Playing with Strings

As you may or may not have guessed, python doesn't work well when playing around with different data types. For instance:

```
In  [1]: 8 + "8"      #from Data8 Lab
Out [1]: Error
```

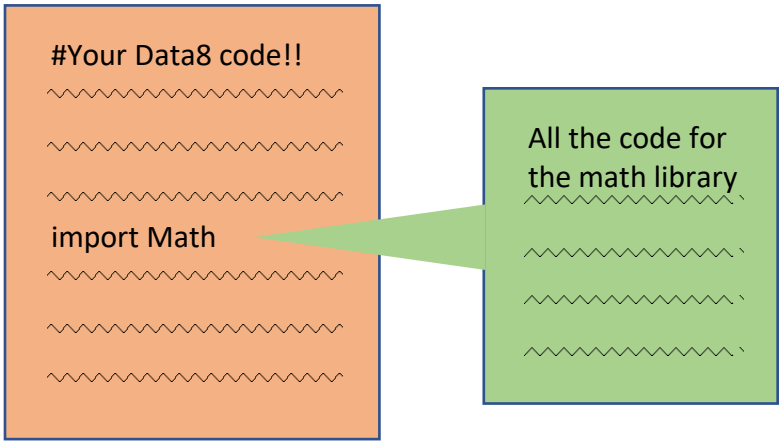
This line errors out because it's a computer. As a human you see the number 8 plus the number 8. However, a computer sees a number plus a string that could be a word. Adding a number and a word doesn't make sense, so to stay on the safe side, python errors out until it's sure it's adding two numbers together. Luckily, we have these nice functions that help us convert strings to numbers:

Takes away quotes	`int(num)`	Converts any string of **digits** to an integer
	`float(num)`	Converts any string of **digits** (maybe with a decimal) to a float (number with decimals)
Adds quotes	`str(anything)`	Converts anything inside into a string

```
In  [2]: 8 + int("8")      #from Data8 Lab
Out [2]: 16
```

So if we use the int() function on the string 8, then we can treat "8" as an integer and add them together.

Importing Code



So how you can think of importing code is that when you type `import <library name>` Then it's like you stick that whole library and all it's code into you're jupyter notebook. However, the one caveat is that in order to access the variables and functions of that library, you must say:

`<library name>.<function name> OR <library name>.<variable name>`

For instance, pi is not an available variable in python. However, you have access to the variable and it's correct value if you import the math library:

```
In  [3]: import math
         math.pi
Out [3]: 3.141592653589793
```

You also have access to its methods. However, sometimes there are some things you won't expect like this:

```
In  [4]: import math
         e = math.e
         pi = math.pi
         i = (-1) ** .5
         math.pow(e, i * pi)
Out [4]: Error, cannot deal with complex numbers
```

But you can get the right answer without the library:

```
In  [5]: import math
         e = math.e
         pi = math.pi
         i = (-1) ** .5
         e ** (i * pi)
Out [5]: (-1.0000000000000002+1.2246467991473535e-16j)
```

Arrays

Arrays are a sequence of "things" that have:

1. An ordering to them such as a numbered ordering (element 0, element 1, etc)
2. A fixed size
3. All the "things" are of the same data type

To start working with arrays we have to import a library we'll be using the rest of the semester:

```
from datascience import *
```

Then we can use the `make_array` function to create arrays. Here are two examples:

```
In  [6]: from datascience import *
         make_array(2, 3, 45, 2, 34)
Out [6]: array([2, 3, 45, 2, 34])
```

```
In  [7]: from datascience import *
         make_array(2, 'Go Bears!')
Out [7]: array(['2', 'Go Bears!'], dtype='<U21')
```

So you can see that making arrays where everything is the same type is totally valid. You can also make arrays of different types, however the resulting array will convert everything to the same type. So in the second example above, the number 2 is actually converted to a string. Python does this because it doesn't know if the string can be a number, but anything can be a string. There are certain rules that python goes off of, but if you make an array that has different types, it's best to check what type it is by checking the type of the first element (by using the function `type(<element>)`

Numpy and np.arange

Just how we imported datascience above, we import numpy to have some more functions with arrays.

```
import numpy as np
```

This means that we imported numpy and everytime we want to access something from the numpy library, we just have to use `np.<expression>` instead of `numpy.<expression>`

`np.arange` is really cool because it creates an array of a certain range. The syntax goes like this:

`np.arange(<start>, <stop>, <step>)`

This means to create an array which starts at some value and stops at (but does not include) a value. Step is the rate at which the array is created. So it would take the starting value, then add step to that as many times as possible until it was greater than or equal to the stop value. Let's see some examples:

```
In  [8]: import numpy as np
         np.arange(1, 3, 1)
Out [8]: array([1, 2])
```

```
In  [9]: import numpy as np
         np.arange(1, 3)     # the default step value is 1
Out [9]: array([1, 2])
```

```
In  [10]: import numpy as np
          np.arange(3)       # the default start value is 0
Out [10]: array([0, 1, 2])   # must specify a stop value
```

```
In  [11]: import numpy as np
          np.arange(1, 10, 2) # start at 1 and add 2 until we get over 10
Out [11]: array([1, 3, 5, 7, 9]) # don't go over or equal to stop value
```

When applying numpy functions to arrays, it applies the function element-wise, unless the function returns a single value.

[Element-wise: the function does the operation to each element of the array individually]

For instance, log10 will compute the log10 of all elements in an array. Likewise, adding 5 to an array would add 5 to every element of the array. However, functions like np.average that only return one value, will not compute things element-wise. This also makes sense since the average of a single element is just that element itself.

```
In  [12]: import numpy as np
          np.arange(1, 10, 2) + 5
Out [12]: array([6, 8, 10, 12, 14])
```

All of the operations we've learned thus far can be used to manipulate arrays, and this is done element-wise.

Here's a great example of how np.log10 works element-wise on arrays (taken from Data8 lab).