

Python for Neuroscientists

Week 1 - Let's go over some basics / get up to speed

Why Python?

Language Ranking: IEEE Spectrum

Rank	Language	Type	Score
1	Python	⊕ ☐ ☑	100.0
2	Java	⊕ ☐ ☑	96.3
3	C	☐ ☑ ☑	94.4
4	C++	☐ ☑ ☑	87.5
5	R	☑	81.5
6	JavaScript	⊕	79.4
7	C#	⊕ ☐ ☑ ☑	74.5
8	Matlab	☑	70.6
9	Swift	☐ ☑	69.1
10	Go	⊕ ☑	68.0
11	Arduino	☑	67.2
12	HTML,CSS	⊕	66.8
13	PHP	⊕	65.1
14	Assembly	☑	63.7
15	SQL	☑	63.4

- Easy to learn
- Free!
- Has been quickly adopted for scientific computing in both industry data science and academic research
- Extensive library support (pandas) for data manipulation
- Vast, ever-growing number of specialized libraries

A few things to know off the bat (especially if you're a MATLABer)

1.) Python is an **object-oriented programming language**.

- *If you've worked primarily in MATLAB, chances are that you've done mostly **procedural** programming, where the focus is on writing a sequential series of steps, or functions, to be executed on single variables or datasets.*
- Nearly every data type you work in Python with will be an **object** with associated data (properties) and tools (methods) - you'll often see these referred to as an object's "attributes". We mean this quite literally - if we say " $x = 29.2$," *x is an object (one called a "float") with associated attributes.*
 - This means that we'll be thinking about our data in terms of logical groupings rather than single pieces to be fed through a "machine".
 - You're probably doing this on an instinctual level already! i.e. "experiments" could be a large group, encompassing different experiment types on single mice. Each mouse also has attributes, including sex, age, data, response, etc. - while these specific values might vary between individual mice, the categories *remain the same*.

A few things to know off the bat (especially if you're a MATLABer)

2.) Python is zero-indexed (like math :P)

- Rather than starting with “1”, Python starts counting with “0.” **This is the case with most programming languages.**

- If we have the array:

[2,3,8,4,8,2,5]

“2” is the **0th** element, and “3” is the **1st** element.

A few things to know off the bat (especially if you're a MATLABer)

3.) Indentations matter...a lot.

- We won't go into this too much now, but do know that once you do much more than assigning variables, indentations are a vital part of Python syntax. Thankfully, if you're using a Python-specific IDE, they'll be taken care of for you most of the time. We're saying it now, though, so that you remember to look if you start getting weird syntax-specific bugs :)

Beyond indentations, you'll find that Python-users (some more than others...) tend to care quite a bit about something called “Pythonic” programming - style elements that make your code as simple, efficient, and readable as possible.

[What we aspire to: PEP8](#) and [FLAKE8 \(same thing, but more specific\)](#)

Sounds good? Let's jump in!

Go ahead and open a new Jupyter Notebook to follow along with - we'll
a quick break while people get set up.

Part I: Variables and Data Types in Python

If you've programmed at all before, you're familiar with variables. In Python, a variable is an object that **stores the address** of a **data type**.

To assign a variable a value, we use the “=” operator.

Let's start with **numerical** data types. In Python, these will be integers (int), floating point numbers (floats), and complex numbers.

In your notebook, assign a variable named “x” the value 10.

Part I: Variables and Data Types in Python

If you've programmed at all before, you're familiar with variables. In Python, a variable is an **object that stores the address of a data type**.

To assign a variable a value, we use the “=” operator.

Let's start with **numerical** data types. In Python, these will be integers (int), floating point numbers (floats), and complex numbers - for now, let's focus on **ints** and **floats**.

In your notebook, assign a variable named “x” the value 10, and view the value you've assigned using Python's built-in print() function.

```
In [91]: x = 10  
         print(x)
```

10

What kind of data type is x?

Part I: Variables and Data Types in Python: Integers

What kind of data type is x?

To determine the data type of an object, you can use Python's built-in “type” function:

```
In [4]: x = 10
```

```
In [5]: type(x)
```

```
Out[5]: int
```

As you can see, Python is telling us that x is an “int,” or integer.

Integers are the most basic numerical data type. They consist of positive and negative **whole numbers**. There's no length limit on an int, as long as it doesn't have a fractional component.

Part I: Variables and Data Types in Python:

Floats

What about if we want to represent a fractional number? Let's go ahead and try typing in a decimal. In your notebook, reassign `x` to any value with a decimal point (e.g. `10.01`) and return the variable's type.

Part I: Variables and Data Types in Python:

Floats

What about if we want to represent a fractional number? Let's go ahead and try typing in a decimal. In your notebook, reassign `x` to any value with a decimal point (e.g. 10.01) and return the variable's type.

```
In [6]: x = 10.01  
        type(x)
```

```
Out[6]: float
```

Neat! Python created a float for us automatically - floating point numbers, or floats, are used when a variable needs to store a fractional number.

So what's the difference between an int and a float? Why even use ints?

Part I: Variables and Data Types in Python:

Floats

```
In [6]: x = 10.01  
        type(x)
```

```
Out[6]: float
```

So what's the difference between an int and a float? Why even use ints?

1.) Memory usage - don't forget that computers store everything in **binary**. In this case, representing the place of values before and after a decimal point can get tricky, and uses more memory. In contrast, only 8 bits are needed to store every whole number between 0 and 255 - processing ints will take much less time!

2.) What are you trying to do with your data? If you're trying to keep a count of a mouse population, will you ever need a float?

Part I: Variables and Data Types in Python: the int() method

2.) What are you trying to do with your data? If you're trying to keep a count of a mouse population, will you ever need a float?

Let's stick on this idea for a second - let's say you're building a tool that keeps count of the number of experiments a lab has run, and want to make sure that people only enter whole numbers.

You can easily turn a float into an in using Python's built-in int() method. Give it a try - turn x from the last example into an int! What happens?

Part I: Variables and Data Types in Python: the int() method

2.) What are you trying to do with your data? If you're trying to keep a count of a mouse population, will you ever need a float?

Let's keep with this idea for a second - let's say you're building a tool that keeps count of the number of experiments a lab has run, and want to make sure that people only enter whole numbers.

You can easily turn a float into an in using Python's built-in int() method. Give it a try - turn x from the last example into an int! What happens?

```
In [19]: x = 10.01  
         type(x)
```

```
Out[19]: float
```

```
In [22]: int(x)
```

```
Out[22]: 10
```

Interesting! int() will round to the nearest whole number, rounding down if the post-decimal value is ≤ 5

Part I: Variables and Data Types in Python:

booleans

Moving on from numerical data types - a **boolean** is a data type that stores either a **True** or **False** value.

To get a feel for this, let's evaluate some basic arithmetic.

In Python, “=” means that you're setting a variable to a value. If you want to use the arithmetic operator “is equal to,” use “==” instead. Alternatively, to check if something is **not equal**, we type “!=”

In your notebook, check if 1 is equal to 1. What happens if you instead test that 1 is equal to 2?

Part I: Variables and Data Types in Python:

booleans

Moving on from numerical data types - a **boolean** is a data type that stores either a **True** or **False** value.

To get a feel for this, let's evaluate some basic arithmetic.

In Python, “=” means that you're setting a variable to a value. If you want to use the arithmetic operator “is equal to,” use “==” instead.

In your notebook, check if 1 is equal to 1. What happens if you instead test that 1 is equal to 2?

```
[8]: a = 1 == 1
      b = 1 == 2
      print(a, b)
      True False
```

These responses are booleans.

A few other things to know - you can cast all native python data types to a boolean using `bool()`, but nearly everything will evaluate as `True`. The only things that will evaluate to `false` are `0` and empty values.

Part I: Variables and Data Types in Python:

strings

This is a theme for this section...but if you've programmed before, you're familiar with strings :)

A string is a data type used for storing **character information**, and is denoted by being placed in quotes ' ____ '. Keep in mind that a variable on its own is not a string. If we say `cat = 'cat'`, what we've done is create a variable "cat" that is equal to the value of the string "cat".

Strings are the first data type we've introduced that has a **length** that can be returned Python's built in `len()` function.

