,  33.22981366,  15.35580524,
        29.33774834,  21.82910547,  31.10047847,  25.08614748,
        22.99168975,  16.99164345,  31.79594689,  26.18878637,
        28.35714286,  15.65471836,  38.16855754,  28.93442623,
        20.36124795,  47.65045342,  45.64315353,  36.40202703,
        21.77625961,  24.76354256,  42.56272401,  26.10261026,
        18.36183618,  14.44547996,  33.33333333,  10.83413231,
        20.92574735,  97.27626459,  22.97955209,  22.78978389,
        37.89370079,  25.17552658,  73.89558233,  37.01825558,
        24.12731006,  21.33757962,  20.55378062,  23.31887202,
        33.66445916,  38.75968992,  56.09480813,  11.75799087,
        35.23920653,  24.46300716,  25.        ,  23.71257485,
        43.37788578,  31.42493639,  46.36363636,  32.58575198,
        24.76697736,  98.75518672,  27.18446602,  96.20786517,
        31.83139535,  81.97932053,  23.79518072,  17.5304878 ,
        21.17263844,  37.16216216,  27.28813559,  26.99490662,
        55.14834206,  35.97785978,   0.        ,  47.15447154,
        47.15447154,  29.75391499,  16.23529412,  48.52941176,
        46.81933842,  32.52688172,  98.95833333,  61.13074205,
        67.0212766 ,  75.51020408,  50.83798883,  98.80952381,
        98.03921569,  98.93617021,  87.65432099,   0.        ,
       100.        ,          nan])
```

Check out the `% Change` column in `compensation`. It shows the percentage increase in the CEO's pay from the previous year. For CEOs with no previous year on record, it instead says "(No previous year)". The values in this column are *strings*, not numbers, so like the `Total Pay` column, it's not usable without a bit of extra work.

Given your current pay and the percentage increase from the previous year, you can compute your previous year's pay. For example, if your pay is $\$120$ this year, and that's an increase of 50% from the previous year, then your previous year's pay was $\frac{\$120}{1+\frac{50}{100}}$, or $80.

**Question 3.5.** Create a new table called `with_previous_compensation`. It should be a copy of `compensation`, but with the "(No previous year)" CEOs filtered out, and with an extra column called `2014 Total Pay ($)`. That column should have each CEO's pay in 2014.

*Hint 1:* You can print out your results after each step to make sure you're on the right track.

*Hint 2:* We've provided a structure that you can use to get to the answer. However, if it's confusing, feel free to delete the current structure and approach the problem your own way!

```
# Definition to turn percent to number
def percent_string_to_num(percent_string):
    """Converts a percentage string to a number."""
    return float(percent_string.strip("%"))/100

# Compensation table where there is a previous year
having_previous_year = compensation.where("% Change", are.not_containing("(No previous year)"))

# Get the percent changes as numbers instead of strings
# We're still working off the table having_previous_year
percent_changes = having_previous_year.apply(percent_string_to_num, "% Change")

# Calculate the previous year's pay
# We're still working off the table having_previous_year
previous_pay = having_previous_year.column(9) / (percent_changes +1)


# Put the previous pay column into the having_previous_year table
with_previous_compensation = having_previous_year.with_column("previous pay", previous_pay)

with_previous_compensation
```

| Rank | Name | Company (Headquarters) | Total Pay | % Change | Cash Pay | Equity Pay | Other Pay | Ratio of CEO pay to average industry worker pay | Total Pay ($) | previous pay |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | Robert A. Iger | Walt Disney (Burbank) | $44.91 | -3% | $24.89 | $17.28 | $2.74 | 477 | 4.491e+07 | 4.6299e+07 |
| 4 | Marissa A. Mayer | Yahoo! (Sunnyvale) | $35.98 | -15% | $1.00 | $34.43 | $0.55 | 342 | 3.598e+07 | 4.23294e+07 |
| 5 | Marc Benioff | salesforce.com (San Francisco) | $33.36 | -16% | $4.65 | $27.26 | $1.45 | 338 | 3.336e+07 | 3.97143e+07 |
| 6 | John H. Hammergren | McKesson (San Francisco) | $24.84 | -4% | $12.10 | $12.37 | $0.37 | 222 | 2.484e+07 | 2.5875e+07 |
| 7 | John S. Watson | Chevron (San Ramon) | $22.04 | -15% | $4.31 | $14.68 | $3.05 | 183 | 2.204e+07 | 2.59294e+07 |
| 8 | Jeffrey Weiner | LinkedIn (Mountain View) | $19.86 | 27% | $2.47 | $17.26 | $0.13 | 182 | 1.986e+07 | 1.56378e+07 |
| 9 | John T. Chambers** | Cisco Systems (San Jose) | $19.62 | 19% | $5.10 | $14.51 | $0.01 | 170 | 1.962e+07 | 1.64874e+07 |

**Question 3.6.** What was the average pay of these CEOs in 2014?

