Lab 1: Expressions

Welcome to Data Science 8: Foundations of Data Science! Each week you will complete a lab assignment like this one; these are designed to be completed during the scheduled lab time, but you can work on them at home as well.

FIXME: Policies.

Labs are required, and you must complete them by the end of the last scheduled lab of the week. You can read more about course policies (https://data8.org/info) on the course website (http://data8.org).

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.

In the middle of the lab, you'll see how to run *automated tests* we've provided to check whether your code is working properly. When you're done, follow the instructions at the end of this notebook to run all the tests and submit your lab.

This lab covers parts of Section 1.4 (http://www.inferentialthinking.com/chapter1/expressions.html) of the online textbook. You should read the book, but not right now. Instead, let's get started!

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 (http://daringfireball.net/projects/markdown/syntax) 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 1.1.1. This paragraph is in its own text cell. Try editing it so that this sentence is the last sentence in the paragraph, and then click the "run cell" ▶| button . This sentence, for example, should be deleted. So should this one.

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

And this one:

```
In [2]: 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 [3]: 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 1.2.1. Change the cell above so that it prints out:

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

Hint: If you're stuck on the Earth symbol for more than a few minutes, try talking to a neighbor or a TA. That's a good idea for any lab problem.

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 1.3.1. 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.

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.

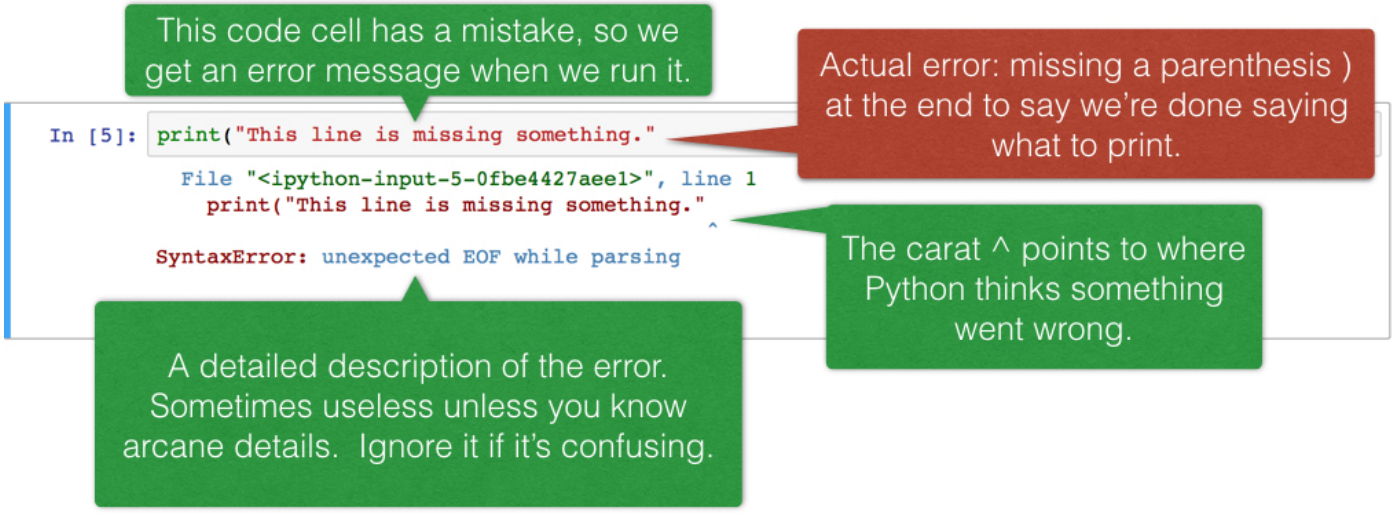
Whenever you write code, you'll make mistakes. When you run a code cell that has errors, Python will sometimes produce error messages to tell you what you did wrong.

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 [4]: print("This line is missing something."
          File "<ipython-input-4-0fbe4427aee1>", line 1
            print("This line is missing something."
                                                   ^
        SyntaxError: unexpected EOF while parsing
```

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. (Of course, if you're frustrated, ask a neighbor or a TA for help.)