In your notebook, create a variable "me", and set it equal to your name (as a string). Now take the `len(me)` - it should return the same number of characters as those in your name! The data types we cover from here on out can also be used with `len()`.

Part I: Variables and Data Types in Python:

using the `dir()` method to access attributes associated with strings

As we've already mentioned, pretty much **everything you work with in Python is an object**. Objects, including strings, each have a set of properties and tools/methods (called “attributes”) associated with them. Some of the methods associated with strings are really useful - they allow us to easily manipulate our string! But how do we know what attributes our string (or any other object)?

To get more information about an object in Python, you can often use the built in `dir()` method - this will return all of an object's attributes. Let's try it now.

Create a variable called “animal” in your notebook, and assign it the value ‘lemming’. Then, run the `dir()` method on it.

Part I: Variables and Data Types in Python:

methods associated with strings

```
In [40]: animal = "lionsing"  
dir(animal)
```

```
Out[40]: ['__add__',  
         '__class__',  
         '__contains__',  
         '__delattr__',  
         '__dir__',  
         '__doc__',  
         '__eq__',  
         '__format__',  
         '__ge__',  
         '__getattribute__',  
         '__getitem__',  
         '__getnewargs__',  
         '__gt__',  
         '__hash__',  
         '__init__',  
         '__init_subclass__',  
         '__iter__',  
         '__le__',  
         '__len__',  
         '__lt__',  
         '__mod__',  
         '__mul__',  
         '__ne__',  
         '__new__',  
         '__reduce__',  
         '__reduce_ex__',  
         '__repr__',  
         '__rmod__',  
         '__setattr__',  
         '__sizeof__',  
         '__str__',  
         '__subclasshook__',  
         'capitalize',  
         'center',  
         'count',  
         'encode',  
         'endswith',  
         'expandtabs',  
         'find',  
         'format',  
         'format_map',  
         'index',  
         'isalnum',  
         'isalpha',  
         'isascii',  
         'isdecimal',  
         'isdigit',  
         'isidentifier',  
         'islower',  
         'isnumeric',  
         'isprintable',  
         'isspace',  
         'istitle',  
         'isupper',  
         'join',  
         'just',  
         'lower',  
         'lstrip',  
         'maketrans',  
         'partition',  
         'replace',  
         'rfind',  
         'rindex',  
         'rjust',  
         'rpartition',  
         'split',  
         'splitlines',  
         'startswith',  
         'strip',  
         'swapcase',  
         'title',  
         'translate',  
         'upper']
```

This returns an awful lot!! For now, let's ignore anything surrounded by underscores and focus just on the attributes starting with “capitalize”.

Part I: Variables and Data Types in Python:

methods associated with strings

```
'capitalize',  
'casefold',  
'center',  
'count',  
'encode',  
'endswith',  
'expandtabs',  
'find',  
'format',  
'format_map',  
'index',  
'isalnum',  
'isalpha',  
'isascii',  
'isdecimal',  
'isdigit',  
'isidentifier',  
'islower',  
'isnumeric',  
'isprintable',  
'isspace',  
'istitle',  
'isupper',  
'join',  
'ljust',  
'lower',  
'lstrip',  
'maketrans',  
'partition',  
'replace',  
'rfind',  
'rindex',  
'rjust',  
'rpartition',  
'rsplit',  
'rstrip',  
'split',  
'splitlines',  
'startswith',  
'strip',  
'swapcase',  
'title',  
'translate',  
'upper',  
'zfill']
```

This is a bit more manageable!

So each of these attributes is called a “method.” But how do we use them? And how do we know what each of them do?

First off, know that the convention for accessing an attribute of an object is to type:

```
object.attribute()
```

Let’s see what happens when we use animal’s **capitalize** method - give it a shot in your notebook.

Part I: Variables and Data Types in Python:

using the built-in help system

```
'capitalize',  
'casefold',  
'center',  
'count',  
'encode',  
'endswith',  
'expandtabs',  
'find',  
'format',  
'format_map',  
'index',  
'isalnum',  
'isalpha',  
'isascii',  
'isdecimal',  
'isdigit',  
'isidentifier',  
'islower',  
'isnumeric',  
'isprintable',  
'isspace',  
'istitle',  
'isupper',  
'join',  
'ljust',  
'lower',  
'lstrip',  
'maketrans',  
'partition',  
'replace',  
'rfind',  
'rindex',  
'rjust',  
'rpartition',  
'rsplit',  
'rstrip',  
'split',  
'splitlines',  
'startswith',  
'strip',  
'swapcase',  
'title',  
'translate',  
'upper',  
'zfill']
```

```
In [42]: print(animal.capitalize())
```

Lemming

Neat! So it appears that the capitalize method is pretty straightforward - it takes the characters stored in your string and capitalizes the first one.

We could have predicted that based on the name of the method...but what about some of the methods that aren't quite as clear?

This is where Python has a really useful built-in help system! If you want to learn more about *all* of the methods associated with strings (or any other data type), just type `help(str)`. Try that in your notebook now.

Part I: Variables and Data Types in Python:

using the built-in help system

```
'capitalize',  
'casefold',  
'center',  
'count',  
'encode',  
'endswith',  
'expandtabs',  
'find',  
'format',  
'format_map',  
'index',  
'isalnum',  
'isalpha',  
'isascii',  
'isdecimal',  
'isdigit',  
'isidentifier',  
'islower',  
'isnumeric',  
'isprintable',  
'isspace',  
'istitle',  
'isupper',  
'join',  
'ljust',  
'lower',  
'lstrip',  
'maketrans',  
'partition',  
'replace',  
'rfind',  
'rindex',  
'rjust',  
'rpartition',  
'rsplit',  
'rstrip',  
'split',  
'splitlines',  
'startswith',  
'strip',  
'swapcase',  
'title',  
'translate',  
'upper',  
'zfill']
```

```
In [45]: help(str)  
  
Help on class str in module builtins:  
  
class str(object)  
    str(object='') -> str  
    str(bytes_or_buffer[, encoding[, errors]]) -> str  
  
    Create a new string object from the given object. If encoding or  
    errors is specified, then the object must expose a data buffer  
    that will be decoded using the given encoding and error handler.  
    Otherwise, returns the result of object.__str__() (if defined)  
    or repr(object).  
    encoding defaults to sys.getdefaultencoding().  
    errors defaults to 'strict'.  
  
    Methods defined here:  
  
    __add__(self, value, /)  
        Return self+value.
```

As you can see, typing `help(str)` returns information on every method associated with strings. You can achieve this same result by typing `str`?

What if you don't want to dig through all of that info, and just want to learn how one specific method, like `replace`, works?

You can narrow down your search by typing `help(str.replace)` or `str.replace`?

Finally, although we know it sounds silly - if you *ever* find that the built-in documentation doesn't provide enough info for you to use a method, **never hesitate to Google-search.**

Part I: Variables and Data Types in Python:

using the built-in help system

```
'capitalize',  
'casefold',  
'center',  
'count',  
'encode',  
'endswith',  
'expandtabs',  
'find',  
'format',  
'format_map',  
'index',  
'isalnum',  
'isalpha',  
'isascii',  
'isdecimal',  
'isdigit',  
'isidentifier',  
'islower',  
'isnumeric',  
'isprintable',  
'isspace',  
'istitle',  
'isupper',  
'join',  
'ljust',  
'lower',  
'lstrip',  
'maketrans',  
'partition',  
'replace',  
'rfind',  
'rindex',  
'rjust',  
'rpartition',  
'rsplit',  
'rstrip',  
'split',  
'splitlines',  
'startswith',  
'strip',  
'swapcase',  
'title',  
'translate',  
'upper',  
'zfill']
```

Let's try using a more complicated method - format.

First, figure out what format is supposed to do - I'll give you second to search built-in and online documentation.

