2019

Complete Python Bootcamp

FROM ZERO TO HERO IN PYTHON 3

Table of Contents

01-Numbers

April 9, 2019

1 Numbers and more in Python!

In this lecture, we will learn about numbers in Python and how to use them.

We'll learn about the following topics:

```
1.) Types of Numbers in Python
2.) Basic Arithmetic
3.) Differences between classic division and floor division
4.) Object Assignment in Python
```

1.1 Types of numbers

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number. For example 2.0 and -2.1 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.

Throughout this course we will be mainly working with integers or simple float number types.

Here is a table of the two main types we will spend most of our time working with some examples:

Examples
Number "Type"
1,2,-5,1000
Integers
1.2,-0.5,2e2,3E2
Floating-point numbers

Now let's start with some basic arithmetic.

1.1.1 Basic Arithmetic

```
In [1]: # Addition
        2+1
```

```
Out[1]: 3
```

```
In [2]: # Subtraction
        2-1

Out[2]: 1

In [3]: # Multiplication
        2*2

Out[3]: 4

In [4]: # Division
        3/2

Out[4]: 1.5

In [5]: # Floor Division
        7//4

Out[5]: 1
```

Whoa! What just happened? Last time I checked, 7 divided by 4 equals 1.75 not 1!

The reason we get this result is because we are using "*floor*" division. The // operator (two forward slashes) truncates the decimal without rounding, and returns an integer result.

So what if we just want the remainder after division?

```
In [6]: # Modulo
        7%4

Out[6]: 3
```

4 goes into 7 once, with a remainder of 3. The % operator returns the remainder after division.

1.1.2 Arithmetic continued

```
In [7]: # Powers
        2**3

Out[7]: 8

In [8]: # Can also do roots this way
        4**0.5

Out[8]: 2.0

In [9]: # Order of Operations followed in Python
        2 + 10 * 10 + 3

Out[9]: 105

In [10]: # Can use parentheses to specify orders
         (2+10) * (10+3)

Out[10]: 156
```

1.2 Variable Assignments

Now that we've seen how to use numbers in Python as a calculator let's see how we can assign names and create variables.

We use a single equals sign to assign labels to variables. Let's see a few examples of how we can do this.

```
In [11]: # Let's create an object called "a" and assign it the number 5
         a = 5
```

Now if I call *a* in my Python script, Python will treat it as the number 5.

```
In [12]: # Adding the objects
         a+a

Out[12]: 10
```

What happens on reassignment? Will Python let us write it over?

```
In [13]: # Reassignment
         a = 10

In [14]: # Check
         a

Out[14]: 10
```

Yes! Python allows you to write over assigned variable names. We can also use the variables themselves when doing the reassignment. Here is an example of what I mean:

```
In [15]: # Check
         a

Out[15]: 10

In [16]: # Use A to redefine A
         a = a + a

In [17]: # Check
         a

Out[17]: 20
```

The names you use when creating these labels need to follow a few rules:

```
1. Names can not start with a number.
2. There can be no spaces in the name, use _ instead.
3. Can't use any of these symbols :'",<>/?|\()!@#$%^&*~-+
4. It's considered best practice (PEP8) that names are lowercase.
5. Avoid using the characters 'l' (lowercase letter el), 'O' (uppercase letter oh),
   or 'I' (uppercase letter eye) as single character variable names.
6. Avoid using words that have special meaning in Python like "list" and "str"
```

Using variable names can be a very useful way to keep track of different variables in Python. For example:

```
In [18]: # Use object names to keep better track of what's going on in your code!
         my_income = 100

         tax_rate = 0.1

         my_taxes = my_income*tax_rate

In [19]: # Show my taxes!
         my_taxes

Out[19]: 10.0
```

So what have we learned? We learned some of the basics of numbers in Python. We also learned how to do arithmetic and use Python as a basic calculator. We then wrapped it up with learning about Variable Assignment in Python.

Up next we'll learn about Strings!

01-Variable Assignment

April 9, 2019

1 Variable Assignment

1.1 Rules for variable names

- names can not start with a number
- names can not contain spaces, use _ intead
- names can not contain any of these symbols:

  ```
  :'",<>/?|\!@#%^&*~-+
  ```

- it's considered best practice (PEP8) that names are lowercase with underscores
- avoid using Python built-in keywords like `list` and `str`
- avoid using the single characters `l` (lowercase letter el), `O` (uppercase letter oh) and `I` (uppercase letter eye) as they can be confused with `1` and `0`

1.2 Dynamic Typing

Python uses *dynamic typing*, meaning you can reassign variables to different data types. This makes Python very flexible in assigning data types; it differs from other languages that are *statically typed*.

```
In [1]: my_dogs = 2

In [2]: my_dogs

Out[2]: 2

In [3]: my_dogs = ['Sammy', 'Frankie']

In [4]: my_dogs

Out[4]: ['Sammy', 'Frankie']
```

1.2.1 Pros and Cons of Dynamic Typing

Pros of Dynamic Typing

- very easy to work with
- faster development time

Cons of Dynamic Typing

- may result in unexpected bugs!
- you need to be aware of `type()`

1.3 Assigning Variables

Variable assignment follows `name = object`, where a single equals sign = is an *assignment operator*

```
In [5]: a = 5

In [6]: a

Out[6]: 5
```

Here we assigned the integer object `5` to the variable name `a`.Let's assign `a` to something else:

```
In [7]: a = 10

In [8]: a

Out[8]: 10
```

You can now use `a` in place of the number `10`:

```
In [9]: a + a

Out[9]: 20
```

1.4 Reassigning Variables

Python lets you reassign variables with a reference to the same object.

```
In [10]: a = a + 10

In [11]: a

Out[11]: 20
```

There's actually a shortcut for this. Python lets you add, subtract, multiply and divide numbers with reassignment using `+=`, `-=`, `*=`, and `/=`.

```
In [12]: a += 10

In [13]: a

Out[13]: 30

In [14]: a *= 2

In [15]: a

Out[15]: 60
```

1.5 Determining variable type with `type()`

You can check what type of object is assigned to a variable using Python's built-in `type()` function. Common data types include: * **int** (for integer) * **float** * **str** (for string) * **list** * **tuple** * **dict** (for dictionary) * **set** * **bool** (for Boolean True/False)

```
In [16]: type(a)

Out[16]: int

In [17]: a = (1,2)

In [18]: type(a)

Out[18]: tuple
```

1.6 Simple Exercise

This shows how variables make calculations more readable and easier to follow.

```
In [19]: my_income = 100
         tax_rate = 0.1
         my_taxes = my_income * tax_rate

In [20]: my_taxes

Out[20]: 10.0
```

Great! You should now understand the basics of variable assignment and reassignment in Python.Up next, we'll learn about strings!

02-Strings

April 9, 2019

1 Strings

Strings are used in Python to record text information, such as names. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

This idea of a sequence is an important one in Python and we will touch upon it later on in the future.

In this lecture we'll learn about the following:

```
1.) Creating Strings
2.) Printing Strings
3.) String Indexing and Slicing
4.) String Properties
5.) String Methods
6.) Print Formatting
```

1.1 Creating a String

To create a string in Python you need to use either single quotes or double quotes. For example:

```
In [1]: # Single word
        'hello'

Out[1]: 'hello'

In [2]: # Entire phrase
        'This is also a string'

Out[2]: 'This is also a string'

In [3]: # We can also use double quote
        "String built with double quotes"

Out[3]: 'String built with double quotes'

In [4]: # Be careful with quotes!
        ' I'm using single quotes, but this will create an error'
```

```
  File "<ipython-input-4-da9a34b3dc31>", line 2
    ' I'm using single quotes, but this will create an error'
       ^
SyntaxError: invalid syntax
```

The reason for the error above is because the single quote in I'm stopped the string. You can use combinations of double and single quotes to get the complete statement.

```
In [5]: "Now I'm ready to use the single quotes inside a string!"

Out[5]: "Now I'm ready to use the single quotes inside a string!"
```

Now let's learn about printing strings!

1.2 Printing a String

Using Jupyter notebook with just a string in a cell will automatically output strings, but the correct way to display strings in your output is by using a print function.

```
In [6]: # We can simply declare a string
        'Hello World'

Out[6]: 'Hello World'

In [7]: # Note that we can't output multiple strings this way
        'Hello World 1'
        'Hello World 2'

Out[7]: 'Hello World 2'
```

We can use a print statement to print a string.

```
In [8]: print('Hello World 1')
        print('Hello World 2')
        print('Use \n to print a new line')
        print('\n')
        print('See what I mean?')

Hello World 1
Hello World 2
Use
 to print a new line


See what I mean?
```

1.3 String Basics

We can also use a function called len() to check the length of a string!

```
In [9]: len('Hello World')

Out[9]: 11
```

Python's built-in len() function counts all of the characters in the string, including spaces and punctuation.

1.4 String Indexing

We know strings are a sequence, which means Python can use indexes to call parts of the sequence. Let's learn how this works.

In Python, we use brackets [] after an object to call its index. We should also note that indexing starts at 0 for Python. Let's create a new object called s and then walk through a few examples of indexing.

```
In [10]: # Assign s as a string
         s = 'Hello World'

In [11]: #Check
         s

Out[11]: 'Hello World'

In [12]: # Print the object
         print(s)

Hello World
```

Let's start indexing!

```
In [13]: # Show first element (in this case a letter)
         s[0]

Out[13]: 'H'

In [14]: s[1]

Out[14]: 'e'

In [15]: s[2]

Out[15]: 'l'
```

We can use a : to perform *slicing* which grabs everything up to a designated point. For example:

```
In [16]: # Grab everything past the first term all the way to the length of s which is len(s)
         s[1:]
```

```
Out[16]: 'ello World'

In [17]: # Note that there is no change to the original s
         s

Out[17]: 'Hello World'

In [18]: # Grab everything UP TO the 3rd index
         s[:3]

Out[18]: 'Hel'
```

Note the above slicing. Here we're telling Python to grab everything from 0 up to 3. It doesn't include the 3rd index. You'll notice this a lot in Python, where statements and are usually in the context of "up to, but not including".

```
In [19]: #Everything
         s[:]

Out[19]: 'Hello World'
```

We can also use negative indexing to go backwards.

```
In [20]: # Last letter (one index behind 0 so it loops back around)
         s[-1]

Out[20]: 'd'

In [21]: # Grab everything but the last letter
         s[:-1]

Out[21]: 'Hello Worl'
```

We can also use index and slice notation to grab elements of a sequence by a specified step size (the default is 1). For instance we can use two colons in a row and then a number specifying the frequency to grab elements. For example:

```
In [22]: # Grab everything, but go in steps size of 1
         s[::1]

Out[22]: 'Hello World'

In [23]: # Grab everything, but go in step sizes of 2
         s[::2]

Out[23]: 'HloWrd'

In [24]: # We can use this to print a string backwards
         s[::-1]

Out[24]: 'dlroW olleH'
```

1.5 String Properties

It's important to note that strings have an important property known as *immutability*. This means that once a string is created, the elements within it can not be changed or replaced. For example:

```
In [25]: s
```

```
Out[25]: 'Hello World'
```

```
In [26]: # Let's try to change the first letter to 'x'
         s[0] = 'x'
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-26-976942677f11> in <module>()
      1 # Let's try to change the first letter to 'x'
----> 2 s[0] = 'x'


TypeError: 'str' object does not support item assignment
```

Notice how the error tells us directly what we can't do, change the item assignment! Something we *can* do is concatenate strings!

```
In [27]: s
```

```
Out[27]: 'Hello World'
```

```
In [28]: # Concatenate strings!
         s + ' concatenate me!'
```

```
Out[28]: 'Hello World concatenate me!'
```

```
In [29]: # We can reassign s completely though!
         s = s + ' concatenate me!'
```

```
In [30]: print(s)
```

```
Hello World concatenate me!
```

```
In [31]: s
```

```
Out[31]: 'Hello World concatenate me!'
```

We can use the multiplication symbol to create repetition!

```
In [32]: letter = 'z'
```

```
In [33]: letter*10
```

```
Out[33]: 'zzzzzzzzzz'
```

1.6 Basic Built-in String methods

Objects in Python usually have built-in methods. These methods are functions inside the object (we will learn about these in much more depth later) that can perform actions or commands on the object itself.

We call methods with a period and then the method name. Methods are in the form:
object.method(parameters)

Where parameters are extra arguments we can pass into the method. Don't worry if the details don't make 100% sense right now. Later on we will be creating our own objects and functions!

Here are some examples of built-in methods in strings:

```
In [34]: s

Out[34]: 'Hello World concatenate me!'

In [35]: # Upper Case a string
         s.upper()

Out[35]: 'HELLO WORLD CONCATENATE ME!'

In [36]: # Lower case
         s.lower()

Out[36]: 'hello world concatenate me!'

In [37]: # Split a string by blank space (this is the default)
         s.split()

Out[37]: ['Hello', 'World', 'concatenate', 'me!']

In [38]: # Split by a specific element (doesn't include the element that was split on)
         s.split('W')

Out[38]: ['Hello ', 'orld concatenate me!']
```

There are many more methods than the ones covered here. Visit the Advanced String section to find out more!

1.7 Print Formatting

We can use the .format() method to add formatted objects to printed string statements.

The easiest way to show this is through an example:

```
In [39]: 'Insert another string with curly brackets: {}'.format('The inserted string')

Out[39]: 'Insert another string with curly brackets: The inserted string'
```

We will revisit this string formatting topic in later sections when we are building our projects!

1.8 Next up: Lists!

03-Print Formatting with Strings

April 9, 2019

1 String Formatting

String formatting lets you inject items into a string rather than trying to chain items together using commas or string concatenation. As a quick comparison, consider:

```
player = 'Thomas'
points = 33

'Last night, '+player+' scored '+str(points)+' points.'  # concatenation

f'Last night, {player} scored {points} points.'          # string formatting
```

There are three ways to perform string formatting. * The oldest method involves placeholders using the modulo `%` character. * An improved technique uses the `.format()` string method. * The newest method, introduced with Python 3.6, uses formatted string literals, called *f-strings*.

Since you will likely encounter all three versions in someone else's code, we describe each of them here.

1.1 Formatting with placeholders

You can use %s to inject strings into your print statements. The modulo `%` is referred to as a "string formatting operator".

```
In [1]: print("I'm going to inject %s here." %'something')
```

```
I'm going to inject something here.
```

You can pass multiple items by placing them inside a tuple after the `%` operator.

```
In [2]: print("I'm going to inject %s text here, and %s text here." %('some','more'))
```

```
I'm going to inject some text here, and more text here.
```

You can also pass variable names:

```
In [3]: x, y = 'some', 'more'
        print("I'm going to inject %s text here, and %s text here."%(x,y))
```

```
I'm going to inject some text here, and more text here.
```

1.1.1 Format conversion methods.

It should be noted that two methods %s and %r convert any python object to a string using two separate methods: `str()` and `repr()`. We will learn more about these functions later on in the course, but you should note that `%r` and `repr()` deliver the *string representation* of the object, including quotation marks and any escape characters.

```
In [4]: print('He said his name was %s.' %'Fred')
        print('He said his name was %r.' %'Fred')
```

```
He said his name was Fred.
He said his name was 'Fred'.
```

As another example, `\t` inserts a tab into a string.

```
In [5]: print('I once caught a fish %s.' %'this \tbig')
        print('I once caught a fish %r.' %'this \tbig')
```

```
I once caught a fish this 	big.
I once caught a fish 'this \tbig'.
```

The `%s` operator converts whatever it sees into a string, including integers and floats. The `%d` operator converts numbers to integers first, without rounding. Note the difference below:

```
In [6]: print('I wrote %s programs today.' %3.75)
        print('I wrote %d programs today.' %3.75)
```

```
I wrote 3.75 programs today.
I wrote 3 programs today.
```

1.1.2 Padding and Precision of Floating Point Numbers

Floating point numbers use the format %5.2f. Here, 5 would be the minimum number of characters the string should contain; these may be padded with whitespace if the entire number does not have this many digits. Next to this, .2f stands for how many numbers to show past the decimal point. Let's see some examples:

```
In [7]: print('Floating point numbers: %5.2f' %(13.144))
```

```
Floating point numbers: 13.14
```

```
In [8]: print('Floating point numbers: %1.0f' %(13.144))
```

```
Floating point numbers: 13
```

```
In [9]: print('Floating point numbers: %1.5f' %(13.144))
```

```
Floating point numbers: 13.14400
```

```
In [10]: print('Floating point numbers: %10.2f' %(13.144))

Floating point numbers:      13.14
```

```
In [11]: print('Floating point numbers: %25.2f' %(13.144))

Floating point numbers:                     13.14
```

For more information on string formatting with placeholders visit https://docs.python.org/3/library/stdtypes.html#old-string-formatting

1.1.3 Multiple Formatting

Nothing prohibits using more than one conversion tool in the same print statement:

```
In [12]: print('First: %s, Second: %5.2f, Third: %r' %('hi!',3.1415,'bye!'))

First: hi!, Second:  3.14, Third: 'bye!'
```

1.2 Formatting with the `.format()` method

A better way to format objects into your strings for print statements is with the string `.format()` method. The syntax is:

```
'String here {} then also {}'.format('something1','something2')
```

For example:

```
In [13]: print('This is a string with an {}'.format('insert'))

This is a string with an insert
```

1.2.1 The .format() method has several advantages over the %s placeholder method:

1. Inserted objects can be called by index position:

```
In [14]: print('The {2} {1} {0}'.format('fox','brown','quick'))

The quick brown fox
```

2. Inserted objects can be assigned keywords:

```
In [15]: print('First Object: {a}, Second Object: {b}, Third Object: {c}'.format(a=1,b='Two',c=

First Object: 1, Second Object: Two, Third Object: 12.3
```

3. Inserted objects can be reused, avoiding duplication:

```
In [16]: print('A %s saved is a %s earned.' %('penny','penny'))
         # vs.
         print('A {p} saved is a {p} earned.'.format(p='penny'))
```

```
A penny saved is a penny earned.
A penny saved is a penny earned.
```

1.2.2 Alignment, padding and precision with `.format()`

Within the curly braces you can assign field lengths, left/right alignments, rounding parameters and more

```
In [17]: print('{0:8} | {1:9}'.format('Fruit', 'Quantity'))
         print('{0:8} | {1:9}'.format('Apples', 3.))
         print('{0:8} | {1:9}'.format('Oranges', 10))
```

```
Fruit    | Quantity
Apples   |       3.0
Oranges  |        10
```

By default, `.format()` aligns text to the left, numbers to the right. You can pass an optional <,^, or > to set a left, center or right alignment:

```
In [18]: print('{0:<8} | {1:^8} | {2:>8}'.format('Left','Center','Right'))
         print('{0:<8} | {1:^8} | {2:>8}'.format(11,22,33))
```

```
Left     |  Center  |    Right
11       |    22    |       33
```

You can precede the aligment operator with a padding character

```
In [19]: print('{0:=<8} | {1:-^8} | {2:.>8}'.format('Left','Center','Right'))
         print('{0:=<8} | {1:-^8} | {2:.>8}'.format(11,22,33))
```

```
Left==== | -Center- | ...Right
11====== | ---22--- | ...33
```

Field widths and float precision are handled in a way similar to placeholders. The following two print statements are equivalent:

```
In [20]: print('This is my ten-character, two-decimal number:%10.2f' %13.579)
         print('This is my ten-character, two-decimal number:{0:10.2f}'.format(13.579))
```

```
This is my ten-character, two-decimal number:     13.58
This is my ten-character, two-decimal number:     13.58
```

Note that there are 5 spaces following the colon, and 5 characters taken up by 13.58, for a total of ten characters.

For more information on the string `.format()` method visit https://docs.python.org/3/library/string.html#formatstrings

1.3 Formatted String Literals (f-strings)

Introduced in Python 3.6, f-strings offer several benefits over the older `.format()` string method described above. For one, you can bring outside variables immediately into to the string rather than pass them as arguments through `.format(var)`.

```
In [21]: name = 'Fred'

         print(f"He said his name is {name}.")
```

```
He said his name is Fred.
```

Pass `!r` to get the string representation:

```
In [22]: print(f"He said his name is {name!r}")
```

```
He said his name is 'Fred'
```

Float formatting follows `"result: {value:{width}.{precision}}"` Where with the `.format()` method you might see `{value:10.4f}`, with f-strings this can become `{value:{10}.{6}}`

```
In [23]: num = 23.45678
         print("My 10 character, four decimal number is:{0:10.4f}".format(num))
         print(f"My 10 character, four decimal number is:{num:{10}.{6}}")
```

```
My 10 character, four decimal number is:   23.4568
My 10 character, four decimal number is:   23.4568
```

Note that with f-strings, *precision* refers to the total number of digits, not just those following the decimal. This fits more closely with scientific notation and statistical analysis. Unfortunately, f-strings do not pad to the right of the decimal, even if precision allows it:

```
In [24]: num = 23.45
         print("My 10 character, four decimal number is:{0:10.4f}".format(num))
         print(f"My 10 character, four decimal number is:{num:{10}.{6}}")
```

```
My 10 character, four decimal number is:   23.4500
My 10 character, four decimal number is:     23.45
```

If this becomes important, you can always use `.format()` method syntax inside an f-string:

```
In [25]: num = 23.45
         print("My 10 character, four decimal number is:{0:10.4f}".format(num))
         print(f"My 10 character, four decimal number is:{num:10.4f}")
```

```
My 10 character, four decimal number is:   23.4500
My 10 character, four decimal number is:   23.4500
```

For more info on formatted string literals visit https://docs.python.org/3/reference/lexical_analysis.html#f-strings

That is the basics of string formatting!

04-Lists

April 9, 2019

1 Lists

Earlier when discussing strings we introduced the concept of a *sequence* in Python. Lists can be thought of the most general version of a *sequence* in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed!

In this section we will learn about:

```
1.) Creating lists
2.) Indexing and Slicing Lists
3.) Basic List Methods
4.) Nesting Lists
5.) Introduction to List Comprehensions
```

Lists are constructed with brackets [] and commas separating every element in the list.
Let's go ahead and see how we can construct lists!

```
In [1]: # Assign a list to an variable named my_list
        my_list = [1,2,3]
```

We just created a list of integers, but lists can actually hold different object types. For example:

```
In [2]: my_list = ['A string',23,100.232,'o']
```

Just like strings, the len() function will tell you how many items are in the sequence of the list.

```
In [3]: len(my_list)

Out[3]: 4
```

1.0.1 Indexing and Slicing

Indexing and slicing work just like in strings. Let's make a new list to remind ourselves of how this works:

```
In [4]: my_list = ['one','two','three',4,5]

In [5]: # Grab element at index 0
        my_list[0]
```

```
Out[5]: 'one'

In [6]: # Grab index 1 and everything past it
        my_list[1:]

Out[6]: ['two', 'three', 4, 5]

In [7]: # Grab everything UP TO index 3
        my_list[:3]

Out[7]: ['one', 'two', 'three']
```

We can also use + to concatenate lists, just like we did for strings.

```
In [8]: my_list + ['new item']

Out[8]: ['one', 'two', 'three', 4, 5, 'new item']
```

Note: This doesn't actually change the original list!

```
In [9]: my_list

Out[9]: ['one', 'two', 'three', 4, 5]
```

You would have to reassign the list to make the change permanent.

```
In [10]: # Reassign
         my_list = my_list + ['add new item permanently']

In [11]: my_list

Out[11]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

We can also use the * for a duplication method similar to strings:

```
In [12]: # Make the list double
         my_list * 2

Out[12]: ['one',
          'two',
          'three',
          4,
          5,
          'add new item permanently',
          'one',
          'two',
          'three',
          4,
          5,
          'add new item permanently']

In [13]: # Again doubling not permanent
         my_list

Out[13]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

1.1 Basic List Methods

If you are familiar with another programming language, you might start to draw parallels between arrays in another language and lists in Python. Lists in Python however, tend to be more flexible than arrays in other languages for a two good reasons: they have no fixed size (meaning we don't have to specify how big a list will be), and they have no fixed type constraint (like we've seen above).

Let's go ahead and explore some more special methods for lists:

```
In [14]: # Create a new list
         list1 = [1,2,3]
```

Use the **append** method to permanently add an item to the end of a list:

```
In [15]: # Append
         list1.append('append me!')

In [16]: # Show
         list1

Out[16]: [1, 2, 3, 'append me!']
```

Use **pop** to "pop off" an item from the list. By default pop takes off the last index, but you can also specify which index to pop off. Let's see an example:

```
In [17]: # Pop off the 0 indexed item
         list1.pop(0)

Out[17]: 1

In [18]: # Show
         list1

Out[18]: [2, 3, 'append me!']

In [19]: # Assign the popped element, remember default popped index is -1
         popped_item = list1.pop()

In [20]: popped_item

Out[20]: 'append me!'

In [21]: # Show remaining list
         list1

Out[21]: [2, 3]
```

It should also be noted that lists indexing will return an error if there is no element at that index. For example:

```
In [22]: list1[100]
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-22-af6d2015fa1f> in <module>()
----> 1 list1[100]


IndexError: list index out of range
```

We can use the **sort** method and the **reverse** methods to also effect your lists:

```
In [23]: new_list = ['a','e','x','b','c']

In [24]: #Show
         new_list

Out[24]: ['a', 'e', 'x', 'b', 'c']

In [25]: # Use reverse to reverse order (this is permanent!)
         new_list.reverse()

In [26]: new_list

Out[26]: ['c', 'b', 'x', 'e', 'a']

In [27]: # Use sort to sort the list (in this case alphabetical order, but for numbers it will
         new_list.sort()

In [28]: new_list

Out[28]: ['a', 'b', 'c', 'e', 'x']
```

1.2 Nesting Lists

A great feature of of Python data structures is that they support *nesting*. This means we can have data structures within data structures. For example: A list inside a list.

Let's see how this works!

```
In [29]: # Let's make three lists
         lst_1=[1,2,3]
         lst_2=[4,5,6]
         lst_3=[7,8,9]

         # Make a list of lists to form a matrix
         matrix = [lst_1,lst_2,lst_3]

In [30]: # Show
         matrix
```

```
Out[30]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

We can again use indexing to grab elements, but now there are two levels for the index. The items in the matrix object, and then the items inside that list!

```
In [31]: # Grab first item in matrix object
         matrix[0]

Out[31]: [1, 2, 3]

In [32]: # Grab first item of the first item in the matrix object
         matrix[0][0]

Out[32]: 1
```

2 List Comprehensions

Python has an advanced feature called list comprehensions. They allow for quick construction of lists. To fully understand list comprehensions we need to understand for loops. So don't worry if you don't completely understand this section, and feel free to just skip it since we will return to this topic later.

But in case you want to know now, here are a few examples!

```
In [33]: # Build a list comprehension by deconstructing a for loop within a []
         first_col = [row[0] for row in matrix]

In [34]: first_col

Out[34]: [1, 4, 7]
```

We used a list comprehension here to grab the first element of every row in the matrix object. We will cover this in much more detail later on!

For more advanced methods and features of lists in Python, check out the Advanced Lists section later on in this course!

05-Dictionaries

April 9, 2019

1 Dictionaries

We've been learning about *sequences* in Python but now we're going to switch gears and learn about *mappings* in Python. If you're familiar with other languages you can think of these Dictionaries as hash tables.

This section will serve as a brief introduction to dictionaries and consist of:

```
1.) Constructing a Dictionary
2.) Accessing objects from a dictionary
3.) Nesting Dictionaries
4.) Basic Dictionary Methods
```

So what are mappings? Mappings are a collection of objects that are stored by a *key*, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.

A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

1.1 Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work!

```
In [1]: # Make a dictionary with {} and : to signify a key and a value
        my_dict = {'key1':'value1','key2':'value2'}

In [2]: # Call values by their key
        my_dict['key2']

Out[2]: 'value2'
```

Its important to note that dictionaries are very flexible in the data types they can hold. For example:

```
In [3]: my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}

In [4]: # Let's call items from the dictionary
        my_dict['key3']

Out[4]: ['item0', 'item1', 'item2']
```

```
In [5]: # Can call an index on that value
        my_dict['key3'][0]

Out[5]: 'item0'

In [6]: # Can then even call methods on that value
        my_dict['key3'][0].upper()

Out[6]: 'ITEM0'
```

We can affect the values of a key as well. For instance:

```
In [7]: my_dict['key1']

Out[7]: 123

In [8]: # Subtract 123 from the value
        my_dict['key1'] = my_dict['key1'] - 123

In [9]: #Check
        my_dict['key1']

Out[9]: 0
```

A quick note, Python has a built-in method of doing a self subtraction or addition (or multiplication or division). We could have also used += or -= for the above statement. For example:

```
In [10]: # Set the object equal to itself minus 123
         my_dict['key1'] -= 123
         my_dict['key1']

Out[10]: -123
```

We can also create keys by assignment. For instance if we started off with an empty dictionary, we could continually add to it:

```
In [11]: # Create a new dictionary
         d = {}

In [12]: # Create a new key through assignment
         d['animal'] = 'Dog'

In [13]: # Can do this with any object
         d['answer'] = 42

In [14]: #Show
         d

Out[14]: {'animal': 'Dog', 'answer': 42}
```

1.2 Nesting with Dictionaries

Hopefully you're starting to see how powerful Python is with its flexibility of nesting objects and calling methods on them. Let's see a dictionary nested inside a dictionary:

```
In [15]: # Dictionary nested inside a dictionary nested inside a dictionary
         d = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

Wow! That's a quite the inception of dictionaries! Let's see how we can grab that value:

```
In [16]: # Keep calling the keys
         d['key1']['nestkey']['subnestkey']

Out[16]: 'value'
```

1.3 A few Dictionary Methods

There are a few methods we can call on a dictionary. Let's get a quick introduction to a few of them:

```
In [17]: # Create a typical dictionary
         d = {'key1':1,'key2':2,'key3':3}

In [18]: # Method to return a list of all keys
         d.keys()

Out[18]: dict_keys(['key1', 'key2', 'key3'])

In [19]: # Method to grab all values
         d.values()

Out[19]: dict_values([1, 2, 3])

In [20]: # Method to return tuples of all items  (we'll learn about tuples soon)
         d.items()

Out[20]: dict_items([('key1', 1), ('key2', 2), ('key3', 3)])
```

Hopefully you now have a good basic understanding how to construct dictionaries. There's a lot more to go into here, but we will revisit dictionaries at later time. After this section all you need to know is how to create a dictionary and how to retrieve values from it.

