lab03

January 12, 2024

[21]:
```python
# Initialize Otter
import otter
grader = otter.Notebook("lab03.ipynb")
```

1 Lab 3: Data Types and Arrays

Welcome to Lab 3!

So far, we've used Python to manipulate numbers and work with tables. But we need to discuss data types to deepen our understanding of how to work with data in Python.

In this lab, you'll first see how to represent and manipulate another fundamental type of data: text. A piece of text is called a *string* in Python. You'll also see how to work with *arrays* of data. An array could contain all the numbers between 0 and 100 or all the words in the chapter of a book. Lastly, you'll create tables and practice analyzing them with your knowledge of table operations. The Python Reference has information that will be useful for this lab.

Submission: Once you're finished, run all cells besides the last one, select File > Save Notebook, and then execute the final cell. Then submit the downloaded zip file, that includes your notebook, according to your instructor's directions.

Set up the tests and imports by running the cell below.

[30]:
```python
# Just run this cell
import numpy as np
import math
from datascience import *
import d8error
```

2 1. Text

Programming doesn't just concern numbers. Text is one of the most common data types used in programs.

Text is represented by a **string value** in Python. The word "string" is a programming term for a sequence of characters. A string might contain a single character, a word, a sentence, or a whole book.

To distinguish text data from actual code, we demarcate strings by putting quotation marks around them. Single quotes (`'`) and double quotes (`"`) are both valid, but the types of opening and closing quotation marks must match. The contents can be any sequence of characters, including numbers and symbols.

We've seen strings before in `print` statements. Below, two different strings are passed as arguments to the `print` function.

[29]:

```
print("I <3", 'Data Science')
```

```
I <3 Data Science
```

Just as names can be given to numbers, names can be given to string values. The names and strings aren't required to be similar in any way. Any name can be assigned to any string.

[28]:

```
one = 'two'
plus = '*'
print(one, plus, one)
```

```
two * two
```

Question 1.1. Yuri Gagarin was the first person to travel through outer space. When he emerged from his capsule upon landing on Earth, he reportedly had the following conversation with a woman and girl who saw the landing:

```
The woman asked: "Can it be that you have come from outer space?"
Gagarin replied: "As a matter of fact, I have!"
```

The cell below contains unfinished code. Fill in the `...`s so that it prints out this conversation *exactly* as it appears above.

[27]:

```
woman_asking = 'The woman asked:'
woman_quote = '"Can it be that you have come from outer space?"'
gagarin_reply = 'Gagarin replied:'
gagarin_quote = 'Ellipsis'

print(woman_asking, woman_quote)
print(gagarin_reply, gagarin_quote)
```

```
The woman asked: "Can it be that you have come from outer space?"
Gagarin replied: Ellipsis
```

[26]:

```
grader.check("q11")
```

[26]:

```
q11 results:
    q11 - 1 result:
         Test case passed

    q11 - 2 result:
         Test case failed
        Trying:
            gagarin_quote
        Expecting:
```

```
    '"As a matter of fact, I have!"'
**********************************************************************
Line 1, in q11 1
Failed example:
    gagarin_quote
Expected:
    '"As a matter of fact, I have!"'
Got:
    'Ellipsis'
```

2.1 1.1. String Methods

Strings can be transformed using **methods**. Recall that methods and functions are not the same thing. Here is the textbook section on string methods: 4.2.1 String Methods.

Here's a sketch of how to call methods on a string:

<expression that evaluates to a string>.<method name>(<argument>, <argument>, ...)

One example of a string method is `replace`, which replaces all instances of some part of the original string (or a *substring*) with a new string.

<original string>.replace(<old substring>, <new substring>)

`replace` returns (evaluates to) a new string, leaving the original string unchanged.

Try to predict the output of this example, then run the cell!

[25]:
```
# Replace one letter
bag = 'bag'
print(bag.replace('g', 't'), bag)
```

```
bat bag
```

You can call functions on the results of other functions. For example, `max(abs(-5), abs(3))` evaluates to 5. Similarly, you can call methods on the results of other method or function calls.

You may have already noticed one difference between functions and methods - a function like `max` does not require a `.` before it's called, but a string method like `replace` does. Here's a handy Python reference on the Data 8 website. It's a good idea to refer to this whenever you're unsure of how to call a function or method.

[24]:
```
# Calling replace on the output of another call to replace
'train'.replace('t', 'ing').replace('in', 'de')
```

[24]: `'degrade'`

Here's a picture of how Python evaluates a "chained" method call like that:

Question 1.1.1. Use `replace` to transform the string `'hitchhiker'` into `'matchmaker'`. Assign your result to `new_word`.

[31]:
```python
new_word = 'hitchhiker'.replace('h', 'm').replace('i', 'a').replace('mm', 'hm')
new_word
```

[31]: 'matchmaker'

[ ]:
```python
grader.check("q111")
```

There are many more string methods in Python, but most programmers don't memorize their names or how to use them. In the "real world," people usually just search the internet for documentation and examples. A complete list of string methods appears in the Python language documentation. Stack Overflow has a huge database of answered questions that often demonstrate how to use these methods to achieve various ends. Material covered in these resources that haven't already been introduced in this class will be out of scope.

2.2 1.2. Converting to and from Strings

Strings and numbers are different *types* of values, even when a string contains the digits of a number. For example, evaluating the following cell causes an error because an integer cannot be added to a string.