Part I: Variables and Data Types in Python:

using the built-in help system

```
'capitalize',  
'casefold',  
'center',  
'count',  
'encode',  
'endswith',  
'expandtabs',  
'find',  
'format',  
'format_map',  
'index',  
'isalnum',  
'isalpha',  
'isascii',  
'isdecimal',  
'isdigit',  
'isidentifier',  
'islower',  
'isnumeric',  
'isprintable',  
'isspace',  
'istitle',  
'isupper',  
'join',  
'ljust',  
'lower',  
'lstrip',  
'maketrans',  
'partition',  
'replace',  
'rfind',  
'rindex',  
'rjust',  
'rpartition',  
'rsplit',  
'rstrip',  
'split',  
'splitlines',  
'startswith',  
'strip',  
'swapcase',  
'title',  
'translate',  
'upper',  
'zfill']
```

Let's try using a more complicated method - format.

First, figure out what format is supposed to do - I'll give you second to search built-in and online documentation.

```
In [50]: str.format?
```

```
Docstring:  
S.format(*args, **kwargs) -> str  
  
Return a formatted version of S, using substitutions from args and kwargs.  
The substitutions are identified by braces ('{' and '}').  
Type:      method_descriptor
```

Personally, I don't find this very useful. So I searched online, and found a few examples of how to use it:



```
# parameters in format function.  
my_string = "{}, is a {} {} science portal for {}"  
  
print (my_string.format("GeeksforGeeks", "computer", "geeks"))
```

Starting with Python 3.1, you can omit the numbers in the replacement fields, in which case the interpreter assumes sequential order. This is referred to as **automatic field numbering**:

Python

```
>>> '{}/{}/{}/{}'.format('foo', 'bar', 'baz')  
'foo/bar/baz'
```


Part I: Variables and Data Types in Python:

the format() method

```
'capitalize',  
'casefold',  
'center',  
'count',  
'encode',  
'endswith',  
'expandtabs',  
'find',  
'format',  
'format_map',  
'index',  
'isalnum',  
'isalpha',  
'isascii',  
'isdecimal',  
'isdigit',  
'isidentifier',  
'islower',  
'isnumeric',  
'isprintable',  
'isspace',  
'istitle',  
'isupper',  
'join',  
'ljust',  
'lower',  
'lstrip',  
'maketrans',  
'partition',  
'replace',  
'rfind',  
'rindex',  
'rjust',  
'rpartition',  
'rsplit',  
'rstrip',  
'split',  
'splitlines',  
'startswith',  
'strip',  
'swapcase',  
'title',  
'translate',  
'upper',  
'zfill']
```

From combining what we know from the documentation with examples that we see online, it seems like “format” takes a string and inserts additional elements into it wherever it sees curly brackets {}.

According to Python’s official documentation and the example we saw from realpython.com, users no longer have to specify the order at which new elements are incorporated - they’ll be placed one at a time into brackets in the order that they’re included in the parentheses at the end of the method.

Let’s give it a try! First, create a new variable called “my_sentence” and set it equal to the string:

‘One species that experiences occasional explosions in animal density is the’

Now, use the format method associated with my_sentence to insert the animal variable that we set to ‘lemming’ earlier.

Part I: Variables and Data Types in Python:

the format() method

'capitalize',
'casefold',
'center',
'count',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'format_map',
'index',
'isalnum',
'isalpha',
'isascii',
'isdecimal',
'isdigit',
'isidentifier',
'islower',
'isnumeric',
'isprintable',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
'lower',
'lstrip',
'maketrans',
'partition',
'replace',
'rfind',
'rindex',
'rjust',
'rpartition',
'rsplit',
'rstrip',
'split',
'splitlines',
'startswith',
'strip',
'swapcase',
'title',
'translate',
'upper',
'zfill']

```
In [55]: animal = "lemming"

mysentence = "One species that experiences occasional explosions in population density is the {}.".format(animal)

print(mysentence)

One animal species that experiences occasional explosions in population density is the lemming.
```

```
In [ ]:
```

Great work!

While this tool might feel a little silly to use when putting together a *single* sentence, you can imagine it might have a lot of utility if you want to create a lot of text output by inserting many elements one at a time!

Part I: Variables and Data Types in Python:

a few other cool things about strings

Outside of the methods that show up when you check out a string's `dir()`, strings can also be manipulated using some of the tools that you'd normally associated with other Python data types.

Make a variable called “`my_verb`”, and set it to ‘`explosion.`’ What happens if you use the `+` operator to add together `animal` and `my_verb`?

Part I: Variables and Data Types in Python:

a few other cool things about strings

Outside of the methods that show up when you check out a string's `dir()`, strings can also be manipulated using some of the tools that you'd normally associated with other Python data types.

Make a variable called “my_verb”, and set it to ‘explosion.’ What happens if you use the + operator to add together animal and my_verb?

```
In [56]: animal = 'lemming'
         my_verb = 'explosion'

         print(animal + my_verb)

lemmingexplosion
```

Ha...

But what if we want a space? This can either be added into the print function, before lemming, or after explosion :)

```
In [57]: animal = 'lemming'
         my_verb = 'explosion'
```

Part I: Variables and Data Types in Python:

a few other cool things about strings

What happens if you use the multiplier *?

Try multiplying animal by 1000.

Part I: Variables and Data Types in Python:

a few other cool things about strings

What happens if you use the multiplier *?

Try multiplying animal by 1000.

```
In [62]: print(animal * 1000)
```

[illegible]

Part I: Variables and Data Types in Python:

methods associated with strings

```
'capitalize',  
'casefold',  
'center',  
'count',  
'encode',  
'endswith',  
'expandtabs',  
'find',  
'format',  
'format_map',  
'index',  
'isalnum',  
'isalpha',  
'isascii',  
'isdecimal',  
'isdigit',  
'isidentifier',  
'islower',  
'isnumeric',  
'isprintable',  
'isspace',  
'istitle',  
'isupper',  
'join',  
'ljust',  
'lower',  
'lstrip',  
'maketrans',  
'partition',  
'replace',  
'rfind',  
'rindex',  
'rjust',  
'rpartition',  
'rsplit',  
'rstrip',  
'split',  
'splitlines',  
'startswith',  
'strip',  
'swapcase',  
'title',  
'translate',  
'upper',  
'zfill']
```

By diving into some of the methods associated with strings, we hope to stress that these methods are *very intentionally* made for string objects.

Strings, as objects, are basically a predesigned empty container that can be filled with any characters you choose. When this container was designed, the creators of Python decided that there were a number of methods that would *always* be useful for a string, and built them into its functionality! **This idea of designing functionality around a logical grouping of information/data is the core concept of object-oriented programming.**

Eventually, you'll start structuring your ways of analyzing data into building these “empty containers (spoiler alert, they're called classes)” for similar data types rather than building an “input -> output”-type flow.

Part I: Variables and Data Types in Python:

lists and tuples

In the last few slides, we've covered basic data types in Python - ints, floats, and strings.

Now it's time to dive in the data types that can store multiple pieces of information.

Lists and **tuples** are some of the most common, useful, and versatile data types that you'll encounter in Python.

A **list** is denoted by brackets [] and consists of *any number* (at least up to your computer's memory capacity) of arbitrary objects (floats, strings, lists, etc.). This is one major advantage in Python - lists can handle everything! Lists also have a number of useful properties - they are **dynamic** and **mutable** (elements in them can be deleted, the elements within an entire list can be multiplied/duplicated *in the list*, more data can be appended to the list, etc). We'll go over this in more detail in the next slide, and some of HW1 will focus heavily on this.

A **tuple** is enclosed in parentheses (), and functions quite similarly to a list. However, tuples are **immutable**. Why even use them, then? A few reasons: 1.) when handling very large tuples/lists, program execution will be quite a bit faster with tuples, and 2.) sometimes you don't want to be able to modify a data type (users or errors in code can mess things up)

Part I: Variables and Data Types in Python:

lists

In your notebook, create a variable called “veggies_numbers”. Set it equal to a list containing 3 strings, an int, and a float: carrot, broccoli, radish, 5, and 20.25.

```
In [2]: vegetables = ["carrot", "broccoli", "radish", 5, 20.25]
```

Now take its length - what do you get?

Part I: Variables and Data Types in Python:

lists

In your notebook, create a variable called “veggies_numbers”. Set it equal to a list containing 3 strings, an int, and a float: carrot, broccoli, radish, 5, and 20.25.

```
In [2]: vegetables = ["carrot", "broccoli", "radish", 5, 20.25]
```

Now take its length - what do you get?

```
In [5]: len(veggie_numbers)
```

```
Out[5]: 5
```

But how do we pull out only a few of these elements?

