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, such as 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.

Deadline: If you are not attending lab physically, you have to complete this lab and submit by Wednesday, February 5th before 8:59 P.M. in order to receive lab credit. Otherwise, please attend the lab you are enrolled in, get checked off with your (u)GSI or learning assistant **AND** submit this assignment by the end of the lab section (with whatever progress you've made) to receive lab credit.

Submission: Once you're finished, select "Save and Checkpoint" in the File menu and then execute the submit cell below (or at the end). The result will contain a link that you can use to check that your assignment has been submitted successfully.

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

In [1]:
```
# Just run this cell
import numpy as np
import math
from datascience import *

# These lines load the tests and sign you into okpy
# When you log-in please hit return (not shift + return) after typing in your e
mail
from client.api.notebook import Notebook
ok = Notebook('lab03.ok')
_ = ok.submit()
```

```
==================================================================
Assignment: Lab 3
OK, version v1.14.19
==================================================================



Saving notebook... No valid file sources found

ERROR  | auth.py:102 | {'error': 'invalid_grant'}

Performing authentication
Please enter your bCourses email: austenzhu@berkeley.edu
Successfully logged in as austenzhu@berkeley.edu
Submit... 0.0% complete
Could not submit: Late Submission of cal/data8/su17/lab02
Backup... 100% complete
```

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.

In [41]:
```
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.

In [42]:
```
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 (https://en.wikiquote.org/wiki/Yuri_Gagarin) 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.

```
BEGIN QUESTION
name: q11
```

In [5]:
```
woman_asking = 'The woman asked:' #SOLUTION
woman_quote = '"Can it be that you have come from outer space?"'
gagarin_reply = 'Gagarin replied:'
gagarin_quote = '"As a matter of fact, I have!"' #SOLUTION

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: "As a matter of fact, I have!"
```

In [6]:
```
# TEST
woman_asking
```

Out[6]:
```
'The woman asked:'
```

In [7]:
```
# TEST
gagarin_quote
```

Out[7]:
```
'"As a matter of fact, I have!"'
```

1.1. String Methods

Strings can be transformed using **methods**. Recall that methods and functions are not technically the same thing, but we'll be using them interchangeably for the purposes of this course.

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!

In [2]:
```
# Replace one letter
hello = 'Hello'
print(hello.replace('o', 'a'), hello)
```

```
Hella Hello
```

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 (http://data8.org/sp20/python-reference.html) 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.

In [9]:
```
# Calling replace on the output of another call to replace
'train'.replace('t', 'ing').replace('in', 'de')
```

Out[9]:
```
'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`.

```
BEGIN QUESTION
name: q111
```

In [10]:
```
new_word = 'hitchhiker'.replace('hi', 'ma') #SOLUTION
new_word
```

Out[10]:
```
'matchmaker'
```

In [11]:
```
# TEST
new_word
```

Out[11]:
```
'matchmaker'
```

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 (https://docs.python.org/3/library/stdtypes.html#string-methods) appears in the Python language documentation. Stack Overflow (http://stackoverflow.com) has a huge database of answered questions that often demonstrate how to use these methods to achieve various ends.

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.

In [50]:
```
8 + "8"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-ea74adbd7634> in <module>
----> 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
`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.

In [12]:
```
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.

```
BEGIN QUESTION
name: q121
```

In [13]:
```
# 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) #SOLUTION
difference
```

Out[13]:
```
30
```

In [14]:
```
# TEST
abs(difference) == 30 or abs(difference) == 30.0
```

Out[14]:
```
True
```

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, not 5.

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 (http://avalon.law.yale.edu/18th_century/rightsof.asp).)