[32]:
```python
8 + "8"
```

2.3 Uh-o it seems we have an error!

It seems we have a TypeError. TypeErrors are usually because of:

1. Using the wrong type of input, i.e `np.arange("3")`

The Data 8 Reference might be helpful to look over for examples and usage: Data 8 Reference

If you are having more trouble please feel free to consult a staff member
or see the error message below

```
Accordion(children=(VBox(children=(VBox(children=(Label(value='Was the message␣
↪you saw useful?'), Dropdown(opt…
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[32], line 1
----> 1 8 + "8"

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

However, there are built-in functions to convert numbers to strings and strings to numbers. Some of these built-in functions have restrictions on the type of argument they take:

Function	Description
`int`	Converts a string of digits or a float to an integer ("int") value

Function	Description
`float`	Converts a string of digits (perhaps with a decimal point) or an int to a decimal ("float") value
`str`	Converts any value to a string

Try to predict what data type and value `example` evaluates to, then run the cell.

[33]:
```
example = 8 + int("10") + float("8")

print(example)
print("This example returned a " + str(type(example)) + "!")
```

```
26.0
This example returned a <class 'float'>!
```

Suppose you're writing a program that looks for dates in a text, and you want your program to find the amount of time that elapsed between two years it has identified. It doesn't make sense to subtract two texts, but you can first convert the text containing the years into numbers.

Question 1.2.1. Finish the code below to compute the number of years that elapsed between `one_year` and `another_year`. Don't just write the numbers `1618` and `1648` (or `30`); use a conversion function to turn the given text data into numbers.

[34]:
```
# Some text data:
one_year = "1618"
another_year = "1648"

# Complete the next line.  Note that we can't just write:
#   another_year - one_year
# If you don't see why, try seeing what happens when you
# write that here.
difference = int(another_year) - int(one_year)
difference
```

[34]:
```
30
```

[ ]:
```
grader.check("q121")
```

2.4 1.3. Passing strings to functions

String values, like numbers, can be arguments to functions and can be returned by functions.

The function `len` (derived from the word "length") takes a single string as its argument and returns the number of characters (including spaces) in the string.

Note that it doesn't count *words*. `len("one small step for man")` evaluates to 22 characters, not 5 words.

Question 1.3.1. Use `len` to find the number of characters in the long string in the next cell. Characters include things like spaces and punctuation. Assign `sentence_length` to that number.

(The string is the first sentence of the English translation of the French Declaration of the Rights of Man.)

[35]:
```
a_very_long_sentence = "The representatives of the French people, organized as␣
 ↪a National Assembly, believing that the ignorance, neglect, or contempt of␣
 ↪the rights of man are the sole cause of public calamities and of the␣
 ↪corruption of governments, have determined to set forth in a solemn␣
 ↪declaration the natural, unalienable, and sacred rights of man, in order␣
 ↪that this declaration, being constantly before all the members of the Social␣
 ↪body, shall remind them continually of their rights and duties; in order␣
 ↪that the acts of the legislative power, as well as those of the executive␣
 ↪power, may be compared at any moment with the objects and purposes of all␣
 ↪political institutions and may thus be more respected, and, lastly, in order␣
 ↪that the grievances of the citizens, based hereafter upon simple and␣
 ↪incontestable principles, shall tend to the maintenance of the constitution␣
 ↪and redound to the happiness of all."
sentence_length = len(a_very_long_sentence)
sentence_length
```

[35]: 896

[ ]:
```
grader.check("q131")
```

3 2. Arrays

Computers are most useful when you can use a small amount of code to *do the same action* to *many different things*.

For example, in the time it takes you to calculate the 18% tip on a restaurant bill, a laptop can calculate 18% tips for every restaurant bill paid by every human on Earth that day. (That's if you're pretty fast at doing arithmetic in your head!)

Arrays are how we put many values in one place so that we can operate on them as a group. For example, if `billions_of_numbers` is an array of numbers, the expression

`.18 * billions_of_numbers`

gives a new array of numbers that contains the result of multiplying **each number** in `billions_of_numbers` by .18. Arrays are not limited to numbers; we can also put all the words in a book into an array of strings.

Concretely, an array is a **collection of values of the same type**.

3.1 2.1. Making arrays

First, let's learn how to manually input values into an array. This typically isn't how programs work. Normally, we create arrays by loading them from an external source, like a data file.

To create an array by hand, call the function `make_array`. Each argument you pass to `make_array` will be in the array it returns. Run this cell to see an example:

[36]:
```
make_array(0.125, 4.75, -1.3)
```

[36]:
```
array([ 0.125,  4.75 , -1.3  ])
```

Each value in an array (in the above case, the numbers 0.125, 4.75, and -1.3) is called an *element* of that array.

Arrays themselves are also values, just like numbers and strings. That means you can assign them to names or use them as arguments to functions. For example, `len(<some_array>)` returns the number of elements in `some_array`.

Question 2.1.1. Make an array containing the numbers 0, 1, -1, and π, in that order. Name it `interesting_numbers`.

Hint: How did you get the value π in lab 2? You can refer to it the same way here. The `math` module has been imported at the top of this notebook.

[37]:
```
interesting_numbers = make_array(0, 1, -1, math.pi)
interesting_numbers
```

[37]:
```
array([ 0.        ,  1.        , -1.        ,  3.14159265])
```

[ ]:
```
grader.check("q211")
```

Question 2.1.2. Make an array containing the five strings `"Hello"`, `","`, `" "`, `"world"`, and `"!"`. (The third one is a single space inside quotes.) Name it `hello_world_components`.

Note: If you evaluate `hello_world_components`, you'll notice some extra information in addition to its contents: `dtype='<U5'`. That's just NumPy's extremely cryptic way of saying that the data types in the array are strings.

[38]:
```
hello_world_components = make_array("Hello", ",", " ", "world", "!")
hello_world_components
```

[38]:
```
array(['Hello', ',', ' ', 'world', '!'],
      dtype='<U5')
```

[ ]:
```
grader.check("q212")
```

3.1.1 `np.arange`

Arrays are provided by a package called NumPy (pronounced "NUM-pie"). The package is called `numpy`, but it's standard to rename it `np` for brevity. You can do that with:

```
import numpy as np
```

Very often in data science, we want to work with many numbers that are evenly spaced within some range. NumPy provides a special function for this called `arange`. The line of code `np.arange(start, stop, step)` evaluates to an array with all the numbers starting at `start` and counting up by `step`, stopping **before** `stop` is reached.

Run the following cells to see some examples!

[39]:
```
# This array starts at 1 and counts up by 2
# and then stops before 6
np.arange(1, 6, 2)
```

[39]:
```
array([1, 3, 5])
```

[40]:
```
# This array doesn't contain 9
# because np.arange stops *before* the stop value is reached
np.arange(4, 9, 1)
```

[40]:
```
array([4, 5, 6, 7, 8])
```

Question 2.1.3. Import `numpy` as `np` and then use `np.arange` to create an array with the multiples of 99 from 0 up to (**and including**) 9999. (So its elements are 0, 99, 198, 297, etc.)

[41]:
```
import numpy as np
multiples_of_99 = np.arange(0, 10000, 99)
multiples_of_99
```

[41]:
```
array([   0,   99,  198,  297,  396,  495,  594,  693,  792,  891,  990,
       1089, 1188, 1287, 1386, 1485, 1584, 1683, 1782, 1881, 1980, 2079,
       2178, 2277, 2376, 2475, 2574, 2673, 2772, 2871, 2970, 3069, 3168,
       3267, 3366, 3465, 3564, 3663, 3762, 3861, 3960, 4059, 4158, 4257,
       4356, 4455, 4554, 4653, 4752, 4851, 4950, 5049, 5148, 5247, 5346,
       5445, 5544, 5643, 5742, 5841, 5940, 6039, 6138, 6237, 6336, 6435,
       6534, 6633, 6732, 6831, 6930, 7029, 7128, 7227, 7326, 7425, 7524,
       7623, 7722, 7821, 7920, 8019, 8118, 8217, 8316, 8415, 8514, 8613,
       8712, 8811, 8910, 9009, 9108, 9207, 9306, 9405, 9504, 9603, 9702,
       9801, 9900, 9999])
```

[ ]:
```
grader.check("q213")
```

3.2 2.2. Working with single elements of arrays ("indexing")

Let's work with a more interesting dataset. The next cell creates an array called `population_amounts` that includes estimated world populations of every year from **1950** to roughly the present. (The estimates come from the US Census Bureau website.)

Rather than type in the data manually, we've loaded them from a file on your computer called `world_population.csv`. You'll learn how to do that later in this lab!

[42]:
```
population_amounts = Table.read_table("world_population.csv").
 ↪column("Population")
population_amounts
```

[42]:
```
array([2557628654, 2594939877, 2636772306, 2682053389, 2730228104,
       2782098943, 2835299673, 2891349717, 2948137248, 3000716593,
       3043001508, 3083966929, 3140093217, 3209827882, 3281201306,
       3350425793, 3420677923, 3490333715, 3562313822, 3637159050,
       3712697742, 3790326948, 3866568653, 3942096442, 4016608813,
       4089083233, 4160185010, 4232084578, 4304105753, 4379013942,
       4451362735, 4534410125, 4614566561, 4695736743, 4774569391,
       4856462699, 4940571232, 5027200492, 5114557167, 5201440110,
       5288955934, 5371585922, 5456136278, 5538268316, 5618682132,
       5699202985, 5779440593, 5857972543, 5935213248, 6012074922,
       6088571383, 6165219247, 6242016348, 6318590956, 6395699509,
       6473044732, 6551263534, 6629913759, 6709049780, 6788214394,
       6866332358, 6944055583, 7022349283, 7101027895, 7178722893,
       7256490011])
```

Here's how we get the first element of `population_amounts`, which is the world population in the first year in the dataset, 1950.

[43]:
```
population_amounts.item(0)
```

[43]:
```
2557628654
```

The value of that expression is the number 2,557,628,654 (around 2.5 billion), because that's the first thing in the array `population_amounts`.

Notice that we wrote `.item(0)`, not `.item(1)`, to get the first element. This is a weird convention in computer science. **0 is called the *index* of the first item.** It's the number of elements that appear *before* that item. So 3 is the index of the 4th item.

Here are some more examples. In the examples, we've given names to the things we get out of `population_amounts`. Read and run each cell.

[44]:
```
# The 13th element in the array is the population
# in 1962 (which is 1950 + 12).
population_1962 = population_amounts.item(12)
population_1962
```

[44]:
```
3140093217
```

[45]:
```
# The 66th element is the population in 2015.
population_2015 = population_amounts.item(65)
population_2015
```

[45]:
```
7256490011
```

[46]:
```
# The array has only 66 elements, so this doesn't work.
# (There's no element with 66 other elements before it.)
population_2016 = population_amounts.item(66)
population_2016
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[46], line 3
      1 # The array has only 66 elements, so this doesn't work.
      2 # (There's no element with 66 other elements before it.)
----> 3 population_2016 = population_amounts.item(66)
      4 population_2016

