

lab02b

March 2, 2022

Name: Allan Gongora

Section: 0131

1 Lab 2b: Arrays

Welcome to Lab 2b!

Last time, we had our first look at Python and Jupyter notebooks. So far, we've only used Python to manipulate numbers. There's a lot more to life than numbers, so Python lets us represent many other types of data in programs.

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 invoke *methods*. A method is very similar to a function. Calling a method looks different because the method is tied to a particular piece of data.

Last, you'll learn more about working with datasets in Python.

Please complete this notebook by filling in the cells provided. Before you begin, execute the following cell to load the provided tests.

```
[1]: pip install gofer-grader
```

```
Requirement already satisfied: gofer-grader in /opt/conda/lib/python3.7/site-  
packages (1.1.0)  
Requirement already satisfied: tornado in /opt/conda/lib/python3.7/site-packages  
(from gofer-grader) (6.1)  
Requirement already satisfied: pygments in /opt/conda/lib/python3.7/site-  
packages (from gofer-grader) (2.11.2)  
Requirement already satisfied: jinja2 in /opt/conda/lib/python3.7/site-packages  
(from gofer-grader) (3.0.3)  
Requirement already satisfied: MarkupSafe>=2.0 in /opt/conda/lib/python3.7/site-  
packages (from jinja2->gofer-grader) (2.0.1)  
Note: you may need to restart the kernel to use updated packages.
```

```
[2]: from gofer.ok import check
```

Recommended Reading: * [Sequences](#)

- 1) For all problems that you must write explanations and sentences for, you **must** provide your answer in the designated space. This can include:
 - A) Sentence responses to questions that ask for an explanation
 - B) Numeric responses to multiple choice questions
 - C) Programming code
- 2) Moreover, throughout this lab and all future ones, please be sure to not re-assign variables throughout the notebook! For example, if you use `max_temperature` in your answer to one question, do not reassign it later on. Otherwise, you will fail tests that you thought you were passing previously!

Once you're finished, select "Save and Checkpoint" in the File menu. Your name and course section number should be in the first and last cell of the assignment. Be sure you have run all cells with code and that the output from that is showing. Then click "Print Preview" in the File menu. Print a copy from there in pdf format. (This means you right click and choose print and choose "save as pdf" from your printer options.) You will need to submit the pdf in Canvas by the deadline.

The gopher grader output and/or output from your coding are essential to helping your instructor grade your work correctly and in a timely manner.

Files submitted that are missing the required output will lose some to all points so double check your pdf before submitting.

1.1 1. Making Arrays

You can type in the data that goes in an array yourself, but that's not typically how programs work. Normally, we create arrays by loading them from an external source, like a data file.

First, though, let's learn how to start from scratch. Execute the following cell so that all the names from the `datascience` module are available to you. The documentation for this module is available at <http://data8.org/datascience>.

```
[3]: from datascience import *
```

```
[4]: import math
```

Now, to create an array, 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:

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

```
[5]: 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* or *item* of that array.

Arrays themselves are also values, just like numbers and strings. That means you can assign them names or use them as arguments to functions.

Question 1.1 Make an array containing the numbers 1, 2, and 3, in that order. Name it `small_numbers`.

```
[6]: small_numbers = make_array(1, 2, 3)
      small_numbers
```

```
[6]: array([1, 2, 3])
```

```
[7]: check('tests/q411.py')
```

```
[7]: <gofer.ok.OKTestsResult at 0x7f52af3702d0>
```

Question 1.2 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 earlier? You can refer to them in exactly the same way here.

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

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

```
[9]: check('tests/q412.py')
```

```
[9]: <gofer.ok.OKTestsResult at 0x7f52af6eea10>
```

Question 1.3 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 print `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 things in the array are strings.

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

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

```
[11]: check('tests/q413.py')
```

```
[11]: <gofer.ok.OKTestsResult at 0x7f52af371610>
```

The `join` method of a string takes an array of strings as its argument and puts all of the elements together into one string. Try it:

```
[12]: '-'.join(make_array('a', 'b', 'c', 'd'))
```

```
[12]: 'a-b-c-d'
```

Question 1.4 Assign `separator` to a string so that the name `hello` is bound to the string 'Hello, world!' in the cell below.