```
BEGIN QUESTION
name: q131
```

In [15]:
```
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 co
nstantly before all the members of the Social body, shall remind them continual
ly 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 respect
ed, 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) #SOLUTION
sentence_length
```

Out[15]: 896

In [55]:
```
# TEST
sentence_length
```

Out[55]: 896

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**.

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:

In [56]:
```
make_array(0.125, 4.75, -1.3)
```

Out[56]: 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, π, and e, in that order. Name it `interesting_numbers`.

Hint: How did you get the values π and e in lab 2? You can refer to them in exactly the same way here.

```
BEGIN QUESTION
name: q211
```

In [17]:
```
interesting_numbers = make_array(0, 1, -1, math.pi, math.e) #SOLUTION
interesting_numbers
```

Out[17]:
```
array([ 0.        ,  1.        , -1.        ,  3.14159265,  2.71828183])
```

In [18]:
```
# TEST
import numpy as np
type(interesting_numbers) == np.ndarray
```

Out[18]:
```
True
```

In [19]:
```
# TEST
len(interesting_numbers)
```

Out[19]:
```
5
```

In [20]:
```
# TEST
import numpy as np
all(interesting_numbers == np.array([0, 1, -1, math.pi, math.e]))
```

Out[20]:
```
True
```

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.

```
BEGIN QUESTION
name: q212
```

In [23]:
```
hello_world_components = make_array("Hello", ",", " ", "world", "!") #SOLUTION
hello_world_components
```

Out[23]:
```
array(['Hello', ',', ' ', 'world', '!'], dtype='<U5')
```

In [24]:
```
# TEST
import numpy as np
type(hello_world_components) == np.ndarray
```

Out[24]:
```
True
```

In [25]:
```
# TEST
len(hello_world_components)
```

Out[25]: 5

In [26]:
```
# TEST
import numpy as np
all(hello_world_components == np.array(["Hello", ",", " ", "world", "!"]))
```

Out[26]: True

np.arange

Arrays are provided by a package called NumPy (http://www.numpy.org/) (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!

In [27]:
```
# This array starts at 1 and counts up by 2
# and then stops before 6
np.arange(1, 6, 2)
```

Out[27]: array([1, 3, 5])

In [28]:
```
# This array doesn't contain 9
# because np.arange stops *before* the stop value is reached
np.arange(4, 9, 1)
```

Out[28]: 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.)

```
BEGIN QUESTION
name: q213
```

In [29]:
```
import numpy as np # SOLUTION
multiples_of_99 = np.arange(0, 9999+99, 99) # SOLUTION
multiples_of_99
```

Out[29]:
```
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])
```

In [30]:
```
# TEST
type(multiples_of_99) == np.ndarray
```

Out[30]:
```
True
```

In [31]:
```
# TEST
len(multiples_of_99)
```

Out[31]:
```
102
```

In [32]:
```
# TEST
all(multiples_of_99 == np.arange(0, 9999+99, 99))
```

Out[32]:
```
True
```

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 (http://www.ncdc.noaa.gov/qclcd/QCLCD?prior=N).

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 2.1.4. 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.

```
BEGIN QUESTION
name: q214
```

In [33]:
```
collection_times = np.arange(0, 31*24*60*60, 60*60) #SOLUTION
collection_times
```

Out[33]:
```
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,
```

In [34]:
```
# TEST
type(collection_times) == np.ndarray
```

Out[34]: True

In [35]:
```
# TEST
len(collection_times)
```

Out[35]: 744

In [36]:
```
# TEST
all(collection_times == np.arange(0, 31*24*60*60, 60*60))
```

Out[36]: True

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 in 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!

In [37]:
```
population_amounts = Table.read_table("world_population.csv").column("Populatio
n")
population_amounts
```

Out[37]:
```
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], dtype=int64)
```

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

In [38]:
```
population_amounts.item(0)
```

Out[38]: 2557628654

The value of that expression is the number 2557628654 (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.

In [39]:
```
# The 13th element in the array is the population
# in 1962 (which is 1950 + 12).
population_1962 = population_amounts.item(12)
population_1962
```

Out[39]: 3140093217

In [40]:
```
# The 66th element is the population in 2015.
population_2015 = population_amounts.item(65)
population_2015
```

Out[40]: 7256490011

In [41]:
```
# 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)
<ipython-input-41-20d13c231c2b> in <module>
      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.

