Lab 2

Welcome to lab 2! Last time, we had our first look at Python and Jupyter notebooks. So far, we've only used Python to manipulate numbers. In this lab, you'll 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*. Methods are very similar to functions, but they look a little different because they're tied to pieces of data (like text or numbers).

Last, you'll see how to work with datasets in Python -- *collections* of data like text or numbers. In this class, we principally use two kinds of collections:

- **Arrays:** An array is a collection of many pieces of a single kind of data, kept in order. An array is like a single column in an Excel spreadsheet.
- **Tables:** A table is a collection of many pieces of different kinds of data. Each kind of data is in its own *column*. A table is like an entire Excel spreadsheet.

This week is about arrays. You'll see tables next week.

First, execute the following cell to set up the automatic tests for this lab.

In [16]:
```
# Test cell; please do not change!
from client.api.assignment import load_assignment
lab02 = load_assignment('lab02.ok')
```

1. Review: The building blocks of Python code

The two building blocks of Python code are *expressions* and *statements*. An expression is a piece of code that is self-contained and (usually) *has a value*. Here are some expressions:

```
3
2 - 3
abs(2 - 3)
max(3, pow(2, abs(2 - 3) + pow(2, 2)))
```

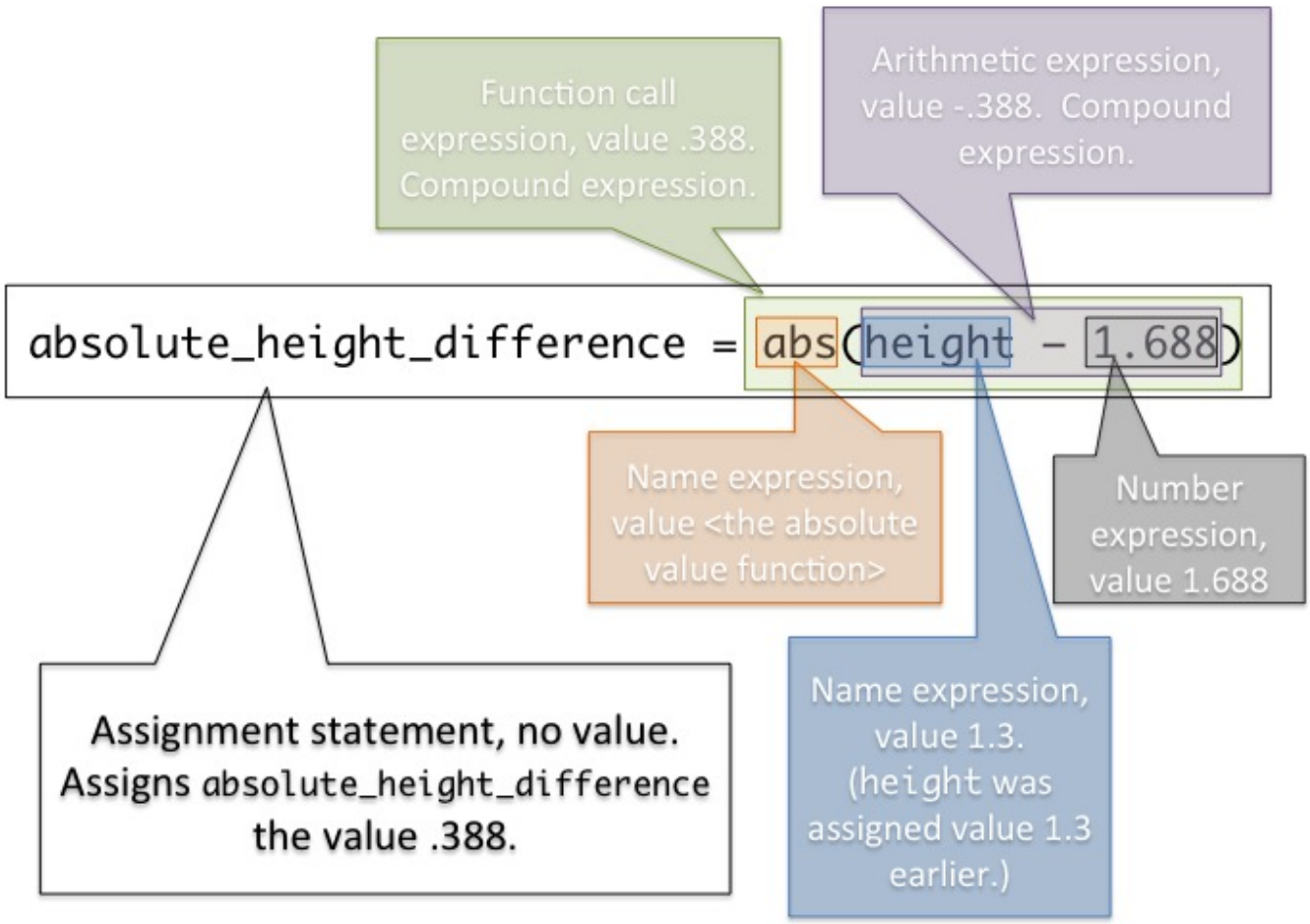
All these expressions but the first are *compound expressions*, meaning that they are actually combinations of several smaller expressions. `2 + 3` combines the expressions `2` and `3` by addition. In that example, `2` and `3` are called *subexpressions* because they're expressions that are part of a larger expression.