Part I: Variables and Data Types in Python:

accessing elements in a list

The two primary ways to pull out elements of a list are **indexing** (pulling a single element based on its position) and **slicing** (pulling out a subset of values based on their indices).

Indexing is done with brackets and the value of Now take its length - what do you get?
the position of whatever element we're trying to access. If our list is called `veggie_numbers`, we'd pull the 0th element out like this:

```
In [4]: veggie_numbers[0]
```

```
Out[4]: 'carrot'
```

Part I: Variables and Data Types in Python:

accessing elements in a list

Slicing is also performed with brackets; however, it requires a starting value followed by a colon, as well as an ending value. Leaving the starting or the ending value blank will default to the beginning or the end of the list, respectively, while using a negative number will set stop/start bounds with respect to the *end* of the list. What are two ways to pull out the middle three values of `veggie_numbers` *in order*?

Part I: Variables and Data Types in Python:

accessing elements in a list

Slicing is also performed with brackets; however, it requires a starting value followed by a colon, as well as an ending value. Leaving the starting or the ending value blank will default to the beginning or the end of the list, respectively, while using a negative number will set stop/start bounds with respect to the *end* of the list. What are two ways to pull out the middle three values of `veggie_numbers` *in order*?

```
In [5]: veggie_numbers[1:4]
```

```
Out[5]: ['broccoli', 'radish', 5]
```

```
In [6]: veggie_numbers[-4:-1]
```

```
Out[6]: ['broccoli', 'radish', 5]
```

Slicing can also take a third element - the **interval** with which you want to select values, with the notation `[start:stop:interval]`. How would we pull out every other element of `veggie_numbers`, starting with the 0th?

Part I: Variables and Data Types in Python:

accessing elements in a list

Slicing is also performed with brackets; however, it requires a starting value followed by a colon, as well as an ending value. Leaving the starting or the ending value blank will default to the beginning or the end of the list, respectively, while using a negative number will set stop/start bounds with respect to the *end* of the list. What are two ways to pull out the middle three values of `veggie_numbers` *in order*?

```
In [5]: veggie_numbers[1:4]
```

```
Out[5]: ['broccoli', 'radish', 5]
```

```
In [6]: veggie_numbers[-4:-1]
```

```
Out[6]: ['broccoli', 'radish', 5]
```

Slicing can also take a third element - the **interval** with which you want to select values, with the notation `[start:stop:interval]`. How would we pull out every other element of `veggie_numbers`, starting with the 0th?

```
In [18]: veggie_numbers[::2]
```

```
Out[18]: ['carrot', 'radish', 20.25]
```

Part I: Variables and Data Types in Python:

dynamic/mutable lists

When we introduced lists, we pointed out that they're **dynamic** and **mutable**. This means that we can take the list object and change its elements, the order of its elements, its size, etc. As objects, lists have some easy-to-use methods associated with them that perform some of these functions!

Use `dir()` on our `veggie_numbers` list to look over its attributes.

Part I: Variables and Data Types in Python:

dynamic/mutable lists

When we introduced lists, we pointed out that they're **dynamic** and **mutable**. This means that we can take the list object and change its elements, the order of its elements, its size, etc. As objects, lists have some easy-to-use methods associated with them that perform some of these functions!

Use `dir()` on our `veggie_numbers` list to look over its attributes.

```
'append',  
'clear',  
'copy',  
'count',  
'extend',  
'index',  
'insert',  
'pop',  
'remove',  
'reverse',  
'sort']
```

Let's give `append` a try - don't be afraid to use `help()`/[/?/google](#).

Append the string 'apple' and the number 1000 to your list.

Part I: Variables and Data Types in Python:

dynamic/mutable lists

You might have noticed something trying this on your own - you can't append more than a single item at a time! Here's what it looks like if you do both at once.

```
In [66]: veggie_numbers.append('apple')
         veggie_numbers.append(1000)
         print(veggie_numbers)

['carrot', 'broccoli', 'radish', 5, 20.25, 'apple', 1000, 'apple', 1000]
```

I'm sure we can come up with a faster way of doing it. But let's remove our first attempt...there are a lot of ways to do this... but for now use the remove method the same way you used append.

Part I: Variables and Data Types in Python:

dynamic/mutable lists

You might have noticed something trying this on your own - you can't append more than a single item at a time! Here's what it looks like if you do both at once.

```
In [66]: veggie_numbers.append('apple')
         veggie_numbers.append(1000)
         print(veggie_numbers)

['carrot', 'broccoli', 'radish', 5, 20.25, 'apple', 1000, 'apple', 1000]
```

I'm sure we can come up with a faster way of doing it. But let's remove our first attempt...there are a lot of ways to do this... but for now use the remove method the same way you used append.

Now, try to append 'apple' and 1000 as a single list, and print veggie_numbers...what happens?

Part I: Variables and Data Types in Python:

dynamic/mutable lists

You might have noticed something trying this on your own - you can't append more than a single item at a time! Here's what it looks like if you do both at once.

```
In [66]: veggie_numbers.append('apple')
         veggie_numbers.append(1000)
         print(veggie_numbers)

['carrot', 'broccoli', 'radish', 5, 20.25, 'apple', 1000, 'apple', 1000]
```

I'm sure we can come up with a faster way of doing it. But let's remove our first attempt...there are a lot of ways to do this... but for now use the remove method the same way you used append.

Now, try to append 'apple' and 1000 as a single list, and print veggie_numbers...what happens?

```
In [74]: veggie_numbers.append(['apple', 1000])
         print(veggie_numbers)

['carrot', 'broccoli', 'radish', 5, 20.25, ['apple', 1000]]
```

We get a list within a list! This sort of thing can be useful in some circumstances, but isn't precisely what we want.

Part I: Variables and Data Types in Python:

dynamic/mutable lists

If we want to instead add 'apple' and 1000 to veggie_numbers in a single step, a little bit of internet searching tells us we can use the `extend()` method - let's give that a try!

Part I: Variables and Data Types in Python:

dynamic/mutable lists

If we want to instead add 'apple' and 1000 to veggie_numbers in a single step, a little bit of internet searching tells us we can use the `extend()` method - let's give that a try!

```
In [83]: veggie_numbers.extend(['apple', 1000])
         print(veggie_numbers)

['carrot', 'broccoli', 'radish', 5, 20.25, 'apple', 1000]
```

Bingo!

Finally, individual items in a list can easily be assigned by taking their index number and setting it equal to whatever new value you'd like, i.e. `my_list[x] = 'new_value'`. Try this in your notebook by re-assigning 'apple' to a vegetable of your choice.

Part I: Variables and Data Types in Python:

dynamic/mutable lists

If we want to instead add 'apple' and 1000 to veggie_numbers in a single step, a little bit of internet searching tells us we can use the `extend()` method - let's give that a try!

```
In [83]: veggie_numbers.extend(['apple', 1000])
         print(veggie_numbers)

['carrot', 'broccoli', 'radish', 5, 20.25, 'apple', 1000]
```

Bingo!

Finally, individual items in a list can easily be assigned by taking their index number and setting it equal to whatever new value you'd like, i.e. `my_list[x] = 'new_value'`. Try this in your notebook by re-assigning 'apple' to a vegetable of your choice.

```
In [90]: veggie_numbers[5]='cauliflower'
         print(veggie_numbers)

['carrot', 'broccoli', 'radish', 5, 20.25, 'cauliflower', 1000]
```

Part I: Variables and Data Types in Python:

a word on tuples

So we don't forget to mention - like lists, tuples can be comprised of multiple data types. Additionally, individual elements within a tuple can be access the same way.

However, tuples **are not mutable** - elements within them can't be moved around, extended, re-assigned, etc.

Part I: Variables and Data Types in Python:

dictionaries!

Hooray, it's time for dictionaries (dicts)!

Dicts are a Python data type that are denoted by curly brackets { } and consist of paired **keys** and **values** separated by a colon :

Let's say we want to create a dictionary called my_dict that describes us as a person.