In [42]:
```
make_array(-1, -3, 4, -2).item(3)
```

Out[42]: -2

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

```
BEGIN QUESTION
name: q221
```

In [43]:
```
population_1973 = population_amounts.item(23) #SOLUTION
population_1973
```

Out[43]: 3942096442

In [44]:
```
# TEST
population_1973
```

Out[44]:
```
3942096442
```

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.

Logarithms

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

> How big was the population in *orders of magnitude* in each year?

Orders of magnitude quantify how big a number is by representing it as the power of another number (for example, representing 104 as $10^{2.017033}$). One way to do this is by using the logarithm function. The logarithm (base 10) of a number increases by 1 every time we multiply the number by 10. It's like a measure of how many decimal digits the number has, or how big it is in orders of magnitude.

We could try to answer our question like this, using the `log10` function from the `math` module and the `item` method you just saw:

In [45]:
```
population_1950_magnitude = math.log10(population_amounts.item(0))
population_1951_magnitude = math.log10(population_amounts.item(1))
population_1952_magnitude = math.log10(population_amounts.item(2))
population_1953_magnitude = math.log10(population_amounts.item(3))
...
```

Out[45]:
```
Ellipsis
```

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 `log10` that takes the logarithm of each element of an array. It takes a single array of numbers as its argument. It returns an array of the same length, where the first element of the result is the logarithm of the first element of the argument, and so on.

Question 2.3.1. Use `np.log10` to compute the logarithms of the world population in every year. Give the result (an array of 66 numbers) the name `population_magnitudes`. Your code should be very short.

```
BEGIN QUESTION
name: q231
```

In [46]:
```
population_magnitudes = np.log10(population_amounts) #SOLUTION
population_magnitudes
```

Out[46]:
```
array([9.40783749, 9.4141273 , 9.42107263, 9.42846742, 9.43619893,
       9.44437257, 9.45259897, 9.46110062, 9.4695477 , 9.47722498,
       9.48330217, 9.48910971, 9.49694254, 9.50648175, 9.51603288,
       9.5251    , 9.53411218, 9.54286695, 9.55173218, 9.56076229,
       9.56968959, 9.57867667, 9.58732573, 9.59572724, 9.60385954,
       9.61162595, 9.61911264, 9.62655434, 9.63388293, 9.64137633,
       9.64849299, 9.6565208 , 9.66413091, 9.67170374, 9.67893421,
       9.68632006, 9.69377717, 9.70132621, 9.70880804, 9.7161236 ,
       9.72336995, 9.73010253, 9.73688521, 9.74337399, 9.74963446,
       9.75581413, 9.7618858 , 9.76774733, 9.77343633, 9.77902438,
       9.7845154 , 9.78994853, 9.7953249 , 9.80062024, 9.80588805,
       9.81110861, 9.81632507, 9.82150788, 9.82666101, 9.83175555,
       9.83672482, 9.84161319, 9.84648243, 9.85132122, 9.85604719,
       9.8607266 ])
```

In [47]:
```
# TEST
# It looks like you're not making an array.  You shouldn't need to
# use .item anywhere in your solution.
import numpy as np
type(population_magnitudes) == np.ndarray
```

Out[47]:
```
True
```

In [48]:
```
# TEST
# You made an array, but it doesn't have the right numbers in it.
import numpy as np
sum(abs(population_magnitudes - np.log10(population_amounts))) < 1e-6
```

Out[48]:
```
True
```

What you just did is called *elementwise* application of `np.log10`, since `np.log10` 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 (https://www.inferentialthinking.com/chapters/05/1/Arrays) on arrays has a useful list of NumPy functions that are designed to work elementwise, like `np.log10`.

