The **data type** of a variable determines how it will behave. Python is a **dynamically typed** language, which means that the type of any given variable may change as your program runs, and this *often* means that you don't need to think about what type a variable has. But sometimes when you encounter an error, it can be helpful to know a bit about data types.

To find the type of a given value (which could be any expression), you can use the built-in `type` function, as in

```
print(type('hello world'))
```

Take a few moments, and use your computers to examine the types of a few values.

Python has a few basic data types that you have already used (and one that you haven't used):

str A *string* type, which is a sequence of characters, as in `"hello"` or `'hello'`. You have probably used these for printing.

int An integer value.

float A real number. The name stands for *floating point* which relates to the internal representation, which is like scientific notation, where the decimal point will change location or "float."

complex A complex number. We won't be using these very often. Even when we do start using complex quantities in quantum mechanics, it will work out better to use real numbers for the real and imaginary parts.

When you do arithmetic `int`s and `float`s will behave similarly, and are often (but not always) interchangeable.

1 Data structures

A **data structure** is a type that can hold other values inside it. Python has four built-in data structures, which we will seldom use, and `numpy` introduces another that we will frequently use.

list An ordered sequence of variables, written like `[3, 2, 1, 'takeoff!']`. The variables need not have the same type, and individual elements may be modified, which modifies the list.

I will encourage you to essentially never use lists, primarily for one reason: multiplying a list by an integer causes it to be repeated. So `3 * [1, 2, 3]` gives you a value of `[1, 2, 3, 1, 2, 3, 1, 2, 3]`. This is a frequent source of errors in this class. We are often wanting to do arithmetic on collections of numbers, and lists make this hard.

tuple A tuple is like a list that you can't modify. Tuples are specified using parentheses (or sometimes with no delimiters), like

```
t = (1, 2, 3)
```

You can't modify the elements of a tuple, and you can't change its size. There are just a couple of scenarios where you will need or want to use a tuple.

dictionary A dictionary is a mapping between values and variables. They are specified using curly braces and colons:

```
instructor = {
    'name': 'David',
    'favorite color': 'blue',
    'number digits': 20,
    4: 'not favorite number',
}
print(instructor['name']) # will print 'David'
```

I encourage you never to use a dictionary in this class.

set Holds a set of values. Again, not used in this class. Is created like

```
students = set(['A', 'B', 'C'])
students.add('D')
if 'B' in students:
   print('Hello B!')
```

numpy array The data structure that we will use most frequently is the array defined in `numpy`. It is what you create when you call `np.linspace(1,10,100)`, and when you print its type you will find `numpy.ndarray`. An array is sort of like a list, but far better for numerical work.

- All the elements of an array have the same type.
- Arrays are fast to do computations with.
- Arithmetic (and most functions) on arrays does arithmetic on each element. This is refered to as *elementwise operations*. This is how you have been plotting things.
- Changing the length of an array is very inconvenient (e.g. appending).

The way that you've seen to create a numpy array is through

```
x = np.linspace(1,2,10)
print(type(x))
```

There are many other functions to create arrays.

2 Methods

Now that we have the idea of types, you are ready to learn about **methods**. A method is a kind of function that has a special syntax (which is annoying), and a special kind of power. I won't be teaching you to create your own methods, but will hopefully show you enough so that you can use methods that you come across in web searches.

Calling methods looks very much like using functions from a package (e.g. `np.linspace`), in that the method call begins with a dot.

```
x = np.linspace(0, 1, 10)
print(x.sum()) # prints 5.0
```

Here `sum` is a method for arrays which adds up all the elements. The first "argument" to a method is the thing to the left of the `.(`, and remaining arguments are given in parentheses like for an ordinary function. This `sum` method has a few optional arguments, so that e.g. we could call

```
x = np.linspace(0, 1, 10)
print(x.sum(where=x<0.35)) # prints 0.666...
```

Methods add three bits of magic beyond what you see in an ordinary function:

1. The code executed depends on the *type* of its first argument.
2. You can create multiple methods that have the same name, and can use them all in the same file. With functions you could have functions with the same name in different files, but not in the same one.
3. You can chain method calls that return values, such that you can read them from left to right in order that they happen.

   ```
   print('Hello world'.replace('Hello', 'Goodbye')) # prints 'Goodbye world'
   print('Hello world'.replace('Hello', 'Goodbye').replace('bye', ' day')) # prints 'Good
   ```

   This can be confusing at first, but at times it can be easier to read than the function version you might imagine:

   ```
   print(replace('Hello world', 'Hello', 'Goodbye')) # prints 'Goodbye world'
   print(replace(replace('Hello world', 'Hello', 'Goodbye'), 'bye', ' day')) # prints 'Go
   ```

   In particular, you can hopefully imagine that if you did several functions chained together in this way, the function name gets further and further separated from the function arguments. Using functions, you'd be wiser probably to write the code something like

   ```
   print(replace('Hello world', 'Hello', 'Goodbye')) # prints 'Goodbye world'
   goodbye = replace('Hello world', 'Hello', 'Goodbye')
   print(replace(goodbye, 'bye', ' day')) # prints 'Good day world'
   ```

   but this also has its own readability issues, because it requires you to create meaningful names for all of the intermediate values, which can be a challenge.