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

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 [ ]: 3.2500
```

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 [4]: print(2)
        3
        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 [ ]:
```
3.25 - 1.5
```

Many basic arithmetic operations are built in to Python. The textbook section on Expressions (http://www.inferentialthinking.com/chapter1/expressions.html) 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. For example, compare the outputs of the cells below. Use parentheses for a happy new year!

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

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

In standard math notation, the first expression is

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

while the second expression is

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

Question 2.1.1. Write a Python expression in this next cell that's equal to $5 \times (3\frac{10}{11}) - 51\frac{1}{3} + 2^{.5 \times 22}$. That's five times three and ten elevenths, minus fifty-one and a third, plus two to the power of half 22. By "$3\frac{10}{11}$" we mean $3 + \frac{10}{11}$, not $3 \times \frac{10}{11}$.

Replace the ellipses (`...`) with your expression. Try to use parentheses only when necessary.

Hint: The correct output should start with a familiar number.

In [19]:
```
5 * (3 + 10 / 11) - (51 + 1/3) + 2**(.5*22)# SOLUTION
```

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.

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 [20]: 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 [ ]: ten
```

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

```
In [ ]:
```

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

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.

In [ ]:
```
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
```

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).

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

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.

According to a famous joke among computer scientists, naming things is one of the two hardest problems in computer science. (The other two are cache invalidation and "off-by-one" errors. And people say computer scientists have an odd sense of humor...)

Question 3.2. Assign the name `seconds` to the number of seconds between midnight January 1, 2010 and midnight January 1, 2020.

Hint: If you're stuck, the next section shows you how to get hints.

In [ ]:
```
# 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.
seconds_in_a_decade = ...

# 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
```

3.1. Checking your code

Now that you know how to name things, you can start using the built-in *tests* to check whether your work is correct. Try not to change the contents of the test cells.

Running the following cell will test whether you have assigned `seconds_in_a_decade` correctly in Question 3.2. If you haven't, this test will tell you the correct answer. Resist the urge to just copy it, and instead try to adjust your expression. (Sometimes the tests will give hints about what went wrong...)

In [ ]:
```
# Test cell; please do not change!
from client.api.assignment import load_assignment
lab01 = load_assignment('lab01.ok')
_ = lab01.grade('q32')
```

All labs will have tests like this one. To get credit for a lab, complete the notebook so that all tests pass, then run the final cell to submit your work.

3.2. Comments

You may have noticed this line in the cell above:

```
# Test cell; please do not change!
```

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.

3.3. Application: A physics experiment

On the Apollo 15 mission to the Moon, astronaut David Scott famously replicated Galileo's physics experiment in which he showed that gravity accelerates objects of different mass at the same rate. Because there is no air resistance for a falling object on the surface of the Moon, even two objects with very different masses and densities should fall at the same rate. David Scott compared a feather and a hammer.

You can run the following cell to watch a video of the experiment.

In [5]:
```
from IPython.display import YouTubeVideo
# The original URL is:
#   https://www.youtube.com/watch?v=U7db6ZeLR5s
YouTubeVideo("U7db6ZeLR5s")
```

Out[5]:

Here's the transcript of the video:

167:22:06 Scott: Well, in my left hand, I have a feather; in my right hand, a hammer. And I guess one of the reasons we got here today was because of a gentleman named Galileo, a long time ago, who made a rather significant discovery about falling objects in gravity fields. And we thought where would be a better place to confirm his findings than on the Moon. And so we thought we'd try it here for you. The feather happens to be, appropriately, a falcon feather for our Falcon. And I'll drop the two of them here and, hopefully, they'll hit the ground at the same time.

167:22:43 Scott: How about that!

167:22:45 Allen: How about that! (Applause in Houston)

167:22:46 Scott: Which proves that Mr. Galileo was correct in his findings.