A *statement* is a whole line of code. Some statements are just expressions. Others *make something happen* rather than *having a value*. After they are run, something in the world has changed. For example, an assignment statement assigns a value to a name. Here are some assignment statements:

```
height = 1.3
the_number_five = abs(-5)
absolute_height_difference = abs(height - 1.688)
```

A key idea in programming is that large, interesting things can be built by combining many simple, uninteresting things. The key to understanding a complicated piece of code is breaking it down into its simple components.

For example, a lot is going on in the last statement above, but it's really just a combination of a few things. This picture describes what's going on.



Question 1.1. In the next cell, assign the name `new_year` to the larger number among the following two numbers: $|2^5 - 2^{11}|$ and $5 \times 13 \times 31$. Try to use just one statement (one line of code).

In [17]:
```
new_year = ...
new_year
```

Check your work by executing the next cell.

In [18]:
```
_ = lab02.grade('q11')
```

2. Text

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

A snippet of text is represented by a *string value* in Python. The word "*string*" is a computer science term for a sequence of characters. A string might contain a single character, a word, a sentence, or a whole book. Strings have quotation marks around them. Single quotes (`'`) and double quotes (`"`) are both valid. 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 [19]:
```
print("I <3", 'Data Science')
```

`print` prints all of its arguments together, separated by single spaces.

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

Question 2.1. The cell below contains an unfinished joke. Replace the `...` with a single-word string so that the final expression prints its punchline. Here's a picture of the joke:

In [20]:
```
why_was = "Because"
six = ...
afraid_of = "eight"
seven = "nine"
print(why_was, six, afraid_of, seven)
```

In [21]:
```
_ = lab02.grade('q21')
```

2.1. String Methods

Strings can be transformed using *methods*, which are functions that involve an existing string and some other arguments. For example, the `replace` method replaces all instances of some part of a string with some replacement. A method is invoked on a string by placing a `.` after the string value, then the name of the method, and finally parentheses containing the arguments.

```
<string>.<method name>(<argument>, <argument>, ...)
```

Otherwise, a method is pretty similar to a function.

Try to predict the output of these examples, then execute them.

In [22]:
```
# Replace one letter
'Hello'.replace('o', 'a')
```

In [23]:
```
# Replace a sequence of letters, which appears twice
'hitchhiker'.replace('hi', 'ma')
```

Once a name is bound to a string value, methods can be invoked on that name as well. The name doesn't change in this case, so a new name is needed to capture the result.

