Lab 1: Expressions

Welcome to Data Science for Political Campaigns! Each week you will complete a lab assignment like this one. You can't learn technical subjects without hands-on practice, so labs are an important part of the course.

Before we get started, there are some administrative details.

Labs are required.

Collaborating on labs is more than okay -- it's encouraged! You should rarely be stuck for more than a few minutes on questions in labs, so ask me or a classmate for help. (Explaining things is beneficial, too -- the best way to solidify your knowledge of a subject is to explain it.) Please don't just share answers, though.

To turn in your lab, you will need to submit a PDF through Canvas. At the bottom, I provide some instructions on how to save your lab as a PDF.

Today's lab

In today's lab, you'll learn how to:

1. navigate Jupyter notebooks (like this one);
2. write and evaluate some basic *expressions* in Python, the computer language of the course;
3. call *functions* to use code other people have written; and
4. break down Python code into smaller parts to understand it.

This lab covers parts of Chapter 3 of the online textbook.

1. Jupyter notebooks

This webpage is called a Jupyter notebook. A notebook is a place to write programs and view their results.

1.1. Text cells

In a notebook, each rectangle containing text or code is called a *cell*.

Text cells (like this one) can be edited by double-clicking on them. They're written in a simple format called Markdown to add formatting and section headings. You don't need to learn Markdown, but you might want to.

After you edit a text cell, click the "run cell" button at the top that looks like ▶| to confirm any changes. (Try not to delete the instructions of the lab.)

Question 1. This paragraph is in its own text cell. Try editing it by typing your name at the end

of the last sentence in the paragraph, and then click the "run cell" ▶| button. Does your name show up properly? Vivian

1.2. Code cells

Other cells contain code in the Python 3 language. Running a code cell will execute all of the code it contains.

To run the code in a code cell, first click on that cell to activate it. It'll be highlighted with a little green or blue rectangle. Next, either press ▶| or hold down the `shift` key and press `return` or `enter`.

Try running this cell:

In [3]:
```
print("Hello, World!")
```

```
Hello, World!
```

And this one:

In [1]:
```
print("\N{WAVING HAND SIGN}, \N{EARTH GLOBE ASIA-AUSTRALIA}!")
```

👋, 🌏!

The fundamental building block of Python code is an expression. Cells can contain multiple lines with multiple expressions. When you run a cell, the lines of code are executed in the order in which they appear. Every `print` expression prints a line. Run the next cell and notice the order of the output.

In [4]:
```
print("First this line is printed,")
print("then the whole \N{EARTH GLOBE ASIA-AUSTRALIA},")
print("and then this one.")
```

```
First this line is printed,
then the whole 🌏,
and then this one.
```

Question 2. Change the cell above so that it prints out:

```
First this line,
then the whole 🌏,
and then this one.
```

1.3. Writing Jupyter notebooks

You can use Jupyter notebooks for your own projects or documents. When you make your own notebook, you'll need to create your own cells for text and code.

To add a cell, click the + button in the menu bar. It'll start out as a text cell. You can change it to a code cell by clicking inside it so it's highlighted, clicking the drop-down box next to the restart (⟳) button in the menu bar, and choosing "Code".

Question 3. Add a code cell below this one. Write code in it that prints out:

```
A whole new cell! ♪🌍♪
```

(That musical note symbol is like the Earth symbol. Its long-form name is `\N{EIGHTH NOTE}`.)

Run your cell to verify that it works.

In [5]:
```
print("A whole new cell! \N{EIGHTH NOTE} \N{EARTH GLOBE ASIA-AUSTRALIA}\N{EIGHTH
```

```
A whole new cell! ♪ 🌏♪
```

1.4. Errors

Python is a language, and like natural human languages, it has rules. It differs from natural language in two important ways:

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

Whenever you write code, you'll make mistakes. When you run a code cell that has errors, Python will sometimes produce error messages to tell you what you did wrong.

Errors are okay; even experienced programmers make many errors. When you make an error, you just have to find the source of the problem, fix it, and move on.