The format would be like so:

```
In [101]: my_stats = {  
           'name': 'Jessie',  
           'age': 29,  
           'residence': 'NYC',  
           'favorite_food': 'nachos'  
         }  
  
print(my_stats)  
  
{'name': 'Jessie', 'age': 29, 'residence': 'NYC', 'favorite_food': 'nachos'}
```

Make a my_stats for yourself - we'll wait for a second.

Part I: Variables and Data Types in Python:

dicts

A few things to know about dicts:

- 1.) They're **unordered**. Try accessing an element of your dict the same way you would in a list (eg `my_stats[0]` - it doesn't work!

Part I: Variables and Data Types in Python:

dicts

A few things to know about dicts:

- 1.) They're **unordered**. Try accessing an element of your dict the same way you would in a list (eg `my_stats[0]` - it doesn't work!
- 2.) Instead, elements of dicts are access by their **keys**. Try typing `my_stats.keys()`

Part I: Variables and Data Types in Python:

dicts

A few things to know about dicts:

- 1.) They're **unordered**. Try accessing an element of your dict the same way you would in a list (eg `my_stats[0]` - it doesn't work!
- 2.) Instead, elements of dicts are access by their **keys**. Try typing `my_stats.keys()`
- 3.) To find the value of a key, you have a few options. The most common is to use brackets - eg `my_stats['name']`. Try printing your favorite food now.

Part I: Variables and Data Types in Python:

dicts

A few things to know about dicts:

- 1.) They're **unordered**. Try accessing an element of your dict the same way you would in a list (eg `my_stats[0]` - it doesn't work!
- 2.) Instead, elements of dicts are access by their **keys**. Try typing `my_stats.keys()`
- 3.) To find the value of a key, you have a few options. The most common is to use brackets - eg `my_stats['name']`. Try printing your favorite food now.
- 4.) Dicts are **mutable** - let's say I moved to Brooklyn. I can reassign this value using the command `my_stats['residence'] = 'Brooklyn'`
 - a.) Furthermore, **keys and values can be added** to a dictionary. Try adding the key "hair color" to your dict now.
 - b.) To remove a key/item, use the pop method - eg `my_stats.pop('hair_color')`. Alternatively, `my_stats.popitem()` will simply remove the last inserted item.

Part I: Variables and Data Types in Python:

dicts

A few things to know about dicts:

- 1.) They're **unordered**. Try accessing an element of your dict the same way you would in a list (eg `my_stats[0]` - it doesn't work!
- 2.) Instead, elements of dicts are access by their **keys**. Try typing `my_stats.keys()`
- 3.) To find the value of a key, you have a few options. The most common is to use brackets - eg `my_stats['name']`. Try printing your favorite food now.
- 4.) Dicts are **mutable** - let's say I moved to Brooklyn. I can reassign this value using the command `my_stats['residence'] = 'Brooklyn'`
 - a.) Furthermore, **keys and values can be added** to a dictionary. Try adding the key "hair color" to your dict now.
 - b.) To remove a key/item, use the pop method - eg `my_stats.pop('hair_color')`. Alternatively, `my_stats.popitem()` will simply remove the last inserted item.
- 5.) Dicts can be nested - totally fine to make a dict the value for the first key in a dict!

Part I: Variables and Data Types in Python:

dicts

This is a relatively simple overview for now; however, don't let this diminish the importance of dictionaries for you! They'll be major players in almost everything we do from here on out :)

Can you come up with some lab scenarios in which being able to store things in a dict format would be useful?

Part I: Variables and Data Types in Python:

sets

A **set** is a python data type consisting of **unordered**, **unique** values. We may not cover them heavily in the course, but they're useful to know!

Like dicts, sets are created using curly brackets. **However, they do not have key/item pairs.** Instead, each value in a set is simply separated by a comma.

Why use a set? Because a set **cannot have duplicate values**, turning things like lists or tuples into sets is a fast, efficient way of removing duplicates. This can be achieved using Python's built-in `set()` function.

```
In [109]: list1 = [1,2,3,'a','b','c','c','d','d',1,2]
          set(list1)
```

```
Out[109]: {1, 2, 3, 'a', 'b', 'c', 'd'}
```

Additionally, sets are useful tools in performing some logical operations on sets (duh!) like unions and intersections.

The end of Part I

WE DID IT - we made it to Part II!

We're going to shift the focus from data types in Python to programming concepts in Python.

Part II: Conditionals/loops, and how to use them in Python

In the first part of the lecture, we focused on assigning variables to different **data types**, and using built-in methods to manipulate those data types.

Now, we're going to move along to writing **rules and tasks in Python** for our computer to execute.

The first concept we'll focus on are **conditional statements**. If you've programmed before, you're certainly familiar with these - our focus will be on how to implement them in Python.

Part II: Conditionals: `if...elif...else`

In an if/else statement, we're telling the computer to execute a command **if** it evaluates a logical statement to be true. In Python, if statements are written on one line and followed by a colon. The task we want the computer to execute is then written **following an indentation** (which, if you've formatted the if statement properly, should automatically appear) on the second line.

```
In [110]: a = 5
          b = 22

          if a < b:
              print('a is less than b')

a is less than b
```

Part II: Conditionals: `if...elif...else`

The first conditional we'll focus on is the `if/else` statement, in which we're telling the computer to execute a command **if** it evaluates a certain statement to be true. In Python, if statements are written on one line and followed by a colon. The task we want the computer to execute is then written **following an indentation** (which, if you've formatted the if statement properly, should automatically appear) on the second line.

```
In [110]: a = 5
          b = 22

          if a < b:
              print('a is less than b')

          a is less than b
```

We can also add in the **`elif`** statement, which essentially says “if the above statement wasn’t true, then evaluate if *this* statement is true and perform the task I’ve outlined below” and the **`else`** statement, which says “if none of the above is true, then perform the task I’ve outlined below”

```
In [112]: a = 54
          b = 22

          if a < b:
              print('a is less than b')
          elif a == b:
              print('a equals b')
```

```
In [115]: a = 54
          b = 54

          if a < b:
              print('a is less than b')
          elif a == b:
              print('a equals b')
```

Part II: Conditionals: `if...elif...else`

The first conditional we'll focus on is the `if/else` statement, in which we're telling the computer to execute a command **if** it evaluates a certain statement to be true. In Python, if statements are written on one line and followed by a colon. The task we want the computer to execute is then written **following an indentation** (which, if you've formatted the if statement properly, should automatically appear) on the second line.

```
In [110]: a = 5
          b = 22

          if a < b:
              print('a is less than b')

          a is less than b
```

We can also add in the **`elif`** statement, which essentially says “if the above statement wasn’t true, then evaluate if *this* statement is true and perform the task I’ve outlined below” and the **`else`** statement, which says “if none of the above is true, then perform the task I’ve outlined below”

```
In [112]: a = 54
          b = 22

          if a < b:
              print('a is less than b')
          elif a == b:
              print('a equals b')
```

```
In [115]: a = 54
          b = 54

          if a < b:
              print('a is less than b')
          elif a == b:
              print('a equals b')
```

Part II: Conditionals: `if...elif...else`

When we consider the logical operation that we want evaluated in an if statement, useful tools include the **and/or operators**.

If we have the values a, b, and c, and want to evaluate them with respect to each other, we could use multiple if statements OR we can write a compound expression.

For example:

```
In [116]: a = 52
          b = 54
          c = 22

          if a < b and a > c:
              print('A is somewhere in-between!')

A is somewhere in-between!
```

Part II: Conditionals: `if...elif...else`

Furthermore, we can use **in** and **not** operators to create our logical statements.

```
In [121]: a = ['apple', 'orange', 'kiwi']  
b = ['eggs', 'milk', 'butter']  
c = ['cucumber', 'celery', 'pepper']  
  
if 'celery' in c:  
    print('c contains various veggies!')
```

c contains various veggies!

```
In [122]: a = ['apple', 'orange', 'kiwi']  
b = ['eggs', 'milk', 'butter']  
c = ['cucumber', 'celery', 'pepper']  
  
if 'eggs' not in c:  
    print('c is probably not my dairy list!')
```

c is probably not my dairy list!