Newton's Law. Using this footage, we can also attempt to confirm another famous bit of physics: Newton's law of universal gravitation. Newton's laws predict that any object dropped near the surface of the Moon should fall

$$\frac{1}{2} G \frac{M}{R^2} t^2 \text{ meters}$$

after t seconds, where G is a universal constant, M is the moon's mass in kilograms, and R is the moon's radius in meters. So if we know G, M, and R, then Newton's laws let us predict how far an object will fall over any amount of time.

To verify the accuracy of this law, we will calculate the difference between the predicted distance the hammer drops and the actual distance. (If they are different, it might be because Newton's laws are wrong, or because our measurements are imprecise, or because there are other factors affecting the hammer for which we haven't accounted.)

Someone studied the video and estimated that the hammer was dropped 113 cm from the surface. Counting frames in the video, the hammer falls for 1.2 seconds (36 frames).

Question 3.3.1. Complete the code in the next cell to fill in the *data* from the experiment.

In [22]:
```
# t, the duration of the fall in the experiment, in seconds.
# Fill this in.
time = 1.2

# The estimated distance the hammer actually fell, in meters.
# Fill this in.
estimated_distance_m = ...
```

In [23]:
```
_ = lab01.grade('q331')
```

Question 3.3.2. Now, complete the code in the next cell to compute the difference between the predicted and estimated distances (in meters) that the hammer fell in this experiment.

This just means translating the formula above ($\frac{1}{2} G \frac{M}{R^2} t^2$) into Python code. You'll have to replace each variable in the math formula with the name we gave that number in Python code.

In [24]:
```
# First, we've written down the values of the 3 universal
# constants that show up in Newton's formula.

# G, the universal constant measuring the strength of gravity.
gravity_constant = 6.674 * 10**-11

# M, the moon's mass, in kilograms.
moon_mass_kg = 7.34767309 * 10**22

# R, the radius of the moon, in meters.
moon_radius_m = 1.737 * 10**6

# The distance the hammer should have fallen over the
# duration of the fall, in meters, according to Newton's
# law of gravity.  The text above describes the formula
# for this distance given by Newton's law.
# **YOU FILL THIS PART IN.**
predicted_distance_m = ...

# Here we've computed the difference between the predicted
# fall distance and the distance we actually measured.
# If you've filled in the above code, this should just work.
difference = predicted_distance_m - estimated_distance_m
difference
```