In [25]:
```
sharp = 'edged'
hot = sharp.replace('ed', 'ma')
print('sharp =', sharp)
print('hot =', hot)
```

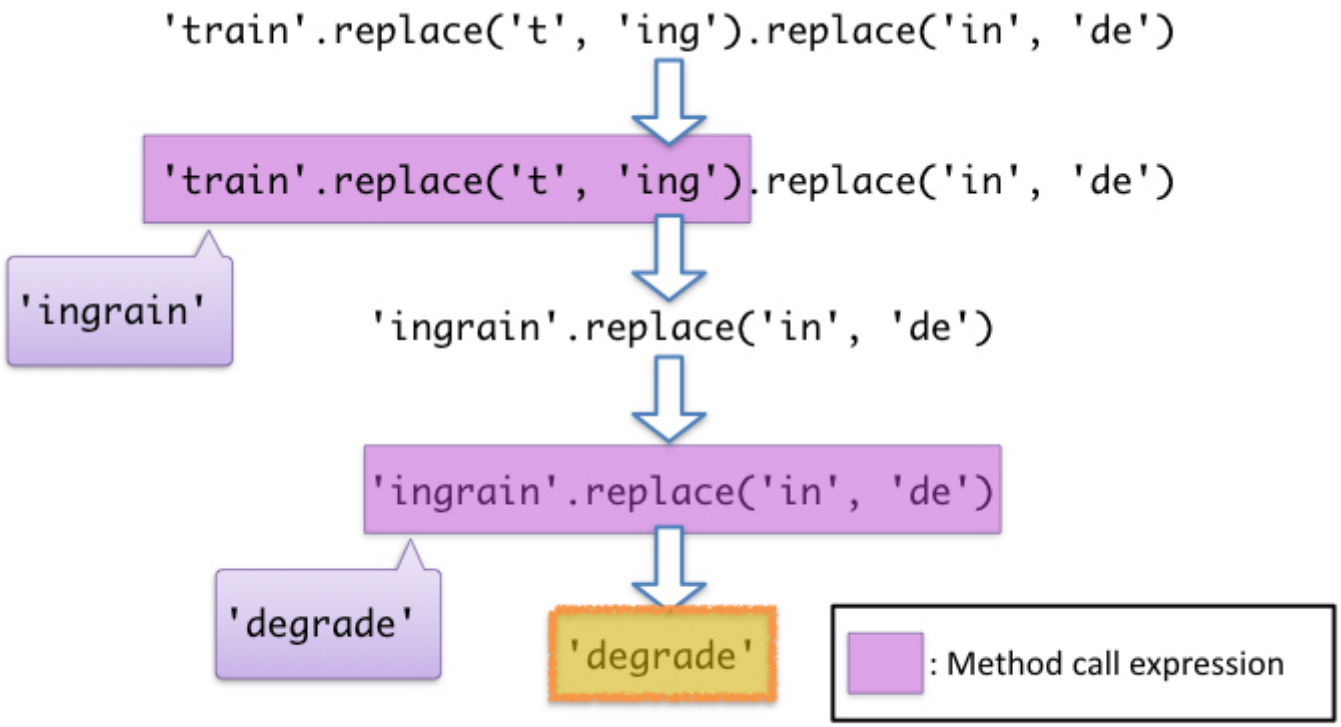
Remember that you could call functions on the results of other functions. For example,

```
max(abs(-5), abs(3))
```

has value 5. Similarly, you can invoke methods on the results of other method (or function) calls.

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

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



Question 2.1.1. Assign strings to the names `you` and `this` so that the final expression evaluates to a 10-letter English word with three double letters in a row.

Hint: After you guess at some values for `you` and `this`, it's helpful to see the value of the variable `the`. Try printing the value of `the` by adding a line like this:

```
print(the)
```

Hint 2: Run the tests if you're stuck. They'll often give you help.