```
[13]: separator = ""
      hello = separator.join(hello_world_components)
      hello
```

```
[13]: 'Hello, world!'
```

```
[14]: check('tests/q414.py')
```

```
[14]: <gofer.ok.OKTestsResult at 0x7f52af37c550>
```

1.1.1 np.arange

Arrays are provided by a package called [NumPy](#) (pronounced “NUM-pie” or, if you prefer to pronounce things incorrectly, “NUM-pee”). 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`. `np.arange(start, stop, space)` produces an array with all the numbers starting at `start` and counting up by `space`, stopping before `stop` is reached.

For example, the value of `np.arange(1, 6, 2)` is an array with elements 1, 3, and 5 – it starts at 1 and counts up by 2, then stops before 6. In other words, it’s equivalent to `make_array(1, 3, 5)`.

`np.arange(4, 9, 1)` is an array with elements 4, 5, 6, 7, and 8. (It doesn’t contain 9 because `np.arange` stops *before* the stop value is reached.)

Question 1.5 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.)

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

```
[15]: 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])
```

```
[16]: check('tests/q4111.py')
```

```
[16]: <gofer.ok.OKTestsResult at 0x7f52af3b9450>
```

1.1.2 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 doesn'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 1.6 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.

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

```
[17]: 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])

```

```
[18]: check('tests/q4112.py')
```

```
[18]: <gofer.ok.OKTestsResult at 0x7f52af3c3510>
```

1.2 2. Working with Single Elements of Arrays (“Indexing”)

Let’s work with a more interesting dataset. The next cell creates an array called `population` 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 next week.

```
[19]: # Don't worry too much about what goes on in this cell.
      from datascience import *
      population = Table.read_table("world_population.csv").column("Population")
      population
```

```
[19]: 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`, which is the world population in the first year in the dataset, 1950.

```
[20]: population.item(0)
```

```
[20]: 2557628654
```

The value of that expression is the number 2557628654 (around 2.5 billion), because that’s the first thing in the array `population`.

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`. Read and run each cell.

```
[21]: # The third element in the array is the population
      # in 1952.
      population_1952 = population.item(2)
      population_1952
```


[21]: 2636772306

```
[22]: # The thirteenth element in the array is the population
      # in 1962 (which is 1950 + 12).
      population_1962 = population.item(12)
      population_1962
```

[22]: 3140093217

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

[23]: 7256490011

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

```
[25]: # 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)
```

[25]: -2

Question 2.1 Set `population_1973` to the world population in 1973, by getting the appropriate element from `population` using `item`.

```
[26]: population_1973 = population.item(23)
      population_1973
```

[26]: 3942096442

```
[27]: check('tests/q421.py')
```

[27]: <gofer.ok.OKTestsResult at 0x7f52af3c4510>

1.3 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.

1.3.1 Logarithms

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

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

The logarithm function is one way of measuring how big a number is. 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:

```
[28]: import math

population_1950_magnitude = math.log10(population.item(0))
population_1951_magnitude = math.log10(population.item(1))
population_1952_magnitude = math.log10(population.item(2))
population_1953_magnitude = math.log10(population.item(3))
...
```

[28]: 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 3.1 Use it 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.

```
[29]: population_magnitudes = np.log10(population)
population_magnitudes
```

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

```
[30]: check('tests/q431.py')
```

```
[30]: <gofer.ok.OKTestsResult at 0x7f52af2953d0>
```

This is called *elementwise* application of the function, since it operates separately on each element of the array it's called on. The textbook's section on arrays has a useful list of NumPy functions that are designed to work elementwise, like `np.log10`.

1.3.2 Arithmetic

Arithmetic also works elementwise on arrays. For example, you can divide all the population numbers by 1 billion to get numbers in billions:

```
[31]: population_in_billions = population / 1000000000
      population_in_billions
```

```
[31]: 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):

```
[32]: restaurant_bills = make_array(20.12, 39.90, 31.01)
      print("Restaurant bills:\t", restaurant_bills)
      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 3.2 Suppose the total charge at a restaurant is the original bill plus the tip. 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`.