06-Tuples

April 9, 2019

1 Tuples

In Python tuples are very similar to lists, however, unlike lists they are *immutable* meaning they can not be changed. You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.

In this section, we will get a brief overview of the following:

```
1.) Constructing Tuples
2.) Basic Tuple Methods
3.) Immutability
4.) When to Use Tuples
```

You'll have an intuition of how to use tuples based on what you've learned about lists. We can treat them very similarly with the major distinction being that tuples are immutable.

1.1 Constructing Tuples

The construction of a tuples use () with elements separated by commas. For example:

```
In [1]: # Create a tuple
        t = (1,2,3)

In [2]: # Check len just like a list
        len(t)

Out[2]: 3

In [3]: # Can also mix object types
        t = ('one',2)

        # Show
        t

Out[3]: ('one', 2)

In [4]: # Use indexing just like we did in lists
        t[0]

Out[4]: 'one'
```

```
In [5]: # Slicing just like a list
        t[-1]
```

```
Out[5]: 2
```

1.2 Basic Tuple Methods

Tuples have built-in methods, but not as many as lists do. Let's look at two of them:

```
In [6]: # Use .index to enter a value and return the index
        t.index('one')
```

```
Out[6]: 0
```

```
In [7]: # Use .count to count the number of times a value appears
        t.count('one')
```

```
Out[7]: 1
```

1.3 Immutability

It can't be stressed enough that tuples are immutable. To drive that point home:

```
In [8]: t[0]= 'change'
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-8-1257c0aa9edd> in <module>()
    ----> 1 t[0]= 'change'


        TypeError: 'tuple' object does not support item assignment
```

Because of this immutability, tuples can't grow. Once a tuple is made we can not add to it.

```
In [9]: t.append('nope')
```

```
        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-9-b75f5b09ac19> in <module>()
    ----> 1 t.append('nope')


        AttributeError: 'tuple' object has no attribute 'append'
```

1.4 When to use Tuples

You may be wondering, "Why bother using tuples when they have fewer available methods?" To be honest, tuples are not used as often as lists in programming, but are used when immutability is necessary. If in your program you are passing around an object and need to make sure it does not get changed, then a tuple becomes your solution. It provides a convenient source of data integrity.

You should now be able to create and use tuples in your programming as well as have an understanding of their immutability.

Up next Sets and Booleans!!

07-Sets and Booleans

April 9, 2019

1 Set and Booleans

There are two other object types in Python that we should quickly cover: Sets and Booleans.

1.1 Sets

Sets are an unordered collection of *unique* elements. We can construct them by using the set() function. Let's go ahead and make a set to see how it works

```
In [1]: x = set()

In [2]: # We add to sets with the add() method
        x.add(1)

In [3]: #Show
        x

Out[3]: {1}
```

Note the curly brackets. This does not indicate a dictionary! Although you can draw analogies as a set being a dictionary with only keys.

We know that a set has only unique entries. So what happens when we try to add something that is already in a set?

```
In [4]: # Add a different element
        x.add(2)

In [5]: #Show
        x

Out[5]: {1, 2}

In [6]: # Try to add the same element
        x.add(1)

In [7]: #Show
        x

Out[7]: {1, 2}
```

Notice how it won't place another 1 there. That's because a set is only concerned with unique elements! We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

```
In [8]: # Create a list with repeats
        list1 = [1,1,2,2,3,4,5,6,1,1]

In [9]: # Cast as set to get unique values
        set(list1)

Out[9]: {1, 2, 3, 4, 5, 6}
```

1.2 Booleans

Python comes with Booleans (with predefined True and False displays that are basically just the integers 1 and 0). It also has a placeholder object called None. Let's walk through a few quick examples of Booleans (we will dive deeper into them later in this course).

```
In [10]: # Set object to be a boolean
         a = True

In [11]: #Show
         a

Out[11]: True
```

We can also use comparison operators to create booleans. We will go over all the comparison operators later on in the course.

```
In [12]: # Output is boolean
         1 > 2

Out[12]: False
```

We can use None as a placeholder for an object that we don't want to reassign yet:

```
In [13]: # None placeholder
         b = None

In [14]: # Show
         print(b)

None
```

Thats it! You should now have a basic understanding of Python objects and data structure types. Next, go ahead and do the assessment test!

08-Files

April 9, 2019

1 Files

Python uses file objects to interact with external files on your computer. These file objects can be any sort of file you have on your computer, whether it be an audio file, a text file, emails, Excel documents, etc. Note: You will probably need to install certain libraries or modules to interact with those various file types, but they are easily available. (We will cover downloading modules later on in the course).

Python has a built-in open function that allows us to open and play with basic file types. First we will need a file though. We're going to use some IPython magic to create a text file!

1.1 IPython Writing a File

This function is specific to jupyter notebooks! Alternatively, quickly create a simple .txt file with sublime text editor.

```
In [1]: %%writefile test.txt
        Hello, this is a quick test file.

Overwriting test.txt
```

1.2 Python Opening a file

Let's being by opening the file test.txt that is located in the same directory as this notebook. For now we will work with files located in the same directory as the notebook or .py script you are using.

It is very easy to get an error on this step:

```
In [1]: myfile = open('whoops.txt')


        ---------------------------------------------------------------------------

        FileNotFoundError                         Traceback (most recent call last)

        <ipython-input-1-dafe28ee473f> in <module>()
    ----> 1 myfile = open('whoops.txt')
```

```
FileNotFoundError: [Errno 2] No such file or directory: 'whoops.txt'
```

To avoid this error,make sure your .txt file is saved in the same location as your notebook, to check your notebook location, use **pwd**:

```
In [2]: pwd
```

```
Out[2]: 'C:\\Users\\Marcial\\Pierian-Data-Courses\\Complete-Python-3-Bootcamp\\00-Python Objec
```

Alternatively, to grab files from any location on your computer, simply pass in the entire file path.

For Windows you need to use double so python doesn't treat the second as an escape character, a file path is in the form:

```
myfile = open("C:\\Users\\YourUserName\\Home\\Folder\\myfile.txt")
```

For MacOS and Linux you use slashes in the opposite direction:

```
myfile = open("/Users/YouUserName/Folder/myfile.txt")
```

```
In [2]: # Open the text.txt we made earlier
        my_file = open('test.txt')
```

```
In [3]: # We can now read the file
        my_file.read()
```

```
Out[3]: 'Hello, this is a quick test file.'
```

```
In [4]: # But what happens if we try to read it again?
        my_file.read()
```

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

This happens because you can imagine the reading "cursor" is at the end of the file after having read it. So there is nothing left to read. We can reset the "cursor" like this:

```
In [5]: # Seek to the start of file (index 0)
        my_file.seek(0)
```

```
Out[5]: 0
```

```
In [6]: # Now read again
        my_file.read()
```

```
Out[6]: 'Hello, this is a quick test file.'
```

You can read a file line by line using the readlines method. Use caution with large files, since everything will be held in memory. We will learn how to iterate over large files later in the course.

```
In [7]: # Readlines returns a list of the lines in the file
        my_file.seek(0)
        my_file.readlines()
```

```
Out[7]: ['Hello, this is a quick test file.']
```

When you have finished using a file, it is always good practice to close it.

```
In [8]: my_file.close()
```

1.3 Writing to a File

By default, the `open()` function will only allow us to read the file. We need to pass the argument `'w'` to write over the file. For example:

```
In [9]: # Add a second argument to the function, 'w' which stands for write.
        # Passing 'w+' lets us read and write to the file

        my_file = open('test.txt','w+')
```

1.3.1 Use caution!

Opening a file with `'w'` or `'w+'` truncates the original, meaning that anything that was in the original file **is deleted**!

```
In [10]: # Write to the file
         my_file.write('This is a new line')

Out[10]: 18

In [11]: # Read the file
         my_file.seek(0)
         my_file.read()

Out[11]: 'This is a new line'

In [12]: my_file.close()  # always do this when you're done with a file
```

1.4 Appending to a File

Passing the argument `'a'` opens the file and puts the pointer at the end, so anything written is appended. Like `'w+'`, `'a+'` lets us read and write to a file. If the file does not exist, one will be created.

```
In [13]: my_file = open('test.txt','a+')
         my_file.write('\nThis is text being appended to test.txt')
         my_file.write('\nAnd another line here.')

Out[13]: 23

In [14]: my_file.seek(0)
         print(my_file.read())

This is a new line
This is text being appended to test.txt
And another line here.


In [15]: my_file.close()
```

1.4.1 Appending with %%writefile

We can do the same thing using IPython cell magic:

```
In [16]: %%writefile -a test.txt

         This is text being appended to test.txt
         And another line here.

Appending to test.txt
```

Add a blank space if you want the first line to begin on its own line, as Jupyter won't recognize escape sequences like \n

1.5 Iterating through a File

Lets get a quick preview of a for loop by iterating over a text file. First let's make a new text file with some IPython Magic:

```
In [17]: %%writefile test.txt
         First Line
         Second Line

Overwriting test.txt
```

Now we can use a little bit of flow to tell the program to for through every line of the file and do something:

```
In [18]: for line in open('test.txt'):
             print(line)

First Line

Second Line
```

Don't worry about fully understanding this yet, for loops are coming up soon. But we'll break down what we did above. We said that for every line in this text file, go ahead and print that line. It's important to note a few things here:

1. We could have called the "line" object anything (see example below).
2. By not calling `.read()` on the file, the whole text file was not stored in memory.
3. Notice the indent on the second line for print. This whitespace is required in Python.

```
In [19]: # Pertaining to the first point above
         for asdf in open('test.txt'):
             print(asdf)
```

```
First Line

Second Line
```

We'll learn a lot more about this later, but up next: Sets and Booleans!

09-Objects and Data Structures Assessment Test

April 9, 2019

1 Objects and Data Structures Assessment Test

1.1 Test your knowledge.

** Answer the following questions **

Write a brief description of all the following Object Types and Data Structures we've learned about:

Numbers:
Strings:
Lists:
Tuples:
Dictionaries:

1.2 Numbers

Write an equation that uses multiplication, division, an exponent, addition, and subtraction that is equal to 100.25.

Hint: This is just to test your memory of the basic arithmetic commands, work backwards from 100.25

In [ ]:

Answer these 3 questions without typing code. Then type code to check your answer.

```
What is the value of the expression 4 * (6 + 5)

What is the value of the expression 4 * 6 + 5

What is the value of the expression 4 + 6 * 5
```

In [ ]:

What is the *type* of the result of the expression 3 + 1.5 + 4?

What would you use to find a number's square root, as well as its square?

In [ ]: # Square root:

In [ ]: # Square:

1.3 Strings

Given the string 'hello' give an index command that returns 'e'. Enter your code in the cell below:

```
In [ ]: s = 'hello'
        # Print out 'e' using indexing
```

Reverse the string 'hello' using slicing:

```
In [ ]: s ='hello'
        # Reverse the string using slicing
```

Given the string hello, give two methods of producing the letter 'o' using indexing.

```
In [ ]: s ='hello'
        # Print out the 'o'

        # Method 1:
```

```
In [ ]: # Method 2:
```

1.4 Lists

Build this list [0,0,0] two separate ways.

```
In [ ]: # Method 1:
```

```
In [ ]: # Method 2:
```

Reassign 'hello' in this nested list to say 'goodbye' instead:

```
In [ ]: list3 = [1,2,[3,4,'hello']]
```

Sort the list below:

```
In [ ]: list4 = [5,3,4,6,1]
```

1.5 Dictionaries

Using keys and indexing, grab the 'hello' from the following dictionaries:

```
In [ ]: d = {'simple_key':'hello'}
        # Grab 'hello'
```

```
In [ ]: d = {'k1':{'k2':'hello'}}
        # Grab 'hello'
```

```
In [ ]: # Getting a little tricker
        d = {'k1':[{'nest_key':['this is deep',['hello']]}]}

        #Grab hello
```

```
In [ ]: # This will be hard and annoying!
        d = {'k1':[1,2,{'k2':['this is tricky',{'tough':[1,2,['hello']]}]}]}
```

Can you sort a dictionary? Why or why not?

1.6 Tuples

What is the major difference between tuples and lists?

How do you create a tuple?

1.7 Sets

What is unique about a set?

Use a set to find the unique values of the list below:

```
In [ ]: list5 = [1,2,2,33,4,4,11,22,3,3,2]
```

1.8 Booleans

For the following quiz questions, we will get a preview of comparison operators. In the table below, a=3 and b=4.

Operator

Description

Example

==

If the values of two operands are equal, then the condition becomes true.

(a == b) is not true.

!=

If values of two operands are not equal, then condition becomes true.

(a != b) is true.

>

If the value of left operand is greater than the value of right operand, then condition becomes true.

(a > b) is not true.

<

If the value of left operand is less than the value of right operand, then condition becomes true.

(a < b) is true.

>=

If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.

(a >= b) is not true.

<=

If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

(a <= b) is true.

What will be the resulting Boolean of the following pieces of code (answer fist then check by typing it in!)

```
In [ ]: # Answer before running cell
        2 > 3
```

```
In [ ]: # Answer before running cell
        3 <= 2
```

```
In [ ]: # Answer before running cell
        3 == 2.0
```

```
In [ ]: # Answer before running cell
        3.0 == 3
```

```
In [ ]: # Answer before running cell
        4**0.5 != 2
```

Final Question: What is the boolean output of the cell block below?

```
In [ ]: # two nested lists
        l_one = [1,2,[3,4]]
        l_two = [1,2,{'k1':4}]

        # True or False?
        l_one[2][0] >= l_two[2]['k1']
```

1.9 Great Job on your first assessment!

10-Objects and Data Structures Assessment Test-Solution

April 9, 2019

1 Objects and Data Structures Assessment Test

1.1 Test your knowledge.

** Answer the following questions **

Write a brief description of all the following Object Types and Data Structures we've learned about:

For the full answers, review the Jupyter notebook introductions of each topic!

Numbers
Strings
Lists
Tuples
Dictionaries

1.2 Numbers

Write an equation that uses multiplication, division, an exponent, addition, and subtraction that is equal to 100.25.

Hint: This is just to test your memory of the basic arithmetic commands, work backwards from 100.25

```
In [1]: # Your answer is probably different
        (60 + (10 ** 2) / 4 * 7) - 134.75
```

```
Out[1]: 100.25
```

Answer these 3 questions without typing code. Then type code to check your answer.

```
What is the value of the expression 4 * (6 + 5)

What is the value of the expression 4 * 6 + 5

What is the value of the expression 4 + 6 * 5
```

```
In [2]: 4 * (6 + 5)
```

```
Out[2]: 44
```

```
In [3]: 4 * 6 + 5
```

```
Out[3]: 29

In [4]: 4 + 6 * 5

Out[4]: 34
```

What is the *type* of the result of the expression 3 + 1.5 + 4?
Answer: Floating Point Number
What would you use to find a number's square root, as well as its square?

```
In [5]: # Square root:
        100 ** 0.5

Out[5]: 10.0

In [6]: # Square:
        10 ** 2

Out[6]: 100
```

1.3 Strings

Given the string 'hello' give an index command that returns 'e'. Enter your code in the cell below:

```
In [7]: s = 'hello'
        # Print out 'e' using indexing

        s[1]

Out[7]: 'e'
```

Reverse the string 'hello' using slicing:

```
In [8]: s ='hello'
        # Reverse the string using slicing

        s[::-1]

Out[8]: 'olleh'
```

Given the string 'hello', give two methods of producing the letter 'o' using indexing.

```
In [9]: s ='hello'
        # Print out the 'o'

        # Method 1:

        s[-1]

Out[9]: 'o'

In [10]: # Method 2:

         s[4]

Out[10]: 'o'
```

1.4 Lists

Build this list [0,0,0] two separate ways.

```
In [11]: # Method 1:
         [0]*3

Out[11]: [0, 0, 0]

In [12]: # Method 2:
         list2 = [0,0,0]
         list2

Out[12]: [0, 0, 0]
```

Reassign 'hello' in this nested list to say 'goodbye' instead:

```
In [13]: list3 = [1,2,[3,4,'hello']]

In [14]: list3[2][2] = 'goodbye'

In [15]: list3

Out[15]: [1, 2, [3, 4, 'goodbye']]
```

Sort the list below:

```
In [16]: list4 = [5,3,4,6,1]

In [17]: # Method 1:
         sorted(list4)

Out[17]: [1, 3, 4, 5, 6]

In [18]: # Method 2:
         list4.sort()
         list4

Out[18]: [1, 3, 4, 5, 6]
```

1.5 Dictionaries

Using keys and indexing, grab the 'hello' from the following dictionaries:

```
In [19]: d = {'simple_key':'hello'}
         # Grab 'hello'

         d['simple_key']

Out[19]: 'hello'
```

```
In [20]: d = {'k1':{'k2':'hello'}}
         # Grab 'hello'

         d['k1']['k2']

Out[20]: 'hello'

In [21]: # Getting a little tricker
         d = {'k1':[{'nest_key':['this is deep',['hello']]}]}

In [22]: # This was harder than I expected...
         d['k1'][0]['nest_key'][1][0]

Out[22]: 'hello'

In [23]: # This will be hard and annoying!
         d = {'k1':[1,2,{'k2':['this is tricky',{'tough':[1,2,['hello']]}]}]}

In [24]: # Phew!
         d['k1'][2]['k2'][1]['tough'][2][0]

Out[24]: 'hello'
```

Can you sort a dictionary? Why or why not?
Answer: No! Because normal dictionaries are *mappings* not a sequence.

1.6 Tuples

What is the major difference between tuples and lists?
Tuples are immutable!
How do you create a tuple?

```
In [25]: t = (1,2,3)
```

1.7 Sets

What is unique about a set?
Answer: They don't allow for duplicate items!
Use a set to find the unique values of the list below:

```
In [26]: list5 = [1,2,2,33,4,4,11,22,3,3,2]

In [27]: set(list5)

Out[27]: {1, 2, 3, 4, 11, 22, 33}
```

1.8 Booleans

For the following quiz questions, we will get a preview of comparison operators. In the table below, a=3 and b=4.

Operator
Description
Example
==
If the values of two operands are equal, then the condition becomes true.
(a == b) is not true.
!=
If values of two operands are not equal, then condition becomes true.
(a != b) is true.
>
If the value of left operand is greater than the value of right operand, then condition becomes true.
(a > b) is not true.
<
If the value of left operand is less than the value of right operand, then condition becomes true.
(a < b) is true.
>=
If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.
(a >= b) is not true.
<=
If the value of left operand is less than or equal to the value of right operand, then condition becomes true.
(a <= b) is true.

What will be the resulting Boolean of the following pieces of code (answer fist then check by typing it in!)

```
In [28]: # Answer before running cell
         2 > 3

Out[28]: False

In [29]: # Answer before running cell
         3 <= 2

Out[29]: False

In [30]: # Answer before running cell
         3 == 2.0

Out[30]: False

In [31]: # Answer before running cell
         3.0 == 3

Out[31]: True
```

```
In [32]: # Answer before running cell
         4**0.5 != 2
```

```
Out[32]: False
```

Final Question: What is the boolean output of the cell block below?

```
In [33]: # two nested lists
         l_one = [1,2,[3,4]]
         l_two = [1,2,{'k1':4}]

         # True or False?
         l_one[2][0] >= l_two[2]['k1']
```

```
Out[33]: False
```

1.9 Great Job on your first assessment!

01-Comparison Operators

April 9, 2019

1 Comparison Operators

In this lecture we will be learning about Comparison Operators in Python. These operators will allow us to compare variables and output a Boolean value (True or False).

If you have any sort of background in Math, these operators should be very straight forward.

First we'll present a table of the comparison operators and then work through some examples:

Table of Comparison Operators

In the table below, a=3 and b=4.

Operator

Description

Example

==

If the values of two operands are equal, then the condition becomes true.

(a == b) is not true.

!=

If values of two operands are not equal, then condition becomes true.

(a != b) is true

>

If the value of left operand is greater than the value of right operand, then condition becomes true.

(a > b) is not true.

<

If the value of left operand is less than the value of right operand, then condition becomes true.

(a < b) is true.

>=

If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.

(a >= b) is not true.

<=

If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

(a <= b) is true.

Let's now work through quick examples of each of these.

Equal

```
In [1]: 2 == 2
```

```
Out[1]: True
```

```
In [2]: 1 == 0
```

```
Out[2]: False
```

Note that == is a comparison operator, while = is an assignment operator.

Not Equal

```
In [3]: 2 != 1
```

```
Out[3]: True
```

```
In [4]: 2 != 2
```

```
Out[4]: False
```

Greater Than

```
In [5]: 2 > 1
```

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

```
In [6]: 2 > 4
```

```
Out[6]: False
```

Less Than

```
In [7]: 2 < 4
```

```
Out[7]: True
```

```
In [8]: 2 < 1
```

```
Out[8]: False
```

Greater Than or Equal to

```
In [9]: 2 >= 2
```

```
Out[9]: True
```

```
In [10]: 2 >= 1
```

```
Out[10]: True
```

Less than or Equal to

```
In [11]: 2 <= 2
```

```
Out[11]: True
```

```
In [12]: 2 <= 4
```

```
Out[12]: True
```

Great! Go over each comparison operator to make sure you understand what each one is saying. But hopefully this was straightforward for you.

Next we will cover chained comparison operators

02-Chained Comparison Operators

April 9, 2019

1 Chained Comparison Operators

An interesting feature of Python is the ability to *chain* multiple comparisons to perform a more complex test. You can use these chained comparisons as shorthand for larger Boolean Expressions.

In this lecture we will learn how to chain comparison operators and we will also introduce two other important statements in Python: **and** and **or**.

Let's look at a few examples of using chains:

```
In [1]: 1 < 2 < 3
```

```
Out[1]: True
```

The above statement checks if 1 was less than 2 **and** if 2 was less than 3. We could have written this using an **and** statement in Python:

```
In [2]: 1<2 and 2<3
```

```
Out[2]: True
```

The **and** is used to make sure two checks have to be true in order for the total check to be true. Let's see another example:

```
In [3]: 1 < 3 > 2
```

```
Out[3]: True
```

The above checks if 3 is larger than both of the other numbers, so you could use **and** to rewrite it as:

```
In [4]: 1<3 and 3>2
```

```
Out[4]: True
```

It's important to note that Python is checking both instances of the comparisons. We can also use **or** to write comparisons in Python. For example:

```
In [5]: 1==2 or 2<3
```

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

Note how it was true; this is because with the **or** operator, we only need one *or* the other to be true. Let's see one more example to drive this home:

```
In [6]: 1==1 or 100==1
```

```
Out[6]: True
```

Great! For an overview of this quick lesson: You should have a comfortable understanding of using **and** and **or** statements as well as reading chained comparison code.

Go ahead and go to the quiz for this section to check your understanding!

01-Introduction to Python Statements

April 9, 2019

1 Introduction to Python Statements

In this lecture we will be doing a quick overview of Python Statements. This lecture will emphasize differences between Python and other languages such as C++.

There are two reasons we take this approach for learning the context of Python Statements:

```
1.) If you are coming from a different language this will rapidly accelerate your understandin
2.) Learning about statements will allow you to be able to read other languages more easily in
```

1.1 Python vs Other Languages

Let's create a simple statement that says: "If a is greater than b, assign 2 to a and 4 to b"

Take a look at these two if statements (we will learn about building out if statements soon).

Version 1 (Other Languages)

```
if (a>b){
    a = 2;
    b = 4;
}
```

Version 2 (Python)

```
if a>b:
    a = 2
    b = 4
```

You'll notice that Python is less cluttered and much more readable than the first version. How does Python manage this?

Let's walk through the main differences:

Python gets rid of () and {} by incorporating two main factors: a *colon* and *whitespace*. The statement is ended with a colon, and whitespace is used (indentation) to describe what takes place in case of the statement.

Another major difference is the lack of semicolons in Python. Semicolons are used to denote statement endings in many other languages, but in Python, the end of a line is the same as the end of a statement.

Lastly, to end this brief overview of differences, let's take a closer look at indentation syntax in Python vs other languages:

1.2 Indentation

Here is some pseudo-code to indicate the use of whitespace and indentation in Python:

Other Languages

```
if (x)
    if(y)
        code-statement;
else
    another-code-statement;
```

Python

```
if x:
    if y:
        code-statement
else:
    another-code-statement
```

Note how Python is so heavily driven by code indentation and whitespace. This means that code readability is a core part of the design of the Python language.

Now let's start diving deeper by coding these sort of statements in Python!

1.3 Time to code!

02-if, elif, and else Statements

April 9, 2019

1 if, elif, else Statements

if Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Verbally, we can imagine we are telling the computer:

"Hey if this case happens, perform some action"

We can then expand the idea further with elif and else statements, which allow us to tell the computer:

"Hey if this case happens, perform some action. Else, if another case happens, perform some other action. Else, if *none* of the above cases happened, perform this action."

Let's go ahead and look at the syntax format for if statements to get a better idea of this:

```
if case1:
    perform action1
elif case2:
    perform action2
else:
    perform action3
```

1.1 First Example

Let's see a quick example of this:

```
In [1]: if True:
            print('It was true!')
```

```
It was true!
```

Let's add in some else logic:

```
In [2]: x = False

        if x:
            print('x was True!')
        else:
            print('I will be printed in any case where x is not true')
```

```
I will be printed in any case where x is not true
```

1.1.1 Multiple Branches

Let's get a fuller picture of how far if, elif, and else can take us!

We write this out in a nested structure. Take note of how the if, elif, and else line up in the code. This can help you see what if is related to what elif or else statements.

We'll reintroduce a comparison syntax for Python.

```
In [3]: loc = 'Bank'

        if loc == 'Auto Shop':
            print('Welcome to the Auto Shop!')
        elif loc == 'Bank':
            print('Welcome to the bank!')
        else:
            print('Where are you?')
```

```
Welcome to the bank!
```

Note how the nested if statements are each checked until a True boolean causes the nested code below it to run. You should also note that you can put in as many elif statements as you want before you close off with an else.

Let's create two more simple examples for the if, elif, and else statements:

```
In [4]: person = 'Sammy'

        if person == 'Sammy':
            print('Welcome Sammy!')
        else:
            print("Welcome, what's your name?")
```

```
Welcome Sammy!
```

```
In [5]: person = 'George'

        if person == 'Sammy':
            print('Welcome Sammy!')
        elif person =='George':
            print('Welcome George!')
        else:
            print("Welcome, what's your name?")
```

```
Welcome George!
```

1.2 Indentation

It is important to keep a good understanding of how indentation works in Python to maintain the structure and order of your code. We will touch on this topic again when we start building out functions!

03-for Loops

April 9, 2019

1 for Loops

A for loop acts as an iterator in Python; it goes through items that are in a *sequence* or any other iterable item. Objects that we've learned about that we can iterate over include strings, lists, tuples, and even built-in iterables for dictionaries, such as keys or values.

We've already seen the for statement a little bit in past lectures but now let's formalize our understanding.

Here's the general format for a for loop in Python:

```
for item in object:
    statements to do stuff
```

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code. This item name can then be referenced inside your loop, for example if you wanted to use if statements to perform checks.

Let's go ahead and work through several example of for loops using a variety of data object types. We'll start simple and build more complexity later on.

1.1 Example 1

Iterating through a list

```
In [1]: # We'll learn how to automate this sort of list in the next lecture
        list1 = [1,2,3,4,5,6,7,8,9,10]

In [2]: for num in list1:
            print(num)
```

```
1
2
3
4
5
6
7
8
9
```

```
10
```

Great! Hopefully this makes sense. Now let's add an if statement to check for even numbers. We'll first introduce a new concept here–the modulo. ### Modulo The modulo allows us to get the remainder in a division and uses the % symbol. For example:

```
In [3]: 17 % 5

Out[3]: 2
```

This makes sense since 17 divided by 5 is 3 remainder 2. Let's see a few more quick examples:

```
In [4]: # 3 Remainder 1
        10 % 3

Out[4]: 1

In [5]: # 2 Remainder 4
        18 % 7

Out[5]: 4

In [6]: # 2 no remainder
        4 % 2

Out[6]: 0
```

Notice that if a number is fully divisible with no remainder, the result of the modulo call is 0. We can use this to test for even numbers, since if a number modulo 2 is equal to 0, that means it is an even number!

Back to the for loops!