As you can imagine, there are a LOT of things you can do with a conditional - you simply need to be able to come up with a logical statement to be evaluated.

Part II: loops

bröther may i have some lööps



Part II: loops

If you've programmed, you've worked with loops.

The idea behind a loop is to **iterate** over a sequence of code until a particular condition has been satisfied.

Loops in Python come in two flavors: **for loops** and **while loops** - we'll start with an example of a for loop.

In your notebook, make a list consisting of the numbers 1 to 10 called "one_to_ten"

Part II: loops

If you've programmed, you've worked with loops.

The idea behind a loop is to **iterate** over a sequence of code until a particular condition has been satisfied.

Loops in Python come in two flavors: **for loops** and **while loops** - we'll start with an example of a for loop.

In your notebook, make a list consisting of the numbers 1 to 10 called "one_to_ten"

Now, let's say we want the computer to print out every value in one_to_ten within a string! How would we write this?

In Python, for loops are structured like so:

```
for i in sequence:  
    task to perform
```

Part II: for loops

```
In [127]: one_to_ten = [1,2,3,4,5,6,7,8,9,10]
```

```
for i in one_to_ten:  
    print('Number:',i)
```

```
Number: 1  
Number: 2  
Number: 3  
Number: 4  
Number: 5  
Number: 6  
Number: 7  
Number: 8  
Number: 9  
Number: 10
```

Part II: for loops

```
In [127]: one_to_ten = [1,2,3,4,5,6,7,8,9,10]
```

```
for i in one_to_ten:  
    print('Number:',i)
```

```
Number: 1  
Number: 2  
Number: 3  
Number: 4  
Number: 5  
Number: 6  
Number: 7  
Number: 8  
Number: 9  
Number: 10
```

Let's give this a try on our own!

For `one_to_ten`, write a for loop that will iterate through the list and print the value. However, if the value `== 5`, have your loop return “We hit the number 5!” instead.

Part II: for loops

```
In [142]: one_to_ten = [1,2,3,4,5,6,7,8,9,10]

for i in one_to_ten:
    if i != 5:
        print(i)
    elif i == 5:
        print('We hit the number five!')
```

```
1
2
3
4
We hit the number five!
6
7
8
9
10
```

This is all well and good... but what if I asked you to write a for loop for `one_to_ten` that takes each element and turns it into that element ***plus twice its own value?***

Part II: for loops

This is all well and good... but what if I asked you to write a for loop for `one_to_ten` that takes each element and turns it into that element ***plus twice its own value***?

```
In [144]: one_to_ten = [1,2,3,4,5,6,7,8,9,10]
```

```
for i in one_to_ten:  
    i = i + (2*i)
```

```
print(one_to_ten)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Why doesn't this work?

Part II: for loops

This is all well and good... but what if I asked you to write a for loop for `one_to_ten` that takes each element and turns it into that element ***plus twice its own value***?

```
In [144]: one_to_ten = [1,2,3,4,5,6,7,8,9,10]
```

```
for i in one_to_ten:  
    i = i + (2*i)
```

```
print(one_to_ten)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

You'll notice that nothing happened in the above statement, because **that's not how you reassign values in a list**. If you think back to lists earlier this lecture, we learned that if we want to replace an element, the format is `listname[x] = newvalue`.

This means that we need a way of keeping count of where we are as we iterate over each element in `one_to_ten`...

Thankfully, Python has a super efficient method of doing this called **`enumerate()`**

Part II: using the `enumerate` method in for loops

If we first search `enumerate`?, Python's built-in help tool tells us this:

```
The enumerate object yields pairs containing a count (from start, which defaults to zero) and a value yielded by the iterable argument.
```

```
enumerate is useful for obtaining an indexed list:
```

```
(0, seq[0]), (1, seq[1]), (2, seq[2]), ...
```

```
Type:                type
```

```
Subclasses:
```

Let's try running `enumerate` in a loop over `one_to_ten` and see what we get!

Part II: using the `enumerate` method in for loops

Let's try running `enumerate` in a loop over `one_to_ten` and see what we get!

```
In [152]: for i in enumerate(one_to_ten):  
          print(i)
```

```
(0, 1)  
(1, 2)  
(2, 3)  
(3, 4)  
(4, 5)  
(5, 6)  
(6, 7)  
(7, 8)  
(8, 9)  
(9, 10)
```

Interesting! So for each element of `one_to_ten`, `enumerate()` returns **tuples** consisting of **the number of the iteration** and **the value of the element**.

Looking at the output we see from `enumerate()` above, can you think of a way to write the for loop we discussed earlier?

Our goal: for each value in `one_to_ten`, assign `one_to_ten[iteration #] = value + (value*2)`

Part II: using the `enumerate` method in for loops

```
In [152]: for i in enumerate(one_to_ten):  
          print(i)
```

```
(0, 1)  
(1, 2)  
(2, 3)  
(3, 4)  
(4, 5)  
(5, 6)  
(6, 7)  
(7, 8)  
(8, 9)  
(9, 10)
```

`idx, i` refer to each element that `enumerate(one_to_ten)` returns

```
In [159]: one_to_ten = [1,2,3,4,5,6,7,8,9,10]  
  
          for idx, i in enumerate(one_to_ten):  
              one_to_ten[idx] = i + (2*i)  
  
          print(one_to_ten)  
  
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

As a side note - it's a good habit to make the variables you're assigning *within your loop* somewhat descriptive! When I use `enumerate()`, I like to use “`idx`” for the index location, just to keep things clear

Part II: using the `enumerate` method in for loops

```
In [152]: for i in enumerate(one_to_ten):  
          print(i)
```

```
(0, 1)  
(1, 2)  
(2, 3)  
(3, 4)  
(4, 5)  
(5, 6)  
(6, 7)  
(7, 8)  
(8, 9)  
(9, 10)
```

`idx, i` refer to each element that `enumerate(one_to_ten)` returns

```
In [159]: one_to_ten = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
          for idx, i in enumerate(one_to_ten):  
              one_to_ten[idx] = i + (2*i)  
  
          print(one_to_ten)  
  
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

As a side note - it's a good habit to make the variables you're assigning *within your loop* somewhat descriptive! When I use `enumerate()`, I like to use “`idx`” for the index location, just to keep things clear

Part II: parallel iteration in for loops using zip()

In the last few slides, we learned how to use `enumerate()` to return both an index and a value if we need both.

Let's consider another circumstance - we have two lists that we want to evaluate together at the same time. Of course, Python has another neat built in tool for this!

Create two lists in your notebook: one titled “mouse_number” and one titled “temperament”. Fill “mouse number” with the values 1 through 5, and “temperament” with your choices of 5 random “nice” and “mean” strings. What happens if you return `i` for every value in `zip(mouse_number, temperament)` ?

Part II: parallel iteration in for loops using zip()

In the last few slides, we learned how to use `enumerate()` to return both an index and a value if we need both.

Let's consider another circumstance - we have two lists that we want to evaluate together at the same time. Of course, Python has another neat built in tool for this!

Create two lists in your notebook: one titled “mouse_number” and one titled “temperament”. Fill “mouse number” with the values 1 through 5, and “temperament” with your choices of 5 random “nice” and “mean” strings. What happens if you return `i` for every value in `zip(mouse_number, temperament)` ?

```
In [163]: mouse_number = [1, 2, 3, 4, 5]
          temperament = ['nice', 'nice', 'mean', 'nice', 'mean']
```

```
for i in zip(mouse_number, temperament):
    print(i)
```

```
(1, 'nice')
(2, 'nice')
(3, 'mean')
(4, 'nice')
(5, 'mean')
```

Part II: parallel iteration in for loops using zip()

```
53]: mouse_number = [1, 2, 3, 4, 5]
    temperament = ['nice', 'nice', 'mean', 'nice', 'mean']

    for i in zip(mouse_number, temperament):
        print(i)
```

```
(1, 'nice')
(2, 'nice')
(3, 'mean')
(4, 'nice')
(5, 'mean')
```

Much like the example we covered using `enumerate()`, can you use the output of `zip(mouse_number, temperament)` to write a for loop that returns the sentence “Mouse number ____ is a ____ mouse.” for each element of both lists?