In [26]:
```
you = ...
this = ...
a = 'beeper'
the = a.replace('p', you)
the.replace('bee', this)
```

In [27]:
```
_ = lab02.grade('q211')
```

Other string methods do not take any arguments at all, because the original string is all that's needed to compute the result. In this case, parentheses are still needed, but there's nothing in between the parentheses. Here are some methods that work that way:

Method name	Value
`lower`	a lowercased version of the string
`upper`	an uppercased version of the string
`capitalize`	a version with the first letter capitalized
`title`	a version with the first letter of every word capitalized

In [28]:
```
'unIverSITy of caliFORnia'.title()
```

Methods can also take arguments that aren't strings. For example, strings have a method called `zfill` that "pads" them with the character `0` so that they reach a certain length. This can be useful for displaying numbers in a uniform format:

In [29]:
```
print("5.12".zfill(6))
print("10.50".zfill(6))
```

All these string methods are useful, 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. (Stack Overflow (http://stackoverflow.com) has a huge database of answered questions.) You can refer back to this lab or the textbook for the ones we mention. Exams for this course will include documentation for any functions you need to use.

2.2. 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 [30]:
```
8 + "8"
```

However, there are built-in functions to convert numbers to strings and strings to numbers.

```
int:   Converts a string of digits to an int value
float: Converts a string of digits, perhaps with a decimal point, to a float
 value
str:   Converts any value to a string
```

Try to predict what the following cell will evaluate to, then evaluate it.

In [1]:
```
8 + int("8")
```

Suppose you're writing a program that looks for dates in a text, and you want your program to find the difference 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 2.2.1. Finish the code below to compute the difference between the two years. Don't just write the numbers `1618` and `1648` (or `30`); use a conversion function to turn the given text data into numbers.

In [2]:
```
# 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 = ...
difference
```

In [33]:
```
_ = lab02.grade('q221')
```

2.3. Strings as function arguments

String values, like numbers, can be arguments to functions and can be returned by functions. The function `len` takes a single string as its argument and returns the number of characters in the string: its **len**gth. Note that it doesn't count *words*. `len("one small step for man")` is 22, not 5.

Question 2.3.1. Use `len` to find out the number of characters in the very long string in the next cell. (It's 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).) The length of a string is the total number of characters in it, including things like spaces and punctuation. Assign `sentence_length` to that number.

In [34]:
```
a_very_long_sentence = "The representatives of the French people, organi
zed as a National Assembly, believing that the ignorance, neglect, or co
ntempt 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 sol
emn declaration the natural, unalienable, and sacred rights of man, in o
rder that this declaration, being constantly before all the members of t
he Social body, shall remind them continually of their rights and dutie
s; in order that the acts of the legislative power, as well as those of
 the executive power, may be compared at any moment with the objects and
 purposes of all political institutions and may thus be more respected,
 and, lastly, in order that the grievances of the citizens, based hereaf
ter upon simple and incontestable principles, shall tend to the maintena
nce of the constitution and redound to the happiness of all."
sentence_length = ...
sentence_length
```

In [35]:
```
_ = lab02.grade('q231')
```

3. Importing code

> What has been will be again,
> what has been done will be done again;
> there is nothing new under the sun.

Most programming involves work that is very similar to work that has been done before. Since writing code is time-consuming, it's good to rely on others' published code when you can. Rather than copy-pasting, Python allows us to *import* other code, creating a *module* that contains all of the names created by that code.

Python includes many useful modules that are just an `import` away. We'll look at the `math` module as a first example.

Suppose we want to very accurately compute the area of a circle with radius 5 meters. For that, we need the constant π, which is roughly 3.14. Conveniently, the `math` module defines `pi` for us:

In [38]:
```
import math
radius = 5
area_of_circle = radius**2 * math.pi
area_of_circle
```

`pi` is defined inside `math`, and the way that we access names that are inside modules is by writing the module's name, then a dot, then the name of the thing we want:

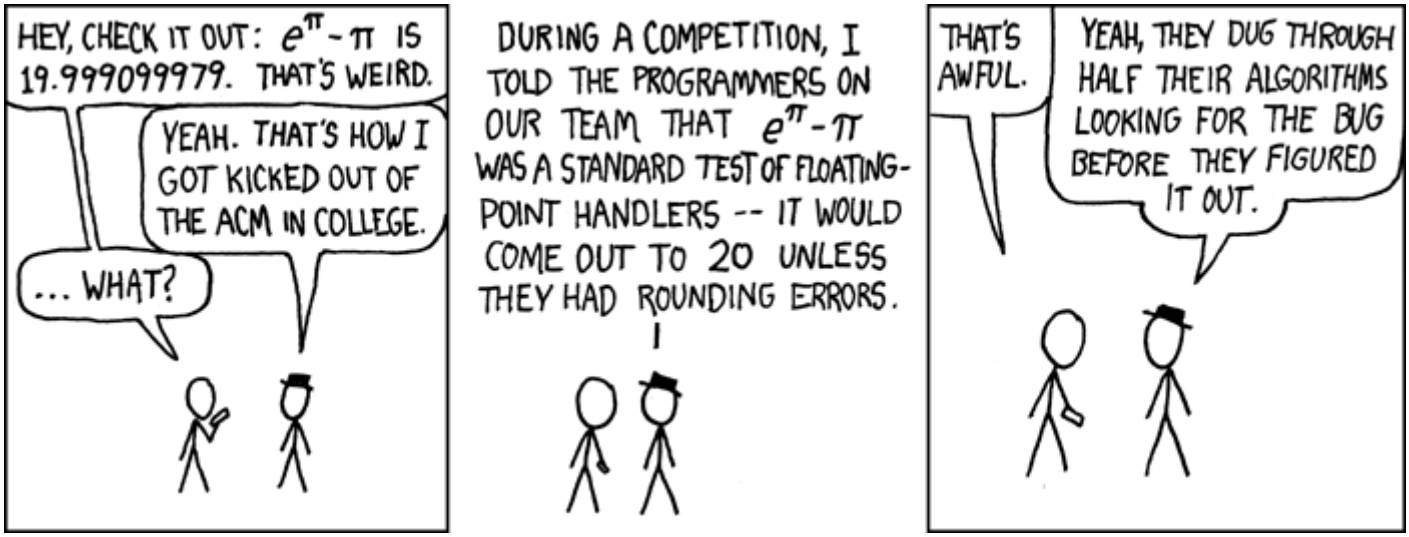
```
<module name>.<name>
```

In order to use a module at all, we must first write the statement `import <module name>`. That statement creates a module object with things like `pi` in it and then assigns the name `math` to that module. Above we have done that for `math`.

Question 3.1. `math` also provides the name `e` for the base of the natural logarithm, which is roughly 2.71. Compute $e^{\pi} - \pi$, giving it the name `near_twenty`.

In [39]:
```
near_twenty = ...
near_twenty
```

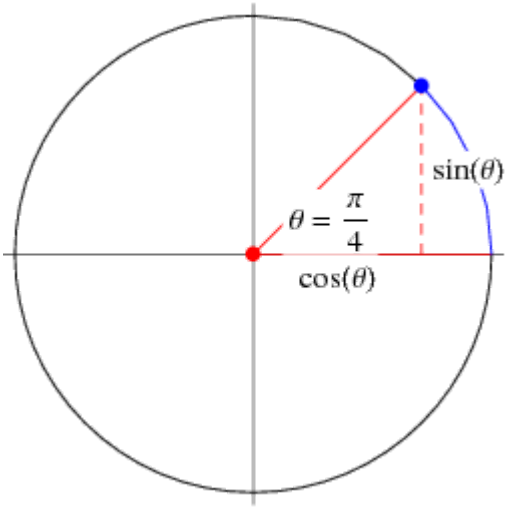
In [40]:
```
_ = lab02.grade('q31')
```



3.1. Importing functions

Modules can provide other named things, including functions. For example, `math` provides the name `sin` for the sine function. Having imported `math` already, we can write `math.sin(3)` to compute the sine of 3. (Note that this sine function considers its argument to be in radians (https://en.wikipedia.org/wiki/Radian), not degrees. 180 degrees are equivalent to π radians.)