1.2 Example 2

Let's print only the even numbers from that list!

```
In [7]: for num in list1:
            if num % 2 == 0:
                print(num)

2
4
6
8
10
```

We could have also put an else statement in there:

```
In [8]: for num in list1:
            if num % 2 == 0:
                print(num)
            else:
                print('Odd number')
```

```
Odd number
2
Odd number
4
Odd number
6
Odd number
8
Odd number
10
```

1.3 Example 3

Another common idea during a for loop is keeping some sort of running tally during multiple loops. For example, let's create a for loop that sums up the list:

```
In [9]: # Start sum at zero
        list_sum = 0

        for num in list1:
            list_sum = list_sum + num

        print(list_sum)
```

```
55
```

Great! Read over the above cell and make sure you understand fully what is going on. Also we could have implemented a += to perform the addition towards the sum. For example:

```
In [10]: # Start sum at zero
         list_sum = 0

         for num in list1:
             list_sum += num

         print(list_sum)
```

```
55
```

1.4 Example 4

We've used for loops with lists, how about with strings? Remember strings are a sequence so when we iterate through them we will be accessing each item in that string.

```
In [11]: for letter in 'This is a string.':
             print(letter)
```

```
T
h
i
s

i
s

a

s
t
r
i
n
g
.
```

1.5 Example 5

Let's now look at how a for loop can be used with a tuple:

```
In [12]: tup = (1,2,3,4,5)

         for t in tup:
             print(t)
```

```
1
2
3
4
5
```

1.6 Example 6

Tuples have a special quality when it comes to for loops. If you are iterating through a sequence that contains tuples, the item can actually be the tuple itself, this is an example of *tuple unpacking*. During the for loop we will be unpacking the tuple inside of a sequence and we can access the individual items inside that tuple!

```
In [13]: list2 = [(2,4),(6,8),(10,12)]

In [14]: for tup in list2:
             print(tup)

(2, 4)
(6, 8)
(10, 12)


In [15]: # Now with unpacking!
         for (t1,t2) in list2:
             print(t1)

2
6
10
```

Cool! With tuples in a sequence we can access the items inside of them through unpacking! The reason this is important is because many objects will deliver their iterables through tuples. Let's start exploring iterating through Dictionaries to explore this further!

1.7 Example 7

```
In [16]: d = {'k1':1,'k2':2,'k3':3}

In [17]: for item in d:
             print(item)

k1
k2
k3
```

Notice how this produces only the keys. So how can we get the values? Or both the keys and the values?

We're going to introduce three new Dictionary methods: **.keys()**, **.values()** and **.items()**

In Python each of these methods return a *dictionary view object*. It supports operations like membership test and iteration, but its contents are not independent of the original dictionary – it is only a view. Let's see it in action:

```
In [18]: # Create a dictionary view object
         d.items()

Out[18]: dict_items([('k1', 1), ('k2', 2), ('k3', 3)])
```

Since the .items() method supports iteration, we can perform *dictionary unpacking* to separate keys and values just as we did in the previous examples.

```
In [19]: # Dictionary unpacking
         for k,v in d.items():
             print(k)
             print(v)
```

```
k1
1
k2
2
k3
3
```

If you want to obtain a true list of keys, values, or key/value tuples, you can *cast* the view as a list:

```
In [20]: list(d.keys())
```

```
Out[20]: ['k1', 'k2', 'k3']
```

Remember that dictionaries are unordered, and that keys and values come back in arbitrary order. You can obtain a sorted list using sorted():

```
In [21]: sorted(d.values())
```

```
Out[21]: [1, 2, 3]
```

1.8 Conclusion

We've learned how to use for loops to iterate through tuples, lists, strings, and dictionaries. It will be an important tool for us, so make sure you know it well and understood the above examples.

More resources

04-while Loops

April 9, 2019

1 while Loops

The while statement in Python is one of most general ways to perform iteration. A while statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

```
while test:
    code statements
else:
    final code statements
```

Let's look at a few simple while loops in action.

```
In [1]: x = 0

        while x < 10:
            print('x is currently: ',x)
            print(' x is still less than 10, adding 1 to x')
            x+=1
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
x is currently:  1
 x is still less than 10, adding 1 to x
x is currently:  2
 x is still less than 10, adding 1 to x
x is currently:  3
 x is still less than 10, adding 1 to x
x is currently:  4
 x is still less than 10, adding 1 to x
x is currently:  5
 x is still less than 10, adding 1 to x
x is currently:  6
 x is still less than 10, adding 1 to x
x is currently:  7
```

```
 x is still less than 10, adding 1 to x
x is currently:  8
 x is still less than 10, adding 1 to x
x is currently:  9
 x is still less than 10, adding 1 to x
```

Notice how many times the print statements occurred and how the while loop kept going until the True condition was met, which occurred once x==10. It's important to note that once this occurred the code stopped. Let's see how we could add an else statement:

```
In [2]: x = 0

        while x < 10:
            print('x is currently: ',x)
            print(' x is still less than 10, adding 1 to x')
            x+=1

        else:
            print('All Done!')
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
x is currently:  1
 x is still less than 10, adding 1 to x
x is currently:  2
 x is still less than 10, adding 1 to x
x is currently:  3
 x is still less than 10, adding 1 to x
x is currently:  4
 x is still less than 10, adding 1 to x
x is currently:  5
 x is still less than 10, adding 1 to x
x is currently:  6
 x is still less than 10, adding 1 to x
x is currently:  7
 x is still less than 10, adding 1 to x
x is currently:  8
 x is still less than 10, adding 1 to x
x is currently:  9
 x is still less than 10, adding 1 to x
All Done!
```

2 break, continue, pass

We can use break, continue, and pass statements in our loops to add additional functionality for various cases. The three statements are defined by:

```
break: Breaks out of the current closest enclosing loop.
continue: Goes to the top of the closest enclosing loop.
pass: Does nothing at all.
```

Thinking about break and continue statements, the general format of the while loop looks like this:

```
while test:
    code statement
    if test:
        break
    if test:
        continue
else:
```

break and continue statements can appear anywhere inside the loop's body, but we will usually put them further nested in conjunction with an if statement to perform an action based on some condition.

Let's go ahead and look at some examples!

```
In [3]: x = 0

        while x < 10:
            print('x is currently: ',x)
            print(' x is still less than 10, adding 1 to x')
            x+=1
            if x==3:
                print('x==3')
            else:
                print('continuing...')
                continue
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
continuing...
x is currently:  1
 x is still less than 10, adding 1 to x
continuing...
x is currently:  2
 x is still less than 10, adding 1 to x
x==3
x is currently:  3
 x is still less than 10, adding 1 to x
continuing...
x is currently:  4
 x is still less than 10, adding 1 to x
continuing...
x is currently:  5
 x is still less than 10, adding 1 to x
```

```
continuing...
x is currently:  6
 x is still less than 10, adding 1 to x
continuing...
x is currently:  7
 x is still less than 10, adding 1 to x
continuing...
x is currently:  8
 x is still less than 10, adding 1 to x
continuing...
x is currently:  9
 x is still less than 10, adding 1 to x
continuing...
```

Note how we have a printed statement when x==3, and a continue being printed out as we continue through the outer while loop. Let's put in a break once x ==3 and see if the result makes sense:

```
In [4]: x = 0

        while x < 10:
            print('x is currently: ',x)
            print(' x is still less than 10, adding 1 to x')
            x+=1
            if x==3:
                print('Breaking because x==3')
                break
            else:
                print('continuing...')
                continue

x is currently:  0
 x is still less than 10, adding 1 to x
continuing...
x is currently:  1
 x is still less than 10, adding 1 to x
continuing...
x is currently:  2
 x is still less than 10, adding 1 to x
Breaking because x==3
```

Note how the other else statement wasn't reached and continuing was never printed!

After these brief but simple examples, you should feel comfortable using while statements in your code.

A word of caution however! It is possible to create an infinitely running loop with while statements. For example:

```
In [ ]: # DO NOT RUN THIS CODE!!!!
        while True:
            print("I'm stuck in an infinite loop!")
```

A quick note: If you *did* run the above cell, click on the Kernel menu above to restart the kernel!

05-Useful-Operators

April 9, 2019

1 Useful Operators

There are a few built-in functions and "operators" in Python that don't fit well into any category, so we will go over them in this lecture, let's begin!

1.1 range

The range function allows you to quickly *generate* a list of integers, this comes in handy a lot, so take note of how to use it! There are 3 parameters you can pass, a start, a stop, and a step size. Let's see some examples:

```
In [1]: range(0,11)
```

```
Out[1]: range(0, 11)
```

Note that this is a **generator** function, so to actually get a list out of it, we need to cast it to a list with **list()**. What is a generator? Its a special type of function that will generate information and not need to save it to memory. We haven't talked about functions or generators yet, so just keep this in your notes for now, we will discuss this in much more detail in later on in your training!

```
In [3]: # Notice how 11 is not included, up to but not including 11, just like slice notation!
        list(range(0,11))
```

```
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
In [4]: list(range(0,12))
```

```
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

```
In [6]: # Third parameter is step size!
        # step size just means how big of a jump/leap/step you
        # take from the starting number to get to the next number.

        list(range(0,11,2))
```

```
Out[6]: [0, 2, 4, 6, 8, 10]
```

```
In [7]: list(range(0,101,10))
```

```
Out[7]: [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

1.2 enumerate

enumerate is a very useful function to use with for loops. Let's imagine the following situation:

```
In [8]: index_count = 0

        for letter in 'abcde':
            print("At index {} the letter is {}".format(index_count,letter))
            index_count += 1
```

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
At index 3 the letter is d
At index 4 the letter is e
```

Keeping track of how many loops you've gone through is so common, that enumerate was created so you don't need to worry about creating and updating this index_count or loop_count variable

```
In [10]: # Notice the tuple unpacking!

         for i,letter in enumerate('abcde'):
             print("At index {} the letter is {}".format(i,letter))
```

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
At index 3 the letter is d
At index 4 the letter is e
```

1.3 zip

Notice the format enumerate actually returns, let's take a look by transforming it to a list()

```
In [12]: list(enumerate('abcde'))
```

```
Out[12]: [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
```

It was a list of tuples, meaning we could use tuple unpacking during our for loop. This data structure is actually very common in Python , especially when working with outside libraries. You can use the **zip()** function to quickly create a list of tuples by "zipping" up together two lists.

```
In [13]: mylist1 = [1,2,3,4,5]
         mylist2 = ['a','b','c','d','e']

In [15]: # This one is also a generator! We will explain this later, but for now let's transfo
         zip(mylist1,mylist2)
```

```
Out[15]: <zip at 0x1d205086f08>

In [17]: list(zip(mylist1,mylist2))

Out[17]: [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

To use the generator, we could just use a for loop

```
In [20]: for item1, item2 in zip(mylist1,mylist2):
             print('For this tuple, first item was {} and second item was {}'.format(item1,ite

For this tuple, first item was 1 and second item was a
For this tuple, first item was 2 and second item was b
For this tuple, first item was 3 and second item was c
For this tuple, first item was 4 and second item was d
For this tuple, first item was 5 and second item was e
```

1.4 in operator

We've already seen the **in** keyword durng the for loop, but we can also use it to quickly check if an object is in a list

```
In [21]: 'x' in ['x','y','z']

Out[21]: True

In [22]: 'x' in [1,2,3]

Out[22]: False
```

1.5 min and max

Quickly check the minimum or maximum of a list with these functions.

```
In [26]: mylist = [10,20,30,40,100]

In [27]: min(mylist)

Out[27]: 10

In [44]: max(mylist)

Out[44]: 100
```

1.6 random

Python comes with a built in random library. There are a lot of functions included in this random library, so we will only show you two useful functions for now.

```
In [29]: from random import shuffle

In [35]: # This shuffles the list "in-place" meaning it won't return
         # anything, instead it will effect the list passed
         shuffle(mylist)

In [36]: mylist

Out[36]: [40, 10, 100, 30, 20]

In [39]: from random import randint

In [41]: # Return random integer in range [a, b], including both end points.
         randint(0,100)

Out[41]: 25

In [42]: # Return random integer in range [a, b], including both end points.
         randint(0,100)

Out[42]: 91
```

1.7 input

```
In [43]: input('Enter Something into this box: ')

Enter Something into this box: great job!


Out[43]: 'great job!'
```

06-List Comprehensions

April 9, 2019

1 List Comprehensions

In addition to sequence operations and list methods, Python includes a more advanced operation called a list comprehension.

List comprehensions allow us to build out lists using a different notation. You can think of it as essentially a one line for loop built inside of brackets. For a simple example: ## Example 1

```
In [1]: # Grab every letter in string
        lst = [x for x in 'word']

In [2]: # Check
        lst

Out[2]: ['w', 'o', 'r', 'd']
```

This is the basic idea of a list comprehension. If you're familiar with mathematical notation this format should feel familiar for example: x^2 : x in { 0,1,2...10 }

Let's see a few more examples of list comprehensions in Python: ## Example 2

```
In [3]: # Square numbers in range and turn into list
        lst = [x**2 for x in range(0,11)]

In [4]: lst

Out[4]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

1.1 Example 3

Let's see how to add in if statements:

```
In [5]: # Check for even numbers in a range
        lst = [x for x in range(11) if x % 2 == 0]

In [6]: lst

Out[6]: [0, 2, 4, 6, 8, 10]
```

1.2 Example 4

Can also do more complicated arithmetic:

```
In [7]: # Convert Celsius to Fahrenheit
        celsius = [0,10,20.1,34.5]

        fahrenheit = [((9/5)*temp + 32) for temp in celsius ]

        fahrenheit

Out[7]: [32.0, 50.0, 68.18, 94.1]
```

1.3 Example 5

We can also perform nested list comprehensions, for example:

```
In [8]: lst = [ x**2 for x in [x**2 for x in range(11)]]
        lst

Out[8]: [0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]
```

Later on in the course we will learn about generator comprehensions. After this lecture you should feel comfortable reading and writing basic list comprehensions.

07-Statements Assessment Test

April 9, 2019

1 Statements Assessment Test

Let's test your knowledge!

Use for, .split(), and if to create a Statement that will print out words that start with 's':

```
In [ ]: st = 'Print only the words that start with s in this sentence'
```

```
In [ ]: #Code here
```

Use range() to print all the even numbers from 0 to 10.

```
In [ ]: #Code Here
```

Use a List Comprehension to create a list of all numbers between 1 and 50 that are divisible by 3.

```
In [ ]: #Code in this cell
        []
```

Go through the string below and if the length of a word is even print "even!"

```
In [ ]: st = 'Print every word in this sentence that has an even number of letters'
```

```
In [ ]: #Code in this cell
```

Write a program that prints the integers from 1 to 100. But for multiples of three print "Fizz" instead of the number, and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

```
In [ ]: #Code in this cell
```

Use List Comprehension to create a list of the first letters of every word in the string below:

```
In [ ]: st = 'Create a list of the first letters of every word in this string'
```

```
In [ ]: #Code in this cell
```

1.0.1 Great Job!

08-Statements Assessment Test - Solutions

April 9, 2019

1 Statements Assessment Solutions

Use for, .split(), and if to create a Statement that will print out words that start with 's':

```
In [1]: st = 'Print only the words that start with s in this sentence'
```

```
In [2]: for word in st.split():
            if word[0] == 's':
                print(word)
```

```
start
s
sentence
```

Use range() to print all the even numbers from 0 to 10.

```
In [3]: list(range(0,11,2))
```

```
Out[3]: [0, 2, 4, 6, 8, 10]
```

Use List comprehension to create a list of all numbers between 1 and 50 that are divisible by 3.

```
In [4]: [x for x in range(1,51) if x%3 == 0]
```

```
Out[4]: [3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]
```

Go through the string below and if the length of a word is even print "even!"

```
In [5]: st = 'Print every word in this sentence that has an even number of letters'
```

```
In [6]: for word in st.split():
            if len(word)%2 == 0:
                print(word+" <-- has an even length!")
```

```
word <-- has an even length!
in <-- has an even length!
this <-- has an even length!
sentence <-- has an even length!
that <-- has an even length!
an <-- has an even length!
even <-- has an even length!
number <-- has an even length!
of <-- has an even length!
```

Write a program that prints the integers from 1 to 100. But for multiples of three print "Fizz" instead of the number, and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

```
In [ ]: for num in range(1,101):
            if num % 3 == 0 and num % 5 == 0:
                print("FizzBuzz")
            elif num % 3 == 0:
                print("Fizz")
            elif num % 5 == 0:
                print("Buzz")
            else:
                print(num)
```

Use a List Comprehension to create a list of the first letters of every word in the string below:

```
In [7]: st = 'Create a list of the first letters of every word in this string'
```

```
In [8]: [word[0] for word in st.split()]
```

```
Out[8]: ['C', 'a', 'l', 'o', 't', 'f', 'l', 'o', 'e', 'w', 'i', 't', 's']
```

1.0.1 Great Job!

09-Guessing Game Challenge

April 9, 2019

1 Guessing Game Challenge

Let's use `while` loops to create a guessing game.

The Challenge:

Write a program that picks a random integer from 1 to 100, and has players guess the number. The rules are:

1. If a player's guess is less than 1 or greater than 100, say "OUT OF BOUNDS"
2. On a player's first turn, if their guess is

- within 10 of the number, return "WARM!"
- further than 10 away from the number, return "COLD!"

3. On all subsequent turns, if a guess is

- closer to the number than the previous guess return "WARMER!"
- farther from the number than the previous guess, return "COLDER!"

4. When the player's guess equals the number, tell them they've guessed correctly *and* how many guesses it took!

You can try this from scratch, or follow the steps outlined below. A separate Solution notebook has been provided. Good luck!

First, pick a random integer from 1 to 100 using the random module and assign it to a variable
Note: `random.randint(a,b)` returns a random integer in range `[a, b]`, including both end points.

```
In [ ]:
```

Next, print an introduction to the game and explain the rules

```
In [ ]:
```

Create a list to store guesses Hint: zero is a good placeholder value. It's useful because it evaluates to "False"

```
In [ ]:
```

Write a `while` loop that asks for a valid guess. Test it a few times to make sure it works.

```
In [ ]: while True:

            pass
```

Write a `while` loop that compares the player's guess to our number. If the player guesses correctly, break from the loop. Otherwise, tell the player if they're warmer or colder, and continue asking for guesses. Some hints: * it may help to sketch out all possible combinations on paper first! * you can use the `abs()` function to find the positive difference between two numbers * if you append all new guesses to the list, then the previous guess is given as `guesses[-2]`

```
In [ ]: while True:

            # we can copy the code from above to take an input

            pass
```

That's it! You've just programmed your first game!

In the next section we'll learn how to turn some of these repetitive actions into *functions* that can be called whenever we need them.

1.0.1 Good Job!

10-Guessing Game Challenge - Solution

April 9, 2019

1 Guessing Game Challenge - Solution

Let's use `while` loops to create a guessing game.

The Challenge:

Write a program that picks a random integer from 1 to 100, and has players guess the number. The rules are:

1. If a player's guess is less than 1 or greater than 100, say "OUT OF BOUNDS"
2. On a player's first turn, if their guess is

- within 10 of the number, return "WARM!"
- further than 10 away from the number, return "COLD!"

3. On all subsequent turns, if a guess is

- closer to the number than the previous guess return "WARMER!"
- farther from the number than the previous guess, return "COLDER!"

4. When the player's guess equals the number, tell them they've guessed correctly *and* how many guesses it took!

First, pick a random integer from 1 to 100 using the random module and assign it to a variable
Note: `random.randint(a,b)` returns a random integer in range `[a, b]`, including both end points.

```
In [1]: import random

        num = random.randint(1,100)
```

Next, print an introduction to the game and explain the rules

```
In [2]: print("WELCOME TO GUESS ME!")
        print("I'm thinking of a number between 1 and 100")
        print("If your guess is more than 10 away from my number, I'll tell you you're COLD")
        print("If your guess is within 10 of my number, I'll tell you you're WARM")
        print("If your guess is farther than your most recent guess, I'll say you're getting C
        print("If your guess is closer than your most recent guess, I'll say you're getting WA
        print("LET'S PLAY!")
```

```
WELCOME TO GUESS ME!
I'm thinking of a number between 1 and 100
If your guess is more than 10 away from my number, I'll tell you you're COLD
If your guess is within 10 of my number, I'll tell you you're WARM
If your guess is farther than your most recent guess, I'll say you're getting COLDER
If your guess is closer than your most recent guess, I'll say you're getting WARMER
LET'S PLAY!
```

Create a list to store guesses Hint: zero is a good placeholder value. It's useful because it evaluates to "False"

```
In [3]: guesses = [0]
```

Write a `while` loop that asks for a valid guess. Test it a few times to make sure it works.

```
In [4]: while True:

            guess = int(input("I'm thinking of a number between 1 and 100.\n  What is your gues

            if guess < 1 or guess > 100:
                print('OUT OF BOUNDS! Please try again: ')
                continue

            break
```

```
I'm thinking of a number between 1 and 100.
  What is your guess? 500
OUT OF BOUNDS! Please try again:
I'm thinking of a number between 1 and 100.
  What is your guess? 50
```

Write a `while` loop that compares the player's guess to our number. If the player guesses correctly, break from the loop. Otherwise, tell the player if they're warmer or colder, and continue asking for guesses. Some hints: * it may help to sketch out all possible combinations on paper first! * you can use the `abs()` function to find the positive difference between two numbers * if you append all new guesses to the list, then the previous guess is given as `guesses[-2]`

```
In [5]: while True:

            # we can copy the code from above to take an input
            guess = int(input("I'm thinking of a number between 1 and 100.\n  What is your gues

            if guess < 1 or guess > 100:
                print('OUT OF BOUNDS! Please try again: ')
                continue
```

```
            # here we compare the player's guess to our number
            if guess == num:
                print(f'CONGRATULATIONS, YOU GUESSED IT IN ONLY {len(guesses)} GUESSES!!')
                break

            # if guess is incorrect, add guess to the list
            guesses.append(guess)

            # when testing the first guess, guesses[-2]==0, which evaluates to False
            # and brings us down to the second section

            if guesses[-2]:
                if abs(num-guess) < abs(num-guesses[-2]):
                    print('WARMER!')
                else:
                    print('COLDER!')

            else:
                if abs(num-guess) <= 10:
                    print('WARM!')
                else:
                    print('COLD!')
```

```
I'm thinking of a number between 1 and 100.
  What is your guess? 50
COLD!
I'm thinking of a number between 1 and 100.
  What is your guess? 75
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 85
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 92
COLDER!
I'm thinking of a number between 1 and 100.
  What is your guess? 80
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 78
COLDER!
I'm thinking of a number between 1 and 100.
  What is your guess? 82
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 83
COLDER!
I'm thinking of a number between 1 and 100.
```

```
  What is your guess? 81
CONGRATULATIONS, YOU GUESSED IT IN ONLY 9 GUESSES!!
```

That's it! You've just programmed your first game!

In the next section we'll learn how to turn some of these repetitive actions into *functions* that can be called whenever we need them.

1.0.1 Good Job!

01-Methods

April 9, 2019

1 Methods

We've already seen a few example of methods when learning about Object and Data Structure Types in Python. Methods are essentially functions built into objects. Later on in the course we will learn about how to create our own objects and methods using Object Oriented Programming (OOP) and classes.

Methods perform specific actions on an object and can also take arguments, just like a function. This lecture will serve as just a brief introduction to methods and get you thinking about overall design methods that we will touch back upon when we reach OOP in the course.

Methods are in the form:

```
object.method(arg1,arg2,etc...)
```

You'll later see that we can think of methods as having an argument 'self' referring to the object itself. You can't see this argument but we will be using it later on in the course during the OOP lectures.

Let's take a quick look at what an example of the various methods a list has:

```
In [1]: # Create a simple list
        lst = [1,2,3,4,5]
```

Fortunately, with iPython and the Jupyter Notebook we can quickly see all the possible methods using the tab key. The methods for a list are:

- append
- count
- extend
- insert
- pop
- remove
- reverse
- sort

Let's try out a few of them:

append() allows us to add elements to the end of a list:

```
In [2]: lst.append(6)
```

```
In [3]: lst
```

```
Out[3]: [1, 2, 3, 4, 5, 6]
```

Great! Now how about count()? The count() method will count the number of occurrences of an element in a list.

```
In [4]: # Check how many times 2 shows up in the list
        lst.count(2)
```

```
Out[4]: 1
```

You can always use Shift+Tab in the Jupyter Notebook to get more help about the method. In general Python you can use the help() function:

```
In [5]: help(lst.count)
```

```
Help on built-in function count:

count(...) method of builtins.list instance
    L.count(value) -> integer -- return number of occurrences of value
```

Feel free to play around with the rest of the methods for a list. Later on in this section your quiz will involve using help and Google searching for methods of different types of objects!

Great! By this lecture you should feel comfortable calling methods of objects in Python!

02-Functions

April 9, 2019

1 Functions

1.1 Introduction to Functions

This lecture will consist of explaining what a function is in Python and how to create one. Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

So what is a function?

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Functions will be one of most basic levels of reusing code in Python, and it will also allow us to start thinking of program design (we will dive much deeper into the ideas of design when we learn about Object Oriented Programming).

1.2 def Statements

Let's see how to build out a function's syntax in Python. It has the following form:

```
In [1]: def name_of_function(arg1,arg2):
            '''
            This is where the function's Document String (docstring) goes
            '''
            # Do stuff here
            # Return desired result
```

We begin with def then a space followed by the name of the function. Try to keep names relevant, for example len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a built-in function in Python (such as len).

Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.

Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code. Lots of other programing languages do not do this, so keep that in mind.

Next you'll see the docstring, this is where you write a basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

After all this you begin writing the code you wish to execute.

The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.

1.2.1 Example 1: A simple print 'hello' function

```
In [2]: def say_hello():
            print('hello')
```

Call the function:

```
In [3]: say_hello()
```

```
hello
```

1.2.2 Example 2: A simple greeting function

Let's write a function that greets people with their name.

```
In [4]: def greeting(name):
            print('Hello %s' %(name))
```

```
In [5]: greeting('Jose')
```

```
Hello Jose
```

1.3 Using return

Let's see some example that use a return statement. return allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

1.3.1 Example 3: Addition function

```
In [6]: def add_num(num1,num2):
            return num1+num2
```

```
In [7]: add_num(4,5)
```

```
Out[7]: 9
```

```
In [8]: # Can also save as variable due to return
        result = add_num(4,5)
```

```
In [9]: print(result)
```

```
9
```

What happens if we input two strings?

```
In [10]: add_num('one','two')

Out[10]: 'onetwo'
```

Note that because we don't declare variable types in Python, this function could be used to add numbers or sequences together! We'll later learn about adding in checks to make sure a user puts in the correct arguments into a function.

Let's also start using break, continue, and pass statements in our code. We introduced these during the while lecture.

Finally let's go over a full example of creating a function to check if a number is prime (a common interview exercise).

We know a number is prime if that number is only evenly divisible by 1 and itself. Let's write our first version of the function to check all the numbers from 1 to N and perform modulo checks.

```
In [11]: def is_prime(num):
             '''
             Naive method of checking for primes.
             '''
             for n in range(2,num):
                 if num % n == 0:
                     print(num,'is not prime')
                     break
             else: # If never mod zero, then prime
                 print(num,'is prime!')

In [12]: is_prime(16)

16 is not prime


In [13]: is_prime(17)

17 is prime!
```

Note how the else lines up under for and not if. This is because we want the for loop to exhaust all possibilities in the range before printing our number is prime.

Also note how we break the code after the first print statement. As soon as we determine that a number is not prime we break out of the for loop.

We can actually improve this function by only checking to the square root of the target number, and by disregarding all even numbers after checking for 2. We'll also switch to returning a boolean value to get an example of using return statements:

```
In [14]: import math

         def is_prime2(num):
             '''
             Better method of checking for primes.
             '''
             if num % 2 == 0 and num > 2:
                 return False
             for i in range(3, int(math.sqrt(num)) + 1, 2):
                 if num % i == 0:
                     return False
             return True

In [15]: is_prime2(18)

Out[15]: False
```

Why don't we have any break statements? It should be noted that as soon as a function *returns* something, it shuts down. A function can deliver multiple print statements, but it will only obey one return.

Great! You should now have a basic understanding of creating your own functions to save yourself from repeatedly writing code!