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

In [6]:
```
print("This line is missing something.")
```

```
This line is missing something.
```

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

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

There's a lot of terminology in programming languages, but you don't need to know it all in order to program effectively. If you see a cryptic message like this, you can often get by without deciphering it.

Question 4. Try to fix the code above so that you can run the cell and see the intended message instead of an error.

1.5. The Kernel

The kernel is a program that executes the code inside your notebook and outputs the results. In the top right of your window, you can see a circle that indicates the status of your kernel. If the circle is empty (⚪), the kernel is idle and ready to execute code. If the circle is filled in (⚫), the kernel is busy running some code.

You may run into problems where your kernel is stuck for an excessive amount of time, your notebook is very slow and unresponsive, or your kernel loses its connection. If this happens, try the following steps:

1. At the top of your screen, click **Kernel**, then **Interrupt**.
2. If that doesn't help, click **Kernel**, then **Restart**. If you do this, you will have to run your code cells from the start of your notebook up until where you paused your work.
3. If that doesn't help, restart your server. First, save your work by clicking **File** at the top left of your screen, then **Save and Checkpoint**. Next, click **Control Panel** at the top right. Choose **Stop My Server** to shut it down, then **My Server** to start it back up. Then, navigate back to the notebook you were working on.

1.6. Submitting your work

All assignments in the course will be distributed as notebooks like this one, and you will submit your work as a PDF. You can download your notebook by opening it, turning Edit mode on, then navigating to File -> Download as -> PDF. You can then submit this PDF via Canvas.

2. Numbers

Quantitative information arises everywhere in data science. In addition to representing commands to print out lines, expressions can represent numbers and methods of combining numbers. The expression `3.2500` evaluates to the number 3.25. (Run the cell and see.)

In [7]:
```
3.2500
```

Out[7]:
```
3.25
```

Notice that we didn't have to `print`. When you run a notebook cell, if the last line has a value, then Jupyter helpfully prints out that value for you. However, it won't print out prior lines automatically.

In [6]:
```
print(2)
3
4
```

```
2
```

Out[6]:
```
4
```

Above, you should see that 4 is the value of the last expression, 2 is printed, but 3 is lost forever because it was neither printed nor last.

You don't want to print everything all the time anyway. But if you feel sorry for 3, change the cell above to print it.

2.1. Arithmetic

The line in the next cell subtracts. Its value is what you'd expect. Run it.

In [8]:
```
3.25 - 1.5
```

Out[8]:
```
1.75
```

Many basic arithmetic operations are built in to Python. The textbook section on Expressions describes all the arithmetic operators used in the course. The common operator that differs from typical math notation is `**`, which raises one number to the power of the other. So, `2**3` stands for 2^3 and evaluates to 8.

The order of operations is what you learned in elementary school, and Python also has parentheses.

In [12]:
```
1+6*5-6*3**2*2**3/4*7
```

Out[12]:
```
-725.0
```

In [9]:
```
1+(6*5-(6*3))**2*((2**3)/4*7)
```

Out[9]:
```
2017.0
```

In standard math notation, the first expression is

$$1 + 6 \times 5 - 6 \times 3^2 \times \frac{2^3}{4} \times 7,$$

while the second expression is

$$1 + (6 \times 5 - (6 \times 3))^2 \times (\frac{(2^3)}{4} \times 7).$$

Question 5. Write a Python expression in this next cell that's equal to $5 \times (\frac{3}{11}) - 50 + 2^{.5 \times 22} - \frac{7}{33}$.

Replace the ellipses (`...`) with your expression.

In [10]:
```
5*(3/11)-50+((2**(.5*22))-(7/33))
```

Out[10]:
```
1999.151515151515
```

3. Names

In natural language, we have terminology that lets us quickly reference very complicated concepts. We don't say, "That's a large mammal with brown fur and sharp teeth!" Instead, we just say, "Bear!"

Similarly, an effective strategy for writing code is to define names for data as we compute it, like a lawyer would define terms for complex ideas at the start of a legal document to simplify the rest of the writing.

In Python, we do this with *assignment statements*. An assignment statement has a name on the left side of an `=` sign and an expression to be evaluated on the right.