Question 3.1.1. A $\frac{\pi}{4}$-radian (45-degree) angle forms a right triangle with equal base and height, pictured below. If the hypotenuse is 1, then the height is $\sin(\frac{\pi}{4})$. Compute that using `sin` and `pi` from the `math` module. Give the result the name `sine_of_pi_over_four`.



(Source: Wolfram MathWorld (http://mathworld.wolfram.com/images/eps-gif/TrigonometryAnglesPi4_1000.gif))

In [41]:
```
sine_of_pi_over_four = ...
sine_of_pi_over_four
```

In [42]:
```
_ = lab02.grade('q311')
```

For your reference, here are some more examples of functions from the `math` module:

In [43]:
```
# Calculating factorials.
math.factorial(5)
```

In [44]:
```
# Calculating logarithms (the logarithm of 8 in base 2).
# The result is 3 because 2 to the power of 3 is 8.
math.log(8, 2)
```

In [45]:
```
# Calculating square roots.
math.sqrt(5)
```

In [46]:
```
# Calculating cosines.
math.cos(math.pi)
```

A function that displays a picture

People have written Python functions that do very cool and complicated things, like crawling web pages for data, transforming videos, or doing machine learning with lots of data. Now that you can import things, when you want to do something with code, first check to see if someone else has done it for you.

Let's see an example of a function that's used for downloading and displaying pictures.

The module `IPython.display` provides a function called `Image`. `Image` takes a single argument, a string that is the URL of the image on the web. It returns an *image* value that this Jupyter notebook understands how to display. To display an image, make it the value of the last expression in a cell, just like you'd display a number or a string.

Question 3.1.2. In the next cell, we've imported the module `IPython.display`. Use its `Image` function to display the image at this URL:

```
https://upload.wikimedia.org/wikipedia/commons/thumb/8/8c/David_-_The_Death_
of_Socrates.jpg/1024px-David_-_The_Death_of_Socrates.jpg
```

Give the name `art` to the output of the function call, then make the last line of the cell `art` to see the image. (It might take a few seconds to load the image. It's a painting called *The Death of Socrates* by Jacques-Louis David, depicting events from a philosophical text by Plato.)

In [47]:
```
import IPython.display
# Replace the ... with a call to the Image function
# in the IPython.display module, which should produce
art = ...
art
```

In [48]:
```
_ = lab02.grade('q312')
```

5. Arrays

Up to now, we haven't done much that you couldn't do yourself by hand, without going through the trouble of learning Python. 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's the result of multiplying each number in `billions_of_numbers` by .18 (18%). Arrays are not limited to numbers; we can also put all the words in a book into an array of strings.

5.1. 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 do it the hard way.

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:

In [1]:
```
make_array(0.125, 4.75, -1.3)
```

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

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

Question 5.1.1. Make an array containing the numbers 1, 2, and 3, in that order. Name it `small_numbers`.

In [65]:
```
small_numbers = ...
small_numbers
```

In [66]:
```
_ = lab02.grade('q511')
```

Question 5.1.2. 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.

In [2]:
```
interesting_numbers = ...
interesting_numbers
```

In [68]:
```
_ = lab02.grade('q512')
```

Question 5.1.3. Make an array containing the five strings `"Hello"`, `","`, `" "` (that's just a single space inside quotes), `"world"`, and `"!"`. 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.

In [69]:
```
hello_world_components = ...
hello_world_components
```

In [70]:
```
_ = lab02.grade('q513')
```

5.1.1. `np.arange`

Arrays are provided by a package called NumPy (http://www.numpy.org/) (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 `arange` stops *before* the stop value is reached.)

Question 5.1.1.1. 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, and we find that the data don't include the timestamps of the readings (the time at which each one was taken). We'll assume 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.

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: 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. Check the length of `collection_times` and make sure it has 31×24 elements.

In [71]:
```
collection_times = ...
collection_times
```

In [72]:
```
_ = lab02.grade('q5111')
```

5.2. 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 (http://www.census.gov/population/international/data/worldpop/table_population.php).)