IndexError: index 66 is out of bounds for axis 0 with size 66
```

Since `make_array` returns an array, we can call `.item(3)` on its output to get its 4th element, just like we "chained" together calls to the method `replace` earlier.

[ ]:
```
make_array(-1, -3, 4, -2).item(3)
```

Question 2.2.1. Set `population_1973` to the world population in 1973, by getting the appropriate element from `population_amounts` using `item`.

[47]:
```
population_1973 = population_amounts.item(23)
population_1973
```

[47]: 3942096442

[ ]:
```
grader.check("q221")
```

3.3 2.3. Doing something to every element of an array

Arrays are primarily useful for doing the same operation many times, so we don't often have to use `.item` and work with single elements.

Rounding Here is one simple question we might ask about world population:

> How big was the population in each year, rounded to the nearest million?

Rounding is often used with large numbers when we don't need as much precision in our numbers. One example of this is when we present data in tables and visualizations.

We could try to answer our question using the `round` function that is built into Python and the `item` method you just saw.

Note: the `round` function takes in two arguments: the number to be rounded, and the number of decimal places to round to. The second argument can be thought of as how many steps right or left you move from the decimal point. Negative numbers tell us to move left, and positive numbers tell us to move right. So, if we have `round(1234.5, -2)`, it means that we should move two places left, and then make all numbers to the right of this place zeroes. This would output the number 1200.0. On the other hand, if we have `round(6.789, 1)`, we should move one place right, and then make all numbers to the right of this place zeroes. This would output the number 6.8.

[48]:
```
population_1950_magnitude = round(population_amounts.item(0), -6)
population_1951_magnitude = round(population_amounts.item(1), -6)
population_1952_magnitude = round(population_amounts.item(2), -6)
population_1953_magnitude = round(population_amounts.item(3), -6)
```

But this is tedious and doesn't really take advantage of the fact that we are using a computer.

Instead, NumPy provides its own version of `round` that rounds each element of an array. It takes in two arguments: a single array of numbers, and the number of decimal places to round to. It returns an array of the same length, where the first element of the result is the first element of the argument rounded, and so on.

Question 2.3.1. Use `np.round` to compute the world population in every year, rounded to the nearest million (6 zeroes). Give the result (an array of 66 numbers) the name `population_rounded`. Your code should be very short.

[49]:
```
population_rounded = np.round(population_amounts, -6)
population_rounded
```

[49]:
```
array([2558000000, 2595000000, 2637000000, 2682000000, 2730000000,
       2782000000, 2835000000, 2891000000, 2948000000, 3001000000,
       3043000000, 3084000000, 3140000000, 3210000000, 3281000000,
       3350000000, 3421000000, 3490000000, 3562000000, 3637000000,
       3713000000, 3790000000, 3867000000, 3942000000, 4017000000,
       4089000000, 4160000000, 4232000000, 4304000000, 4379000000,
       4451000000, 4534000000, 4615000000, 4696000000, 4775000000,
       4856000000, 4941000000, 5027000000, 5115000000, 5201000000,
       5289000000, 5372000000, 5456000000, 5538000000, 5619000000,
       5699000000, 5779000000, 5858000000, 5935000000, 6012000000,
       6089000000, 6165000000, 6242000000, 6319000000, 6396000000,
       6473000000, 6551000000, 6630000000, 6709000000, 6788000000,
       6866000000, 6944000000, 7022000000, 7101000000, 7179000000,
       7256000000])
```

[ ]:
```
grader.check("q231")
```

What you just did is called **elementwise** application of `np.round`, since `np.round` operates separately on each element of the array that it's called on. Here's a picture of what's going on:

The textbook's section on arrays has a useful list of NumPy functions that are designed to work elementwise, like `np.round`.

Arithmetic Arithmetic also works elementwise on arrays, meaning that if you perform an arithmetic operation (like subtraction, division, etc) on an array, Python will do the operation to every element of the array individually and return an array of all of the results. For example, you can divide all the population numbers by 1 billion to get numbers in billions:

[50]:
```
population_in_billions = population_amounts / 1000000000
population_in_billions
```

```
[50]: array([ 2.55762865,  2.59493988,  2.63677231,  2.68205339,  2.7302281 ,
         2.78209894,  2.83529967,  2.89134972,  2.94813725,  3.00071659,
         3.04300151,  3.08396693,  3.14009322,  3.20982788,  3.28120131,
         3.35042579,  3.42067792,  3.49033371,  3.56231382,  3.63715905,
         3.71269774,  3.79032695,  3.86656865,  3.94209644,  4.01660881,
         4.08908323,  4.16018501,  4.23208458,  4.30410575,  4.37901394,
         4.45136274,  4.53441012,  4.61456656,  4.69573674,  4.77456939,
         4.8564627 ,  4.94057123,  5.02720049,  5.11455717,  5.20144011,
         5.28895593,  5.37158592,  5.45613628,  5.53826832,  5.61868213,
         5.69920299,  5.77944059,  5.85797254,  5.93521325,  6.01207492,
         6.08857138,  6.16521925,  6.24201635,  6.31859096,  6.39569951,
         6.47304473,  6.55126353,  6.62991376,  6.70904978,  6.78821439,
         6.86633236,  6.94405558,  7.02234928,  7.10102789,  7.17872289,
         7.25649001])
```

You can do the same with addition, subtraction, multiplication, and exponentiation (`**`).

3.4 3. Creating Tables

An array is useful for describing a single attribute of each element in a collection. For example, let's say our collection is all US States. Then an array could describe the land area of each state.

Tables extend this idea by containing multiple columns stored and represented as arrays, each one describing a different attribute for every element of a collection. In this way, tables allow us to not only store data about many entities but to also contain several kinds of data about each entity.

For example, in the cell below we have two arrays. The first one, `population_amounts`, was defined above in section 2.2 and contains the world population in each year (estimated by the US Census Bureau). The second array, `years`, contains the years themselves. These elements are in order, so the year and the world population for that year have the same index in their corresponding arrays.

```
[51]: # Just run this cell

years = np.arange(1950, 2015+1)
print("Population column:", population_amounts)
print("Years column:", years)
```

```
Population column: [2557628654 2594939877 2636772306 2682053389 2730228104
2782098943
 2835299673 2891349717 2948137248 3000716593 3043001508 3083966929
 3140093217 3209827882 3281201306 3350425793 3420677923 3490333715
 3562313822 3637159050 3712697742 3790326948 3866568653 3942096442
 4016608813 4089083233 4160185010 4232084578 4304105753 4379013942
 4451362735 4534410125 4614566561 4695736743 4774569391 4856462699
 4940571232 5027200492 5114557167 5201440110 5288955934 5371585922
 5456136278 5538268316 5618682132 5699202985 5779440593 5857972543
 5935213248 6012074922 6088571383 6165219247 6242016348 6318590956
 6395699509 6473044732 6551263534 6629913759 6709049780 6788214394
 6866332358 6944055583 7022349283 7101027895 7178722893 7256490011]
```

```
Years column: [1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962
1963 1964
 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979
 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994
 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009
 2010 2011 2012 2013 2014 2015]
```

Suppose we want to answer this question:

> In which year did the world's population cross 6 billion?

You could technically answer this question just from staring at the arrays, but it's a bit convoluted, since you would have to count the position where the population first crossed 6 billion, then find the corresponding element in the years array. In cases like these, it might be easier to put the data into a *`Table`*, a 2-dimensional type of dataset.

The expression below:

- creates an empty table using the expression `Table()`,
- adds two columns by calling `with_columns` with four arguments,
- assigns the result to the name `population`, and finally
- evaluates `population` so that we can see the table.

The strings `"Year"` and `"Population"` are column labels that we have chosen. The names `population_amounts` and `years` were assigned above to two arrays of the **same length**. The function `with_columns` (you can find the documentation here) takes in alternating strings (to represent column labels) and arrays (representing the data in those columns). The strings and arrays are separated by commas.

[52]:
```python
population = Table().with_columns(
    "Population", population_amounts,
    "Year", years
)
population
```

[52]:
```
Population | Year
2557628654 | 1950
2594939877 | 1951
2636772306 | 1952
2682053389 | 1953
2730228104 | 1954
2782098943 | 1955
2835299673 | 1956
2891349717 | 1957
2948137248 | 1958
3000716593 | 1959
… (56 rows omitted)
```

Now the data is combined into a single table! It's much easier to parse this data. If you need to know what the population was in 1959, for example, you can tell from a single glance.

Question 3.1. In the cell below, we've created 2 arrays. Using the steps above, assign `top_10_movies` to a table that has two columns called "Name" and "Rating", which hold `top_10_movie_names` and `top_10_movie_ratings` respectively.

[53]:

```
top_10_movie_names = make_array(
        'The Shawshank Redemption (1994)',
        'The Godfather (1972)',
        'The Godfather: Part II (1974)',
        'Pulp Fiction (1994)',
        "Schindler's List (1993)",
        'The Lord of the Rings: The Return of the King (2003)',
        '12 Angry Men (1957)',
        'The Dark Knight (2008)',
        'Il buono, il brutto, il cattivo (1966)',
        'The Lord of the Rings: The Fellowship of the Ring (2001)')
top_10_movie_ratings = make_array(9.2, 9.2, 9., 8.9, 8.9, 8.9, 8.9, 8.9, 8.9, 8.
 ↪8)

top_10_movies = Table().with_columns(
"Name", top_10_movie_names,
"Rating", top_10_movie_ratings
)

# We've put this next line here
# so your table will get printed out
# when you run this cell.
top_10_movies
```

[53]:

```
Name                                                     | Rating
The Shawshank Redemption (1994)                          | 9.2
The Godfather (1972)                                     | 9.2
The Godfather: Part II (1974)                            | 9
Pulp Fiction (1994)                                      | 8.9
Schindler's List (1993)                                  | 8.9
The Lord of the Rings: The Return of the King (2003)     | 8.9
12 Angry Men (1957)                                      | 8.9
The Dark Knight (2008)                                   | 8.9
Il buono, il brutto, il cattivo (1966)                   | 8.9
The Lord of the Rings: The Fellowship of the Ring (2001) | 8.8
```

[ ]:

```
grader.check("q31")
```

Loading a table from a file In most cases, we aren't going to go through the trouble of typing in all the data manually. Instead, we load them in from an external source, like a data file. There are many formats for data files, but CSV ("comma-separated values") is the most common.

`Table.read_table(...)` takes one argument (a path to a data file in **string** format) and returns a table.

Question 3.2. `imdb.csv` contains a table of information about the 250 highest-rated movies on IMDb. Load it as a table called `imdb`.

(You may remember working with this table in Lab 2!)

[55]:
```python
imdb =Table.read_table("imdb.csv")
imdb
```

[55]:
```
Votes  | Rating | Title                     | Year | Decade
88355  | 8.4    | M                         | 1931 | 1930
132823 | 8.3    | Singin' in the Rain       | 1952 | 1950
74178  | 8.3    | All About Eve             | 1950 | 1950
635139 | 8.6    | Léon                      | 1994 | 1990
145514 | 8.2    | The Elephant Man          | 1980 | 1980
425461 | 8.3    | Full Metal Jacket         | 1987 | 1980
441174 | 8.1    | Gone Girl                 | 2014 | 2010
850601 | 8.3    | Batman Begins             | 2005 | 2000
37664  | 8.2    | Judgment at Nuremberg     | 1961 | 1960
46987  | 8      | Relatos salvajes          | 2014 | 2010
… (240 rows omitted)
```

[ ]:
```python
grader.check("q32")
```

Where did `imdb.csv` come from? Take a look at this lab's folder. You should see a file called `imdb.csv`.

Open up the `imdb.csv` file in that folder and look at the format. What do you notice? The `.csv` filename ending says that this file is in the CSV (comma-separated value) format.

3.5 4. More Table Operations!

Now that you've worked with arrays, let's add a few more methods to the list of table operations that you saw in Lab 2.

3.5.1 `column`

`column` takes the column name of a table (in string format) as its argument and returns the values in that column as an **array**.

[56]:
```python
# Returns an array of movie names
top_10_movies.column('Name')
```

[56]:
```
array(['The Shawshank Redemption (1994)', 'The Godfather (1972)',
       'The Godfather: Part II (1974)', 'Pulp Fiction (1994)',
       "Schindler's List (1993)",
       'The Lord of the Rings: The Return of the King (2003)',
       '12 Angry Men (1957)', 'The Dark Knight (2008)',
       'Il buono, il brutto, il cattivo (1966)',
       'The Lord of the Rings: The Fellowship of the Ring (2001)'],
      dtype='<U56')
```

3.5.2 take

The table method `take` takes as its argument an array of numbers. Each number should be the index of a row in the table. It returns a **new table** with only those rows.

You'll usually want to use `take` in conjunction with `np.arange` to take the first few rows of a table.

[57]:
```
# Take first 5 movies of top_10_movies
top_10_movies.take(np.arange(0, 5, 1))
```

[57]:
```
Name                             | Rating
The Shawshank Redemption (1994) | 9.2
The Godfather (1972)            | 9.2
The Godfather: Part II (1974)   | 9
Pulp Fiction (1994)             | 8.9
Schindler's List (1993)         | 8.9
```

The next three questions will give you practice with combining the operations you've learned in this lab and the previous one to answer questions about the `population` and `imdb` tables. First, check out the `population` table from section 2.

[58]:
```
# Run this cell to display the population table.
population
```

[58]:
```
Population | Year
2557628654 | 1950
2594939877 | 1951
2636772306 | 1952
2682053389 | 1953
2730228104 | 1954
2782098943 | 1955
2835299673 | 1956
2891349717 | 1957
2948137248 | 1958
3000716593 | 1959
… (56 rows omitted)
```

Question 4.1. Check out the `population` table from section 2 of this lab. Compute the year when the world population first went above 6 billion. Assign the year to `year_population_crossed_6_billion`.

[61]:
```
year_population_crossed_6_billion = Table().with_columns("Population",
population_in_billions, "Years", years)
year_population_crossed_6_billion = year_population_crossed_6_billion.
  ↪where('Population', are.above(6))
year_population_crossed_6_billion = 1999
```

[ ]:
```
grader.check("q41")
```

Question 4.2. Find the average rating for movies released before the year 2000 and the average rating for movies released in the year 2000 or after for the movies in `imdb`.

Hint: Think of the steps you need to do (take the average, find the ratings, find movies released in 20th/21st centuries), and try to put them in an order that makes sense.

[64]:
```
num_before_2000 = imdb.where('Year', are.below(2000)).num_rows
num_after_1999 = imdb.where('Year', are.above(1999)).num_rows
before_2000 = sum(imdb.where('Year', are.below(2000)).column('Rating'))
num_before_2000
after_or_in_2000 = sum(imdb.where('Year', are.above(1999)).column('Rating'))
num_after_1999
# print("Average before 2000 rating:", before_2000)
# print("Average after or in 2000 rating:", after_or_in_2000)
before_2000
```

[64]:
```
1415.6000000000004
```

[65]:
```
grader.check("q42")
```

[65]:
```
q42 results:
    q42 - 1 result:
        Test case failed
        Trying:
            abs(before_2000 - 8.2783625730994146) < 1e-5
        Expecting:
            True
        **********************************************************************
        Line 1, in q42 0
        Failed example:
            abs(before_2000 - 8.2783625730994146) < 1e-5
        Expected:
            True
        Got:
            False