In [12]:
```
ten = 3 * 2 + 4
```

When you run that cell, Python first evaluates the first line. It computes the value of the expression `3 * 2 + 4`, which is the number 10. Then it gives that value the name `ten`. At that point, the code in the cell is done running.

After you run that cell, the value 10 is bound to the name `ten`:

In [13]:
```
ten
```

Out[13]:
```
10
```

The statement `ten = 3 * 2 + 4` is not asserting that `ten` is already equal to `3 * 2 + 4`, as we might expect by analogy with math notation. Rather, that line of code changes what `ten` means; it now refers to the value 10, whereas before it meant nothing at all.

If the designers of Python had been ruthlessly pedantic, they might have made us write

```
define the name ten to hereafter have the value of 3 * 2 + 4
```

instead. You will probably appreciate the brevity of " = "! But keep in mind that this is the real meaning.

Question 6. Try writing code that uses a name (like `eleven`) that hasn't been assigned to anything. You'll see an error!

In [12]:
```
eleven = 11
eleven
```

Out[12]:
```
11
```

A common pattern in Jupyter notebooks is to assign a value to a name and then immediately evaluate the name in the last line in the cell so that the value is displayed as output.

In [13]:
```
close_to_pi = 355/113
close_to_pi
```

Out[13]:
```
3.1415929203539825
```

Another common pattern is that a series of lines in a single cell will build up a complex computation in stages, naming the intermediate results. Writing code likes this can make it easier for your future self to understand what you were doing at the time. It also helps if any of your inputs change, such as getting a raise.

In [19]:
```
bimonthly_salary = 840
monthly_salary = 2 * bimonthly_salary
number_of_months_in_a_year = 12
yearly_salary = number_of_months_in_a_year * monthly_salary
yearly_salary
```

Out[19]:
```
20160
```

Names in Python can have letters (upper- and lower-case letters are both okay and count as different letters), underscores, and numbers. The first character can't be a number (otherwise a name might look like a number). And names can't contain spaces, since spaces are used to separate pieces of code from each other.

Other than those rules, what you name something doesn't matter *to Python*. For example, this cell does the same thing as the above cell, except everything has a different name:

In [14]:
```
a = 840
b = 2 * a
c = 12
d = c * b
d
```

Out[14]:
```
20160
```

However, names are very important for making your code *readable* to yourself and others. The cell above is shorter, but it's totally useless without an explanation of what it does.

Question 7. Assign the name `seconds_in_a_decade` to the number of seconds between midnight January 1, 2010 and midnight January 1, 2020.

In [18]:
```
# Change the next line so that it computes the number of
# seconds in a decade and assigns that number the name
# seconds_in_a_decade.
sec = 1
minu = 60 * sec
hour = 60 * minu
seconds_in_a_day = 24*hour
seconds_in_a_year = 365*seconds_in_a_day
seconds_in_a_leap = 366* seconds_in_a_day
```

```
seconds_in_a_decade = (8*seconds_in_a_year) + (2*seconds_in_a_leap)

# We've put this line in this cell so that it will print
# the value you've given to seconds_in_a_decade when you
# run it.  You don't need to change this.
seconds_in_a_decade
```

Out[18]: 315532800

3.1. Comments

You may have noticed this line in the cell above:

```
# We've put this line in this cell so that it will print
```

That is called a *comment*. It doesn't make anything happen in Python; Python ignores anything on a line after a #. Instead, it's there to communicate something about the code to you, the human reader. Comments are extremely useful.

3.2. Application: Number of voters needed to win

Everything you do during the campaign, from knocking on a door, to sending out a tweet, should be in service of reaching your win number on Election Day. Time, people, and money are every

campaign's most important resources. In order to make sure those resources are used in the most efficient and effective way possible, we use targeting. Targeting starts with knowing how many votes you'll need to win. So let's get to work!

Question 8. Assume that the percentage of voter turnout from the last three similar elections has been 40%, 45%, and 52%. What has the **average** voter turnout been?