03-Function Practice Exercises

April 9, 2019

1 Function Practice Exercises

Problems are arranged in increasing difficulty: * Warmup - these can be solved using basic comparisons and methods * Level 1 - these may involve if/then conditional statements and simple methods * Level 2 - these may require iterating over sequences, usually with some kind of loop * Challenging - these will take some creativity to solve

1.1 WARMUP SECTION:

LESSER OF TWO EVENS: Write a function that returns the lesser of two given numbers *if* both numbers are even, but returns the greater if one or both numbers are odd

```
lesser_of_two_evens(2,4) --> 2
lesser_of_two_evens(2,5) --> 5
```

```
In [ ]: def lesser_of_two_evens(a,b):
            pass
```

```
In [ ]: # Check
        lesser_of_two_evens(2,4)
```

```
In [ ]: # Check
        lesser_of_two_evens(2,5)
```

ANIMAL CRACKERS: Write a function takes a two-word string and returns True if both words begin with same letter

```
animal_crackers('Levelheaded Llama') --> True
animal_crackers('Crazy Kangaroo') --> False
```

```
In [ ]: def animal_crackers(text):
            pass
```

```
In [ ]: # Check
        animal_crackers('Levelheaded Llama')
```

```
In [ ]: # Check
        animal_crackers('Crazy Kangaroo')
```

MAKES TWENTY: Given two integers, return True if the sum of the integers is 20 *or* if one of the integers is 20. If not, return False

```
makes_twenty(20,10) --> True
makes_twenty(12,8) --> True
makes_twenty(2,3) --> False
```

```
In [ ]: def makes_twenty(n1,n2):
            pass
```

```
In [ ]: # Check
        makes_twenty(20,10)
```

```
In [ ]: # Check
        makes_twenty(2,3)
```

2 LEVEL 1 PROBLEMS

OLD MACDONALD: Write a function that capitalizes the first and fourth letters of a name

```
old_macdonald('macdonald') --> MacDonald
```

Note: `'macdonald'.capitalize()` returns `'Macdonald'`

```
In [ ]: def old_macdonald(name):
            pass
```

```
In [ ]: # Check
        old_macdonald('macdonald')
```

MASTER YODA: Given a sentence, return a sentence with the words reversed

```
master_yoda('I am home') --> 'home am I'
master_yoda('We are ready') --> 'ready are We'
```

Note: The .join() method may be useful here. The .join() method allows you to join together strings in a list with some connector string. For example, some uses of the .join() method:

```
>>> "--".join(['a','b','c'])
>>> 'a--b--c'
```

This means if you had a list of words you wanted to turn back into a sentence, you could just join them with a single space string:

```
>>> " ".join(['Hello','world'])
>>> "Hello world"
```

```
In [ ]: def master_yoda(text):
            pass
```

```
In [ ]: # Check
        master_yoda('I am home')
```

```
In [ ]: # Check
        master_yoda('We are ready')
```

ALMOST THERE: Given an integer n, return True if n is within 10 of either 100 or 200

```
almost_there(90) --> True
almost_there(104) --> True
almost_there(150) --> False
almost_there(209) --> True
```

NOTE: `abs(num)` returns the absolute value of a number

```
In [ ]: def almost_there(n):
            pass
```

```
In [ ]: # Check
        almost_there(104)
```

```
In [ ]: # Check
        almost_there(150)
```

```
In [ ]: # Check
        almost_there(209)
```

3 LEVEL 2 PROBLEMS

FIND 33: Given a list of ints, return True if the array contains a 3 next to a 3 somewhere.

```
has_33([1, 3, 3])  True
has_33([1, 3, 1, 3])  False
has_33([3, 1, 3])  False
```

```
In [ ]: def has_33(nums):
            pass
```

```
In [ ]: # Check
        has_33([1, 3, 3])
```

```
In [ ]: # Check
        has_33([1, 3, 1, 3])
```

```
In [ ]: # Check
        has_33([3, 1, 3])
```

PAPER DOLL: Given a string, return a string where for every character in the original there are three characters

```
paper_doll('Hello') --> 'HHHeeellllllooo'
paper_doll('Mississippi') --> 'MMMiiissssssiiippppppiii'
```

```
In [ ]: def paper_doll(text):
            pass
```

```
In [ ]: # Check
        paper_doll('Hello')
```

```
In [ ]: # Check
        paper_doll('Mississippi')
```

BLACKJACK: Given three integers between 1 and 11, if their sum is less than or equal to 21, return their sum. If their sum exceeds 21 *and* there's an eleven, reduce the total sum by 10. Finally, if the sum (even after adjustment) exceeds 21, return 'BUST'

```
blackjack(5,6,7) --> 18
blackjack(9,9,9) --> 'BUST'
blackjack(9,9,11) --> 19
```

```
In [ ]: def blackjack(a,b,c):
            pass
```

```
In [ ]: # Check
        blackjack(5,6,7)
```

```
In [ ]: # Check
        blackjack(9,9,9)
```

```
In [ ]: # Check
        blackjack(9,9,11)
```

SUMMER OF '69: Return the sum of the numbers in the array, except ignore sections of numbers starting with a 6 and extending to the next 9 (every 6 will be followed by at least one 9). Return 0 for no numbers.

```
summer_69([1, 3, 5]) --> 9
summer_69([4, 5, 6, 7, 8, 9]) --> 9
summer_69([2, 1, 6, 9, 11]) --> 14
```

```
In [ ]: def summer_69(arr):
            pass
```

```
In [ ]: # Check
        summer_69([1, 3, 5])
```

```
In [ ]: # Check
        summer_69([4, 5, 6, 7, 8, 9])
```

```
In [ ]: # Check
        summer_69([2, 1, 6, 9, 11])
```

4 CHALLENGING PROBLEMS

SPY GAME: Write a function that takes in a list of integers and returns True if it contains 007 in order

```
spy_game([1,2,4,0,0,7,5]) --> True
spy_game([1,0,2,4,0,5,7]) --> True
spy_game([1,7,2,0,4,5,0]) --> False
```

```
In [ ]: def spy_game(nums):
            pass

In [ ]: # Check
        spy_game([1,2,4,0,0,7,5])

In [ ]: # Check
        spy_game([1,0,2,4,0,5,7])

In [ ]: # Check
        spy_game([1,7,2,0,4,5,0])
```

COUNT PRIMES: Write a function that returns the *number* of prime numbers that exist up to and including a given number

```
count_primes(100) --> 25
```

By convention, 0 and 1 are not prime.

```
In [ ]: def count_primes(num):
            pass

In [ ]: # Check
        count_primes(100)
```

4.0.1 Just for fun:

PRINT BIG: Write a function that takes in a single letter, and returns a 5x5 representation of that letter

```
print_big('a')

out:   *
      * *
     *****
     *   *
     *   *
```

HINT: Consider making a dictionary of possible patterns, and mapping the alphabet to specific 5-line combinations of patterns. For purposes of this exercise, it's ok if your dictionary stops at "E".

```
In [ ]: def print_big(letter):
            pass

In [ ]: print_big('a')
```

4.1 Great Job!

04-Function Practice Exercises - Solutions

April 9, 2019

1 Function Practice Exercises - Solutions

Problems are arranged in increasing difficulty: * Warmup - these can be solved using basic comparisons and methods * Level 1 - these may involve if/then conditional statements and simple methods * Level 2 - these may require iterating over sequences, usually with some kind of loop * Challenging - these will take some creativity to solve

1.1 WARMUP SECTION:

LESSER OF TWO EVENS: Write a function that returns the lesser of two given numbers *if* both numbers are even, but returns the greater if one or both numbers are odd

```
lesser_of_two_evens(2,4) --> 2
lesser_of_two_evens(2,5) --> 5
```

```
In [1]: def lesser_of_two_evens(a,b):
            if a%2 == 0 and b%2 == 0:
                return min(a,b)
            else:
                return max(a,b)
```

```
In [2]: # Check
        lesser_of_two_evens(2,4)
```

```
Out[2]: 2
```

```
In [3]: # Check
        lesser_of_two_evens(2,5)
```

```
Out[3]: 5
```

ANIMAL CRACKERS: Write a function takes a two-word string and returns True if both words begin with same letter

```
animal_crackers('Levelheaded Llama') --> True
animal_crackers('Crazy Kangaroo') --> False
```

```
In [4]: def animal_crackers(text):
            wordlist = text.split()
            return wordlist[0][0] == wordlist[1][0]
```

```
In [5]: # Check
        animal_crackers('Levelheaded Llama')
```

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

```
In [6]: # Check
        animal_crackers('Crazy Kangaroo')
```

```
Out[6]: False
```

MAKES TWENTY: Given two integers, return True if the sum of the integers is 20 *or* if one of the integers is 20. If not, return False

```
makes_twenty(20,10) --> True
makes_twenty(12,8) --> True
makes_twenty(2,3) --> False
```

```
In [7]: def makes_twenty(n1,n2):
            return (n1+n2)==20 or n1==20 or n2==20
```

```
In [8]: # Check
        makes_twenty(20,10)
```

```
Out[8]: True
```

```
In [9]: # Check
        makes_twenty(12,8)
```

```
Out[9]: True
```

```
In [10]: #Check
         makes_twenty(2,3)
```

```
Out[10]: False
```

2 LEVEL 1 PROBLEMS

OLD MACDONALD: Write a function that capitalizes the first and fourth letters of a name

```
old_macdonald('macdonald') --> MacDonald
```

Note: `'macdonald'.capitalize()` returns `'Macdonald'`

```
In [11]: def old_macdonald(name):
             if len(name) > 3:
                 return name[:3].capitalize() + name[3:].capitalize()
             else:
                 return 'Name is too short!'
```

```
In [12]: # Check
         old_macdonald('macdonald')
```

```
Out[12]: 'MacDonald'
```

MASTER YODA: Given a sentence, return a sentence with the words reversed

```
master_yoda('I am home') --> 'home am I'
master_yoda('We are ready') --> 'ready are We'
```

```
In [13]: def master_yoda(text):
             return ' '.join(text.split()[::-1])
```

```
In [14]: # Check
         master_yoda('I am home')
```

```
Out[14]: 'home am I'
```

```
In [15]: # Check
         master_yoda('We are ready')
```

```
Out[15]: 'ready are We'
```

ALMOST THERE: Given an integer n, return True if n is within 10 of either 100 or 200

```
almost_there(90) --> True
almost_there(104) --> True
almost_there(150) --> False
almost_there(209) --> True
```

NOTE: `abs(num)` returns the absolute value of a number

```
In [16]: def almost_there(n):
             return ((abs(100 - n) <= 10) or (abs(200 - n) <= 10))
```

```
In [17]: # Check
         almost_there(90)
```

```
Out[17]: True
```

```
In [18]: # Check
         almost_there(104)
```

```
Out[18]: True
```

```
In [19]: # Check
         almost_there(150)
```

```
Out[19]: False
```

```
In [20]: # Check
         almost_there(209)
```

```
Out[20]: True
```

3 LEVEL 2 PROBLEMS

FIND 33: Given a list of ints, return True if the array contains a 3 next to a 3 somewhere.

```
has_33([1, 3, 3])  True
has_33([1, 3, 1, 3])  False
has_33([3, 1, 3])  False
```

```
In [21]: def has_33(nums):
             for i in range(0, len(nums)-1):

                 # nicer looking alternative in commented code
                 #if nums[i] == 3 and nums[i+1] == 3:

                 if nums[i:i+2] == [3,3]:
                     return True

             return False
```

```
In [22]: # Check
         has_33([1, 3, 3])
```

```
Out[22]: True
```

```
In [23]: # Check
         has_33([1, 3, 1, 3])
```

```
Out[23]: False
```

```
In [24]: # Check
         has_33([3, 1, 3])
```

```
Out[24]: False
```

PAPER DOLL: Given a string, return a string where for every character in the original there are three characters

```
paper_doll('Hello') --> 'HHHeeellllllooo'
paper_doll('Mississippi') --> 'MMMiiissssssiiippppppiii'
```

```
In [25]: def paper_doll(text):
             result = ''
             for char in text:
                 result += char * 3
             return result
```

```
In [26]: # Check
         paper_doll('Hello')
```

```
Out[26]: 'HHHeeellllllooo'
```

```
In [27]: # Check
         paper_doll('Mississippi')
```

```
Out[27]: 'MMMiiissssssiiissssssiiippppppiii'
```

BLACKJACK: Given three integers between 1 and 11, if their sum is less than or equal to 21, return their sum. If their sum exceeds 21 *and* there's an eleven, reduce the total sum by 10. Finally, if the sum (even after adjustment) exceeds 21, return 'BUST'

```
blackjack(5,6,7) --> 18
blackjack(9,9,9) --> 'BUST'
blackjack(9,9,11) --> 19
```

```
In [28]: def blackjack(a,b,c):

             if sum((a,b,c)) <= 21:
                 return sum((a,b,c))
             elif sum((a,b,c)) <=31 and 11 in (a,b,c):
                 return sum((a,b,c)) - 10
             else:
                 return 'BUST'
```

```
In [29]: # Check
         blackjack(5,6,7)
```

```
Out[29]: 18
```

```
In [30]: # Check
         blackjack(9,9,9)
```

```
Out[30]: 'BUST'
```

```
In [31]: # Check
         blackjack(9,9,11)
```

```
Out[31]: 19
```

SUMMER OF '69: Return the sum of the numbers in the array, except ignore sections of numbers starting with a 6 and extending to the next 9 (every 6 will be followed by at least one 9). Return 0 for no numbers.

```
summer_69([1, 3, 5]) --> 9
summer_69([4, 5, 6, 7, 8, 9]) --> 9
summer_69([2, 1, 6, 9, 11]) --> 14
```

```
In [32]: def summer_69(arr):
             total = 0
             add = True
             for num in arr:
                 while add:
                     if num != 6:
                         total += num
                         break
                     else:
                         add = False
```

```
            while not add:
                if num != 9:
                    break
                else:
                    add = True
                    break
        return total
```

```
In [33]: # Check
         summer_69([1, 3, 5])
```

```
Out[33]: 9
```

```
In [34]: # Check
         summer_69([4, 5, 6, 7, 8, 9])
```

```
Out[34]: 9
```

```
In [35]: # Check
         summer_69([2, 1, 6, 9, 11])
```

```
Out[35]: 14
```

4 CHALLENGING PROBLEMS

SPY GAME: Write a function that takes in a list of integers and returns True if it contains 007 in order

```
spy_game([1,2,4,0,0,7,5]) --> True
spy_game([1,0,2,4,0,5,7]) --> True
spy_game([1,7,2,0,4,5,0]) --> False
```

```
In [36]: def spy_game(nums):

             code = [0,0,7,'x']

             for num in nums:
                 if num == code[0]:
                     code.pop(0)   # code.remove(num) also works

             return len(code) == 1
```

```
In [37]: # Check
         spy_game([1,2,4,0,0,7,5])
```

```
Out[37]: True
```

```
In [38]: # Check
         spy_game([1,0,2,4,0,5,7])
```

```
Out[38]: True
```

```
In [39]: # Check
         spy_game([1,7,2,0,4,5,0])
```

```
Out[39]: False
```

COUNT PRIMES: Write a function that returns the *number* of prime numbers that exist up to and including a given number

```
count_primes(100) --> 25
```

By convention, 0 and 1 are not prime.

```
In [40]: def count_primes(num):
             primes = [2]
             x = 3
             if num < 2:  # for the case of num = 0 or 1
                 return 0
             while x <= num:
                 for y in range(3,x,2):  # test all odd factors up to x-1
                     if x%y == 0:
                         x += 2
                         break
                 else:
                     primes.append(x)
                     x += 2
             print(primes)
             return len(primes)
```

```
In [41]: # Check
         count_primes(100)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 9
```

```
Out[41]: 25
```

BONUS: Here's a faster version that makes use of the prime numbers we're collecting as we go!

```
In [42]: def count_primes2(num):
             primes = [2]
             x = 3
             if num < 2:
                 return 0
             while x <= num:
                 for y in primes:  # use the primes list!
                     if x%y == 0:
                         x += 2
```

```
                    break
            else:
                primes.append(x)
                x += 2
        print(primes)
        return len(primes)
```

```
In [43]: count_primes2(100)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 9
```

```
Out[43]: 25
```

4.0.1 Just for fun, not a real problem :)

PRINT BIG: Write a function that takes in a single letter, and returns a 5x5 representation of that letter

```
print_big('a')
```

```
out:   *
      * *
     *****
     *   *
     *   *
```

HINT: Consider making a dictionary of possible patterns, and mapping the alphabet to specific 5-line combinations of patterns. For purposes of this exercise, it's ok if your dictionary stops at "E".

```
In [44]: def print_big(letter):
            patterns = {1:'  *  ',2:' * * ',3:'*   *',4:'*****',5:'**** ',6:'   * ',7:' *   '
            alphabet = {'A':[1,2,4,3,3],'B':[5,3,5,3,5],'C':[4,9,9,9,4],'D':[5,3,3,3,5],'E':[4
            for pattern in alphabet[letter.upper()]:
                print(patterns[pattern])
```

```
In [45]: print_big('a')
```

```
  *
 * *
*****
*   *
*   *
```

4.1 Great Job!

05-Lambda-Expressions-Map-and-Filter

April 9, 2019

1 Lambda Expressions, Map, and Filter

Now its time to quickly learn about two built in functions, filter and map. Once we learn about how these operate, we can learn about the lambda expression, which will come in handy when you begin to develop your skills further!

1.1 map function

The **map** function allows you to "map" a function to an iterable object. That is to say you can quickly call the same function to every item in an iterable, such as a list. For example:

```
In [1]: def square(num):
            return num**2

In [2]: my_nums = [1,2,3,4,5]

In [5]: map(square,my_nums)

Out[5]: <map at 0x205baec21d0>

In [7]: # To get the results, either iterate through map()
        # or just cast to a list
        list(map(square,my_nums))

Out[7]: [1, 4, 9, 16, 25]
```

The functions can also be more complex

```
In [8]: def splicer(mystring):
            if len(mystring) % 2 == 0:
                return 'even'
            else:
                return mystring[0]

In [9]: mynames = ['John','Cindy','Sarah','Kelly','Mike']

In [10]: list(map(splicer,mynames))

Out[10]: ['even', 'C', 'S', 'K', 'even']
```

1.2 filter function

The filter function returns an iterator yielding those items of iterable for which function(item) is true. Meaning you need to filter by a function that returns either True or False. Then passing that into filter (along with your iterable) and you will get back only the results that would return True when passed to the function.

```
In [12]: def check_even(num):
             return num % 2 == 0

In [13]: nums = [0,1,2,3,4,5,6,7,8,9,10]

In [15]: filter(check_even,nums)

Out[15]: <filter at 0x205baed4710>

In [16]: list(filter(check_even,nums))

Out[16]: [0, 2, 4, 6, 8, 10]
```

1.3 lambda expression

One of Pythons most useful (and for beginners, confusing) tools is the lambda expression. lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using def.

Function objects returned by running lambda expressions work exactly the same as those created and assigned by defs. There is key difference that makes lambda useful in specialized roles:

lambda's body is a single expression, not a block of statements.

- The lambda's body is similar to what we would put in a def body's return statement. We simply type the result as an expression instead of explicitly returning it. Because it is limited to an expression, a lambda is less general that a def. We can only squeeze design, to limit program nesting. lambda is designed for coding simple functions, and def handles the larger tasks.

Lets slowly break down a lambda expression by deconstructing a function:

```
In [17]: def square(num):
             result = num**2
             return result

In [18]: square(2)

Out[18]: 4
```

We could simplify it:

```
In [19]: def square(num):
             return num**2

In [20]: square(2)
```

```
Out[20]: 4
```

We could actually even write this all on one line.

```
In [21]: def square(num): return num**2

In [22]: square(2)

Out[22]: 4
```

This is the form a function that a lambda expression intends to replicate. A lambda expression can then be written as:

```
In [23]: lambda num: num ** 2

Out[23]: <function __main__.<lambda>>

In [25]: # You wouldn't usually assign a name to a lambda expression, this is just for demonst
         square = lambda num: num **2

In [26]: square(2)

Out[26]: 4
```

So why would use this? Many function calls need a function passed in, such as map and filter. Often you only need to use the function you are passing in once, so instead of formally defining it, you just use the lambda expression. Let's repeat some of the examples from above with a lambda expression

```
In [29]: list(map(lambda num: num ** 2, my_nums))

Out[29]: [1, 4, 9, 16, 25]

In [30]: list(filter(lambda n: n % 2 == 0,nums))

Out[30]: [0, 2, 4, 6, 8, 10]
```

Here are a few more examples, keep in mind the more comples a function is, the harder it is to translate into a lambda expression, meaning sometimes its just easier (and often the only way) to create the def keyword function.

** Lambda expression for grabbing the first character of a string: **

```
In [31]: lambda s: s[0]

Out[31]: <function __main__.<lambda>>
```

** Lambda expression for reversing a string: **

```
In [32]: lambda s: s[::-1]

Out[32]: <function __main__.<lambda>>
```

You can even pass in multiple arguments into a lambda expression. Again, keep in mind that not every function can be translated into a lambda expression.

```
In [34]: lambda x,y : x + y

Out[34]: <function __main__.<lambda>>
```

You will find yourself using lambda expressions often with certain non-built-in libraries, for example the pandas library for data analysis works very well with lambda expressions.

06-Nested Statements and Scope

April 9, 2019

1 Nested Statements and Scope

Now that we have gone over writing our own functions, it's important to understand how Python deals with the variable names you assign. When you create a variable name in Python the name is stored in a *name-space*. Variable names also have a *scope*, the scope determines the visibility of that variable name to other parts of your code.

Let's start with a quick thought experiment; imagine the following code:

```
In [1]: x = 25

        def printer():
            x = 50
            return x

        # print(x)
        # print(printer())
```

What do you imagine the output of printer() is? 25 or 50? What is the output of print x? 25 or 50?

```
In [2]: print(x)
```

```
25
```

```
In [3]: print(printer())
```

```
50
```

Interesting! But how does Python know which **x** you're referring to in your code? This is where the idea of scope comes in. Python has a set of rules it follows to decide what variables (such as **x** in this case) you are referencing in your code. Lets break down the rules:

This idea of scope in your code is very important to understand in order to properly assign and call variable names.

In simple terms, the idea of scope can be described by 3 general rules:

1. Name assignments will create or change local names by default.

2. Name references search (at most) four scopes, these are:

    - local
    - enclosing functions
    - global
    - built-in

3. Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes.

The statement in #2 above can be defined by the LEGB rule.

LEGB Rule:

L: Local — Names assigned in any way within a function (def or lambda), and not declared global in that function.

E: Enclosing function locals — Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) — Names preassigned in the built-in names module : open, range, SyntaxError,...

1.1 Quick examples of LEGB

1.1.1 Local

```
In [4]: # x is local here:
        f = lambda x:x**2
```

1.1.2 Enclosing function locals

This occurs when we have a function inside a function (nested functions)

```
In [5]: name = 'This is a global name'

        def greet():
            # Enclosing function
            name = 'Sammy'

            def hello():
                print('Hello '+name)

            hello()

        greet()
```

```
Hello Sammy
```

Note how Sammy was used, because the hello() function was enclosed inside of the greet function!

1.1.3 Global

Luckily in Jupyter a quick way to test for global variables is to see if another cell recognizes the variable!

```
In [6]: print(name)

This is a global name
```

1.1.4 Built-in

These are the built-in function names in Python (don't overwrite these!)

```
In [7]: len

Out[7]: <function len>
```

1.2 Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are local to the function. This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

Example:

```
In [8]: x = 50

        def func(x):
            print('x is', x)
            x = 2
            print('Changed local x to', x)

        func(x)
        print('x is still', x)

x is 50
Changed local x to 2
x is still 50
```

The first time that we print the value of the name **x** with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the function definition.

Next, we assign the value 2 to **x**. The name **x** is local to our function. So, when we change the value of **x** in the function, the **x** defined in the main block remains unaffected.

With the last print statement, we display the value of **x** as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

1.3 The global statement

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is global. We do this using the global statement. It is impossible to assign a value to a variable defined outside a function without the global statement.

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the global statement makes it amply clear that the variable is defined in an outermost block.

Example:

```
In [9]: x = 50

        def func():
            global x
            print('This function is now using the global x!')
            print('Because of global x is: ', x)
            x = 2
            print('Ran func(), changed global x to', x)

        print('Before calling func(), x is: ', x)
        func()
        print('Value of x (outside of func()) is: ', x)
```

```
Before calling func(), x is:  50
This function is now using the global x!
Because of global x is:  50
Ran func(), changed global x to 2
Value of x (outside of func()) is:  2
```

The global statement is used to declare that **x** is a global variable - hence, when we assign a value to **x** inside the function, that change is reflected when we use the value of **x** in the main block.

You can specify more than one global variable using the same global statement e.g. global x, y, z.

1.4 Conclusion

You should now have a good understanding of Scope (you may have already intuitively felt right about Scope which is great!) One last mention is that you can use the **globals()** and **locals()** functions to check what are your current local and global variables.

Another thing to keep in mind is that everything in Python is an object! I can assign variables to functions just like I can with numbers! We will go over this again in the decorator section of the course!

07-args and kwargs

April 9, 2019

1 *args and **kwargs

Work with Python long enough, and eventually you will encounter `*args` and `**kwargs`. These strange terms show up as parameters in function definitions. What do they do? Let's review a simple function:

```
In [1]: def myfunc(a,b):
            return sum((a,b))*.05

        myfunc(40,60)
```

```
Out[1]: 5.0
```

This function returns 5% of the sum of **a** and **b**. In this example, **a** and **b** are *positional* arguments; that is, 40 is assigned to **a** because it is the first argument, and 60 to **b**. Notice also that to work with multiple positional arguments in the `sum()` function we had to pass them in as a tuple.

What if we want to work with more than two numbers? One way would be to assign a *lot* of parameters, and give each one a default value.

```
In [2]: def myfunc(a=0,b=0,c=0,d=0,e=0):
            return sum((a,b,c,d,e))*.05

        myfunc(40,60,20)
```

```
Out[2]: 6.0
```

Obviously this is not a very efficient solution, and that's where `*args` comes in.

1.1 *args

When a function parameter starts with an asterisk, it allows for an *arbitrary number* of arguments, and the function takes them in as a tuple of values. Rewriting the above function:

```
In [3]: def myfunc(*args):
            return sum(args)*.05

        myfunc(40,60,20)
```

```
Out[3]: 6.0
```

Notice how passing the keyword "args" into the `sum()` function did the same thing as a tuple of arguments.

It is worth noting that the word "args" is itself arbitrary - any word will do so long as it's preceded by an asterisk. To demonstrate this:

```
In [4]: def myfunc(*spam):
            return sum(spam)*.05

        myfunc(40,60,20)
```

```
Out[4]: 6.0
```

1.2 **kwargs

Similarly, Python offers a way to handle arbitrary numbers of *keyworded* arguments. Instead of creating a tuple of values, `**kwargs` builds a dictionary of key/value pairs. For example:

```
In [5]: def myfunc(**kwargs):
            if 'fruit' in kwargs:
                print(f"My favorite fruit is {kwargs['fruit']}")  # review String Formatting a
            else:
                print("I don't like fruit")

        myfunc(fruit='pineapple')
```

```
My favorite fruit is pineapple
```

```
In [6]: myfunc()
```

```
I don't like fruit
```

1.3 *args and **kwargs combined

You can pass `*args` and `**kwargs` into the same function, but `*args` have to appear before `**kwargs`

```
In [7]: def myfunc(*args, **kwargs):
            if 'fruit' and 'juice' in kwargs:
                print(f"I like {' and '.join(args)} and my favorite fruit is {kwargs['fruit']}"
                print(f"May I have some {kwargs['juice']} juice?")
            else:
                pass

        myfunc('eggs','spam',fruit='cherries',juice='orange')
```

```
I like eggs and spam and my favorite fruit is cherries
May I have some orange juice?
```

Placing keyworded arguments ahead of positional arguments raises an exception:

```
In [8]: myfunc(fruit='cherries',juice='orange','eggs','spam')


          File "<ipython-input-8-fc6ff65addcc>", line 1
        myfunc(fruit='cherries',juice='orange','eggs','spam')
                                               ^
    SyntaxError: positional argument follows keyword argument
```

As with "args", you can use any name you'd like for keyworded arguments - "kwargs" is just a popular convention.

That's it! Now you should understand how `*args` and `**kwargs` provide the flexibilty to work with arbitrary numbers of arguments!

08-Functions and Methods Homework

April 9, 2019

1 Functions and Methods Homework

Complete the following questions: ____ **Write a function that computes the volume of a sphere given its radius.**

The volume of a sphere is given as

$$\frac{4}{3}r^3$$

```
In [1]: def vol(rad):
            pass
```

```
In [2]: # Check
        vol(2)
```

```
Out[2]: 33.49333333333333
```

Write a function that checks whether a number is in a given range (inclusive of high and low)

```
In [3]: def ran_check(num,low,high):
            pass
```

```
In [4]: # Check
        ran_check(5,2,7)
```

```
5 is in the range between 2 and 7
```

If you only wanted to return a boolean:

```
In [5]: def ran_bool(num,low,high):
            pass
```

```
In [6]: ran_bool(3,1,10)
```

```
Out[6]: True
```

Write a Python function that accepts a string and calculates the number of upper case letters and lower case letters.

```
Sample String : 'Hello Mr. Rogers, how are you this fine Tuesday?'
Expected Output :
No. of Upper case characters : 4
No. of Lower case Characters : 33
```

HINT: Two string methods that might prove useful: **.isupper()** and **.islower()**
If you feel ambitious, explore the Collections module to solve this problem!

```
In [7]: def up_low(s):
            pass

In [8]: s = 'Hello Mr. Rogers, how are you this fine Tuesday?'
        up_low(s)

Original String :  Hello Mr. Rogers, how are you this fine Tuesday?
No. of Upper case characters :  4
No. of Lower case Characters :  33
```

Write a Python function that takes a list and returns a new list with unique elements of the first list.

```
Sample List : [1,1,1,1,2,2,3,3,3,3,4,5]
Unique List : [1, 2, 3, 4, 5]

In [9]: def unique_list(lst):
            pass

In [10]: unique_list([1,1,1,1,2,2,3,3,3,3,4,5])

Out[10]: [1, 2, 3, 4, 5]
```

Write a Python function to multiply all the numbers in a list.

```
Sample List : [1, 2, 3, -4]
Expected Output : -24

In [11]: def multiply(numbers):
             pass

In [12]: multiply([1,2,3,-4])

Out[12]: -24
```

Write a Python function that checks whether a passed in string is palindrome or not.

Note: A palindrome is word, phrase, or sequence that reads the same backward as forward, e.g., madam or nurses run.

```
In [13]: def palindrome(s):
             pass

In [14]: palindrome('helleh')

Out[14]: True
```

Hard: Write a Python function to check whether a string is pangram or not.

```
Note : Pangrams are words or sentences containing every letter of the alphabet at least once.
For example : "The quick brown fox jumps over the lazy dog"
```

Hint: Look at the string module

```
In [15]: import string

         def ispangram(str1, alphabet=string.ascii_lowercase):
             pass

In [16]: ispangram("The quick brown fox jumps over the lazy dog")

Out[16]: True

In [17]: string.ascii_lowercase

Out[17]: 'abcdefghijklmnopqrstuvwxyz'
```

Great Job!