Hint: don’t forget to use the format method associated with strings! E.g. “Insert one here: {} and two here {}".format(1, 2)

Part II: parallel iteration in for loops using zip()

```
In [163]: mouse_number = [1, 2, 3, 4, 5]
          temperament = ['nice', 'nice', 'mean', 'nice', 'mean']
```

```
for i in zip(mouse_number, temperament):
    print(i)
```

```
(1, 'nice')
(2, 'nice')
(3, 'mean')
(4, 'nice')
(5, 'mean')
```

Num, temp refer to each element that zip(mouse_number, temperament) returns

```
In [164]: for num, temp in zip(mouse_number, temperament):
          print("Mouse number {} is a {} mouse".format(num, temp))
```

```
Mouse number 1 is a nice mouse
Mouse number 2 is a nice mouse
Mouse number 3 is a mean mouse
Mouse number 4 is a nice mouse
Mouse number 5 is a mean mouse
```

For each iteration, the “num” and “temp” variables inserted by the format method will change.

Part II: list comprehensions

Finally, after introducing the concept of a for loop, we want to introduce a **really cool, Python-specific tool called a list comprehension**.

Let's think back to the `one_to_ten` variable you made - we had originally set ourselves the task of using a for loop to turn every element in `one_to_ten` equal to itself plus twice its value.

A list comprehension is a tool that condenses the logic of a for loop in a concise, single line of code. It's formatted like a list - that is, in brackets `[]` - but contains a for loop.

If we wanted to use a list comprehension to accomplish the same goal, it would look like this:

```
[i + (2*i) for i in one_to_ten]
```

However, we have to remember that we haven't set it equal to anything!

To accomplish the same task that we originally outlined, we would set it equal to `one_to_ten` in order to re-assign our original list.

Part II: list comprehensions

```
In [168]: one_to_ten = [1,2,3,4,5,6,7,8,9,10]
```

```
In [171]: one_to_ten = [i+(i*2) for i in one_to_ten]  
print(one_to_ten)
```

```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```


Part II: parallel iteration in for loops using zip()

Just from these examples, it's clear that Python has built in as much functionality as possible to streamline the use of for loops - we've only covered a few of the possibilities! As always, never hesitate to start searching the internet for more efficient ways to accomplish your goal.

Part II: while loops

Once again, if you've programmed before, while loops ought to be just as familiar to you as for loops. In Python, the two are structured the same way; however, while a for loop iterates until a condition is met, a **while loop iterations while a condition is true**.

Example:

```
In [167]: t = 0
          dt = 1
          step = 3

          while t <= 10:
              step = step + step
              t = t + dt
              print("At timepoint {}, Sheila took {} steps.".format(t,step))
```

```
At timepoint 1, Sheila took 6 steps.
At timepoint 2, Sheila took 12 steps.
At timepoint 3, Sheila took 24 steps.
At timepoint 4, Sheila took 48 steps.
At timepoint 5, Sheila took 96 steps.
At timepoint 6, Sheila took 192 steps.
At timepoint 7, Sheila took 384 steps.
At timepoint 8, Sheila took 768 steps.
At timepoint 9, Sheila took 1536 steps.
At timepoint 10, Sheila took 3072 steps.
At timepoint 11, Sheila took 6144 steps.
```

The end of Part II!

In part two, we covered some ways to make rules for your computer in Python, and how to iterate over a sequence in order to perform specific operations on numerous pieces of data.

Now, we'll get to the FINAL PART (omg) of today's class: writing functions!

Part III: Functions, and importing your own functions from a .py file

Functions are one of the most basic ways of taking data (**input**), manipulating it, and returning an **output**.

Functions can be called repeatedly, and used on *any* input, so long as that input is of the same type that the function was written for.

In Python, functions are **defined** using the command “def”, followed by the name of the function, followed by parentheses containing the necessary input.

The contents of the function - what you want it to do to the input you’ve given it- must then be indented.

```
def myFunction(input1,input2):  
    print(input1)  
    print(input2)
```

Part III: Functions, and importing your own functions from a .py file

Additionally, it is good form to include a **docstring** in your functions. A docstring is simply text surrounded by three quotes on either side, eg.

```
Def myFunction(input1, input2): ←———— This is the docstring.  
    """Prints input"""  
    print(input1)  
    print(input2)
```

A truly excellent docstring will outline what a function does, the arguments it takes in, and provides an example.

<https://www.python.org/dev/peps/pep-0257/> is an excellent resource for how to format yours!

```
def complex(real=0.0, imag=0.0):  
    """Form a complex number.  
  
    Keyword arguments:  
    real -- the real part (default 0.0)  
    imag -- the imaginary part (default 0.0)  
    """  
  
    if imag == 0.0 and real == 0.0:  
        return complex_zero
```

Part III: Functions, and importing your own functions from a .py file

Let's write a really simple function together just to get used to the idea.

In your notebook, define a function called “myname” that takes a single argument and, when called, prints the argument along with the phrase “is my name”

Part III: Functions, and importing your own functions from a .py file

Let's write a really simple function together just to get used to the idea.

In your notebook, define a function called “myname” that takes a single argument and, when called, prints the argument along with the phrase “is my name”

```
In [177]: def myName(input1):  
          print(input1 + " is my name")
```

```
In [178]: myName('jessie')  
  
jessie is my name
```

This is a very simple example - what if we want the function to actually evaluate something and **return a value**?

Simple - we simply specify what we want returned using the `return()` statement!

Part III: Functions, and importing your own functions from a .py file

Let's write a function together that takes a temperature in Celsius and converts it to Fahrenheit.

The equation is:

$$(x^{\circ}\text{C} * 9/5) + 32$$

Let's ask ourselves: what should the function take as input?

What should it return?

Part III: Functions, and importing your own functions from a .py file

Let's write a function together that takes a temperature in Celsius and converts it to Fahrenheit.

The equation is:

$$(x^{\circ}\text{C} * 9/5) + 32$$

Let's ask ourselves: what should the function take as input? Degrees Celsius!

What should it return? Degrees Fahrenheit!

So, we'd want to have something along the lines of:

```
def tempConverter(degC):  
  
    degF = (degC * 9/5) + 32  
  
    return(degF)
```

Part III: Functions, and importing your own functions from a .py file

Try running `tempConverter` in your notebook- instead of just typing `tempConverter(input)` , can you assign a variable to it and print the variable?

Part III: Functions, and importing your own functions from a .py file

Try running tempConverter in your notebook- instead of just typing `tempConverter(input)` , can you assign a variable to it and print the variable?

```
In [181]: def tempConverter(degC):  
           degF = (degC * 9/5) + 32  
  
           return(degF)  
  
f = tempConverter(5)  
print(f)  
  