In [19]:
```
average_turnout = (.4+.45+.52)/(3)
average_turnout
```

Out[19]: 0.4566666666666667

Question 9. Assume there are 14,527 people registered to vote. How many people do you expect to vote in this election?

In [20]:
```
expected_turnout = 14527 * average_turnout
expected_turnout
```

Out[20]: 6633.996666666668

Question 10. How many votes do you need to get in order to win? Please explain.

The needed votes to win is 50% + 1 of the voters who turn out.

In [22]:
```
needed_votes_to_win = ((expected_turnout/2)+1)
needed_votes_to_win
```

Out[22]: 3317.998333333334

4. Calling functions

The most common way to combine or manipulate values in Python is by calling functions. Python comes with many built-in functions that perform common operations.

For example, the `abs` function takes a single number as its argument and returns the absolute value of that number. The absolute value of a number is its distance from 0 on the number line, so `abs(5)` is 5 and `abs(-5)` is also 5.

In [27]:
```
abs(5)
```

Out[27]: 5

In [28]:
```
abs(-5)
```

Out[28]: 5

4.1. Application: Computing walking distances

Chunhua is on the corner of 7th Avenue and 42nd Street in Midtown Manhattan, and she wants to know far she'd have to walk to get to Gramercy School on the corner of 10th Avenue and 34th Street.

She can't cut across blocks diagonally, since there are buildings in the way. She has to walk along the sidewalks. Using the map below, she sees she'd have to walk 3 avenues (long blocks) and 8 streets (short blocks). In terms of the given numbers, she computed 3 as the difference between 7 and 10, *in absolute value*, and 8 similarly.

Chunhua also knows that blocks in Manhattan are all about 80m by 274m (avenues are farther apart than streets). So in total, she'd have to walk $(80 \times |42 - 34| + 274 \times |7 - 10|)$ meters to get to the park.

Question 11. Finish the line `num_avenues_away = ...` in the next cell so that the cell calculates the distance Chunhua must walk and gives it the name `manhattan_distance`. Everything else has been filled in for you. **Use the `abs` function.**

In [23]:
```
# Here's the number of streets away:
num_streets_away = abs(42-34)

# Compute the number of avenues away in a similar way:
num_avenues_away = abs(7-10)

street_length_m = 80
avenue_length_m = 274

# Now we compute the total distance Chunhua must walk.
manhattan_distance = street_length_m*num_streets_away + avenue_length_m*num_aven

# We've included this line so that you see the distance
# you've computed when you run this cell.  You don't need
# to change it, but you can if you want.
manhattan_distance
```

Out[23]: 1462

Multiple arguments

Some functions take multiple arguments, separated by commas. For example, the built-in `max` function returns the maximum argument passed to it.

In [24]:
```
max(2, -3, 4, -5)
```

Out[24]: 4

5. Understanding nested expressions

Function calls and arithmetic expressions can themselves contain expressions. You saw an example in the last question:

```
abs(42-34)
```

has 2 number expressions in a subtraction expression in a function call expression. And you probably wrote something like `abs(7-10)` to compute `num_avenues_away`.

Nested expressions can turn into complicated-looking code. However, the way in which complicated expressions break down is very regular.

Suppose we are interested in heights that are very unusual. We'll say that a height is unusual to the extent that it's far away on the number line from the average human height. An estimate of the average adult human height (averaging, we hope, over all humans on Earth today) is 1.688 meters.

So if Aditya is 1.21 meters tall, then his height is $|1.21 - 1.688|$, or $.478$, meters away from the average. Here's a picture of that:

And here's how we'd write that in one line of Python code:

In [ ]:
```
abs(1.21 - 1.688)
```

What's going on here? `abs` takes just one argument, so the stuff inside the parentheses is all part of that *single argument*. Specifically, the argument is the value of the expression `1.21 - 1.688`. The value of that expression is `-.478`. That value is the argument to `abs`. The absolute value of that is `.478`, so `.478` is the value of the full expression `abs(1.21 - 1.688)`.

Picture simplifying the expression in several steps:

1. `abs(1.21 - 1.688)`
2. `abs(-.478)`
3. `.478`

In fact, that's basically what Python does to compute the value of the expression.