09-Functions and Methods Homework - Solutions

April 9, 2019

1 Functions and Methods Homework Solutions

Write a function that computes the volume of a sphere given its radius.

```
In [1]: def vol(rad):
            return (4/3)*(3.14)*(rad**3)
```

```
In [2]: # Check
        vol(2)
```

```
Out[2]: 33.49333333333333
```

Write a function that checks whether a number is in a given range (inclusive of high and low)

```
In [3]: def ran_check(num,low,high):
            #Check if num is between low and high (including low and high)
            if num in range(low,high+1):
                print('{} is in the range between {} and {}'.format(num,low,high))
            else:
                print('The number is outside the range.')
```

```
In [4]: # Check
        ran_check(5,2,7)
```

```
5 is in the range between 2 and 7
```

If you only wanted to return a boolean:

```
In [5]: def ran_bool(num,low,high):
            return num in range(low,high+1)
```

```
In [6]: ran_bool(3,1,10)
```

```
Out[6]: True
```

Write a Python function that accepts a string and calculates the number of upper case letters and lower case letters.

```
Sample String : 'Hello Mr. Rogers, how are you this fine Tuesday?'
Expected Output :
No. of Upper case characters : 4
No. of Lower case Characters : 33
```

If you feel ambitious, explore the Collections module to solve this problem!

```
In [7]: def up_low(s):
            d={"upper":0, "lower":0}
            for c in s:
                if c.isupper():
                    d["upper"]+=1
                elif c.islower():
                    d["lower"]+=1
                else:
                    pass
            print("Original String : ", s)
            print("No. of Upper case characters : ", d["upper"])
            print("No. of Lower case Characters : ", d["lower"])

In [8]: s = 'Hello Mr. Rogers, how are you this fine Tuesday?'
        up_low(s)

Original String :  Hello Mr. Rogers, how are you this fine Tuesday?
No. of Upper case characters :  4
No. of Lower case Characters :  33
```

Write a Python function that takes a list and returns a new list with unique elements of the first list.

```
Sample List : [1,1,1,1,2,2,3,3,3,3,4,5]
Unique List : [1, 2, 3, 4, 5]

In [9]: def unique_list(lst):
            # Also possible to use list(set())
            x = []
            for a in lst:
                if a not in x:
                    x.append(a)
            return x
```

```
In [10]: unique_list([1,1,1,1,2,2,3,3,3,3,4,5])

Out[10]: [1, 2, 3, 4, 5]
```

Write a Python function to multiply all the numbers in a list.

```
Sample List : [1, 2, 3, -4]
Expected Output : -24
```

```
In [11]: def multiply(numbers):
             total = 1
             for x in numbers:
                 total *= x
             return total

In [12]: multiply([1,2,3,-4])

Out[12]: -24
```

Write a Python function that checks whether a passed string is palindrome or not.

Note: A palindrome is word, phrase, or sequence that reads the same backward as forward, e.g., madam or nurses run.

```
In [13]: def palindrome(s):

             s = s.replace(' ','') # This replaces all spaces ' ' with no space ''. (Fixes iss
             return s == s[::-1]   # Check through slicing

In [14]: palindrome('nurses run')

Out[14]: True

In [15]: palindrome('abcba')

Out[15]: True
```

Hard:

Write a Python function to check whether a string is pangram or not.

```
Note : Pangrams are words or sentences containing every letter of the alphabet at least once.
For example : "The quick brown fox jumps over the lazy dog"
```

Hint: Look at the string module

```
In [16]: import string

         def ispangram(str1, alphabet=string.ascii_lowercase):
             alphaset = set(alphabet)
             return alphaset <= set(str1.lower())

In [17]: ispangram("The quick brown fox jumps over the lazy dog")

Out[17]: True

In [18]: string.ascii_lowercase

Out[18]: 'abcdefghijklmnopqrstuvwxyz'
```

01-Milestone Project 1 - Assignment

April 9, 2019

1 Milestone Project 1

1.1 Congratulations on making it to your first milestone!

You've already learned a ton and are ready to work on a real project.

Your assignment: Create a Tic Tac Toe game. You are free to use any IDE you like.

Here are the requirements:

- 2 players should be able to play the game (both sitting at the same computer)
- The board should be printed out every time a player makes a move
- You should be able to accept input of the player position and then place a symbol on the board

Feel free to use Google to help you figure anything out (but don't just Google "Tic Tac Toe in Python" otherwise you won't learn anything!) Keep in mind that this project can take anywhere between several hours to several days.

There are 4 Jupyter Notebooks related to this assignment:

- This Assignment Notebook
- A "Walkthrough Steps Workbook" Notebook
- A "Complete Walkthrough Solution" Notebook
- An "Advanced Solution" Notebook

I encourage you to just try to start the project on your own without referencing any of the notebooks. If you get stuck, check out the next lecture which is a text lecture with helpful hints and steps. If you're still stuck after that, then check out the Walkthrough Steps Workbook, which breaks up the project in steps for you to solve. Still stuck? Then check out the Complete Walkthrough Solution video for more help on approaching the project!

There are parts of this that will be a struggle...and that is good! I have complete faith that if you have made it this far through the course you have all the tools and knowledge to tackle this project. Remember, it's totally open book, so take your time, do a little research, and remember:

1.2 HAVE FUN!

02-Milestone Project 1 - Walkthrough Steps Workbook

April 9, 2019

1 Milestone Project 1: Walkthrough Steps Workbook

Below is a set of steps for you to follow to try to create the Tic Tac Toe Milestone Project game!

Some suggested tools before you get started: To take input from a user:

```
player1 = input("Please pick a marker 'X' or 'O'")
```

Note that input() takes in a string. If you need an integer value, use

```
position = int(input('Please enter a number'))
```

To clear the screen between moves:

```
from IPython.display import clear_output
clear_output()
```

Note that clear_output() will only work in jupyter. To clear the screen in other IDEs, consider:

```
print('\n'*100)
```

This scrolls the previous board up out of view. Now on to the program!

Step 1: Write a function that can print out a board. Set up your board as a list, where each index 1-9 corresponds with a number on a number pad, so you get a 3 by 3 board representation.

```
In [ ]: from IPython.display import clear_output

        def display_board(board):

            pass
```

TEST Step 1: run your function on a test version of the board list, and make adjustments as necessary

```
In [ ]: test_board = ['#','X','O','X','O','X','O','X','O','X']
        display_board(test_board)
```

Step 2: Write a function that can take in a player input and assign their marker as 'X' or 'O'. Think about using *while* loops to continually ask until you get a correct answer.

```
In [ ]: def player_input():

            pass
```

TEST Step 2: run the function to make sure it returns the desired output

```
In [ ]: player_input()
```

Step 3: Write a function that takes in the board list object, a marker ('X' or 'O'), and a desired position (number 1-9) and assigns it to the board.

```
In [ ]: def place_marker(board, marker, position):

            pass
```

TEST Step 3: run the place marker function using test parameters and display the modified board

```
In [ ]: place_marker(test_board,'$',8)
        display_board(test_board)
```

Step 4: Write a function that takes in a board and a mark (X or O) and then checks to see if that mark has won.

```
In [ ]: def win_check(board, mark):

            pass
```

TEST Step 4: run the win_check function against our test_board - it should return True

```
In [ ]: win_check(test_board,'X')
```

Step 5: Write a function that uses the random module to randomly decide which player goes first. You may want to lookup random.randint() Return a string of which player went first.

```
In [ ]: import random

        def choose_first():
            pass
```

Step 6: Write a function that returns a boolean indicating whether a space on the board is freely available.

```
In [ ]: def space_check(board, position):

            pass
```

Step 7: Write a function that checks if the board is full and returns a boolean value. True if full, False otherwise.

```
In [ ]: def full_board_check(board):

            pass
```

Step 8: Write a function that asks for a player's next position (as a number 1-9) and then uses the function from step 6 to check if it's a free position. If it is, then return the position for later use.

```
In [ ]: def player_choice(board):

            pass
```

Step 9: Write a function that asks the player if they want to play again and returns a boolean True if they do want to play again.

```
In [ ]: def replay():

            pass
```

Step 10: Here comes the hard part! Use while loops and the functions you've made to run the game!

```
In [ ]: print('Welcome to Tic Tac Toe!')

        #while True:
            # Set the game up here
            #pass

            #while game_on:
                #Player 1 Turn


                # Player2's turn.

                    #pass

            #if not replay():
                #break
```

1.1 Good Job!

03-Milestone Project 1 - Complete Walkthrough Solution

April 9, 2019

1 Milestone Project 1: Full Walk-through Code Solution

Below is the filled in code that goes along with the complete walk-through video. Check out the corresponding lecture videos for more information on this code!

Step 1: Write a function that can print out a board. Set up your board as a list, where each index 1-9 corresponds with a number on a number pad, so you get a 3 by 3 board representation.

```
In [1]: from IPython.display import clear_output

        def display_board(board):
            clear_output()  # Remember, this only works in jupyter!

            print('   |   |')
            print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
            print('   |   |')
            print('-----------')
            print('   |   |')
            print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
            print('   |   |')
            print('-----------')
            print('   |   |')
            print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
            print('   |   |')
```

TEST Step 1: run your function on a test version of the board list, and make adjustments as necessary

```
In [2]: test_board = ['#','X','O','X','O','X','O','X','O','X']
        display_board(test_board)
```

```
   |   |
 X | O | X
   |   |
-----------
   |   |
 O | X | O
   |   |
```

```
-----------
   |   |
 X | O | X
   |   |
```

Step 2: Write a function that can take in a player input and assign their marker as 'X' or 'O'. Think about using *while* loops to continually ask until you get a correct answer.

```
In [3]: def player_input():
            marker = ''

            while not (marker == 'X' or marker == 'O'):
                marker = input('Player 1: Do you want to be X or O? ').upper()

            if marker == 'X':
                return ('X', 'O')
            else:
                return ('O', 'X')
```

TEST Step 2: run the function to make sure it returns the desired output

```
In [4]: player_input()
```

```
Player 1: Do you want to be X or O? X
```

```
Out[4]: ('X', 'O')
```

Step 3: Write a function that takes in the board list object, a marker ('X' or 'O'), and a desired position (number 1-9) and assigns it to the board.

```
In [5]: def place_marker(board, marker, position):
            board[position] = marker
```

TEST Step 3: run the place marker function using test parameters and display the modified board

```
In [6]: place_marker(test_board,'$',8)
        display_board(test_board)
```

```
   |   |
 X | $ | X
   |   |
-----------
   |   |
 O | X | O
   |   |
-----------
   |   |
 X | O | X
   |   |
```

Step 4: Write a function that takes in a board and checks to see if someone has won.

```
In [7]: def win_check(board,mark):

            return ((board[7] == mark and board[8] == mark and board[9] == mark) or # across t
            (board[4] == mark and board[5] == mark and board[6] == mark) or # across the middl
            (board[1] == mark and board[2] == mark and board[3] == mark) or # across the botto
            (board[7] == mark and board[4] == mark and board[1] == mark) or # down the middle
            (board[8] == mark and board[5] == mark and board[2] == mark) or # down the middle
            (board[9] == mark and board[6] == mark and board[3] == mark) or # down the right s
            (board[7] == mark and board[5] == mark and board[3] == mark) or # diagonal
            (board[9] == mark and board[5] == mark and board[1] == mark)) # diagonal
```

TEST Step 4: run the win_check function against our test_board - it should return True

```
In [8]: win_check(test_board,'X')
```

```
Out[8]: True
```

Step 5: Write a function that uses the random module to randomly decide which player goes first. You may want to lookup random.randint() Return a string of which player went first.

```
In [9]: import random

        def choose_first():
            if random.randint(0, 1) == 0:
                return 'Player 2'
            else:
                return 'Player 1'
```

Step 6: Write a function that returns a boolean indicating whether a space on the board is freely available.

```
In [10]: def space_check(board, position):

             return board[position] == ' '
```

Step 7: Write a function that checks if the board is full and returns a boolean value. True if full, False otherwise.

```
In [11]: def full_board_check(board):
             for i in range(1,10):
                 if space_check(board, i):
                     return False
             return True
```

Step 8: Write a function that asks for a player's next position (as a number 1-9) and then uses the function from step 6 to check if its a free position. If it is, then return the position for later use.

```
In [12]: def player_choice(board):
             position = 0

             while position not in [1,2,3,4,5,6,7,8,9] or not space_check(board, position):
                 position = int(input('Choose your next position: (1-9) '))

             return position
```

Step 9: Write a function that asks the player if they want to play again and returns a boolean True if they do want to play again.

```
In [13]: def replay():

             return input('Do you want to play again? Enter Yes or No: ').lower().startswith('y
```

Step 10: Here comes the hard part! Use while loops and the functions you've made to run the game!

```
In [14]: print('Welcome to Tic Tac Toe!')

         while True:
             # Reset the board
             theBoard = [' '] * 10
             player1_marker, player2_marker = player_input()
             turn = choose_first()
             print(turn + ' will go first.')

             play_game = input('Are you ready to play? Enter Yes or No.')

             if play_game.lower()[0] == 'y':
                 game_on = True
             else:
                 game_on = False

             while game_on:
                 if turn == 'Player 1':
                     # Player1's turn.

                     display_board(theBoard)
                     position = player_choice(theBoard)
                     place_marker(theBoard, player1_marker, position)

                     if win_check(theBoard, player1_marker):
                         display_board(theBoard)
                         print('Congratulations! You have won the game!')
                         game_on = False
                     else:
                         if full_board_check(theBoard):
                             display_board(theBoard)
```

```
                            print('The game is a draw!')
                            break
                        else:
                            turn = 'Player 2'

                else:
                    # Player2's turn.

                    display_board(theBoard)
                    position = player_choice(theBoard)
                    place_marker(theBoard, player2_marker, position)

                    if win_check(theBoard, player2_marker):
                        display_board(theBoard)
                        print('Player 2 has won!')
                        game_on = False
                    else:
                        if full_board_check(theBoard):
                            display_board(theBoard)
                            print('The game is a draw!')
                            break
                        else:
                            turn = 'Player 1'

            if not replay():
                break
```

```
   |   |
   | O | O
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
 X | X | X
   |   |
Congratulations! You have won the game!
Do you want to play again? Enter Yes or No: No
```

1.1 Good Job!

04-OPTIONAL -Milestone Project 1 - Advanced Solution

April 9, 2019

1 Tic Tac Toe - Advanced Solution

This solution follows the same basic format as the Complete Walkthrough Solution, but takes advantage of some of the more advanced statements we have learned. Feel free to download the notebook to understand how it works!

```
In [1]: # Specifically for the iPython Notebook environment for clearing output
        from IPython.display import clear_output
        import random

        # Global variables
        theBoard = [' '] * 10   # a list of empty spaces
        available = [str(num) for num in range(0,10)] # a List Comprehension
        players = [0,'X','O']   # note that players[1] == 'X' and players[-1] == 'O'
```

```
In [2]: def display_board(a,b):
            print('Available   TIC-TAC-TOE\n'+
                  '  moves\n\n  '+
                  a[7]+'|'+a[8]+'|'+a[9]+'        '+b[7]+'|'+b[8]+'|'+b[9]+'\n  '+
                  '-----        -----\n  '+
                  a[4]+'|'+a[5]+'|'+a[6]+'        '+b[4]+'|'+b[5]+'|'+b[6]+'\n  '+
                  '-----        -----\n  '+
                  a[1]+'|'+a[2]+'|'+a[3]+'        '+b[1]+'|'+b[2]+'|'+b[3]+'\n')
        display_board(available,theBoard)
```

```
Available   TIC-TAC-TOE
  moves

  7|8|9         | |
  -----        -----
  4|5|6         | |
  -----        -----
  1|2|3         | |
```

```
In [11]: def display_board(a,b):
             print(f'Available   TIC-TAC-TOE\n  moves\n\n  {a[7]}|{a[8]}|{a[9]}        {b[7]}|
         display_board(available,theBoard)
```

```
Available   TIC-TAC-TOE
  moves

  7|8|9         | |
  -----        -----
  4|5|6         | |
  -----        -----
  1|2|3         | |
```

```
In [3]: def place_marker(avail,board,marker,position):
            board[position] = marker
            avail[position] = ' '

In [4]: def win_check(board,mark):

            return ((board[7] ==  board[8] ==  board[9] == mark) or # across the top
            (board[4] ==  board[5] ==  board[6] == mark) or # across the middle
            (board[1] ==  board[2] ==  board[3] == mark) or # across the bottom
            (board[7] ==  board[4] ==  board[1] == mark) or # down the middle
            (board[8] ==  board[5] ==  board[2] == mark) or # down the middle
            (board[9] ==  board[6] ==  board[3] == mark) or # down the right side
            (board[7] ==  board[5] ==  board[3] == mark) or # diagonal
            (board[9] ==  board[5] ==  board[1] == mark)) # diagonal

In [5]: def random_player():
            return random.choice((-1, 1))

        def space_check(board,position):
            return board[position] == ' '

        def full_board_check(board):
            return ' ' not in board[1:]

In [6]: def player_choice(board,player):
            position = 0

            while position not in [1,2,3,4,5,6,7,8,9] or not space_check(board, position):
                try:
                    position = int(input('Player %s, choose your next position: (1-9) '%(player
                except:
                    print("I'm sorry, please try again.")

            return position

In [7]: def replay():

            return input('Do you want to play again? Enter Yes or No: ').lower().startswith('y
```

```
In [ ]: while True:
            clear_output()
            print('Welcome to Tic Tac Toe!')

            toggle = random_player()
            player = players[toggle]
            print('For this round, Player %s will go first!' %(player))

            game_on = True
            input('Hit Enter to continue')
            while game_on:
                display_board(available,theBoard)
                position = player_choice(theBoard,player)
                place_marker(available,theBoard,player,position)

                if win_check(theBoard, player):
                    display_board(available,theBoard)
                    print('Congratulations! Player '+player+' wins!')
                    game_on = False
                else:
                    if full_board_check(theBoard):
                        display_board(available,theBoard)
                        print('The game is a draw!')
                        break
                    else:
                        toggle *= -1
                        player = players[toggle]
                        clear_output()

            # reset the board and available moves list
            theBoard = [' '] * 10
            available = [str(num) for num in range(0,10)]

            if not replay():
                break
```

```
Welcome to Tic Tac Toe!
For this round, Player X will go first!
```

```
In [ ]:
```

01-Object Oriented Programming

April 9, 2019

1 Object Oriented Programming

Object Oriented Programming (OOP) tends to be one of the major obstacles for beginners when they are first starting to learn Python.

There are many, many tutorials and lessons covering OOP so feel free to Google search other lessons, and I have also put some links to other useful tutorials online at the bottom of this Notebook.

For this lesson we will construct our knowledge of OOP in Python by building on the following topics:

- Objects
- Using the *class* keyword
- Creating class attributes
- Creating methods in a class
- Learning about Inheritance
- Learning about Polymorphism
- Learning about Special Methods for classes

Lets start the lesson by remembering about the Basic Python Objects. For example:

```
In [1]: lst = [1,2,3]
```

Remember how we could call methods on a list?

```
In [2]: lst.count(2)
```

```
Out[2]: 1
```

What we will basically be doing in this lecture is exploring how we could create an Object type like a list. We've already learned about how to create functions. So let's explore Objects in general:

1.1 Objects

In Python, *everything is an object*. Remember from previous lectures we can use type() to check the type of object something is:

```
In [3]: print(type(1))
        print(type([]))
        print(type(()))
        print(type({}))
```

```
<class 'int'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

So we know all these things are objects, so how can we create our own Object types? That is where the class keyword comes in. ## class User defined objects are created using the class keyword. The class is a blueprint that defines the nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class. For example, above we created the object lst which was an instance of a list object.

Let see how we can use class:

```
In [4]: # Create a new object type called Sample
        class Sample:
            pass

        # Instance of Sample
        x = Sample()

        print(type(x))
```

```
<class '__main__.Sample'>
```

By convention we give classes a name that starts with a capital letter. Note how x is now the reference to our new instance of a Sample class. In other words, we **instantiate** the Sample class.

Inside of the class we currently just have pass. But we can define class attributes and methods.

An **attribute** is a characteristic of an object. A **method** is an operation we can perform with the object.

For example, we can create a class called Dog. An attribute of a dog may be its breed or its name, while a method of a dog may be defined by a .bark() method which returns a sound.

Let's get a better understanding of attributes through an example.

1.2 Attributes

The syntax for creating an attribute is:

```
self.attribute = something
```

There is a special method called:

```
__init__()
```

This method is used to initialize the attributes of an object. For example:

```
In [5]: class Dog:
            def __init__(self,breed):
                self.breed = breed

        sam = Dog(breed='Lab')
        frank = Dog(breed='Huskie')
```

Lets break down what we have above.The special method

```
__init__()
```

is called automatically right after the object has been created:

```
def __init__(self, breed):
```

Each attribute in a class definition begins with a reference to the instance object. It is by convention named self. The breed is the argument. The value is passed during the class instantiation.

```
 self.breed = breed
```

Now we have created two instances of the Dog class. With two breed types, we can then access these attributes like this:

```
In [6]: sam.breed

Out[6]: 'Lab'

In [7]: frank.breed

Out[7]: 'Huskie'
```

Note how we don't have any parentheses after breed; this is because it is an attribute and doesn't take any arguments.

In Python there are also *class object attributes*. These Class Object Attributes are the same for any instance of the class. For example, we could create the attribute *species* for the Dog class. Dogs, regardless of their breed, name, or other attributes, will always be mammals. We apply this logic in the following manner:

```
In [8]: class Dog:

            # Class Object Attribute
            species = 'mammal'

            def __init__(self,breed,name):
                self.breed = breed
                self.name = name

In [9]: sam = Dog('Lab','Sam')

In [10]: sam.name

Out[10]: 'Sam'
```

Note that the Class Object Attribute is defined outside of any methods in the class. Also by convention, we place them first before the init.

```
In [11]: sam.species

Out[11]: 'mammal'
```

1.3 Methods

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are a key concept of the OOP paradigm. They are essential to dividing responsibilities in programming, especially in large applications.

You can basically think of methods as functions acting on an Object that take the Object itself into account through its *self* argument.

Let's go through an example of creating a Circle class:

```
In [12]: class Circle:
             pi = 3.14

             # Circle gets instantiated with a radius (default is 1)
             def __init__(self, radius=1):
                 self.radius = radius
                 self.area = radius * radius * Circle.pi

             # Method for resetting Radius
             def setRadius(self, new_radius):
                 self.radius = new_radius
                 self.area = new_radius * new_radius * self.pi

             # Method for getting Circumference
             def getCircumference(self):
                 return self.radius * self.pi * 2


         c = Circle()

         print('Radius is: ',c.radius)
         print('Area is: ',c.area)
         print('Circumference is: ',c.getCircumference())
```

```
Radius is:  1
Area is:  3.14
Circumference is:  6.28
```

In the __init__ method above, in order to calculate the area attribute, we had to call Circle.pi. This is because the object does not yet have its own .pi attribute, so we call the Class Object Attribute pi instead. In the setRadius method, however, we'll be working with an existing Circle object that does have its own pi attribute. Here we can use either Circle.pi or self.pi. Now let's change the radius and see how that affects our Circle object:

```
In [13]: c.setRadius(2)

         print('Radius is: ',c.radius)
         print('Area is: ',c.area)
         print('Circumference is: ',c.getCircumference())
```

```
Radius is:  2
Area is:  12.56
Circumference is:  12.56
```

Great! Notice how we used self. notation to reference attributes of the class within the method calls. Review how the code above works and try creating your own method.

1.4 Inheritance

Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called derived classes, the classes that we derive from are called base classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

Let's see an example by incorporating our previous work on the Dog class:

```
In [14]: class Animal:
             def __init__(self):
                 print("Animal created")

             def whoAmI(self):
                 print("Animal")

             def eat(self):
                 print("Eating")


         class Dog(Animal):
             def __init__(self):
                 Animal.__init__(self)
                 print("Dog created")

             def whoAmI(self):
                 print("Dog")

             def bark(self):
                 print("Woof!")
```

```
In [15]: d = Dog()
```

```
Animal created
Dog created
```

```
In [16]: d.whoAmI()
```

```
Dog
```

```
In [17]: d.eat()
```

```
Eating
```

```
In [18]: d.bark()
```

```
Woof!
```

In this example, we have two classes: Animal and Dog. The Animal is the base class, the Dog is the derived class.

The derived class inherits the functionality of the base class.

- It is shown by the eat() method.

The derived class modifies existing behavior of the base class.

- shown by the whoAmI() method.

Finally, the derived class extends the functionality of the base class, by defining a new bark() method.

1.5 Polymorphism

We've learned that while functions can take in different arguments, methods belong to the objects they act on. In Python, *polymorphism* refers to the way in which different object classes can share the same method name, and those methods can be called from the same place even though a variety of different objects might be passed in. The best way to explain this is by example:

```
In [19]: class Dog:
             def __init__(self, name):
                 self.name = name

             def speak(self):
                 return self.name+' says Woof!'

         class Cat:
             def __init__(self, name):
                 self.name = name

             def speak(self):
                 return self.name+' says Meow!'

         niko = Dog('Niko')
         felix = Cat('Felix')

         print(niko.speak())
         print(felix.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Here we have a Dog class and a Cat class, and each has a `.speak()` method. When called, each object's `.speak()` method returns a result unique to the object.

There a few different ways to demonstrate polymorphism. First, with a for loop:

```
In [20]: for pet in [niko,felix]:
             print(pet.speak())

Niko says Woof!
Felix says Meow!
```

Another is with functions:

```
In [21]: def pet_speak(pet):
             print(pet.speak())

         pet_speak(niko)
         pet_speak(felix)

Niko says Woof!
Felix says Meow!
```

In both cases we were able to pass in different object types, and we obtained object-specific results from the same mechanism.

A more common practice is to use abstract classes and inheritance. An abstract class is one that never expects to be instantiated. For example, we will never have an Animal object, only Dog and Cat objects, although Dogs and Cats are derived from Animals:

```
In [22]: class Animal:
             def __init__(self, name):    # Constructor of the class
                 self.name = name

             def speak(self):              # Abstract method, defined by convention only
                 raise NotImplementedError("Subclass must implement abstract method")


         class Dog(Animal):

             def speak(self):
                 return self.name+' says Woof!'

         class Cat(Animal):

             def speak(self):
                 return self.name+' says Meow!'

         fido = Dog('Fido')
         isis = Cat('Isis')
```

```
print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

Real life examples of polymorphism include: * opening different file types - different tools are needed to display Word, pdf and Excel files * adding different objects - the + operator performs arithmetic and concatenation

1.6 Special Methods

Finally let's go over special methods. Classes in Python can implement certain operations with special method names. These methods are not actually called directly but by Python specific language syntax. For example let's create a Book class:

```
In [23]: class Book:
             def __init__(self, title, author, pages):
                 print("A book is created")
                 self.title = title
                 self.author = author
                 self.pages = pages

             def __str__(self):
                 return "Title: %s, author: %s, pages: %s" %(self.title, self.author, self.pag

             def __len__(self):
                 return self.pages

             def __del__(self):
                 print("A book is destroyed")
```

```
In [24]: book = Book("Python Rocks!", "Jose Portilla", 159)

         #Special Methods
         print(book)
         print(len(book))
         del book
```

```
A book is created
Title: Python Rocks!, author: Jose Portilla, pages: 159
159
A book is destroyed
```

The __init__(), __str__(), __len__() and __del__() methods

These special methods are defined by their use of underscores. They allow us to use Python specific functions on objects created through our class.

Great! After this lecture you should have a basic understanding of how to create your own objects with class in Python. You will be utilizing this heavily in your next milestone project!

For more great resources on this topic, check out:

Jeff Knupp's Post

Mozilla's Post

Tutorial's Point

Official Documentation

02-Object Oriented Programming Homework

April 9, 2019

1 Object Oriented Programming

1.1 Homework Assignment

Problem 1 Fill in the Line class methods to accept coordinates as a pair of tuples and return the slope and distance of the line.

```
In [1]: class Line:

            def __init__(self,coor1,coor2):
                pass

            def distance(self):
                pass

            def slope(self):
                pass
```

```
In [2]: # EXAMPLE OUTPUT

        coordinate1 = (3,2)
        coordinate2 = (8,10)

        li = Line(coordinate1,coordinate2)
```

```
In [3]: li.distance()

Out[3]: 9.433981132056603
```

```
In [4]: li.slope()

Out[4]: 1.6
```

Problem 2 Fill in the class

```
In [5]: class Cylinder:

            def __init__(self,height=1,radius=1):
                pass

            def volume(self):
                pass

            def surface_area(self):
                pass
```

```
In [6]: # EXAMPLE OUTPUT
        c = Cylinder(2,3)
```

```
In [7]: c.volume()
```

```
Out[7]: 56.52
```

```
In [8]: c.surface_area()
```

```
Out[8]: 94.2
```

03-Object Oriented Programming Homework - Solution

April 9, 2019

1 Object Oriented Programming

1.1 Homework Assignment

Problem 1 Fill in the Line class methods to accept coordinates as a pair of tuples and return the slope and distance of the line.

```
In [1]: class Line(object):

            def __init__(self,coor1,coor2):
                self.coor1 = coor1
                self.coor2 = coor2

            def distance(self):
                x1,y1 = self.coor1
                x2,y2 = self.coor2
                return ((x2-x1)**2 + (y2-y1)**2)**0.5

            def slope(self):
                x1,y1 = self.coor1
                x2,y2 = self.coor2
                return (y2-y1)/(x2-x1)
```

```
In [2]: coordinate1 = (3,2)
        coordinate2 = (8,10)

        li = Line(coordinate1,coordinate2)
```

```
In [3]: li.distance()
```

```
Out[3]: 9.433981132056603
```

```
In [4]: li.slope()
```

```
Out[4]: 1.6
```

Problem 2 Fill in the class

```
In [5]: class Cylinder:

            def __init__(self,height=1,radius=1):
                self.height = height
                self.radius = radius

            def volume(self):
                return self.height*3.14*(self.radius)**2

            def surface_area(self):
                top = 3.14 * (self.radius)**2
                return (2*top) + (2*3.14*self.radius*self.height)

In [6]: c = Cylinder(2,3)

In [7]: c.volume()

Out[7]: 56.52

In [8]: c.surface_area()

Out[8]: 94.2
```

04-OOP Challenge

April 9, 2019

1 Object Oriented Programming Challenge

For this challenge, create a bank account class that has two attributes:

- owner
- balance

and two methods:

- deposit
- withdraw

As an added requirement, withdrawals may not exceed the available balance.

Instantiate your class, make several deposits and withdrawals, and test to make sure the account can't be overdrawn.