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:

In [49]:
```
population_in_billions = population_amounts / 1000000000
population_in_billions
```

Out[49]:
```
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 (`**`). For example, you can calculate a tip on several restaurant bills at once (in this case just 3):

In [50]:
```
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 2.3.2. 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` .

```
BEGIN QUESTION
name: q232
```

In [51]:
```
total_charges = 1.2 * restaurant_bills #SOLUTION
total_charges
```

Out[51]:
```
array([24.144, 47.88 , 37.212])
```

In [52]:
```
# TEST
# It looks like you're not making an array.  You shouldn't need to
# use .item anywhere in your solution.
import numpy as np
type(total_charges) == np.ndarray
```

Out[52]:
```
True
```

In [53]:
```
# TEST
# You made an array, but it doesn't have the right numbers in it.
import numpy as np
sum(abs(total_charges - np.array([24.144, 47.88, 37.212]))) < 1e-6
```

Out[53]: True

Question 2.3.3. The array `more_restaurant_bills` contains 100,000 bills! Compute the total charge for each one. How is your code different?

```
BEGIN QUESTION
name: q233
```

In [54]:
```
more_restaurant_bills = Table.read_table("more_restaurant_bills.csv").column("B
ill")
more_total_charges = 1.2 * more_restaurant_bills #SOLUTION
more_total_charges
```

Out[54]: array([20.244, 20.892, 12.216, ..., 19.308, 18.336, 35.664])

In [55]:
```
# TEST
# It looks like you're not making an array.  You shouldn't need to
# use .item anywhere in your solution.
import numpy as np
type(more_total_charges) == np.ndarray
```

Out[55]: True

In [56]:
```
# TEST
# You made an array, but it doesn't have the right numbers in it.
import numpy as np
sum(abs(more_total_charges - 1.2 * more_restaurant_bills)) < 1e-6
```

Out[56]: True

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

```
BEGIN QUESTION
name: q234
```

In [57]:
```
sum_of_bills = sum(more_total_charges) #SOLUTION
sum_of_bills
```

Out[57]: 1795730.0640000193

In [58]:
```
# TEST
round(sum_of_bills, 2) == 1795730.06
```

Out[58]: True

Question 2.3.5. 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.

```
BEGIN QUESTION
name: q235
```

In [60]:
```
powers_of_2 = 2 ** np.arange(30) #SOLUTION
powers_of_2
```

Out[60]:
```
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],
      dtype=int32)
```

In [61]:
```
# TEST
all(powers_of_2 == 2 ** np.arange(30))
```

Out[61]:
```
True
```

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 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.

In [62]:
```
# 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 2782
098943
 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 (http://data8.org/datascience/tables.html)) takes in alternating strings (to represent column labels) and arrays (representing the data in those columns). The strings and arrays are separated by commas.

In [63]:
```
population = Table().with_columns(
    "Population", population_amounts,
    "Year", years
)
population
```

Out[63]:

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 "Rating" and "Name", which hold `top_10_movie_ratings` and `top_10_movie_names` respectively.

```
BEGIN QUESTION
name: q31
```

In [64]:
```
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_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_movies = Table().with_columns("Rating", top_10_movie_ratings, "Name", to
p_10_movie_names) #SOLUTION

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

Out[64]:

Rating	Name
9.2	The Shawshank Redemption (1994)
9.2	The Godfather (1972)
9	The Godfather: Part II (1974)
8.9	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)
8.8	The Lord of the Rings: The Fellowship of the Ring (2001)

In [65]:
```
# TEST
type(top_10_movies) == tables.Table
```

Out[65]:
```
True
```

In [66]:
```
# TEST
top_10_movies.select('Rating', 'Name').sort('Name')
```

Out[66]:

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

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!)

```
BEGIN QUESTION
name: q32
```

In [69]:
```
imdb = Table.read_table('imdb.csv') #SOLUTION
imdb
```

Out[69]:

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)