Question 12. Say that Botan's height is 1.85 meters. In the next cell, use `abs` to compute the absolute value of the difference between Botan's height and the average human height. Give that value the name `botan_distance_from_average_m`.

In [30]:
```
# Replace the ... with an expression to compute the absolute
# value of the difference between Botan's height (1.85m) and
# the average human height.
botan_distance_from_average_m = abs(1.688-1.85)
```

```
# Again, we've written this here so that the distance you
# compute will get printed when you run this cell.
botan_distance_from_average_m
```

Out[30]: 0.16200000000000014

5.1. More nesting

Now say that we want to compute the most unusual height among Aditya's and Botan's heights. We'll use the function `max`, which (again) takes two numbers as arguments and returns the larger of the two arguments. Combining that with the `abs` function, we can compute the biggest distance from the average among the two heights:

In [25]:
```
# Just read and run this cell.

aditya_height_m = 1.21
botan_height_m = 1.85
average_adult_human_height_m = 1.688

# The biggest distance from the average human height, among the two heights:
biggest_distance_m = max(abs(aditya_height_m - average_adult_human_height_m), ab

# Print out our results in a nice readable format:
print("The biggest distance from the average height among these two people is",
```

```
The biggest distance from the average height among these two people is 0.478 met
ers.
```

The line where `biggest_distance_m` is computed looks complicated, but we can break it down into simpler components just like we did before.

The basic recipe is repeated simplification of small parts of the expression:

- We start with the simplest components whose values we know, like plain names or numbers. (Examples: `aditya_height_m` or `5`.)
- **Find a simple-enough group of expressions:** We look for a group of simple expressions that are directly connected to each other in the code, for example by arithmetic or as arguments to a function call.
- **Evaluate that group:** We evaluate the arithmetic expressions or function calls they're part of, and replace the whole group with whatever we compute. (Example: `aditya_height_m - average_adult_human_height_m` becomes `-.478`.)
- **Repeat:** We continue this process, using the values of the glommed-together stuff as our new basic components. (Example: `abs(-.478)` becomes `.478`, and `max(.478, .162)` later becomes `.478`.)
- We keep doing that until we've evaluated the whole expression.

Ok, your turn.

Question 13. Given the heights of the Splash Triplets from the Golden State Warriors, write an expression that computes the smallest difference between any of the three heights. Your

expression shouldn't have any numbers in it, only function calls and the names `klay`, `steph`, and `kevin`. Give the value of your expression the name `min_height_difference`.

In [26]:
```
# The three players' heights, in meters:
klay =  2.01 # Klay Thompson is 6'7"
steph = 1.91 # Steph Curry is 6'3"
kevin = 2.06 # Kevin Durant is officially 6'9", but many suspect that he is tall
             # (Further complicating matters, membership of the "Splash Triplets
             #  is disputed, since it was originally used in reference to
             #  Klay Thompson, Steph Curry, and Draymond Green.)

# We'd like to look at all 3 pairs of heights, compute the absolute
# difference between each pair, and then find the smallest of those
# 3 absolute differences.  This is left to you!  If you're stuck,
# try computing the value for each step of the process (like the
# difference between Klay's heigh and Steph's height) on a separate
# line and giving it a name (like klay_steph_height_diff).
min_height_difference = min(abs(klay-steph),abs(klay-kevin), abs(steph-kevin))
min_height_difference
```

Out[26]: 0.050000000000000266

Congratulations!

You are done with the lab. Before you finish and submit, please fill out this brief evaluation:

- I spent around 3 hours on this lab.
- This lab was just about the right difficulty.

To turn in your lab, you will need to submit a PDF through Canvas. There are a few options to do this. The best option will depend on your computer and web browser.

- Option 1: File -> Download as -> PDF.
- Option 2: File -> Print Preview -> This opens a new tab; in the new tab -> File -> Export as a PDF.
- Option 3: File -> Download as -> HTML -> Double-click and open the HTML file -> Export as a PDF.

Whichever option you take, please review the PDF before you upload it. Make sure everything looks good and prints properly. If you run into any issues, please reach out for help.