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.

In [43]:
```
# Don't worry too much about what goes on in this cell.
from datascience import Table
population = Table.read_table("world_population.csv").column("Populatio
n")
population
```

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

In [44]:
```
population.item(0)
```

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.

In [45]:
```
# The third element in population is the population
# in 1952.
population_1950 = population.item(2)
population_1950
```

In [46]:
```
# The thirteenth element in population is the population
# in 1962 (which is 1950 + 12).
population_1962 = population.item(12)
population_1962
```

In [47]:
```
# The 66th element is the population in 2015.
population_2015 = population.item(65)
population_2015
```

In [48]:
```
# 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
```

In [49]:
```
# Since make_array returns an array, we can immediately
# call .item(3) on it to get its 4th element.
make_array(-1, -3, 4, -2).item(3)
```

Question 5.2.1. Set `population_1973` to the world population in 1973, by getting the appropriate element from `population` using `item`.

In [50]:
```
population_1973 = ...
population_1973
```

In [ ]:
```
_ = lab02.grade('q521')
```

Run the next cell to visualize the elements of `population` and their indices. You'll learn next week how to make tables like this!

In [27]:
```
Table.read_table("world_population.csv").show()
```

Question 5.2.2. What's the index of the 31st item in `population`? Try to answer the question without looking at the table or the data!

In [28]:
```
index_of_31st_item = ...
```

In [ ]:
```
_ = lab02.grade("q522")
```

5.3. 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 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?

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:

In [ ]:
```
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))
...
```

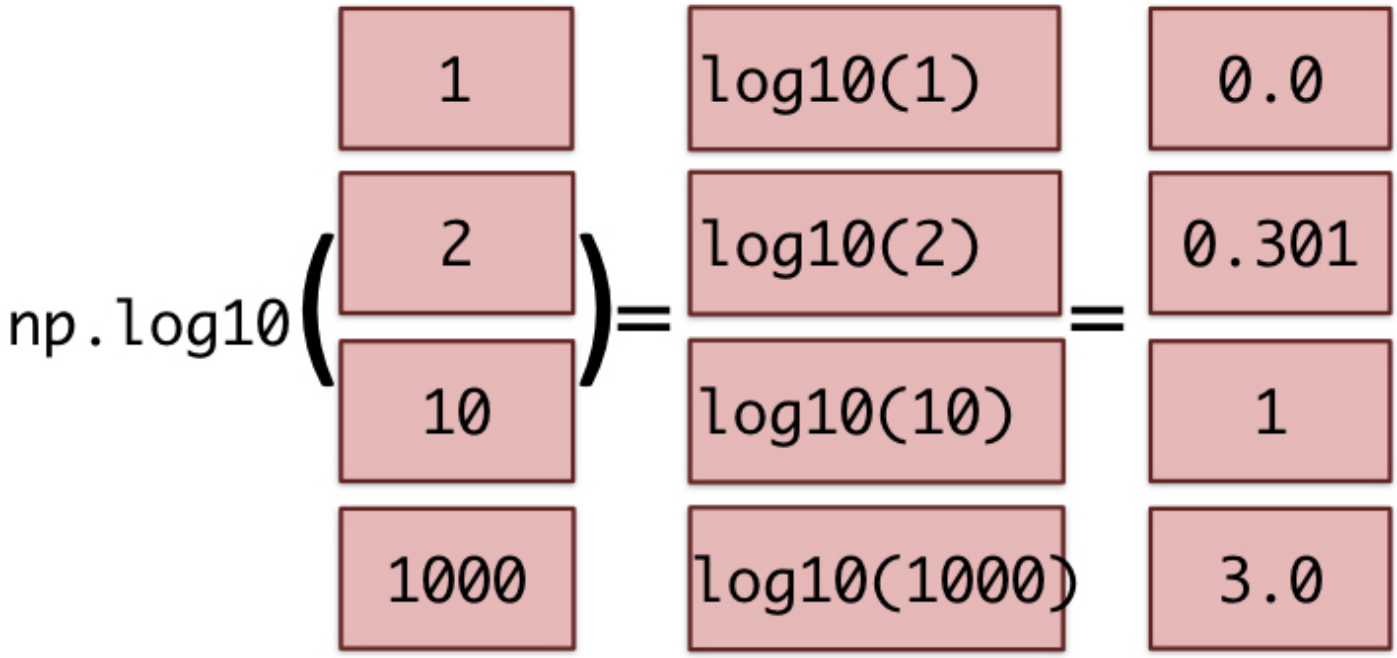
But this is tedious and doesn't really take advantage of the fact that we are using a computer.