In [12]:
```
_ = lab01.grade('q332')
```

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

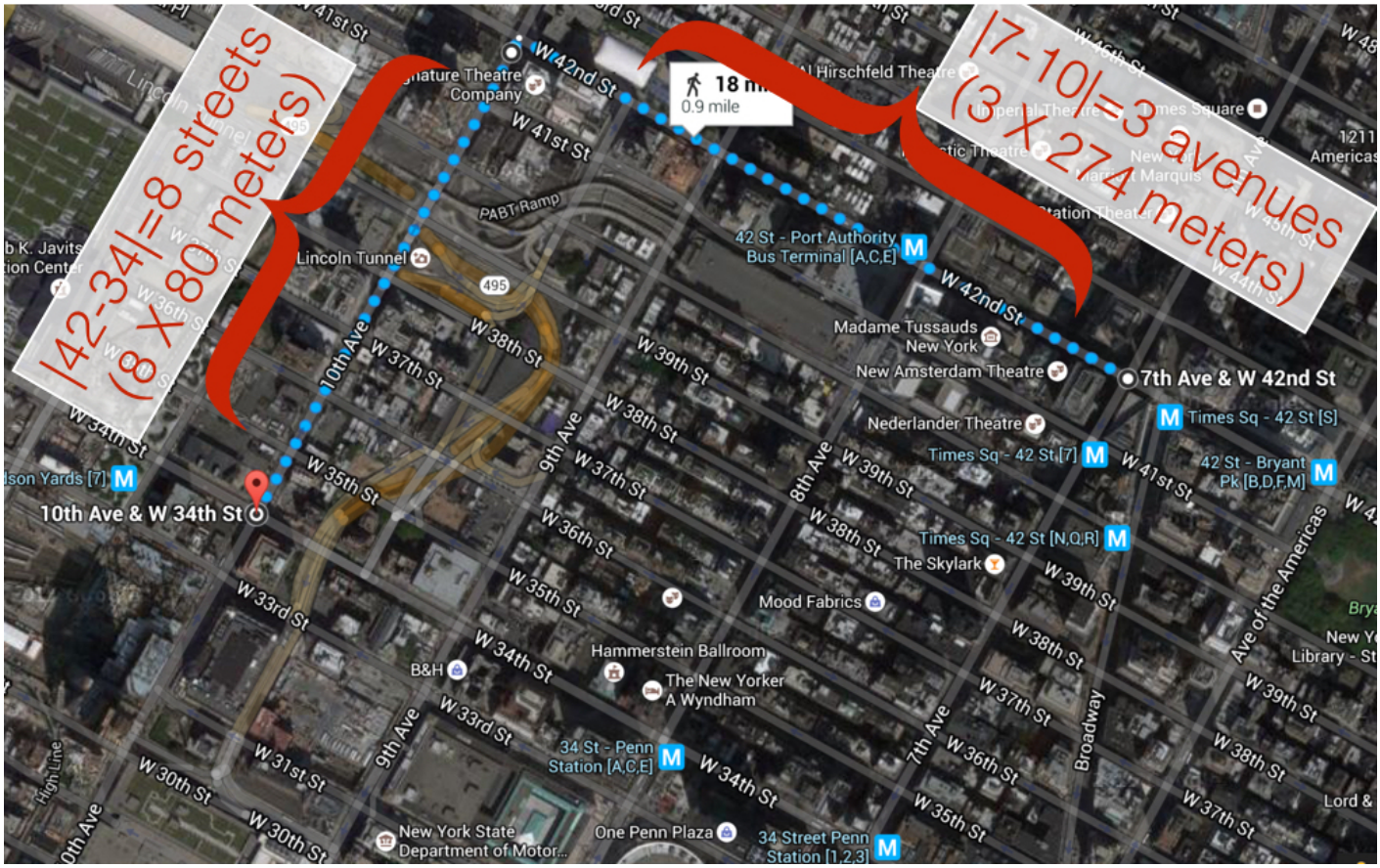
In [ ]:
```
abs(-5)
```

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 4.1.1. 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 [25]:
```
# 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 = ...

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_avenues_away

# 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
```

Be sure to run the next cell to test your code.