    q42 - 2 result:
        Test case failed
        Trying:
            abs(after_or_in_2000 - 8.2379746835443033) < 1e-5
        Expecting:
            True
        **********************************************************************
        Line 1, in q42 1
        Failed example:
            abs(after_or_in_2000 - 8.2379746835443033) < 1e-5
        Expected:
            True
```

```
Got:
    False
```

Question 4.3. Here's a challenge: Find the number of movies that came out in *even* years.

Hint: The operator `%` computes the remainder when dividing by a number. So `5 % 2` is 1 and `6 % 2` is 0. A number is even if the remainder is 0 when you divide by 2.

Hint 2: `%` can be used on arrays, operating elementwise like `+` or `*`. So `make_array(5, 6, 7) % 2` is `array([1, 0, 1])`.

Hint 3: Add a new column called "Year Remainder" that's the remainder when each movie's release year is divided by 2. To do this, you can use `tbl.with_column(col_name, col_values)`: a table method that takes in the name of the new column and an array of values and returns a copy of the original `tbl` with the new column. Then, use `where` to find rows where that new column is equal to 0. Finally, use `num_rows` to count the number of such rows.

Note: These steps can be chained in one single statement, or broken up across several lines with intermediate names assigned. You're always welcome to break down problems however you wish!

[66]:
```
num_even_year_movies = imdb.with_column('Rem', imdb.column('Year') % 2).
↪where('Rem', 0).num_rows
num_even_year_movies
```

[66]:
```
127
```

[67]:
```
grader.check("q43")
```

[67]:
```
q43 results: All test cases passed!
```

3.6 5. More Array Practice

The following questions will be great practice for learning how to generate arrays in different ways, as well as executing different methods of array arithmetic!

3.6.1 5.1 `np.arange` (cont.)

Temperature readings NOAA (the US National Oceanic and Atmospheric Administration) operates weather stations that measure surface temperatures at different sites around the United States. The hourly readings are publicly available.

Suppose we download all the hourly data from the Oakland, California site for the month of December 2015. To analyze the data, we want to know when each reading was taken, but we find that the data don't include the timestamps of the readings (the time at which each one was taken).

However, we know the first reading was taken at the first instant of December 2015 (midnight on December 1st) and each subsequent reading was taken exactly 1 hour after the last.

Question 5.1.1. Create an array of the *time, in seconds, since the start of the month* at which **each hourly reading** was taken. Name it `collection_times`.

Hint 1: There were 31 days in December, which is equivalent to (31×24) hours or $(31 \times 24 \times 60 \times 60)$ seconds. So your array should have 31×24 elements in it.

Hint 2: The `len` function works on arrays, too! If your `collection_times` isn't passing the tests, check its length and make sure it has 31×24 elements.

[68]:
```
collection_times = np.arange(0, 31*24*60*60, 60*60)
collection_times
```

[68]:
```
array([     0,   3600,   7200,  10800,  14400,  18000,  21600,
        25200,  28800,  32400,  36000,  39600,  43200,  46800,
        50400,  54000,  57600,  61200,  64800,  68400,  72000,
        75600,  79200,  82800,  86400,  90000,  93600,  97200,
       100800, 104400, 108000, 111600, 115200, 118800, 122400,
       126000, 129600, 133200, 136800, 140400, 144000, 147600,
       151200, 154800, 158400, 162000, 165600, 169200, 172800,
       176400, 180000, 183600, 187200, 190800, 194400, 198000,
       201600, 205200, 208800, 212400, 216000, 219600, 223200,
       226800, 230400, 234000, 237600, 241200, 244800, 248400,
       252000, 255600, 259200, 262800, 266400, 270000, 273600,
       277200, 280800, 284400, 288000, 291600, 295200, 298800,
       302400, 306000, 309600, 313200, 316800, 320400, 324000,
       327600, 331200, 334800, 338400, 342000, 345600, 349200,
       352800, 356400, 360000, 363600, 367200, 370800, 374400,
       378000, 381600, 385200, 388800, 392400, 396000, 399600,
       403200, 406800, 410400, 414000, 417600, 421200, 424800,
       428400, 432000, 435600, 439200, 442800, 446400, 450000,
       453600, 457200, 460800, 464400, 468000, 471600, 475200,
       478800, 482400, 486000, 489600, 493200, 496800, 500400,
       504000, 507600, 511200, 514800, 518400, 522000, 525600,
       529200, 532800, 536400, 540000, 543600, 547200, 550800,
       554400, 558000, 561600, 565200, 568800, 572400, 576000,
       579600, 583200, 586800, 590400, 594000, 597600, 601200,
       604800, 608400, 612000, 615600, 619200, 622800, 626400,
       630000, 633600, 637200, 640800, 644400, 648000, 651600,
       655200, 658800, 662400, 666000, 669600, 673200, 676800,
       680400, 684000, 687600, 691200, 694800, 698400, 702000,
       705600, 709200, 712800, 716400, 720000, 723600, 727200,
       730800, 734400, 738000, 741600, 745200, 748800, 752400,
       756000, 759600, 763200, 766800, 770400, 774000, 777600,
       781200, 784800, 788400, 792000, 795600, 799200, 802800,