```
In [1]: class Account:
            pass
```

```
In [2]: # 1. Instantiate the class
        acct1 = Account('Jose',100)
```

```
In [3]: # 2. Print the object
        print(acct1)
```

```
Account owner:   Jose
Account balance: $100
```

```
In [4]: # 3. Show the account owner attribute
        acct1.owner
```

```
Out[4]: 'Jose'
```

```
In [5]: # 4. Show the account balance attribute
        acct1.balance
```

```
Out[5]: 100
```

```
In [6]: # 5. Make a series of deposits and withdrawals
        acct1.deposit(50)
```

```
Deposit Accepted
```

```
In [7]: acct1.withdraw(75)
```

```
Withdrawal Accepted
```

```
In [8]: # 6. Make a withdrawal that exceeds the available balance
        acct1.withdraw(500)
```

```
Funds Unavailable!
```

1.1 Good job!

05-OOP Challenge - Solution

April 9, 2019

1 Object Oriented Programming Challenge - Solution

For this challenge, create a bank account class that has two attributes:

- owner
- balance

and two methods:

- deposit
- withdraw

As an added requirement, withdrawals may not exceed the available balance.
Instantiate your class, make several deposits and withdrawals, and test to make sure the account can't be overdrawn.

```
In [1]: class Account:
            def __init__(self,owner,balance=0):
                self.owner = owner
                self.balance = balance

            def __str__(self):
                return f'Account owner:   {self.owner}\nAccount balance: ${self.balance}'

            def deposit(self,dep_amt):
                self.balance += dep_amt
                print('Deposit Accepted')

            def withdraw(self,wd_amt):
                if self.balance >= wd_amt:
                    self.balance -= wd_amt
                    print('Withdrawal Accepted')
                else:
                    print('Funds Unavailable!')
```

```
In [2]: # 1. Instantiate the class
        acct1 = Account('Jose',100)
```

```
In [3]: # 2. Print the object
        print(acct1)

Account owner:   Jose
Account balance: $100


In [4]: # 3. Show the account owner attribute
        acct1.owner

Out[4]: 'Jose'

In [5]: # 4. Show the account balance attribute
        acct1.balance

Out[5]: 100

In [6]: # 5. Make a series of deposits and withdrawals
        acct1.deposit(50)

Deposit Accepted


In [7]: acct1.withdraw(75)

Withdrawal Accepted


In [8]: # 6. Make a withdrawal that exceeds the available balance
        acct1.withdraw(500)

Funds Unavailable!
```

1.1 Good job!

Useful_Info_Notebook

April 9, 2019

1 Modules and Packages

In this section we briefly: * code out a basic module and show how to import it into a Python script * run a Python script from a Jupyter cell * show how command line arguments can be passed into a script

Check out the video lectures for more info and resources for this.

The best online resource is the official docs: https://docs.python.org/3/tutorial/modules.html#packages

But I really like the info here: https://python4astronomers.github.io/installation/packages.html

1.1 Writing modules

```
In [1]: %%writefile file1.py
        def myfunc(x):
            return [num for num in range(x) if num%2==0]
        list1 = myfunc(11)

Writing file1.py
```

file1.py is going to be used as a module.

Note that it doesn't print or return anything, it just defines a function called *myfunc* and a variable called *list1*. ## Writing scripts

```
In [2]: %%writefile file2.py
        import file1
        file1.list1.append(12)
        print(file1.list1)

Writing file2.py
```

file2.py is a Python script.

First, we import our **file1** module (note the lack of a .py extension) Next, we access the *list1* variable inside **file1**, and perform a list method on it. `.append(12)` proves we're working with a Python list object, and not just a string. Finally, we tell our script to print the modified list. ## Running scripts

```
In [3]: ! python file2.py
```

```
[0, 2, 4, 6, 8, 10, 12]
```

Here we run our script from the command line. The exclamation point is a Jupyter trick that lets you run command line statements from inside a jupyter cell.

```
In [4]: import file1
        print(file1.list1)
```

```
[0, 2, 4, 6, 8, 10]
```

The above cell proves that we never altered **file1.py**, we just appended a number to the list *after* it was brought into **file2**.

1.2 Passing command line arguments

Python's `sys` module gives you access to command line arguments when calling scripts.

```
In [5]: %%writefile file3.py
        import sys
        import file1
        num = int(sys.argv[1])
        print(file1.myfunc(num))
```

```
Writing file3.py
```

Note that we selected the second item in the list of arguments with `sys.argv[1]`. This is because the list created with `sys.argv` always starts with the name of the file being used.

```
In [6]: ! python file3.py 21
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Here we're passing 21 to be the upper range value used by the *myfunc* function in **list1.py**

1.3 Understanding modules

Modules in Python are simply Python files with the .py extension, which implement a set of functions. Modules are imported from other modules using the import command.

To import a module, we use the import command. Check out the full list of built-in modules in the Python standard library here.

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton" - they are initialized only once.

If we want to import the math module, we simply import the name of the module:

```
In [7]: # import the library
        import math

In [8]: # use it (ceiling rounding)
        math.ceil(2.4)

Out[8]: 3
```

1.4 Exploring built-in modules

Two very important functions come in handy when exploring modules in Python - the dir and help functions.

We can look for which functions are implemented in each module by using the dir function:

```
In [9]: print(dir(math))

['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asi
```

When we find the function in the module we want to use, we can read about it more using the help function, inside the Python interpreter:

```
In [10]: help(math.ceil)

Help on built-in function ceil in module math:

ceil(...)
    ceil(x)

    Return the ceiling of x as an Integral.
    This is the smallest integer >= x.
```

1.5 Writing modules

Writing Python modules is very simple. To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.

1.6 Writing packages

Packages are name-spaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

Each package in Python is a directory which MUST contain a special file called ***_init_*.py**. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the ***_init_*.py** file inside the foo directory.

To use the module bar, we can import it in two ways:

```
In [ ]: # Just an example, this won't work
        import foo.bar

In [ ]: # OR could do it this way
        from foo import bar
```

In the first method, we must use the foo prefix whenever we access the module bar. In the second method, we don't, because we import the module to our module's name-space.

The ***_init_*.py** file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the ***_all_*** variable, like so:

```
In [ ]: __init__.py:

        __all__ = ["bar"]
```

01-Errors and Exceptions Handling

April 9, 2019

1 Errors and Exception Handling

In this lecture we will learn about Errors and Exception Handling in Python. You've definitely already encountered errors by this point in the course. For example:

```
In [1]: print('Hello)
```

```
          File "<ipython-input-1-db8c9988558c>", line 1
        print('Hello)
                     ^
    SyntaxError: EOL while scanning string literal
```

Note how we get a SyntaxError, with the further description that it was an EOL (End of Line Error) while scanning the string literal. This is specific enough for us to see that we forgot a single quote at the end of the line. Understanding these various error types will help you debug your code much faster.

This type of error and description is known as an Exception. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal.

You can check out the full list of built-in exceptions here. Now let's learn how to handle errors and exceptions in our own code.

1.1 try and except

The basic terminology and syntax used to handle errors in Python are the try and except statements. The code which can cause an exception to occur is put in the try block and the handling of the exception is then implemented in the except block of code. The syntax follows:

```
try:
   You do your operations here...
   ...
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
```

```
   If there is ExceptionII, then execute this block.
   ...
else:
   If there is no exception then execute this block.
```

We can also just check for any exception with just using except: To get a better understanding of all this let's check out an example: We will look at some code that opens and writes a file:

```
In [2]: try:
            f = open('testfile','w')
            f.write('Test write this')
        except IOError:
            # This will only check for an IOError exception and then execute this print statem
            print("Error: Could not find file or read data")
        else:
            print("Content written successfully")
            f.close()
```

```
Content written successfully
```

Now let's see what would happen if we did not have write permission (opening only with 'r'):

```
In [3]: try:
            f = open('testfile','r')
            f.write('Test write this')
        except IOError:
            # This will only check for an IOError exception and then execute this print statem
            print("Error: Could not find file or read data")
        else:
            print("Content written successfully")
            f.close()
```

```
Error: Could not find file or read data
```

Great! Notice how we only printed a statement! The code still ran and we were able to continue doing actions and running code blocks. This is extremely useful when you have to account for possible input errors in your code. You can be prepared for the error and keep running code, instead of your code just breaking as we saw above.

We could have also just said except: if we weren't sure what exception would occur. For example:

```
In [4]: try:
            f = open('testfile','r')
            f.write('Test write this')
        except:
            # This will check for any exception and then execute this print statement
            print("Error: Could not find file or read data")
```

```
        else:
            print("Content written successfully")
            f.close()
```

```
Error: Could not find file or read data
```

Great! Now we don't actually need to memorize that list of exception types! Now what if we kept wanting to run code after the exception occurred? This is where finally comes in. ## finally The finally: block of code will always be run regardless if there was an exception in the try code block. The syntax is:

```
try:
   Code block here
   ...
   Due to any exception, this code may be skipped!
finally:
   This code block would always be executed.
```

For example:

```
In [5]: try:
            f = open("testfile", "w")
            f.write("Test write statement")
            f.close()
        finally:
            print("Always execute finally code blocks")
```

```
Always execute finally code blocks
```

We can use this in conjunction with except. Let's see a new example that will take into account a user providing the wrong input:

```
In [6]: def askint():
            try:
                val = int(input("Please enter an integer: "))
            except:
                print("Looks like you did not enter an integer!")

            finally:
                print("Finally, I executed!")
            print(val)
```

```
In [7]: askint()
```

```
Please enter an integer: 5
Finally, I executed!
5
```

```
In [8]: askint()

Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)

<ipython-input-8-cc291aa76c10> in <module>()
----> 1 askint()


<ipython-input-6-c97dd1c75d24> in askint()
      7     finally:
      8         print("Finally, I executed!")
----> 9     print(val)


UnboundLocalError: local variable 'val' referenced before assignment
```

Notice how we got an error when trying to print val (because it was never properly assigned). Let's remedy this by asking the user and checking to make sure the input type is an integer:

```
In [9]: def askint():
            try:
                val = int(input("Please enter an integer: "))
            except:
                print("Looks like you did not enter an integer!")
                val = int(input("Try again-Please enter an integer: "))
            finally:
                print("Finally, I executed!")
            print(val)

In [10]: askint()

Please enter an integer: five
Looks like you did not enter an integer!
Try again-Please enter an integer: four
Finally, I executed!
```

```
---------------------------------------------------------------------------
```

```
    ValueError                                Traceback (most recent call last)

    <ipython-input-9-92b5f751eb01> in askint()
      2     try:
----> 3         val = int(input("Please enter an integer: "))
      4     except:


    ValueError: invalid literal for int() with base 10: 'five'


During handling of the above exception, another exception occurred:


    ValueError                                Traceback (most recent call last)

    <ipython-input-10-cc291aa76c10> in <module>()
----> 1 askint()


    <ipython-input-9-92b5f751eb01> in askint()
      4     except:
      5         print("Looks like you did not enter an integer!")
----> 6         val = int(input("Try again-Please enter an integer: "))
      7     finally:
      8         print("Finally, I executed!")


    ValueError: invalid literal for int() with base 10: 'four'
```

Hmmm…that only did one check. How can we continually keep checking? We can use a while loop!

```
In [11]: def askint():
             while True:
                 try:
                     val = int(input("Please enter an integer: "))
                 except:
                     print("Looks like you did not enter an integer!")
                     continue
                 else:
                     print("Yep that's an integer!")
                     break
                 finally:
                     print("Finally, I executed!")
                 print(val)

In [12]: askint()
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: four
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: 3
Yep that's an integer!
Finally, I executed!
```

So why did our function print "Finally, I executed!" after each trial, yet it never printed `val` itself? This is because with a try/except/finally clause, any continue or break statements are reserved until *after* the try clause is completed. This means that even though a successful input of **3** brought us to the else: block, and a break statement was thrown, the try clause continued through to finally: before breaking out of the while loop. And since print(val) was outside the try clause, the break statement prevented it from running.

Let's make one final adjustment:

```
In [13]: def askint():
             while True:
                 try:
                     val = int(input("Please enter an integer: "))
                 except:
                     print("Looks like you did not enter an integer!")
                     continue
                 else:
                     print("Yep that's an integer!")
                     print(val)
                     break
                 finally:
                     print("Finally, I executed!")

In [14]: askint()

Please enter an integer: six
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: 6
Yep that's an integer!
6
Finally, I executed!
```

Great! Now you know how to handle errors and exceptions in Python with the try, except, else, and finally notation!

02-Errors and Exceptions Homework

April 9, 2019

1 Errors and Exceptions Homework

1.0.1 Problem 1

Handle the exception thrown by the code below by using try and except blocks.

```
In [1]: for i in ['a','b','c']:
            print(i**2)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-1-c35f41ad7311> in <module>()
      1 for i in ['a','b','c']:
----> 2     print(i**2)


TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

1.0.2 Problem 2

Handle the exception thrown by the code below by using try and except blocks. Then use a finally block to print 'All Done.'

```
In [2]: x = 5
        y = 0

        z = x/y
```

```
---------------------------------------------------------------------------

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-2-6f985c4c80dd> in <module>()
```

```
      2 y = 0
      3
----> 4 z = x/y


    ZeroDivisionError: division by zero
```

1.0.3 Problem 3

Write a function that asks for an integer and prints the square of it. Use a while loop with a try, except, else block to account for incorrect inputs.

```
In [3]: def ask():
            pass

In [4]: ask()

Input an integer: null
An error occurred! Please try again!
Input an integer: 2
Thank you, your number squared is:  4
```

2 Great Job!

03-Errors and Exceptions Homework - Solution

April 9, 2019

1 Errors and Exceptions Homework - Solution

1.0.1 Problem 1

Handle the exception thrown by the code below by using try and except blocks.

```
In [1]: try:
            for i in ['a','b','c']:
                print(i**2)
        except:
            print("An error occurred!")
```

```
An error occurred!
```

1.0.2 Problem 2

Handle the exception thrown by the code below by using try and except blocks. Then use a finally block to print 'All Done.'

```
In [2]: x = 5
        y = 0
        try:
            z = x/y
        except ZeroDivisionError:
            print("Can't divide by Zero!")
        finally:
            print('All Done!')
```

```
Can't divide by Zero!
All Done!
```

1.0.3 Problem 3

Write a function that asks for an integer and prints the square of it. Use a while loop with a try, except, else block to account for incorrect inputs.

```
In [3]: def ask():

            while True:
                try:
                    n = int(input('Input an integer: '))
                except:
                    print('An error occurred! Please try again!')
                    continue
                else:
                    break


            print('Thank you, your number squared is: ',n**2)
```

```
In [4]: ask()
```

```
Input an integer: null
An error occurred! Please try again!
Input an integer: 2
Thank you, your number squared is:  4
```

2 Great Job!

04-Unit Testing

April 9, 2019

1 Unit Testing

Equally important as writing good code is writing good tests. Better to find bugs yourself than have them reported to you by end users!

For this section we'll be working with files outside the notebook. We'll save our code to a .py file, and then save our test script to another .py file. Normally we would code these files using a text editor like Brackets or Atom, or inside an IDE like Spyder or Pycharm. But, since we're here, let's use Jupyter!

Recall that with some IPython magic we can write the contents of a cell to a file using `%%writefile`. Something we haven't seen yet; you can run terminal commands from a jupyter cell using `!`

1.1 Testing tools

There are dozens of good testing libraries out there. Most are third-party packages that require an install, such as:

- pylint
- pyflakes
- pep8

These are simple tools that merely look at your code, and they'll tell you if there are style issues or simple problems like variable names being called before assignment.

A far better way to test your code is to write tests that send sample data to your program, and compare what's returned to a desired outcome.Two such tools are available from the standard library:

- unittest
- doctest

Let's look at pylint first, then we'll do some heavier lifting with unittest.

1.2 `pylint`

`pylint` tests for style as well as some very basic program logic.

First, if you don't have it already (and you probably do, as it's part of the Anaconda distribution), you should install `pylint`.Once that's done feel free to comment out the cell, you won't need it anymore.

```
In [ ]: ! pip install pylint
```

Let's save a very simple script:

```
In [1]: %%writefile simple1.py
        a = 1
        b = 2
        print(a)
        print(B)
```

```
Overwriting simple1.py
```

Now let's check it using pylint

```
In [2]: ! pylint simple1.py
```

```
************* Module simple1
C:  4, 0: Final newline missing (missing-final-newline)
C:  1, 0: Missing module docstring (missing-docstring)
C:  1, 0: Invalid constant name "a" (invalid-name)
C:  2, 0: Invalid constant name "b" (invalid-name)
E:  4, 6: Undefined variable 'B' (undefined-variable)

-------------------------------------------------------------------

Your code has been rated at -12.50/10 (previous run: 8.33/10, -20.83)




No config file found, using default configuration
```

Pylint first lists some styling issues - it would like to see an extra newline at the end, modules and function definitions should have descriptive docstrings, and single characters are a poor choice for variable names.

More importantly, however, pylint identified an error in the program - a variable called before assignment. This needs fixing.

Note that pylint scored our program a negative 12.5 out of 10. Let's try to improve that!

```
In [3]: %%writefile simple1.py
        """
        A very simple script.
        """

        def myfunc():
            """
```

```
        An extremely simple function.
        """
        first = 1
        second = 2
        print(first)
        print(second)

    myfunc()
```

```
Overwriting simple1.py
```

```
In [4]: ! pylint simple1.py
```

```
************* Module simple1
C: 14, 0: Final newline missing (missing-final-newline)

-------------------------------------------------------------------

Your code has been rated at 8.33/10 (previous run: -12.50/10, +20.83)
```

```
No config file found, using default configuration
```

Much better! Our score climbed to 8.33 out of 10. Unfortunately, the final newline has to do with how jupyter writes to a file, and there's not much we can do about that here. Still, pylint helped us troubleshoot some of our problems. But what if the problem was more complex?

```
In [5]: %%writefile simple2.py
        """
        A very simple script.
        """

        def myfunc():
            """
            An extremely simple function.
            """
            first = 1
            second = 2
            print(first)
            print('second')

        myfunc()
```

```
Overwriting simple2.py
```

```
In [6]: ! pylint simple2.py
```

```
************* Module simple2
C: 14, 0: Final newline missing (missing-final-newline)
W: 10, 4: Unused variable 'second' (unused-variable)

-------------------------------------------------------------------

Your code has been rated at 6.67/10 (previous run: 6.67/10, +0.00)



No config file found, using default configuration
```

pylint tells us there's an unused variable in line 10, but it doesn't know that we might get an unexpected output from line 12! For this we need a more robust set of tools. That's where `unittest` comes in.

1.3 `unittest`

`unittest` lets you write your own test programs. The goal is to send a specific set of data to your program, and analyze the returned results against an expected result.

Let's generate a simple script that capitalizes words in a given string. We'll call it **cap.py**.

```
In [7]: %%writefile cap.py
        def cap_text(text):
            return text.capitalize()
```

```
Overwriting cap.py
```

Now we'll write a test script. We can call it whatever we want, but **test_cap.py** seems an obvious choice.

When writing test functions, it's best to go from simple to complex, as each function will be run in order. Here we'll test simple, one-word strings, followed by a test of multiple word strings.

```
In [8]: %%writefile test_cap.py
        import unittest
        import cap

        class TestCap(unittest.TestCase):

            def test_one_word(self):
                text = 'python'
                result = cap.cap_text(text)
                self.assertEqual(result, 'Python')
```

```
            def test_multiple_words(self):
                text = 'monty python'
                result = cap.cap_text(text)
                self.assertEqual(result, 'Monty Python')

        if __name__ == '__main__':
            unittest.main()

Overwriting test_cap.py


In [9]: ! python test_cap.py

F.
======================================================================
FAIL: test_multiple_words (__main__.TestCap)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_cap.py", line 14, in test_multiple_words
    self.assertEqual(result, 'Monty Python')
AssertionError: 'Monty python' != 'Monty Python'
- Monty python
?       ^
+ Monty Python
?       ^


----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

What happened? It turns out that the `.capitalize()` method only capitalizes the first letter of the first word in a string. Doing a little research on string methods, we find that `.title()` might give us what we want.

```
In [10]: %%writefile cap.py
         def cap_text(text):
             return text.title()  # replace .capitalize() with .title()

Overwriting cap.py


In [11]: ! python test_cap.py

..
----------------------------------------------------------------------
```

```
Ran 2 tests in 0.000s

OK
```

Hey, it passed! But have we tested all cases? Let's add another test to **test_cap.py** to see if it handles words with apostrophes, like *don't*.

In a text editor this would be easy, but in Jupyter we have to start from scratch.

```
In [12]: %%writefile test_cap.py
         import unittest
         import cap

         class TestCap(unittest.TestCase):

             def test_one_word(self):
                 text = 'python'
                 result = cap.cap_text(text)
                 self.assertEqual(result, 'Python')

             def test_multiple_words(self):
                 text = 'monty python'
                 result = cap.cap_text(text)
                 self.assertEqual(result, 'Monty Python')

             def test_with_apostrophes(self):
                 text = "monty python's flying circus"
                 result = cap.cap_text(text)
                 self.assertEqual(result, "Monty Python's Flying Circus")

         if __name__ == '__main__':
             unittest.main()

Overwriting test_cap.py


In [13]: ! python test_cap.py

..F
======================================================================
FAIL: test_with_apostrophes (__main__.TestCap)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_cap.py", line 19, in test_with_apostrophes
    self.assertEqual(result, "Monty Python's Flying Circus")
AssertionError: "Monty Python'S Flying Circus" != "Monty Python's Flying Circus"
- Monty Python'S Flying Circus
?              ^
+ Monty Python's Flying Circus
```

```
?                ^


----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (failures=1)
```

Now we have to find a solution that handles apostrophes! There is one (look up `capwords` from the `string` module) but we'll leave that as an exercise for the reader.

Great! Now you should have a basic understanding of unit testing!

01-Milestone Project 2 - Assignment

April 9, 2019

1 Milestone Project 2 - Blackjack Game

In this milestone project you will be creating a Complete BlackJack Card Game in Python.
Here are the requirements:

- You need to create a simple text-based BlackJack game
- The game needs to have one player versus an automated dealer.
- The player can stand or hit.
- The player must be able to pick their betting amount.
- You need to keep track of the player's total money.
- You need to alert the player of wins, losses, or busts, etc...

And most importantly:

- **You must use OOP and classes in some portion of your game. You can not just use functions in your game. Use classes to help you define the Deck and the Player's hand. There are many right ways to do this, so explore it well!**

Feel free to expand this game. Try including multiple players. Try adding in Double-Down and card splits! Remember to you are free to use any resources you want and as always:

2 HAVE FUN!

02-Milestone Project 2 - Walkthrough Steps Workbook

April 9, 2019

1 Milestone Project 2 - Walkthrough Steps Workbook

Below is a set of steps for you to follow to try to create the Blackjack Milestone Project game!

1.1 Game Play

To play a hand of Blackjack the following steps must be followed: 1. Create a deck of 52 cards 2. Shuffle the deck 3. Ask the Player for their bet 4. Make sure that the Player's bet does not exceed their available chips 5. Deal two cards to the Dealer and two cards to the Player 6. Show only one of the Dealer's cards, the other remains hidden 7. Show both of the Player's cards 8. Ask the Player if they wish to Hit, and take another card 9. If the Player's hand doesn't Bust (go over 21), ask if they'd like to Hit again. 10. If a Player Stands, play the Dealer's hand. The dealer will always Hit until the Dealer's value meets or exceeds 17 11. Determine the winner and adjust the Player's chips accordingly 12. Ask the Player if they'd like to play again

1.2 Playing Cards

A standard deck of playing cards has four suits (Hearts, Diamonds, Spades and Clubs) and thirteen ranks (2 through 10, then the face cards Jack, Queen, King and Ace) for a total of 52 cards per deck. Jacks, Queens and Kings all have a rank of 10. Aces have a rank of either 11 or 1 as needed to reach 21 without busting. As a starting point in your program, you may want to assign variables to store a list of suits, ranks, and then use a dictionary to map ranks to values.

1.3 The Game

1.3.1 Imports and Global Variables

** Step 1: Import the random module. This will be used to shuffle the deck prior to dealing. Then, declare variables to store suits, ranks and values. You can develop your own system, or copy ours below. Finally, declare a Boolean value to be used to control while loops. This is a common practice used to control the flow of the game.**

```
suits = ('Hearts', 'Diamonds', 'Spades', 'Clubs')
ranks = ('Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack', 'Queen
values = {'Two':2, 'Three':3, 'Four':4, 'Five':5, 'Six':6, 'Seven':7, 'Eight':8, 'Nine':9, 'Ten
          'Queen':10, 'King':10, 'Ace':11}
```

```
In [ ]: import random

        suits = pass
        ranks = pass
        values = pass

        playing = True
```

1.3.2 Class Definitions

Consider making a Card class where each Card object has a suit and a rank, then a Deck class to hold all 52 Card objects, and can be shuffled, and finally a Hand class that holds those Cards that have been dealt to each player from the Deck.

Step 2: Create a Card Class A Card object really only needs two attributes: suit and rank. You might add an attribute for "value" - we chose to handle value later when developing our Hand class.In addition to the Card's __init__ method, consider adding a __str__ method that, when asked to print a Card, returns a string in the form "Two of Hearts"

```
In [ ]: class Card:

            def __init__(self):
                pass

            def __str__(self):
                pass
```

Step 3: Create a Deck Class Here we might store 52 card objects in a list that can later be shuffled. First, though, we need to *instantiate* all 52 unique card objects and add them to our list. So long as the Card class definition appears in our code, we can build Card objects inside our Deck __init__ method. Consider iterating over sequences of suits and ranks to build out each card. This might appear inside a Deck class __init__ method:

```
for suit in suits:
    for rank in ranks:
```

In addition to an __init__ method we'll want to add methods to shuffle our deck, and to deal out cards during gameplay. OPTIONAL: We may never need to print the contents of the deck during gameplay, but having the ability to see the cards inside it may help troubleshoot any problems that occur during development. With this in mind, consider adding a __str__ method to the class definition.

```
In [ ]: class Deck:

            def __init__(self):
                self.deck = []  # start with an empty list
                for suit in suits:
                    for rank in ranks:
                        pass
```

```
            def __str__(self):
                pass

            def shuffle(self):
                random.shuffle(self.deck)

            def deal(self):
                pass
```

TESTING: Just to see that everything works so far, let's see what our Deck looks like!

```
In [ ]: test_deck = Deck()
        print(test_deck)
```

Great! Now let's move on to our Hand class.

Step 4: Create a Hand Class In addition to holding Card objects dealt from the Deck, the Hand class may be used to calculate the value of those cards using the values dictionary defined above. It may also need to adjust for the value of Aces when appropriate.

```
In [ ]: class Hand:
            def __init__(self):
                self.cards = []  # start with an empty list as we did in the Deck class
                self.value = 0   # start with zero value
                self.aces = 0    # add an attribute to keep track of aces

            def add_card(self,card):
                pass

            def adjust_for_ace(self):
                pass
```

Step 5: Create a Chips Class In addition to decks of cards and hands, we need to keep track of a Player's starting chips, bets, and ongoing winnings. This could be done using global variables, but in the spirit of object oriented programming, let's make a Chips class instead!

```
In [ ]: class Chips:

            def __init__(self):
                self.total = 100  # This can be set to a default value or supplied by a user i
                self.bet = 0

            def win_bet(self):
                pass

            def lose_bet(self):
                pass
```

1.3.3 Function Defintions

A lot of steps are going to be repetitive. That's where functions come in! The following steps are guidelines - add or remove functions as needed in your own program.

Step 6: Write a function for taking bets Since we're asking the user for an integer value, this would be a good place to use try/except. Remember to check that a Player's bet can be covered by their available chips.

```
In [ ]: def take_bet():

            pass
```

Step 7: Write a function for taking hits Either player can take hits until they bust. This function will be called during gameplay anytime a Player requests a hit, or a Dealer's hand is less than 17. It should take in Deck and Hand objects as arguments, and deal one card off the deck and add it to the Hand. You may want it to check for aces in the event that a player's hand exceeds 21.

```
In [ ]: def hit(deck,hand):

            pass
```

Step 8: Write a function prompting the Player to Hit or Stand This function should accept the deck and the player's hand as arguments, and assign playing as a global variable. If the Player Hits, employ the hit() function above. If the Player Stands, set the playing variable to False - this will control the behavior of a while loop later on in our code.

```
In [ ]: def hit_or_stand(deck,hand):
            global playing  # to control an upcoming while loop

            pass
```

Step 9: Write functions to display cards When the game starts, and after each time Player takes a card, the dealer's first card is hidden and all of Player's cards are visible. At the end of the hand all cards are shown, and you may want to show each hand's total value. Write a function for each of these scenarios.

```
In [ ]: def show_some(player,dealer):

            pass

        def show_all(player,dealer):

            pass
```

Step 10: Write functions to handle end of game scenarios Remember to pass player's hand, dealer's hand and chips as needed.

```
In [ ]: def player_busts():
            pass
```

```
        def player_wins():
            pass

        def dealer_busts():
            pass

        def dealer_wins():
            pass

        def push():
            pass
```

1.3.4 And now on to the game!!

```
In [ ]: while True:
            # Print an opening statement


            # Create & shuffle the deck, deal two cards to each player



            # Set up the Player's chips


            # Prompt the Player for their bet


            # Show cards (but keep one dealer card hidden)


            while playing:  # recall this variable from our hit_or_stand function

                # Prompt for Player to Hit or Stand


                # Show cards (but keep one dealer card hidden)


                # If player's hand exceeds 21, run player_busts() and break out of loop


                    break

            # If Player hasn't busted, play Dealer's hand until Dealer reaches 17


                # Show all cards
```

```
            # Run different winning scenarios


        # Inform Player of their chips total

        # Ask to play again

            break
```

And that's it! Remember, these steps may differ significantly from your own solution. That's OK! Keep working on different sections of your program until you get the desired results. It takes a lot of time and patience! As always, feel free to post questions and comments to the QA Forums. # Good job!