In [ ]:
```
_ = lab01.grade('q411')
```

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

5. Understanding nested expressions

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

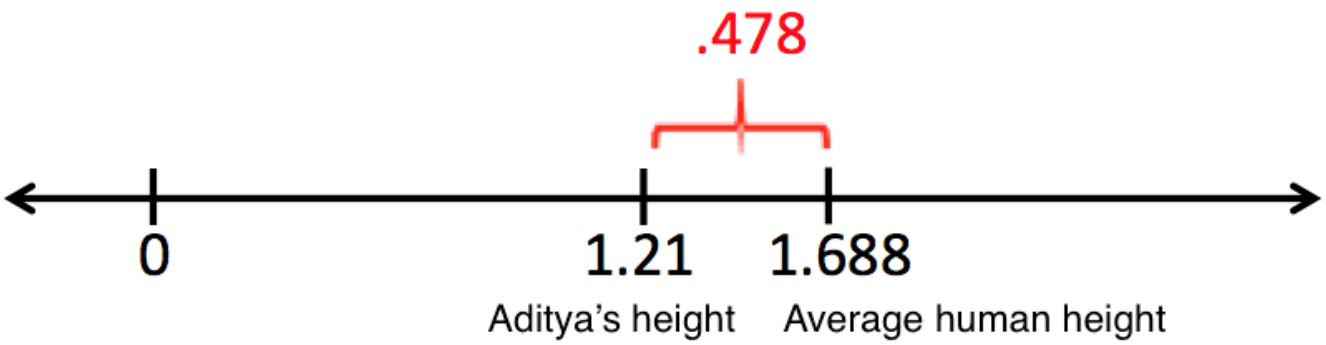
```
abs(42-34)
```

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 (http://press.endocrine.org/doi/full/10.1210/jcem.86.9.7875?ck=nck&) 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)
```

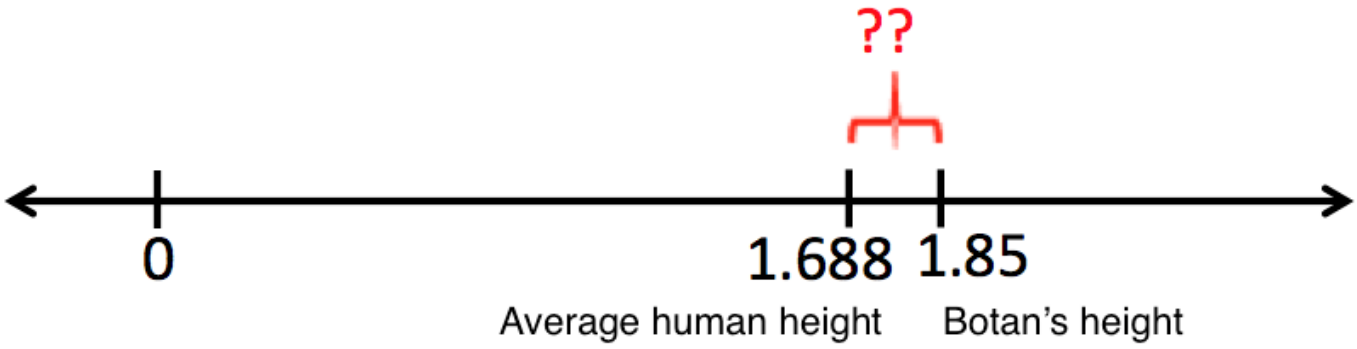
What's going on here? `abs` takes just one argument, so the stuff inside the parentheses is all part of that *single argument*. Specifically, the argument is the value of the expression `1.21 - 1.688`. The value of that expression is `-.478`. That value is the argument to `abs`. The absolute value of 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 5.1. 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 [ ]:
```
# 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 = ...

# 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
```

In [ ]:
```
_ = lab01.grade('q51')
```

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 [26]:
```
# 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 heig
hts:
biggest_distance_m = max(abs(aditya_height_m - average_adult_human_heigh
t_m), abs(botan_height_m - average_adult_human_height_m))

# Print out our results in a nice readable format:
print("The biggest distance from the average height among these two peop
le is", biggest_distance_m, "meters.")
```

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.

You can run the next cell to see a slideshow of that process.

In [27]:
```
from IPython.display import IFrame
IFrame('https://docs.google.com/presentation/d/1urkX-nRsD8VJvcOnJsjmCy0J
pv752Ssn5Pphg2sMC-0/embed?start=false&loop=false&delayms=3000', 800,
600)
```

Ok, your turn.

Exercise 5.1.1. Given the heights of the last three U.S. Presidents, 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 `obama`, `clinton`, and `bush`.

In [29]:
```
# The three Presidents' heights, in meters:
obama = 1.85
clinton = 1.88
bush = 1.82

# We look at all 3 pairs of heights, compute the absolute difference bet
ween
# each pair, and then find the smallest of those 3 absolute differences.
# The last part of the code that does this is left to you.  Replace the
# "..." with something appropriate.
min_height_difference = ...
```

In [28]:
```
_ = lab01.grade('q511')
```

You're done with Lab 1! Be sure to run the tests and verify that they all pass, then choose **Save and Checkpoint** from the **File** menu, then run this final cell.

In [ ]:
```
# Run this cell to submit your work *after* you have passed all of the t
est cells.
# It's ok to run this cell multiple times. Only your final submission wi
ll be scored.

!TZ=America/Los_Angeles ipython nbconvert --output=".lab01_$(date +%m%d
_%H%M)_submission.html" lab01.ipynb
```