       806400, 810000, 813600, 817200, 820800, 824400, 828000,
       831600, 835200, 838800, 842400, 846000, 849600, 853200,
       856800, 860400, 864000, 867600, 871200, 874800, 878400,
       882000, 885600, 889200, 892800, 896400, 900000, 903600,
       907200, 910800, 914400, 918000, 921600, 925200, 928800,
       932400, 936000, 939600, 943200, 946800, 950400, 954000,
       957600, 961200, 964800, 968400, 972000, 975600, 979200,
```

982800, 986400, 990000, 993600, 997200, 1000800, 1004400,
1008000, 1011600, 1015200, 1018800, 1022400, 1026000, 1029600,
1033200, 1036800, 1040400, 1044000, 1047600, 1051200, 1054800,
1058400, 1062000, 1065600, 1069200, 1072800, 1076400, 1080000,
1083600, 1087200, 1090800, 1094400, 1098000, 1101600, 1105200,
1108800, 1112400, 1116000, 1119600, 1123200, 1126800, 1130400,
1134000, 1137600, 1141200, 1144800, 1148400, 1152000, 1155600,
1159200, 1162800, 1166400, 1170000, 1173600, 1177200, 1180800,
1184400, 1188000, 1191600, 1195200, 1198800, 1202400, 1206000,
1209600, 1213200, 1216800, 1220400, 1224000, 1227600, 1231200,
1234800, 1238400, 1242000, 1245600, 1249200, 1252800, 1256400,
1260000, 1263600, 1267200, 1270800, 1274400, 1278000, 1281600,
1285200, 1288800, 1292400, 1296000, 1299600, 1303200, 1306800,
1310400, 1314000, 1317600, 1321200, 1324800, 1328400, 1332000,
1335600, 1339200, 1342800, 1346400, 1350000, 1353600, 1357200,
1360800, 1364400, 1368000, 1371600, 1375200, 1378800, 1382400,
1386000, 1389600, 1393200, 1396800, 1400400, 1404000, 1407600,
1411200, 1414800, 1418400, 1422000, 1425600, 1429200, 1432800,
1436400, 1440000, 1443600, 1447200, 1450800, 1454400, 1458000,
1461600, 1465200, 1468800, 1472400, 1476000, 1479600, 1483200,
1486800, 1490400, 1494000, 1497600, 1501200, 1504800, 1508400,
1512000, 1515600, 1519200, 1522800, 1526400, 1530000, 1533600,
1537200, 1540800, 1544400, 1548000, 1551600, 1555200, 1558800,
1562400, 1566000, 1569600, 1573200, 1576800, 1580400, 1584000,
1587600, 1591200, 1594800, 1598400, 1602000, 1605600, 1609200,
1612800, 1616400, 1620000, 1623600, 1627200, 1630800, 1634400,
1638000, 1641600, 1645200, 1648800, 1652400, 1656000, 1659600,
1663200, 1666800, 1670400, 1674000, 1677600, 1681200, 1684800,
1688400, 1692000, 1695600, 1699200, 1702800, 1706400, 1710000,
1713600, 1717200, 1720800, 1724400, 1728000, 1731600, 1735200,
1738800, 1742400, 1746000, 1749600, 1753200, 1756800, 1760400,
1764000, 1767600, 1771200, 1774800, 1778400, 1782000, 1785600,
1789200, 1792800, 1796400, 1800000, 1803600, 1807200, 1810800,
1814400, 1818000, 1821600, 1825200, 1828800, 1832400, 1836000,
1839600, 1843200, 1846800, 1850400, 1854000, 1857600, 1861200,
1864800, 1868400, 1872000, 1875600, 1879200, 1882800, 1886400,
1890000, 1893600, 1897200, 1900800, 1904400, 1908000, 1911600,
1915200, 1918800, 1922400, 1926000, 1929600, 1933200, 1936800,
1940400, 1944000, 1947600, 1951200, 1954800, 1958400, 1962000,
1965600, 1969200, 1972800, 1976400, 1980000, 1983600, 1987200,
1990800, 1994400, 1998000, 2001600, 2005200, 2008800, 2012400,
2016000, 2019600, 2023200, 2026800, 2030400, 2034000, 2037600,
2041200, 2044800, 2048400, 2052000, 2055600, 2059200, 2062800,
2066400, 2070000, 2073600, 2077200, 2080800, 2084400, 2088000,
2091600, 2095200, 2098800, 2102400, 2106000, 2109600, 2113200,
2116800, 2120400, 2124000, 2127600, 2131200, 2134800, 2138400,
2142000, 2145600, 2149200, 2152800, 2156400, 2160000, 2163600,

```
       2167200, 2170800, 2174400, 2178000, 2181600, 2185200, 2188800,
       2192400, 2196000, 2199600, 2203200, 2206800, 2210400, 2214000,
       2217600, 2221200, 2224800, 2228400, 2232000, 2235600, 2239200,
       2242800, 2246400, 2250000, 2253600, 2257200, 2260800, 2264400,
       2268000, 2271600, 2275200, 2278800, 2282400, 2286000, 2289600,
       2293200, 2296800, 2300400, 2304000, 2307600, 2311200, 2314800,
       2318400, 2322000, 2325600, 2329200, 2332800, 2336400, 2340000,
       2343600, 2347200, 2350800, 2354400, 2358000, 2361600, 2365200,
       2368800, 2372400, 2376000, 2379600, 2383200, 2386800, 2390400,
       2394000, 2397600, 2401200, 2404800, 2408400, 2412000, 2415600,
       2419200, 2422800, 2426400, 2430000, 2433600, 2437200, 2440800,
       2444400, 2448000, 2451600, 2455200, 2458800, 2462400, 2466000,
       2469600, 2473200, 2476800, 2480400, 2484000, 2487600, 2491200,
       2494800, 2498400, 2502000, 2505600, 2509200, 2512800, 2516400,
       2520000, 2523600, 2527200, 2530800, 2534400, 2538000, 2541600,
       2545200, 2548800, 2552400, 2556000, 2559600, 2563200, 2566800,
       2570400, 2574000, 2577600, 2581200, 2584800, 2588400, 2592000,
       2595600, 2599200, 2602800, 2606400, 2610000, 2613600, 2617200,
       2620800, 2624400, 2628000, 2631600, 2635200, 2638800, 2642400,
       2646000, 2649600, 2653200, 2656800, 2660400, 2664000, 2667600,
       2671200, 2674800])
```

[69]:
```
grader.check("q511")
```

[69]: q511 results: All test cases passed!

3.6.2 5.2 Doing something to every element of an array (cont.)

Calculate a tip on several restaurant bills at once (in this case just 3):

[70]:
```
restaurant_bills = make_array(20.12, 39.90, 31.01)
print("Restaurant bills:\t", restaurant_bills)