03-Milestone Project 2 - Complete Walkthrough Solution

April 9, 2019

1 Milestone Project 2 - Complete Walkthrough Solution

This notebook walks through a proposed solution to the Blackjack Game milestone project. The approach to solving and the specific code used are only suggestions - there are many different ways to code this out, and yours is likely to be different!

1.1 Game Play

To play a hand of Blackjack the following steps must be followed: 1. Create a deck of 52 cards 2. Shuffle the deck 3. Ask the Player for their bet 4. Make sure that the Player's bet does not exceed their available chips 5. Deal two cards to the Dealer and two cards to the Player 6. Show only one of the Dealer's cards, the other remains hidden 7. Show both of the Player's cards 8. Ask the Player if they wish to Hit, and take another card 9. If the Player's hand doesn't Bust (go over 21), ask if they'd like to Hit again. 10. If a Player Stands, play the Dealer's hand. The dealer will always Hit until the Dealer's value meets or exceeds 17 11. Determine the winner and adjust the Player's chips accordingly 12. Ask the Player if they'd like to play again

1.2 Playing Cards

A standard deck of playing cards has four suits (Hearts, Diamonds, Spades and Clubs) and thirteen ranks (2 through 10, then the face cards Jack, Queen, King and Ace) for a total of 52 cards per deck. Jacks, Queens and Kings all have a rank of 10. Aces have a rank of either 11 or 1 as needed to reach 21 without busting. As a starting point in your program, you may want to assign variables to store a list of suits, ranks, and then use a dictionary to map ranks to values.

1.3 The Game

1.3.1 Imports and Global Variables

** Step 1: Import the random module. This will be used to shuffle the deck prior to dealing. Then, declare variables to store suits, ranks and values. You can develop your own system, or copy ours below. Finally, declare a Boolean value to be used to control while loops. This is a common practice used to control the flow of the game.**

```
suits = ('Hearts', 'Diamonds', 'Spades', 'Clubs')
ranks = ('Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack', 'Queen
values = {'Two':2, 'Three':3, 'Four':4, 'Five':5, 'Six':6, 'Seven':7, 'Eight':8, 'Nine':9, 'Ten
          'Queen':10, 'King':10, 'Ace':11}
```

```
In [1]: import random

        suits = ('Hearts', 'Diamonds', 'Spades', 'Clubs')
        ranks = ('Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack
        values = {'Two':2, 'Three':3, 'Four':4, 'Five':5, 'Six':6, 'Seven':7, 'Eight':8, 'Nine
                  'Queen':10, 'King':10, 'Ace':11}

        playing = True
```

1.3.2 Class Definitions

Consider making a Card class where each Card object has a suit and a rank, then a Deck class to hold all 52 Card objects, and can be shuffled, and finally a Hand class that holds those Cards that have been dealt to each player from the Deck.

Step 2: Create a Card Class A Card object really only needs two attributes: suit and rank. You might add an attribute for "value" - we chose to handle value later when developing our Hand class.In addition to the Card's __init__ method, consider adding a __str__ method that, when asked to print a Card, returns a string in the form "Two of Hearts"

```
In [2]: class Card:

            def __init__(self,suit,rank):
                self.suit = suit
                self.rank = rank

            def __str__(self):
                return self.rank + ' of ' + self.suit
```

Step 3: Create a Deck Class Here we might store 52 card objects in a list that can later be shuffled. First, though, we need to *instantiate* all 52 unique card objects and add them to our list. So long as the Card class definition appears in our code, we can build Card objects inside our Deck __init__ method. Consider iterating over sequences of suits and ranks to build out each card. This might appear inside a Deck class __init__ method:

```
for suit in suits:
    for rank in ranks:
```

In addition to an __init__ method we'll want to add methods to shuffle our deck, and to deal out cards during gameplay. OPTIONAL: We may never need to print the contents of the deck during gameplay, but having the ability to see the cards inside it may help troubleshoot any problems that occur during development. With this in mind, consider adding a __str__ method to the class definition.

```
In [3]: class Deck:

            def __init__(self):
                self.deck = []  # start with an empty list
                for suit in suits:
                    for rank in ranks:
```

```
                    self.deck.append(Card(suit,rank))  # build Card objects and add them t

        def __str__(self):
            deck_comp = ''  # start with an empty string
            for card in self.deck:
                deck_comp += '\n '+card.__str__() # add each Card object's print string
            return 'The deck has:' + deck_comp

        def shuffle(self):
            random.shuffle(self.deck)

        def deal(self):
            single_card = self.deck.pop()
            return single_card
```

TESTING: Just to see that everything works so far, let's see what our Deck looks like!

```
In [4]: test_deck = Deck()
        print(test_deck)
```

```
The deck has:
 Two of Hearts
 Three of Hearts
 Four of Hearts
 Five of Hearts
 Six of Hearts
 Seven of Hearts
 Eight of Hearts
 Nine of Hearts
 Ten of Hearts
 Jack of Hearts
 Queen of Hearts
 King of Hearts
 Ace of Hearts
 Two of Diamonds
 Three of Diamonds
 Four of Diamonds
 Five of Diamonds
 Six of Diamonds
 Seven of Diamonds
 Eight of Diamonds
 Nine of Diamonds
 Ten of Diamonds
 Jack of Diamonds
 Queen of Diamonds
 King of Diamonds
 Ace of Diamonds
 Two of Spades
```

```
Three of Spades
Four of Spades
Five of Spades
Six of Spades
Seven of Spades
Eight of Spades
Nine of Spades
Ten of Spades
Jack of Spades
Queen of Spades
King of Spades
Ace of Spades
Two of Clubs
Three of Clubs
Four of Clubs
Five of Clubs
Six of Clubs
Seven of Clubs
Eight of Clubs
Nine of Clubs
Ten of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Clubs
```

Great! Now let's move on to our Hand class.

Step 4: Create a Hand Class In addition to holding Card objects dealt from the Deck, the Hand class may be used to calculate the value of those cards using the values dictionary defined above. It may also need to adjust for the value of Aces when appropriate.

```
In [5]: class Hand:
            def __init__(self):
                self.cards = []  # start with an empty list as we did in the Deck class
                self.value = 0   # start with zero value
                self.aces = 0    # add an attribute to keep track of aces

            def add_card(self,card):
                self.cards.append(card)
                self.value += values[card.rank]

            def adjust_for_ace(self):
                pass
```

TESTING: Before we tackle the issue of changing Aces, let's make sure we can add two cards to a player's hand and obtain their value:

```
In [6]: test_deck = Deck()
        test_deck.shuffle()
```

```
test_player = Hand()
test_player.add_card(test_deck.deal())
test_player.add_card(test_deck.deal())
test_player.value
```

```
Out[6]: 14
```

Let's see what these two cards are:

```
In [7]: for card in test_player.cards:
            print(card)
```

```
Nine of Hearts
Five of Hearts
```

Great! Now let's tackle the Aces issue. If a hand's value exceeds 21 but it contains an Ace, we can reduce the Ace's value from 11 to 1 and continue playing.

```
In [8]: class Hand:

            def __init__(self):
                self.cards = []  # start with an empty list as we did in the Deck class
                self.value = 0   # start with zero value
                self.aces = 0    # add an attribute to keep track of aces

            def add_card(self,card):
                self.cards.append(card)
                self.value += values[card.rank]
                if card.rank == 'Ace':
                    self.aces += 1  # add to self.aces

            def adjust_for_ace(self):
                while self.value > 21 and self.aces:
                    self.value -= 10
                    self.aces -= 1
```

We added code to the add_card method to bump self.aces whenever an ace is brought into the hand, and added code to the adjust_for_aces method that decreases the number of aces any time we make an adjustment to stay under 21.

Step 5: Create a Chips Class In addition to decks of cards and hands, we need to keep track of a Player's starting chips, bets, and ongoing winnings. This could be done using global variables, but in the spirit of object oriented programming, let's make a Chips class instead!

```
In [9]: class Chips:

            def __init__(self):
                self.total = 100  # This can be set to a default value or supplied by a user i
                self.bet = 0
```

```
    def win_bet(self):
        self.total += self.bet

    def lose_bet(self):
        self.total -= self.bet
```

A NOTE ABOUT OUR DEFAULT TOTAL VALUE: Alternatively, we could have passed a default total value as an parameter in the __init__. This would have let us pass in an override value at the time the object was created rather than wait until later to change it. The code would have looked like this:

```
def __init__(self,total=100):
    self.total = total
    self.bet = 0
```

Either technique is fine, it only depends on how you plan to start your game parameters.

1.3.3 Function Defintions

A lot of steps are going to be repetitive. That's where functions come in! The following steps are guidelines - add or remove functions as needed in your own program.

Step 6: Write a function for taking bets Since we're asking the user for an integer value, this would be a good place to use try/except. Remember to check that a Player's bet can be covered by their available chips.

```
In [10]: def take_bet(chips):

             while True:
                 try:
                     chips.bet = int(input('How many chips would you like to bet? '))
                 except ValueError:
                     print('Sorry, a bet must be an integer!')
                 else:
                     if chips.bet > chips.total:
                         print("Sorry, your bet can't exceed",chips.total)
                     else:
                         break
```

We used a while loop here to continually prompt the user for input until we received an integer value that was within the Player's betting limit.

A QUICK NOTE ABOUT FUNCTIONS: If we knew in advance what we were going to call our Player's Chips object, we could have written the above function like this:

```
def take_bet():
    while True:
        try:
            player_chips.bet = int(input('How many chips would you like to bet? '))
        except ValueError:
```

```
            print('Sorry, a bet must be an integer!')
        else:
            if player_chips.bet > player_chips.total:
                print("Sorry, your bet can't exceed",player_chips.total)
            else:
                break
```

and then we could call the function without passing any arguments. This is generally not a good idea! It's better to have functions be self-contained, able to accept any incoming value than depend on some future naming convention. Also, this makes it easier to add players in future versions of our program!

Step 7: Write a function for taking hits Either player can take hits until they bust. This function will be called during gameplay anytime a Player requests a hit, or a Dealer's hand is less than 17. It should take in Deck and Hand objects as arguments, and deal one card off the deck and add it to the Hand. You may want it to check for aces in the event that a player's hand exceeds 21.

```
In [11]: def hit(deck,hand):

             hand.add_card(deck.deal())
             hand.adjust_for_ace()
```

Step 8: Write a function prompting the Player to Hit or Stand This function should accept the deck and the player's hand as arguments, and assign playing as a global variable. If the Player Hits, employ the hit() function above. If the Player Stands, set the playing variable to False - this will control the behavior of a while loop later on in our code.

```
In [12]: def hit_or_stand(deck,hand):
             global playing  # to control an upcoming while loop

             while True:
                 x = input("Would you like to Hit or Stand? Enter 'h' or 's' ")

                 if x[0].lower() == 'h':
                     hit(deck,hand)  # hit() function defined above

                 elif x[0].lower() == 's':
                     print("Player stands. Dealer is playing.")
                     playing = False

                 else:
                     print("Sorry, please try again.")
                     continue
                 break
```

Step 9: Write functions to display cards When the game starts, and after each time Player takes a card, the dealer's first card is hidden and all of Player's cards are visible. At the end of the hand all cards are shown, and you may want to show each hand's total value. Write a function for each of these scenarios.

04-Milestone Project 2 - Solution Code

April 9, 2019

1 Milestone Project 2 - Solution Code

Below is an implementation of a simple game of Blackjack. Notice the use of OOP and classes for the cards and hands.

In [3]:
```
# IMPORT STATEMENTS AND VARIABLE DECLARATIONS:

import random

suits = ('Hearts', 'Diamonds', 'Spades', 'Clubs')
ranks = ('Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack
values = {'Two':2, 'Three':3, 'Four':4, 'Five':5, 'Six':6, 'Seven':7, 'Eight':8,
            'Nine':9, 'Ten':10, 'Jack':10, 'Queen':10, 'King':10, 'Ace':11}

playing = True

# CLASS DEFINTIONS:

class Card:

    def __init__(self,suit,rank):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        return self.rank + ' of ' + self.suit


class Deck:

    def __init__(self):
        self.deck = []  # start with an empty list
        for suit in suits:
            for rank in ranks:
                self.deck.append(Card(suit,rank))

    def __str__(self):
```

```
        deck_comp = ''  # start with an empty string
        for card in self.deck:
            deck_comp += '\n '+card.__str__() # add each Card object's print string
        return 'The deck has:' + deck_comp

    def shuffle(self):
        random.shuffle(self.deck)

    def deal(self):
        single_card = self.deck.pop()
        return single_card


class Hand:

    def __init__(self):
        self.cards = []  # start with an empty list as we did in the Deck class
        self.value = 0   # start with zero value
        self.aces = 0    # add an attribute to keep track of aces

    def add_card(self,card):
        self.cards.append(card)
        self.value += values[card.rank]
        if card.rank == 'Ace':
            self.aces += 1  # add to self.aces

    def adjust_for_ace(self):
        while self.value > 21 and self.aces:
            self.value -= 10
            self.aces -= 1


class Chips:

    def __init__(self):
        self.total = 100
        self.bet = 0

    def win_bet(self):
        self.total += self.bet

    def lose_bet(self):
        self.total -= self.bet


# FUNCTION DEFINITIONS:

def take_bet(chips):
```

```
    while True:
        try:
            chips.bet = int(input('How many chips would you like to bet? '))
        except ValueError:
            print('Sorry, a bet must be an integer!')
        else:
            if chips.bet > chips.total:
                print("Sorry, your bet can't exceed",chips.total)
            else:
                break

def hit(deck,hand):
    hand.add_card(deck.deal())
    hand.adjust_for_ace()

def hit_or_stand(deck,hand):
    global playing

    while True:
        x = input("Would you like to Hit or Stand? Enter 'h' or 's' ")

        if x[0].lower() == 'h':
            hit(deck,hand)  # hit() function defined above

        elif x[0].lower() == 's':
            print("Player stands. Dealer is playing.")
            playing = False

        else:
            print("Sorry, please try again.")
            continue
        break


def show_some(player,dealer):
    print("\nDealer's Hand:")
    print(" <card hidden>")
    print('',dealer.cards[1])
    print("\nPlayer's Hand:", *player.cards, sep='\n ')

def show_all(player,dealer):
    print("\nDealer's Hand:", *dealer.cards, sep='\n ')
    print("Dealer's Hand =",dealer.value)
    print("\nPlayer's Hand:", *player.cards, sep='\n ')
    print("Player's Hand =",player.value)

def player_busts(player,dealer,chips):
```

```
    print("Player busts!")
    chips.lose_bet()

def player_wins(player,dealer,chips):
    print("Player wins!")
    chips.win_bet()

def dealer_busts(player,dealer,chips):
    print("Dealer busts!")
    chips.win_bet()

def dealer_wins(player,dealer,chips):
    print("Dealer wins!")
    chips.lose_bet()

def push(player,dealer):
    print("Dealer and Player tie! It's a push.")

# GAMEPLAY!

while True:
    print('Welcome to BlackJack! Get as close to 21 as you can without going over!\n\
    Dealer hits until she reaches 17. Aces count as 1 or 11.')

    # Create & shuffle the deck, deal two cards to each player
    deck = Deck()
    deck.shuffle()

    player_hand = Hand()
    player_hand.add_card(deck.deal())
    player_hand.add_card(deck.deal())

    dealer_hand = Hand()
    dealer_hand.add_card(deck.deal())
    dealer_hand.add_card(deck.deal())

    # Set up the Player's chips
    player_chips = Chips()  # remember the default value is 100

    # Prompt the Player for their bet:
    take_bet(player_chips)

    # Show the cards:
    show_some(player_hand,dealer_hand)

    while playing:  # recall this variable from our hit_or_stand function

        # Prompt for Player to Hit or Stand
```

```
            hit_or_stand(deck,player_hand)
            show_some(player_hand,dealer_hand)

            if player_hand.value > 21:
                player_busts(player_hand,dealer_hand,player_chips)
                break

        # If Player hasn't busted, play Dealer's hand
        if player_hand.value <= 21:

            while dealer_hand.value < 17:
                hit(deck,dealer_hand)

            # Show all cards
            show_all(player_hand,dealer_hand)

            # Test different winning scenarios
            if dealer_hand.value > 21:
                dealer_busts(player_hand,dealer_hand,player_chips)

            elif dealer_hand.value > player_hand.value:
                dealer_wins(player_hand,dealer_hand,player_chips)

            elif dealer_hand.value < player_hand.value:
                player_wins(player_hand,dealer_hand,player_chips)

            else:
                push(player_hand,dealer_hand)

        # Inform Player of their chips total
        print("\nPlayer's winnings stand at",player_chips.total)

        # Ask to play again
        new_game = input("Would you like to play another hand? Enter 'y' or 'n' ")
        if new_game[0].lower()=='y':
            playing=True
            continue
        else:
            print("Thanks for playing!")
            break
```

```
Welcome to BlackJack! Get as close to 21 as you can without going over!
    Dealer hits until she reaches 17. Aces count as 1 or 11.
How many chips would you like to bet? 50

Dealer's Hand:
 <card hidden>
 Seven of Diamonds
```

```
Player's Hand:
 Jack of Clubs
 Three of Diamonds
Would you like to Hit or Stand? Enter 'h' or 's' h

Dealer's Hand:
 <card hidden>
 Seven of Diamonds

Player's Hand:
 Jack of Clubs
 Three of Diamonds
 Six of Hearts
Would you like to Hit or Stand? Enter 'h' or 's' s
Player stands. Dealer is playing.

Dealer's Hand:
 <card hidden>
 Seven of Diamonds

Player's Hand:
 Jack of Clubs
 Three of Diamonds
 Six of Hearts

Dealer's Hand:
 Ace of Hearts
 Seven of Diamonds
Dealer's Hand = 18

Player's Hand:
 Jack of Clubs
 Three of Diamonds
 Six of Hearts
Player's Hand = 19
Player wins!

Player's winnings stand at 150
Would you like to play another hand? Enter 'y' or 'n' n
Thanks for playing!
```

In [ ]:

01-Map

April 9, 2019

1 map()

map() is a built-in Python function that takes in two or more arguments: a function and one or more iterables, in the form:

```
map(function, iterable, ...)
```

map() returns an *iterator* - that is, map() returns a special object that yields one result at a time as needed. We will learn more about iterators and generators in a future lecture. For now, since our examples are so small, we will cast map() as a list to see the results immediately.

When we went over list comprehensions we created a small expression to convert Celsius to Fahrenheit. Let's do the same here but use map:

```
In [1]: def fahrenheit(celsius):
            return (9/5)*celsius + 32

        temps = [0, 22.5, 40, 100]
```

Now let's see map() in action:

```
In [2]: F_temps = map(fahrenheit, temps)

        #Show
        list(F_temps)

Out[2]: [32.0, 72.5, 104.0, 212.0]
```

In the example above, map() applies the fahrenheit function to every item in temps. However, we don't have to define our functions beforehand; we can use a lambda expression instead:

```
In [3]: list(map(lambda x: (9/5)*x + 32, temps))

Out[3]: [32.0, 72.5, 104.0, 212.0]
```

Great! We got the same result! Using map with lambda expressions is much more common since the entire purpose of map() is to save effort on having to create manual for loops.

1.0.1 map() with multiple iterables

map() can accept more than one iterable. The iterables should be the same length - in the event that they are not, map() will stop as soon as the shortest iterable is exhausted.

For instance, if our function is trying to add two values **x** and **y**, we can pass a list of **x** values and another list of **y** values to map(). The function (or lambda) will be fed the 0th index from each list, and then the 1st index, and so on until the n-th index is reached.

Let's see this in action with two and then three lists:

```
In [4]: a = [1,2,3,4]
        b = [5,6,7,8]
        c = [9,10,11,12]

        list(map(lambda x,y:x+y,a,b))

Out[4]: [6, 8, 10, 12]

In [5]: # Now all three lists
        list(map(lambda x,y,z:x+y+z,a,b,c))

Out[5]: [15, 18, 21, 24]
```

We can see in the example above that the parameter **x** gets its values from the list **a**, while **y** gets its values from **b** and **z** from list **c**. Go ahead and play with your own example to make sure you fully understand mapping to more than one iterable.

Great job! You should now have a basic understanding of the map() function.

02-Reduce

April 9, 2019

1 reduce()

Many times students have difficulty understanding reduce() so pay careful attention to this lecture. The function reduce(function, sequence) continually applies the function to the sequence. It then returns a single value.

If seq = [s1, s2, s3, ... , sn], calling reduce(function, sequence) works like this:

- At first the first two elements of seq will be applied to function, i.e. func(s1,s2)
- The list on which reduce() works looks now like this: [function(s1, s2), s3, ... , sn]
- In the next step the function will be applied on the previous result and the third element of the list, i.e. function(function(s1, s2),s3)
- The list looks like this now: [function(function(s1, s2),s3), ... , sn]
- It continues like this until just one element is left and return this element as the result of reduce()

Let's see an example:

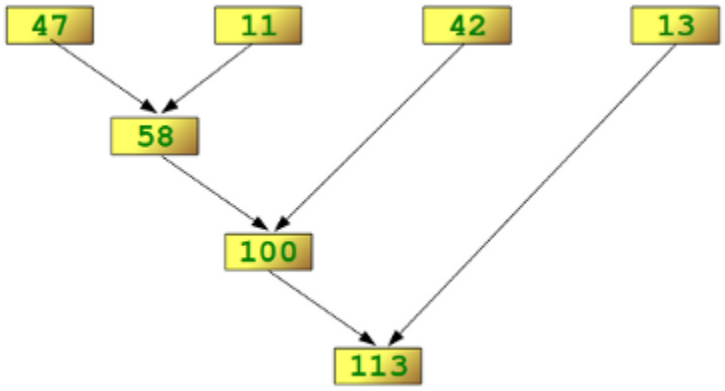
```
In [1]: from functools import reduce

        lst =[47,11,42,13]
        reduce(lambda x,y: x+y,lst)
```

```
Out[1]: 113
```

Lets look at a diagram to get a better understanding of what is going on here:

```
In [2]: from IPython.display import Image
        Image('http://www.python-course.eu/images/reduce_diagram.png')
```

```
Out[2]:
```



Note how we keep reducing the sequence until a single final value is obtained. Lets see another example:

```
In [3]: #Find the maximum of a sequence (This already exists as max())
        max_find = lambda a,b: a if (a > b) else b

In [4]: #Find max
        reduce(max_find,lst)

Out[4]: 47
```

Hopefully you can see how useful reduce can be in various situations. Keep it in mind as you think about your code projects!

03-Filter

April 9, 2019

1 filter

The function filter(function, list) offers a convenient way to filter out all the elements of an iterable, for which the function returns True.

The function filter(function,list) needs a function as its first argument. The function needs to return a Boolean value (either True or False). This function will be applied to every element of the iterable. Only if the function returns True will the element of the iterable be included in the result.

Like map(), filter() returns an *iterator* - that is, filter yields one result at a time as needed. Iterators and generators will be covered in an upcoming lecture. For now, since our examples are so small, we will cast filter() as a list to see our results immediately.

Let's see some examples:

```
In [1]: #First let's make a function
        def even_check(num):
            if num%2 ==0:
                return True
```

Now let's filter a list of numbers. Note: putting the function into filter without any parentheses might feel strange, but keep in mind that functions are objects as well.

```
In [2]: lst =range(20)

        list(filter(even_check,lst))
```

```
Out[2]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

filter() is more commonly used with lambda functions, because we usually use filter for a quick job where we don't want to write an entire function. Let's repeat the example above using a lambda expression:

```
In [3]: list(filter(lambda x: x%2==0,lst))
```

```
Out[3]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Great! You should now have a solid understanding of filter() and how to apply it to your code!

04-Zip

April 9, 2019

1 zip

zip() makes an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

zip() is equivalent to:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

zip() should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables.

Let's see it in action in some examples:

1.1 Examples

```
In [1]: x = [1,2,3]
        y = [4,5,6]

        # Zip the lists together
        list(zip(x,y))
```

```
Out[1]: [(1, 4), (2, 5), (3, 6)]
```

Note how tuples are returned. What if one iterable is longer than the other?

```
In [2]: x = [1,2,3]
        y = [4,5,6,7,8]

        # Zip the lists together
        list(zip(x,y))

Out[2]: [(1, 4), (2, 5), (3, 6)]
```

Note how the zip is defined by the shortest iterable length. Its generally advised not to zip unequal length iterables unless your very sure you only need partial tuple pairings.

What happens if we try to zip together dictionaries?

```
In [3]: d1 = {'a':1,'b':2}
        d2 = {'c':4,'d':5}

        list(zip(d1,d2))

Out[3]: [('a', 'c'), ('b', 'd')]
```

This makes sense because simply iterating through the dictionaries will result in just the keys. We would have to call methods to mix keys and values:

```
In [4]: list(zip(d2,d1.values()))

Out[4]: [('c', 1), ('d', 2)]
```

Great! Finally lets use zip() to switch the keys and values of the two dictionaries:

```
In [5]: def switcharoo(d1,d2):
            dout = {}

            for d1key,d2val in zip(d1,d2.values()):
                dout[d1key] = d2val

            return dout

In [6]: switcharoo(d1,d2)

Out[6]: {'a': 4, 'b': 5}
```

Great! You can use zip to save a lot of typing in many situations! You should now have a good understanding of zip() and some possible use cases.

05-Enumerate

April 9, 2019

1 enumerate()

In this lecture we will learn about an extremely useful built-in function: enumerate(). Enumerate allows you to keep a count as you iterate through an object. It does this by returning a tuple in the form (count,element). The function itself is equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

1.1 Example

```
In [1]: lst = ['a','b','c']

        for number,item in enumerate(lst):
            print(number)
            print(item)
```

```
0
a
1
b
2
c
```

enumerate() becomes particularly useful when you have a case where you need to have some sort of tracker. For example:

```
In [2]: for count,item in enumerate(lst):
            if count >= 2:
                break
            else:
                print(item)
```

```
a
b
```

enumerate() takes an optional "start" argument to override the default value of zero:

```
In [3]: months = ['March','April','May','June']

        list(enumerate(months,start=3))

Out[3]: [(3, 'March'), (4, 'April'), (5, 'May'), (6, 'June')]
```

Great! You should now have a good understanding of enumerate and its potential use cases.

06-all() and any()

April 9, 2019

1 all() and any()

all() and any() are built-in functions in Python that allow us to conveniently check for boolean matching in an iterable. all() will return True if all elements in an iterable are True. It is the same as this function code:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any() will return True if any of the elements in the iterable are True. It is equivalent to the following function code:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

Let's see a few examples of these functions. They should be fairly straightforward:

```
In [1]: lst = [True,True,False,True]
```

```
In [2]: all(lst)
```

```
Out[2]: False
```

Returns False because not all elements are True.

```
In [3]: any(lst)
```

```
Out[3]: True
```

Returns True because at least one of the elements in the list is True

There you have it, you should have an understanding of how to use any() and all() in your code.

07-Complex

April 9, 2019

1 complex()

complex() returns a complex number with the value real + imag*1j or converts a string or number to a complex number.

If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If imag is omitted, it defaults to zero and the constructor serves as a numeric conversion like int and float. If both arguments are omitted, returns 0j.

If you are doing math or engineering that requires complex numbers (such as dynamics, control systems, or impedance of a circuit) this is a useful tool to have in Python.

Let's see some examples:

```
In [1]: # Create 2+3j
        complex(2,3)

Out[1]: (2+3j)

In [2]: complex(10,1)

Out[2]: (10+1j)
```

We can also pass strings:

```
In [3]: complex('12+2j')

Out[3]: (12+2j)
```

That's really all there is to this useful function. Keep it in mind if you are ever dealing with complex numbers in Python!

08-Built-in Functions Assessment Test

April 9, 2019

1 Built-in Functions Test

1.0.1 For this test, you should use built-in functions and be able to write the requested functions in one line.

1.0.2 Problem 1

Use map() to create a function which finds the length of each word in the phrase (broken by spaces) and returns the values in a list.