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

```
[33]: array([24.144, 47.88 , 37.212])
```

```
[34]: check('tests/q432.py')
```

```
[34]: <gofer.ok.OKTestsResult at 0x7f52af2959d0>
```

Question 3.3 `more_restaurant_bills.csv` contains 100,000 bills! Compute the total charge for each one. How is your code different?

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

```
[35]: array([20.244, 20.892, 12.216, ..., 19.308, 18.336, 35.664])
```

```
[36]: check('tests/q433.py')
```

```
[36]: <gofer.ok.OKTestsResult at 0x7f52af407cd0>
```

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

```
[37]: sum_of_bills = sum(more_restaurant_bills * 1.2)
      sum_of_bills
```

```
[37]: 1795730.0640000193
```

```
[38]: check('tests/q434.py')
```

```
[38]: <gofer.ok.OKTestsResult at 0x7f52af370c90>
```

Question 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 15 powers of 2, starting from 2^0 .

```
[39]: powers_of_2 = 2 ** np.arange(15)
      powers_of_2
```

```
[39]: array([ 1,  2,  4,  8, 16, 32, 64, 128, 256,
          512, 1024, 2048, 4096, 8192, 16384])
```

```
[40]: check('tests/q435.py')
```

```
[40]: <gofer.ok.OKTestsResult at 0x7f52af3c4e90>
```

1.4 4. Submission

Once you're finished, select "Save and Checkpoint" in the File menu. Your name and course section number should be in the first and last cell of the assignment. Be sure you have run all cells with

code and that the output from that is showing.

Double check that you have completed all of the free response questions as the auto-grader does NOT check that and YOU are responsible for knowing those questions are there and completing them as part of the grade for this lab. When ready, click “Print Preview” in the File menu. Print a copy from there in pdf format. (This means you right click and choose print and choose “save as pdf” from your printer options.) You will need to submit the pdf in Canvas by the deadline.

The gopher grader output and/or output from your coding are essential to helping your instructor grade your work correctly and in a timely manner.

Files submitted that are missing the required output will lose some to all points so double check your pdf before submitting.

```
[41]: # For your convenience, you can run this cell to run all the tests at once!
import glob
from gofer.ok import grade_notebook
if not globals().get('__GOFER_GRADER__', False):
    display(grade_notebook('lab02b.ipynb', sorted(glob.glob('tests/q4*.py'))))
```

Restaurant bills: [20.12 39.9 31.01]

Tips: [4.024 7.98 6.202]

['tests/q411.py', 'tests/q412.py', 'tests/q413.py', 'tests/q414.py',
'tests/q4111.py', 'tests/q4112.py', 'tests/q421.py', 'tests/q431.py',
'tests/q432.py', 'tests/q433.py', 'tests/q434.py', 'tests/q435.py']

Question 1:

<gofer.ok.OKTestsResult at 0x7f52af2c5a10>

Question 2:

<gofer.ok.OKTestsResult at 0x7f52aef73b90>

Question 3:

<gofer.ok.OKTestsResult at 0x7f52aef73a10>

Question 4:

<gofer.ok.OKTestsResult at 0x7f52aef73e50>

Question 5:

<gofer.ok.OKTestsResult at 0x7f52aef73d50>

Question 6:

<gofer.ok.OKTestsResult at 0x7f52aef73dd0>

Question 7:

<gofer.ok.OKTestsResult at 0x7f52aef79290>

Question 8:

<gofer.ok.OKTestsResult at 0x7f52aef79d50>

Question 9:

<gofer.ok.OKTestsResult at 0x7f52aef79d90>

Question 10:

<gofer.ok.OKTestsResult at 0x7f52aef79950>

Question 11:

<gofer.ok.OKTestsResult at 0x7f52aef798d0>

Question 12:

<gofer.ok.OKTestsResult at 0x7f52aef79e50>

1.0

Name: Allan Gongora

Section: 0131