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 5.3.1. 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.

In [57]:
```
population_magnitudes = ...
population_magnitudes
```

In [86]:
```
_ = lab02.grade('q531')
```



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`.

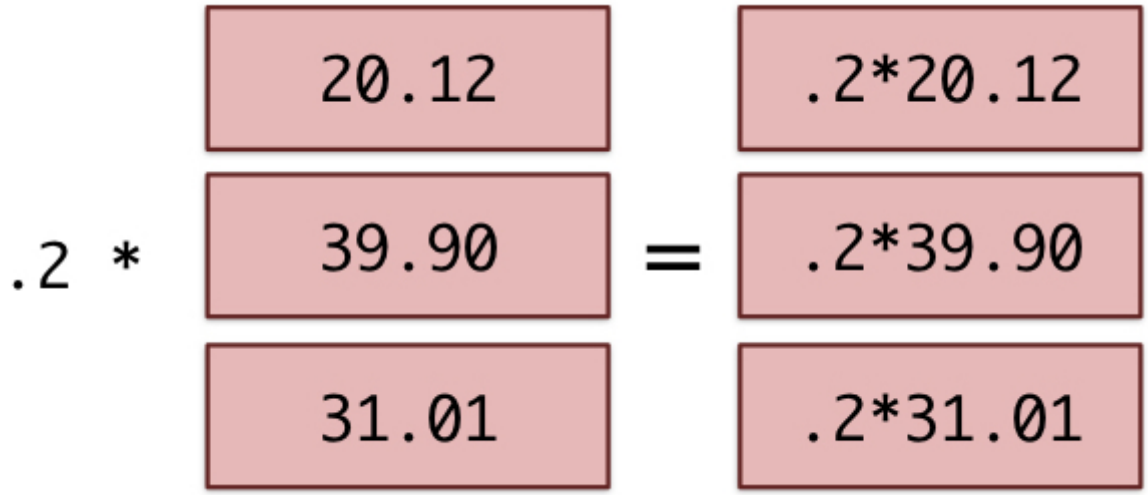
Arithmetic

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

In [ ]:
```
population_in_billions = world_population / 1000000000
population_in_billions
```

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 [31]:
```
restaurant_bills = Table.read_table("restaurant_bills.csv").column("Bil
l")
print("Restaurant bills:", restaurant_bills)
tips = .2 * restaurant_bills
print("Tips:", tips)
```



Question 5.3.2. 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`.

In [32]:
```
total_charges = ...
total_charges
```

Question 5.3.3. `more_restaurant_bills.csv` contains 100,000 bills! Compute the total charge for each one. How is your code different?

In [41]:
```
more_restaurant_bills = Table.read_table("more_restaurant_bills.csv").co
lumn("Bill")
more_total_charges = 1.2 * more_restaurant_bills
more_total_charges
```

In [40]:
```
Table().with_column("Bill",
np.round(15*abs(np.random.normal(size=100000)) + 3, 2)).to_csv("more_res
taurant_bills.csv")
```

Congratulations, you're done with lab 2! Be sure to **run all the tests** (the next cell has a shortcut for that), and then **run the last cell to submit your work**.

In [ ]:
```
# For your convenience, you can run this cell to run all the tests at on
ce!
import os
_ = [lab02.grade(q[:-3]) for q in os.listdir("tests") if
q.startswith('q')]
```