41.0
```

Part III: Functions, and importing your own functions from a .py file

Great! At this point, we've written a function in Python!

The last basic tool we want to leave you with is an understanding of how all this all gets put together.

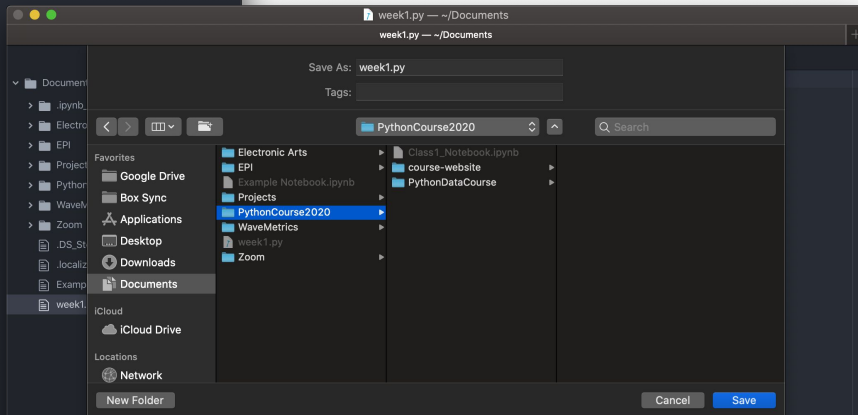
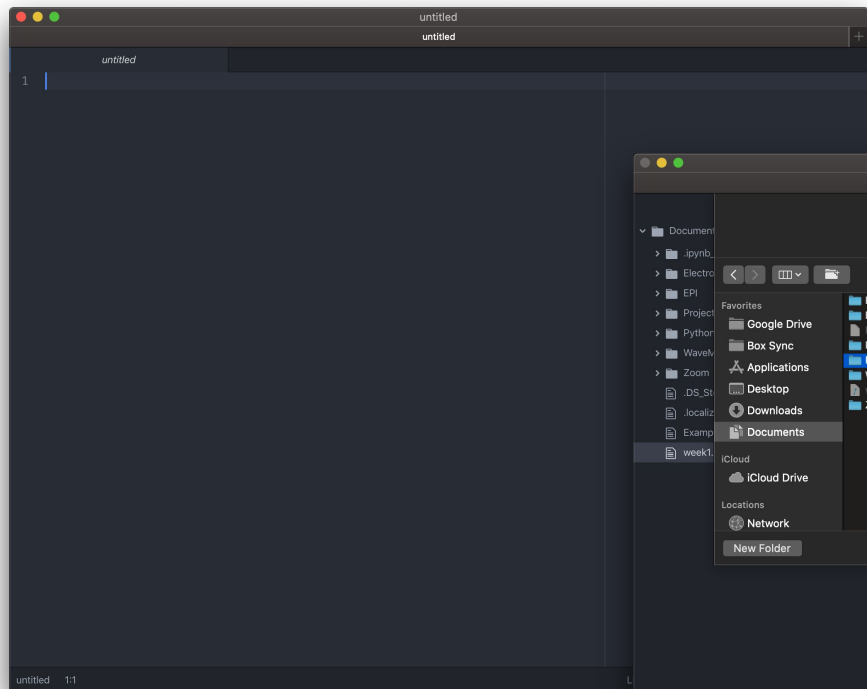
If you remember the preclass homework, we had you **import** a few Python modules - how do we translate something we've written into something that we, to, could import?

Let's consider the function we just wrote, and pretend it's a groundbreaking mathematical analysis that you want to eventually put into a package!

First, open the **text editor** that you downloaded, and open a new file.

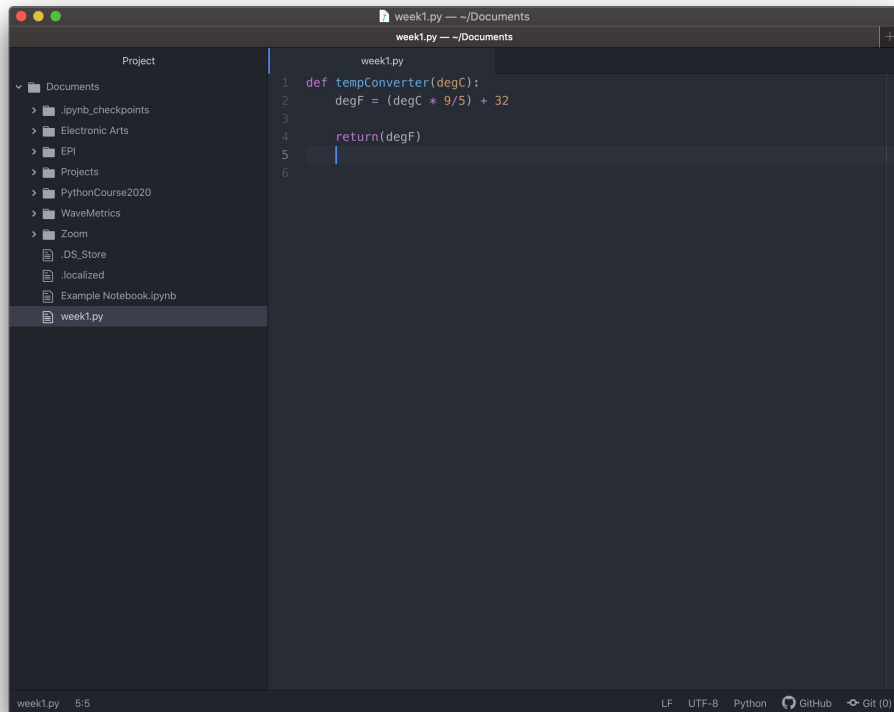
Part III: Functions, and importing your own functions from a .py file

Open a blank new window, and save it as “week1.py” in *the same directory* as you’ve saved the notebook you’re currently working in.



Part III: Functions, and importing your own functions from a .py file

Now, copy and paste in our brilliant temperature conversion function!



The screenshot shows a code editor window titled "week1.py -- ~/Documents". The left sidebar displays a file explorer with a "Project" view, showing a directory structure under "Documents". The main editor area shows the following Python code in "week1.py":

```
1 def tempConverter(degC):  
2     degF = (degC * 9/5) + 32  
3  
4     return(degF)  
5  
6
```

The status bar at the bottom indicates "week1.py 5/5", "LF UTF-8 Python", and icons for GitHub and Git (0).

Part III: Functions, and importing your own functions from a .py file

Finally, restart your notebook's kernel, and **try importing week1**.

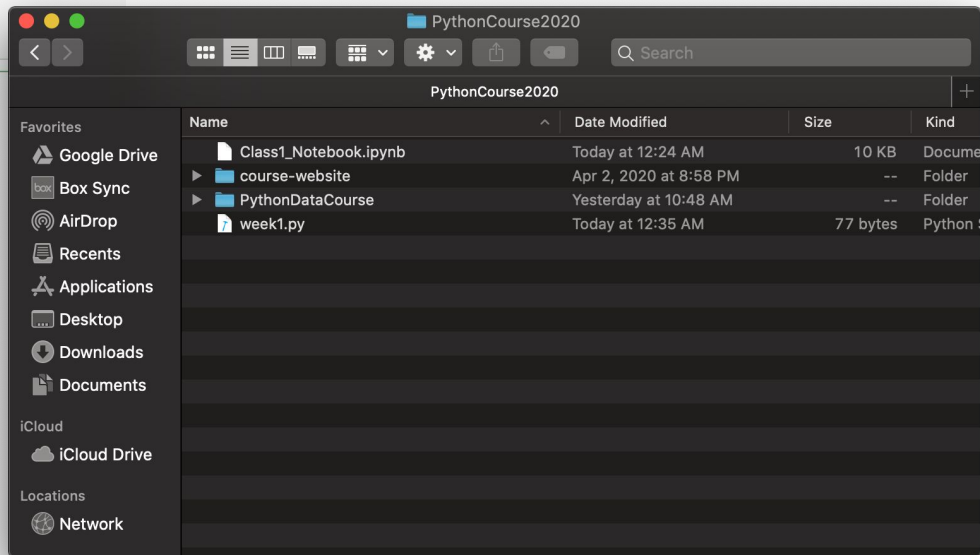
Part III: Functions, and importing your own functions from a .py file

Finally, restart your notebook's kernel, and **try importing week1**.

```
In [2]: import week1 as w1
```

Because our .py file and our notebook **are in the same directory**, we don't have to do anything special here at all!! “Import” will simply notice the “week1.py” file in the same directory, and will import week1 based on name alone.

Please note that *this is not good practice in the long run* - we should **never** put our code in a .py file and then get reliant on storing that file locally. The **correct way to do this** will be the focus of the next class!



Part III: Functions, and importing your own functions from a .py file

Dire warnings aside, let's see if our homemade, locally-imported function works!

Try using the function using the same conventions you learned in the pre-class slides (eg.

`variable = packagename.function()`

```
In [2]: import week1 as w1
```

```
In [6]: f = w1.tempConverter(7)
        print(f)
```

```
44.6
```

Huzzah!

Part III: Functions, and importing your own functions from a .py file

At this point, you've been given a *very* fast rundown of the basics that you'll need for the rest of the course - *this was by no means comprehensive!!*

We encourage you to ask us about **anything** that felt a little shaky during the course, and to also do some research on your own to practice things you might not be familiar with.

Finally, we'll be posting a short problem set later today that will ask you to work on each of these skills! This will be the only problem set sent out in PDF format -the rest will be jupyter notebooks obtained from the course git repository :)

Thanks everyone!