In [70]:
```
# TEST
type(imdb) == tables.Table
```

Out[70]:
```
True
```

In [71]:
```
# TEST
imdb.num_rows == 250
```

Out[71]:
```
True
```

In [72]:
```
# TEST
imdb.select('Votes', 'Rating', 'Title', 'Year', 'Decade').sort(0).take(range(2,
5))
```

Out[72]:

Votes	Rating	Title	Year	Decade
31003	8.1	Le salaire de la peur	1953	1950
32385	8	La battaglia di Algeri	1966	1960
35983	8.1	The Best Years of Our Lives	1946	1940

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 (http://edoceo.com/utilitas/csv-file-format).

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.

`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**.

In [73]:
```
# Returns an array of movie names
top_10_movies.column('Name')
```

Out[73]:
```
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')
```

`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.

In [74]:
```
# Take first 5 movies of top_10_movies
top_10_movies.take(np.arange(0, 5, 1))
```

Out[74]:

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

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.

In [75]:
```
# Run this cell to display the population table.
population
```

Out[75]:

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` .

```
BEGIN QUESTION
name: q41
```

In [76]:
```
year_population_crossed_6_billion = population.where('Population', are.above_or
_equal_to(6*10**9)).column('Year').item(0) #SOLUTION
year_population_crossed_6_billion
```

Out[76]: 1999

In [77]:
```
# TEST
# Oops, your name is assigned to the wrong data type!
import numpy as np
type(year_population_crossed_6_billion) == int or type(year_population_crossed_
6_billion) == np.int32
```

Out[77]: True

In [78]:
```
# TEST
year_population_crossed_6_billion == 1999
```

Out[78]: True

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.

```
BEGIN QUESTION
name: q42
```

In [79]:
```
before_2000 = np.mean(imdb.where('Year', are.below(2000)).column('Rating')) #SOLUTION
after_or_in_2000 = np.mean(imdb.where('Year', are.above_or_equal_to(2000)).column('Rating')) #SOLUTION
print("Average before 2000 rating:", before_2000)
print("Average after or in 2000 rating:", after_or_in_2000)
```

```
Average before 2000 rating: 8.278362573099415
Average after or in 2000 rating: 8.237974683544303
```

In [80]:
```
# TEST
abs(before_2000 - 8.2783625730994146) < 1e-5
```

Out[80]: True

In [81]:
```
# TEST
abs(after_or_in_2000 - 8.2379746835443033) < 1e-5
```

Out[81]: True

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: Create a column called "Year Remainder" that's the remainder when each movie's release year is divided by 2. Make a copy of `imdb` that includes that column (`imdb.with_column(...)` returns a new table). Then use `where` to find rows where that new column is equal to 0. Then 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!

```
BEGIN QUESTION
name: q43
```

In [82]:
```
# BEGIN SOLUTION NO PROMPT
# You could first create a new table with the "Year Remainder" column and then
reduce its size and find its number of rows on a new line.
# You could also use the "Year" column as an array and apply the % operator on
it, and then compare the array with 0 and finally count the number of Trues tha
t result from the comparison.
# END SOLUTION
num_even_year_movies = imdb.with_column("Year Remainder", imdb.column("Year") %
2).where("Year Remainder", are.equal_to(0)).num_rows # SOLUTION
num_even_year_movies
```

Out[82]: 127

In [83]:
```
# TEST
num_even_year_movies == 127
```

Out[83]: True

Congratulations, you're done with lab 3! Be sure to

- **run all the tests** (the next cell has a shortcut for that),
- **Save and Checkpoint** from the `File` menu,
- **run the last cell to submit your work**,
- and ask one of the staff members to check you off.

In [ ]:
```
# For your convenience, you can run this cell to run all the tests at once!
import os
_ = [ok.grade(q[:-3]) for q in os.listdir("tests") if q.startswith('q')]
_ = ok.submit()
```

In [ ]:
```
_ = ok.submit()
```