# Array multiplication
tips = .2 * restaurant_bills
print("Tips:\t\t\t", tips)
```

```
Restaurant bills:        [ 20.12   39.9    31.01]
Tips:                    [ 4.024   7.98    6.202]
```

Question 5.2.1 Suppose the total charge at a restaurant is the original bill plus the tip. If the tip is 20%, that means we can multiply the original bill by 1.2 to get the total charge. Compute the total charge for each bill in `restaurant_bills`, and assign the resulting array to `total_charges`.

[71]:
```
total_charges = restaurant_bills*1.2
total_charges
```

[71]: array([24.144, 47.88 , 37.212])

[72]:
```
grader.check("q521")
```

[72]: q521 results: All test cases passed!

Question 5.2.2. The array `more_restaurant_bills` contains 100,000 bills! Compute the total charge for each one in `more_restaurant_bills`. How is your code different?

[73]:
```
more_restaurant_bills = Table.read_table("more_restaurant_bills.csv").
    ↪column('Bill')
more_total_charges = more_restaurant_bills*1.2
more_total_charges
```

[73]: array([20.244, 20.892, 12.216, …, 19.308, 18.336, 35.664])

[74]:
```
grader.check("q522")
```

[74]: q522 results: All test cases passed!

The function `sum` takes a single array of numbers as its argument. It returns the sum of all the numbers in that array (so it returns a single number, not an array).

Question 5.2.3. What was the sum of all the bills in `more_restaurant_bills`, *including tips*?

[75]:
```
sum_of_bills = sum(more_total_charges)
sum_of_bills
```

[75]: 1795730.0640000193

[76]:
```
grader.check("q523")
```

[76]: q523 results: All test cases passed!

Question 5.2.4. The powers of 2 ($2^0 = 1$, $2^1 = 2$, $2^2 = 4$, etc) arise frequently in computer science. (For example, you may have noticed that storage on smartphones or USBs come in powers of 2, like 16 GB, 32 GB, or 64 GB.) Use `np.arange` and the exponentiation operator `**` to compute the first 30 powers of 2, starting from `2^0`.

Hint 1: `np.arange(1, 2**30, 1)` creates an array with 2^{30} elements and **will crash your kernel**.

Hint 2: Part of your solution will involve `np.arange`, but your array shouldn't have more than 30 elements.

[77]:
```
import numpy as np
powers_of_2 = 2**np.arange(0, 30, 1)
powers_of_2
```

[77]:
```
array([       1,        2,        4,        8,       16,       32,
              64,      128,      256,      512,     1024,     2048,
            4096,     8192,    16384,    32768,    65536,   131072,
          262144,   524288,  1048576,  2097152,  4194304,  8388608,
```

```
16777216,  33554432,  67108864, 134217728, 268435456, 536870912])
```

[78]:
```
grader.check("q524")
```

[78]:
```
q524 results: All test cases passed!
```

Maple Syrup wanted to congratulate you on finishing Lab 3!!

To double-check your work, the cell below will rerun all of the autograder tests.

3.7 6. Submission

Important submission steps: 1. Run the tests and verify that they all pass. 2. Choose **Save Notebook** from the **File** menu, then **run the final cell**. 3. Click the link to download the zip file. 4. Then submit the zip file to the corresponding assignment according to your instructor's directions.

It is your responsibility to make sure your work is saved before running the last cell.

3.8 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

[79]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export(pdf=False, run_tests=True)
```

```
Running your submission against local test cases…



Your submission received the following results when run against available test
cases:

    q11 results:
        q11 - 1 result:
            Test case passed

        q11 - 2 result:
            Test case failed
            Trying:
                gagarin_quote
            Expecting:
                '"As a matter of fact, I have!"'
**********************************************************************
            Line 1, in q11 1
            Failed example:
```

```
            gagarin_quote
        Expected:
            '"As a matter of fact, I have!"'
        Got:
            'Ellipsis'

q111 results: All test cases passed!

q121 results: All test cases passed!

q131 results: All test cases passed!

q211 results: All test cases passed!

q212 results: All test cases passed!

q213 results: All test cases passed!

q221 results: All test cases passed!

q231 results: All test cases passed!

q31 results: All test cases passed!

q32 results: All test cases passed!

q41 results: All test cases passed!

q42 results:
    q42 - 1 result:
         Test case failed
        Trying:
            abs(before_2000 - 8.2783625730994146) < 1e-5
        Expecting:
            True
**********************************************************************
        Line 1, in q42 0
        Failed example:
            abs(before_2000 - 8.2783625730994146) < 1e-5
        Expected:
            True
        Got:
            False

    q42 - 2 result:
         Test case failed
        Trying:
            abs(after_or_in_2000 - 8.2379746835443033) < 1e-5
```

```
            Expecting:
                True
**********************************************************************
            Line 1, in q42 1
            Failed example:
                abs(after_or_in_2000 - 8.2379746835443033) < 1e-5
            Expected:
                True
            Got:
                False

    q43 results: All test cases passed!

    q511 results: All test cases passed!

    q521 results: All test cases passed!

    q522 results: All test cases passed!

    q523 results: All test cases passed!

    q524 results: All test cases passed!

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