```
average_pay_2014 = np.average(with_previous_compensation.column(10))
average_pay_2014
```

```
11649176.115603436
```

**Why is `apply` useful?**

For operations like arithmetic, or the functions in the NumPy library, you don't need to use `apply`, because they automatically work on each element of an array. But there are many things that don't. The string manipulation we did in today's lab is one example. Since you can write any code you want in a function, `apply` gives you total control over how you operate on data.

## 4. Histograms

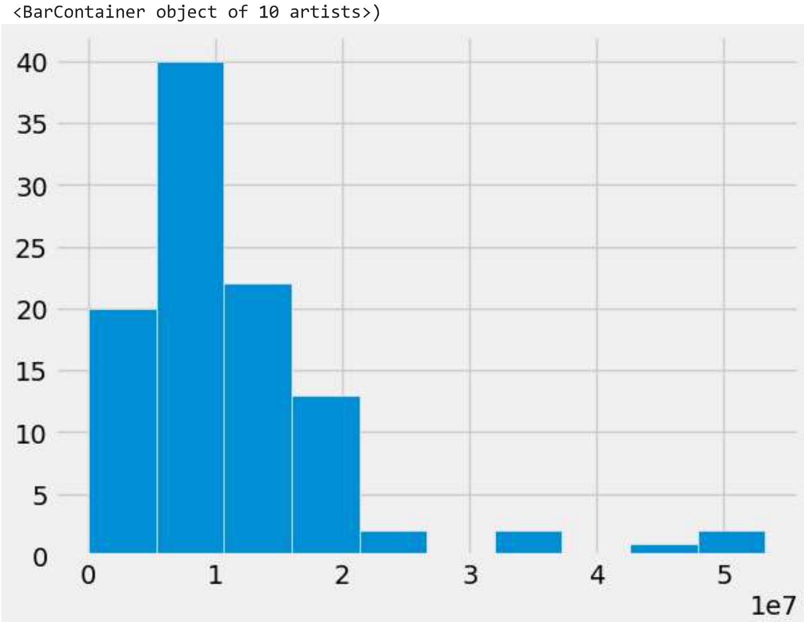
Earlier, we computed the average pay among the CEOs in our 102-CEO dataset. The average doesn't tell us everything about the amounts CEOs are paid, though. Maybe just a few CEOs make the bulk of the money, even among these 102.

We can use a *histogram* method to display the *distribution* of a set of numbers. The table method `hist` takes a single argument, the name of a column of numbers. It produces a histogram of the numbers in that column.

**Question 4.1.** Make a histogram of the total pay of the CEOs in `compensation`. Check with your neighbor or a staff member to make sure you have the right plot.

```
plt.hist(compensation.column(9))
```

```
(array([ 20.,  40.,  22.,  13.,   2.,   0.,   2.,   0.,   1.,   2.]),
 array([        0.,   5325000.,  10650000.,  15975000.,  21300000.,
        26625000.,  31950000.,  37275000.,  42600000.,  47925000.,
        53250000.]),
 <BarContainer object of 10 artists>)
```




**Question 4.2.** How many CEOs made more than $30 million in total pay? Find the value using code, then check that the value you found is consistent with what you see in the histogram.

*Hint:* Use the table method `where` and the property `num_rows`.

```
num_ceos_more_than_30_million_2 = len(compensation.where("Total Pay ($)", are.above(30000000)))
num_ceos_more_than_30_million_2
```

```
10
```

When you have completed your assignment,

1. Change the file name by by adding "Submitted" at the end. For example, if I were a student submitting lab01, the file would initially be called "lab01Liz." After completing it, I would change the name to "lab01LizSubmitted."
2. Print your notebook as a pdf. Go to File -> Print -> save as pdf. Then upload the pdf to the corresponding assignment (lab/hw) on Canvas.

This assignment is due Thursday, February 15th at 11:59pm. No late assignments will be accepted.