The function will have an input of a string, and output a list of integers.

```
In [1]: def word_lengths(phrase):

            pass
```

```
In [2]: word_lengths('How long are the words in this phrase')
```

```
Out[2]: [3, 4, 3, 3, 5, 2, 4, 6]
```

1.0.3 Problem 2

Use reduce() to take a list of digits and return the number that they correspond to. For example, [1, 2, 3] corresponds to one-hundred-twenty-three. *Do not convert the integers to strings!*

```
In [3]: from functools import reduce

        def digits_to_num(digits):

            pass
```

```
In [4]: digits_to_num([3,4,3,2,1])
```

```
Out[4]: 34321
```

1.0.4 Problem 3

Use filter to return the words from a list of words which start with a target letter.

```
In [5]: def filter_words(word_list, letter):

            pass
```

```
In [6]: l = ['hello','are','cat','dog','ham','hi','go','to','heart']
        filter_words(l,'h')
```

```
Out[6]: ['hello', 'ham', 'hi', 'heart']
```

1.0.5 Problem 4

Use zip() and a list comprehension to return a list of the same length where each value is the two strings from L1 and L2 concatenated together with connector between them. Look at the example output below:

```
In [7]: def concatenate(L1, L2, connector):

            pass
```

```
In [8]: concatenate(['A','B'],['a','b'],'-')
```

```
Out[8]: ['A-a', 'B-b']
```

1.0.6 Problem 5

Use enumerate() and other skills to return a dictionary which has the values of the list as keys and the index as the value. You may assume that a value will only appear once in the given list.

```
In [9]: def d_list(L):

            pass
```

```
In [10]: d_list(['a','b','c'])
```

```
Out[10]: {'a': 0, 'b': 1, 'c': 2}
```

1.0.7 Problem 6

Use enumerate() and other skills from above to return the count of the number of items in the list whose value equals its index.

```
In [11]: def count_match_index(L):

             pass
```

```
In [12]: count_match_index([0,2,2,1,5,5,6,10])
```

```
Out[12]: 4
```

2 Great Job!

09-Built-in Functions Assessment Test - Solution

April 9, 2019

1 Built-in Functions Test Solutions

1.0.1 For this test, you should use built-in functions and be able to write the requested functions in one line.

1.0.2 Problem 1

Use map() to create a function which finds the length of each word in the phrase (broken by spaces) and return the values in a list.

The function will have an input of a string, and output a list of integers.

```
In [1]: def word_lengths(phrase):

            return list(map(len, phrase.split()))
```

```
In [2]: word_lengths('How long are the words in this phrase')
```

```
Out[2]: [3, 4, 3, 3, 5, 2, 4, 6]
```

1.0.3 Problem 2

Use reduce() to take a list of digits and return the number that they correspond to. For example, [1,2,3] corresponds to one-hundred-twenty-three. *Do not convert the integers to strings!*

```
In [3]: from functools import reduce

        def digits_to_num(digits):

            return reduce(lambda x,y:x*10 + y,digits)
```

```
In [4]: digits_to_num([3,4,3,2,1])
```

```
Out[4]: 34321
```

1.0.4 Problem 3

Use filter() to return the words from a list of words which start with a target letter.

```
In [5]: def filter_words(word_list, letter):

            return list(filter(lambda word:word[0]==letter,word_list))
```

```
In [6]: words = ['hello','are','cat','dog','ham','hi','go','to','heart']
        filter_words(words,'h')

Out[6]: ['hello', 'ham', 'hi', 'heart']
```

1.0.5 Problem 4

Use zip() and a list comprehension to return a list of the same length where each value is the two strings from L1 and L2 concatenated together with a connector between them. Look at the example output below:

```
In [7]: def concatenate(L1, L2, connector):

            return [word1+connector+word2 for (word1,word2) in zip(L1,L2)]

In [8]: concatenate(['A','B'],['a','b'],'-')

Out[8]: ['A-a', 'B-b']
```

1.0.6 Problem 5

Use enumerate() and other skills to return a dictionary which has the values of the list as keys and the index as the value. You may assume that a value will only appear once in the given list.

```
In [9]: def d_list(L):

            return {key:value for value,key in enumerate(L)}

In [10]: d_list(['a','b','c'])

Out[10]: {'a': 0, 'b': 1, 'c': 2}
```

1.0.7 Problem 6

Use enumerate() and other skills from above to return the count of the number of items in the list whose value equals its index.

```
In [11]: def count_match_index(L):

             return len([num for count,num in enumerate(L) if num==count])

In [12]: count_match_index([0,2,2,1,5,5,6,10])

Out[12]: 4
```

2 Great Job!

01-Decorators

April 9, 2019

1 Decorators

Decorators can be thought of as functions which modify the *functionality* of another function. They help to make your code shorter and more "Pythonic".

To properly explain decorators we will slowly build up from functions. Make sure to run every cell in this Notebook for this lecture to look the same on your own computer.So let's break down the steps:

1.1 Functions Review

```
In [1]: def func():
            return 1

In [2]: func()

Out[2]: 1
```

1.2 Scope Review

Remember from the nested statements lecture that Python uses Scope to know what a label is referring to. For example:

```
In [ ]: s = 'Global Variable'

        def check_for_locals():
            print(locals())
```

Remember that Python functions create a new scope, meaning the function has its own namespace to find variable names when they are mentioned within the function. We can check for local variables and global variables with the locals() and globals() functions. For example:

```
In [ ]: print(globals())
```

Here we get back a dictionary of all the global variables, many of them are predefined in Python. So let's go ahead and look at the keys:

```
In [ ]: print(globals().keys())
```

Note how **s** is there, the Global Variable we defined as a string:

```
In [ ]: globals()['s']
```

Now let's run our function to check for local variables that might exist inside our function (there shouldn't be any)

```
In [ ]: check_for_locals()
```

Great! Now lets continue with building out the logic of what a decorator is. Remember that in Python **everything is an object**. That means functions are objects which can be assigned labels and passed into other functions. Lets start with some simple examples:

```
In [3]: def hello(name='Jose'):
            return 'Hello '+name

In [4]: hello()

Out[4]: 'Hello Jose'
```

Assign another label to the function. Note that we are not using parentheses here because we are not calling the function **hello**, instead we are just passing a function object to the **greet** variable.

```
In [5]: greet = hello

In [6]: greet

Out[6]: <function __main__.hello>

In [7]: greet()

Out[7]: 'Hello Jose'
```

So what happens when we delete the name **hello**?

```
In [8]: del hello

In [9]: hello()


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-9-a75d7781aaeb> in <module>()
    ----> 1 hello()


        NameError: name 'hello' is not defined


In [10]: greet()

Out[10]: 'Hello Jose'
```

Even though we deleted the name **hello**, the name **greet** *still points to* our original function object. It is important to know that functions are objects that can be passed to other objects!

1.3 Functions within functions

Great! So we've seen how we can treat functions as objects, now let's see how we can define functions inside of other functions:

```
In [11]: def hello(name='Jose'):
             print('The hello() function has been executed')

             def greet():
                 return '\t This is inside the greet() function'

             def welcome():
                 return "\t This is inside the welcome() function"

             print(greet())
             print(welcome())
             print("Now we are back inside the hello() function")
```

```
In [12]: hello()
```

```
The hello() function has been executed
         This is inside the greet() function
         This is inside the welcome() function
Now we are back inside the hello() function
```

```
In [13]: welcome()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-13-a401d7101853> in <module>()
----> 1 welcome()


NameError: name 'welcome' is not defined
```

Note how due to scope, the welcome() function is not defined outside of the hello() function. Now lets learn about returning functions from within functions: ## Returning Functions

```
In [14]: def hello(name='Jose'):

             def greet():
                 return '\t This is inside the greet() function'

             def welcome():
```

```
            return "\t This is inside the welcome() function"

        if name == 'Jose':
            return greet
        else:
            return welcome
```

Now let's see what function is returned if we set x = hello(), note how the empty parentheses means that name has been defined as Jose.

```
In [15]: x = hello()

In [16]: x

Out[16]: <function __main__.hello.<locals>.greet>
```

Great! Now we can see how x is pointing to the greet function inside of the hello function.

```
In [17]: print(x())

        This is inside the greet() function
```

Let's take a quick look at the code again.

In the if/else clause we are returning greet and welcome, not greet() and welcome().

This is because when you put a pair of parentheses after it, the function gets executed; whereas if you don't put parentheses after it, then it can be passed around and can be assigned to other variables without executing it.

When we write x = hello(), hello() gets executed and because the name is Jose by default, the function greet is returned. If we change the statement to x = hello(name = "Sam") then the welcome function will be returned. We can also do print(hello()()) which outputs *This is inside the greet() function*.

1.4 Functions as Arguments

Now let's see how we can pass functions as arguments into other functions:

```
In [18]: def hello():
             return 'Hi Jose!'

         def other(func):
             print('Other code would go here')
             print(func())

In [19]: other(hello)

Other code would go here
Hi Jose!
```

Great! Note how we can pass the functions as objects and then use them within other functions. Now we can get started with writing our first decorator:

1.5 Creating a Decorator

In the previous example we actually manually created a Decorator. Here we will modify it to make its use case clear:

```
In [20]: def new_decorator(func):

             def wrap_func():
                 print("Code would be here, before executing the func")

                 func()

                 print("Code here will execute after the func()")

             return wrap_func

         def func_needs_decorator():
             print("This function is in need of a Decorator")
```

```
In [21]: func_needs_decorator()
```

```
This function is in need of a Decorator
```

```
In [22]: # Reassign func_needs_decorator
         func_needs_decorator = new_decorator(func_needs_decorator)
```

```
In [23]: func_needs_decorator()
```

```
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```

So what just happened here? A decorator simply wrapped the function and modified its behavior. Now let's understand how we can rewrite this code using the @ symbol, which is what Python uses for Decorators:

```
In [24]: @new_decorator
         def func_needs_decorator():
             print("This function is in need of a Decorator")
```

```
In [25]: func_needs_decorator()
```

```
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```

Great! You've now built a Decorator manually and then saw how we can use the @ symbol in Python to automate this and clean our code. You'll run into Decorators a lot if you begin using Python for Web Development, such as Flask or Django!

02-Decorators Homework

April 9, 2019

1 Decorators Homework (Optional)

Since you won't run into decorators until further in your coding career, this homework is optional. Check out the Web Framework Flask. You can use Flask to create web pages with Python (as long as you know some HTML and CSS) and they use decorators a lot! Learn how they use view decorators. Don't worry if you don't completely understand everything about Flask, the main point of this optional homework is that you have an awareness of decorators in Web Frameworks. That way if you decide to become a "Full-Stack" Python Web Developer, you won't find yourself perplexed by decorators. You can also check out Django another (and more popular) web framework for Python which is a bit more heavy duty.

Also for some additional info:

A framework is a type of software library that provides generic functionality which can be extended by the programmer to build applications. Flask and Django are good examples of frameworks intended for web development.

A framework is distinguished from a simple library or API. An API is a piece of software that a developer can use in his or her application. A framework is more encompassing: your entire application is structured around the framework (i.e. it provides the framework around which you build your software).

1.1 Great job!

01-Iterators and Generators

April 9, 2019

1 Iterators and Generators

In this section of the course we will be learning the difference between iteration and generation in Python and how to construct our own Generators with the *yield* statement. Generators allow us to generate as we go along, instead of holding everything in memory.

We've touched on this topic in the past when discussing certain built-in Python functions like **range()**, **map()** and **filter()**.

Let's explore a little deeper. We've learned how to create functions with def and the return statement. Generator functions allow us to write a function that can send back a value and then later resume to pick up where it left off. This type of function is a generator in Python, allowing us to generate a sequence of values over time. The main difference in syntax will be the use of a yield statement.

In most aspects, a generator function will appear very similar to a normal function. The main difference is when a generator function is compiled they become an object that supports an iteration protocol. That means when they are called in your code they don't actually return a value and then exit. Instead, generator functions will automatically suspend and resume their execution and state around the last point of value generation. The main advantage here is that instead of having to compute an entire series of values up front, the generator computes one value and then suspends its activity awaiting the next instruction. This feature is known as *state suspension*.

To start getting a better understanding of generators, let's go ahead and see how we can create some.

```
In [1]: # Generator function for the cube of numbers (power of 3)
        def gencubes(n):
            for num in range(n):
                yield num**3
```

```
In [2]: for x in gencubes(10):
            print(x)
```

```
0
1
8
27
64
125
216
343
```

```
512
729
```

Great! Now since we have a generator function we don't have to keep track of every single cube we created.

Generators are best for calculating large sets of results (particularly in calculations that involve loops themselves) in cases where we don't want to allocate the memory for all of the results at the same time.

Let's create another example generator which calculates fibonacci numbers:

```
In [3]: def genfibon(n):
            """
            Generate a fibonnaci sequence up to n
            """
            a = 1
            b = 1
            for i in range(n):
                yield a
                a,b = b,a+b

In [4]: for num in genfibon(10):
            print(num)

1
1
2
3
5
8
13
21
34
55
```

What if this was a normal function, what would it look like?

```
In [5]: def fibon(n):
            a = 1
            b = 1
            output = []

            for i in range(n):
                output.append(a)
                a,b = b,a+b

            return output
```

```
In [6]: fibon(10)

Out[6]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Notice that if we call some huge value of n (like 100000) the second function will have to keep track of every single result, when in our case we actually only care about the previous result to generate the next one!

1.1 next() and iter() built-in functions

A key to fully understanding generators is the next() function and the iter() function.

The next() function allows us to access the next element in a sequence. Lets check it out:

```
In [7]: def simple_gen():
            for x in range(3):
                yield x

In [8]: # Assign simple_gen
        g = simple_gen()

In [9]: print(next(g))

0


In [10]: print(next(g))

1


In [11]: print(next(g))

2


In [12]: print(next(g))


        ---------------------------------------------------------------------------

        StopIteration                             Traceback (most recent call last)

        <ipython-input-12-1dfb29d6357e> in <module>()
    ----> 1 print(next(g))


        StopIteration:
```

After yielding all the values next() caused a StopIteration error. What this error informs us of is that all the values have been yielded.

You might be wondering that why don't we get this error while using a for loop? A for loop automatically catches this error and stops calling next().

Let's go ahead and check out how to use iter(). You remember that strings are iterables:

```
In [13]: s = 'hello'

         #Iterate over string
         for let in s:
             print(let)
```

```
h
e
l
l
o
```

But that doesn't mean the string itself is an *iterator*! We can check this with the next() function:

```
In [14]: next(s)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-14-61c30b5fe1d5> in <module>()
----> 1 next(s)


TypeError: 'str' object is not an iterator
```

Interesting, this means that a string object supports iteration, but we can not directly iterate over it as we could with a generator function. The iter() function allows us to do just that!

```
In [15]: s_iter = iter(s)

In [16]: next(s_iter)

Out[16]: 'h'

In [17]: next(s_iter)

Out[17]: 'e'
```

Great! Now you know how to convert objects that are iterable into iterators themselves!

The main takeaway from this lecture is that using the yield keyword at a function will cause the function to become a generator. This change can save you a lot of memory for large use cases. For more information on generators check out:

Stack Overflow Answer

Another StackOverflow Answer

02-Iterators and Generators Homework

April 9, 2019

1 Iterators and Generators Homework

1.0.1 Problem 1

Create a generator that generates the squares of numbers up to some number N.

```
In [1]: def gensquares(N):

            pass
```

```
In [2]: for x in gensquares(10):
            print(x)
```

```
0
1
4
9
16
25
36
49
64
81
```

1.0.2 Problem 2

Create a generator that yields "n" random numbers between a low and high number (that are inputs). Note: Use the random library. For example:

```
In [3]: import random

        random.randint(1,10)
```

```
Out[3]: 9
```

```
In [4]: def rand_num(low,high,n):

            pass
```

```
In [5]: for num in rand_num(1,10,12):
            print(num)
```

```
6
1
10
5
8
2
8
5
4
5
1
4
```

1.0.3 Problem 3

Use the iter() function to convert the string below into an iterator:

```
In [ ]: s = 'hello'

        #code here
```

1.0.4 Problem 4

Explain a use case for a generator using a yield statement where you would not want to use a normal function with a return statement.

1.0.5 Extra Credit!

Can you explain what *gencomp* is in the code below? (Note: We never covered this in lecture! You will have to do some Googling/Stack Overflowing!)

```
In [6]: my_list = [1,2,3,4,5]

        gencomp = (item for item in my_list if item > 3)

        for item in gencomp:
            print(item)
```

```
4
5
```

Hint: Google *generator comprehension*!

2 Great Job!

03-Iterators and Generators Homework - Solution

April 9, 2019

1 Iterators and Generators Homework - Solution

1.0.1 Problem 1

Create a generator that generates the squares of numbers up to some number N.

```
In [1]: def gensquares(N):
            for i in range(N):
                yield i**2
```

```
In [2]: for x in gensquares(10):
            print(x)
```

```
0
1
4
9
16
25
36
49
64
81
```

1.0.2 Problem 2

Create a generator that yields "n" random numbers between a low and high number (that are inputs).Note: Use the random library. For example:

```
In [3]: import random

        random.randint(1,10)
```

```
Out[3]: 3
```

```
In [4]: def rand_num(low,high,n):

            for i in range(n):
                yield random.randint(low, high)
```

```
In [5]: for num in rand_num(1,10,12):
            print(num)
```

```
3
9
6
10
8
4
5
5
5
3
5
8
```

1.0.3 Problem 3

Use the iter() function to convert the string below into an iterator:

```
In [6]: s = 'hello'

        s = iter(s)

        print(next(s))
```

```
h
```

1.0.4 Problem 4

Explain a use case for a generator using a yield statement where you would not want to use a normal function with a return statement.

If the output has the potential of taking up a large amount of memory and you only intend to iterate through it, you would want to use a generator. (Multiple answers are acceptable here!)

1.0.5 Extra Credit!

Can you explain what *gencomp* is in the code below? (Note: We never covered this in lecture!)

```
In [7]: my_list = [1,2,3,4,5]

        gencomp = (item for item in my_list if item > 3)

        for item in gencomp:
            print(item)
```

```
4
5
```

Hint: Google *generator comprehension*!

2 Great Job!

01-Final Capstone Project

April 9, 2019

1 Final Capstone Projects

Please refer to the **Final Capstone Projects** folder to get all the info on final capstone project ideas and possible solutions!

02-Final Capstone Project Ideas

April 9, 2019

1 List of Capstone Projects

This list contains around 100 project ideas for you to try out in Python! Some of them are straightforward and others we have done before (such as a Fibonacci Sequence or FizzBuzz).

Pick a simple project that you think you can finish in a day to start off with, then pick another project that you think will take you more than a week and extensive Googling!

There are some sample solutions in this folder as well so feel free to explore and remember:
HAVE FUN!

1.1 Numbers

Find PI to the Nth Digit - Enter a number and have the program generate (pi) up to that many decimal places. Keep a limit to how far the program will go.

Find e to the Nth Digit - Just like the previous problem, but with e instead of (pi). Enter a number and have the program generate e up to that many decimal places. Keep a limit to how far the program will go.

Fibonacci Sequence - Enter a number and have the program generate the Fibonacci sequence to that number or to the Nth number.

Prime Factorization - Have the user enter a number and find all Prime Factors (if there are any) and display them.

Next Prime Number - Have the program find prime numbers until the user chooses to stop asking for the next one.

Find Cost of Tile to Cover W x H Floor - Calculate the total cost of tile it would take to cover a floor plan of width and height, using a cost entered by the user.

Mortgage Calculator - Calculate the monthly payments of a fixed term mortgage over given Nth terms at a given interest rate. Also figure out how long it will take the user to pay back the loan. For added complexity, add an option for users to select the compounding interval (Monthly, Weekly, Daily, Continually).

Change Return Program - The user enters a cost and then the amount of money given. The program will figure out the change and the number of quarters, dimes, nickels, pennies needed for the change.

Binary to Decimal and Back Converter - Develop a converter to convert a decimal number to binary or a binary number to its decimal equivalent.

Calculator - A simple calculator to do basic operators. Make it a scientific calculator for added complexity.

Unit Converter (temp, currency, volume, mass and more) - Converts various units between one another. The user enters the type of unit being entered, the type of unit they want to convert to and then the value. The program will then make the conversion.

Alarm Clock - A simple clock where it plays a sound after X number of minutes/seconds or at a particular time.

Distance Between Two Cities - Calculates the distance between two cities and allows the user to specify a unit of distance. This program may require finding coordinates for the cities like latitude and longitude.

Credit Card Validator - Takes in a credit card number from a common credit card vendor (Visa, MasterCard, American Express, Discoverer) and validates it to make sure that it is a valid number (look into how credit cards use a checksum).

Tax Calculator - Asks the user to enter a cost and either a country or state tax. It then returns the tax plus the total cost with tax.

Factorial Finder - The Factorial of a positive integer, n, is defined as the product of the sequence n, n-1, n-2, ...1 and the factorial of zero, 0, is defined as being 1. Solve this using both loops and recursion.

Complex Number Algebra - Show addition, multiplication, negation, and inversion of complex numbers in separate functions. (Subtraction and division operations can be made with pairs of these operations.) Print the results for each operation tested.

Happy Numbers - A happy number is defined by the following process. Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers, while those that do not end in 1 are unhappy numbers. Display an example of your output here. Find first 8 happy numbers.

Number Names - Show how to spell out a number in English. You can use a preexisting implementation or roll your own, but you should support inputs up to at least one million (or the maximum value of your language's default bounded integer type, if that's less). *Optional: Support for inputs other than positive integers (like zero, negative integers, and floating-point numbers).*

Coin Flip Simulation - Write some code that simulates flipping a single coin however many times the user decides. The code should record the outcomes and count the number of tails and heads.

Limit Calculator - Ask the user to enter f(x) and the limit value, then return the value of the limit statement *Optional: Make the calculator capable of supporting infinite limits.*

Fast Exponentiation - Ask the user to enter 2 integers a and b and output aˆb (i.e. pow(a,b)) in O(lg n) time complexity.

1.2 Classic Algorithms

Collatz Conjecture - Start with a number $n > 1$. Find the number of steps it takes to reach one using the following process: If n is even, divide it by 2. If n is odd, multiply it by 3 and add 1.

Sorting - Implement two types of sorting algorithms: Merge sort and bubble sort.

Closest pair problem - The closest pair of points problem or closest pair problem is a problem of computational geometry: given n points in metric space, find a pair of points with the smallest distance between them.

Sieve of Eratosthenes - The sieve of Eratosthenes is one of the most efficient ways to find all of the smaller primes (below 10 million or so).

1.3 Graph

Graph from links - Create a program that will create a graph or network from a series of links.

Eulerian Path - Create a program which will take as an input a graph and output either a Eulerian path or a Eulerian cycle, or state that it is not possible. A Eulerian Path starts at one node and traverses every edge of a graph through every node and finishes at another node. A Eulerian cycle is a eulerian Path that starts and finishes at the same node.

Connected Graph - Create a program which takes a graph as an input and outputs whether every node is connected or not.

Dijkstra's Algorithm - Create a program that finds the shortest path through a graph using its edges.

Minimum Spanning Tree - Create a program which takes a connected, undirected graph with weights and outputs the minimum spanning tree of the graph i.e., a subgraph that is a tree, contains all the vertices, and the sum of its weights is the least possible.

1.4 Data Structures

Inverted index - An Inverted Index is a data structure used to create full text search. Given a set of text files, implement a program to create an inverted index. Also create a user interface to do a search using that inverted index which returns a list of files that contain the query term / terms. The search index can be in memory.

1.5 Text

Fizz Buzz - Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

Reverse a String - Enter a string and the program will reverse it and print it out.

Pig Latin - Pig Latin is a game of alterations played on the English language game. To create the Pig Latin form of an English word the initial consonant sound is transposed to the end of the word and an ay is affixed (Ex.: "banana" would yield anana-bay). Read Wikipedia for more information on rules.

Count Vowels - Enter a string and the program counts the number of vowels in the text. For added complexity have it report a sum of each vowel found.

Check if Palindrome - Checks if the string entered by the user is a palindrome. That is that it reads the same forwards as backwards like "racecar"

Count Words in a String - Counts the number of individual words in a string. For added complexity read these strings in from a text file and generate a summary.

Text Editor - Notepad style application that can open, edit, and save text documents. *Optional: Add syntax highlighting and other features.*

RSS Feed Creator - Given a link to RSS/Atom Feed, get all posts and display them.

Quote Tracker (market symbols etc) - A program which can go out and check the current value of stocks for a list of symbols entered by the user. The user can set how often the stocks are checked. For CLI, show whether the stock has moved up or down. *Optional: If GUI, the program can show green up and red down arrows to show which direction the stock value has moved.*

Guestbook / Journal - A simple application that allows people to add comments or write journal entries. It can allow comments or not and timestamps for all entries. Could also be made

into a shout box. *Optional: Deploy it on Google App Engine or Heroku or any other PaaS (if possible, of course).*

Vigenere / Vernam / Ceasar Ciphers - Functions for encrypting and decrypting data messages. Then send them to a friend.

Regex Query Tool - A tool that allows the user to enter a text string and then in a separate control enter a regex pattern. It will run the regular expression against the source text and return any matches or flag errors in the regular expression.

1.6 Networking

FTP Program - A file transfer program which can transfer files back and forth from a remote web sever.

Bandwidth Monitor - A small utility program that tracks how much data you have uploaded and downloaded from the net during the course of your current online session. See if you can find out what periods of the day you use more and less and generate a report or graph that shows it.

Port Scanner - Enter an IP address and a port range where the program will then attempt to find open ports on the given computer by connecting to each of them. On any successful connections mark the port as open.

Mail Checker (POP3 / IMAP) - The user enters various account information include web server and IP, protocol type (POP3 or IMAP) and the application will check for email at a given interval.

Country from IP Lookup - Enter an IP address and find the country that IP is registered in. *Optional: Find the Ip automatically.*

Whois Search Tool - Enter an IP or host address and have it look it up through whois and return the results to you.

Site Checker with Time Scheduling - An application that attempts to connect to a website or server every so many minutes or a given time and check if it is up. If it is down, it will notify you by email or by posting a notice on screen.

1.7 Classes

Product Inventory Project - Create an application which manages an inventory of products. Create a product class which has a price, id, and quantity on hand. Then create an *inventory* class which keeps track of various products and can sum up the inventory value.

Airline / Hotel Reservation System - Create a reservation system which books airline seats or hotel rooms. It charges various rates for particular sections of the plane or hotel. Example, first class is going to cost more than coach. Hotel rooms have penthouse suites which cost more. Keep track of when rooms will be available and can be scheduled.

Company Manager - Create an hierarchy of classes - abstract class Employee and subclasses HourlyEmployee, SalariedEmployee, Manager and Executive. Every one's pay is calculated differently, research a bit about it. After you've established an employee hierarchy, create a Company class that allows you to manage the employees. You should be able to hire, fire and raise employees.

Bank Account Manager - Create a class called Account which will be an abstract class for three other classes called CheckingAccount, SavingsAccount and BusinessAccount. Manage credits and debits from these accounts through an ATM style program.

Patient / Doctor Scheduler - Create a patient class and a doctor class. Have a doctor that can handle multiple patients and setup a scheduling program where a doctor can only handle 16

patients during an 8 hr work day.

Recipe Creator and Manager - Create a recipe class with ingredients and a put them in a recipe manager program that organizes them into categories like deserts, main courses or by ingredients like chicken, beef, soups, pies etc.

Image Gallery - Create an image abstract class and then a class that inherits from it for each image type. Put them in a program which displays them in a gallery style format for viewing.

Shape Area and Perimeter Classes - Create an abstract class called Shape and then inherit from it other shapes like diamond, rectangle, circle, triangle etc. Then have each class override the area and perimeter functionality to handle each shape type.

Flower Shop Ordering To Go - Create a flower shop application which deals in flower objects and use those flower objects in a bouquet object which can then be sold. Keep track of the number of objects and when you may need to order more.

Family Tree Creator - Create a class called Person which will have a name, when they were born and when (and if) they died. Allow the user to create these Person classes and put them into a family tree structure. Print out the tree to the screen.

1.8 Threading

Create A Progress Bar for Downloads - Create a progress bar for applications that can keep track of a download in progress. The progress bar will be on a separate thread and will communicate with the main thread using delegates.

Bulk Thumbnail Creator - Picture processing can take a bit of time for some transformations. Especially if the image is large. Create an image program which can take hundreds of images and converts them to a specified size in the background thread while you do other things. For added complexity, have one thread handling re-sizing, have another bulk renaming of thumbnails etc.

1.9 Web

Page Scraper - Create an application which connects to a site and pulls out all links, or images, and saves them to a list. *Optional: Organize the indexed content and don't allow duplicates. Have it put the results into an easily searchable index file.*

Online White Board - Create an application which allows you to draw pictures, write notes and use various colors to flesh out ideas for projects. *Optional: Add feature to invite friends to collaborate on a white board online.*

Get Atomic Time from Internet Clock - This program will get the true atomic time from an atomic time clock on the Internet. Use any one of the atomic clocks returned by a simple Google search.

Fetch Current Weather - Get the current weather for a given zip/postal code. *Optional: Try locating the user automatically.*

Scheduled Auto Login and Action - Make an application which logs into a given site on a schedule and invokes a certain action and then logs out. This can be useful for checking web mail, posting regular content, or getting info for other applications and saving it to your computer.

E-Card Generator - Make a site that allows people to generate their own little e-cards and send them to other people. Do not use Flash. Use a picture library and perhaps insightful mottos or quotes.

Content Management System - Create a content management system (CMS) like Joomla, Drupal, PHP Nuke etc. Start small. *Optional: Allow for the addition of modules/addons.*

Web Board (Forum) - Create a forum for you and your buddies to post, administer and share thoughts and ideas.

CAPTCHA Maker - Ever see those images with letters a numbers when you signup for a service and then asks you to enter what you see? It keeps web bots from automatically signing up and spamming. Try creating one yourself for online forms.

1.10 Files

Quiz Maker - Make an application which takes various questions from a file, picked randomly, and puts together a quiz for students. Each quiz can be different and then reads a key to grade the quizzes.

Sort Excel/CSV File Utility - Reads a file of records, sorts them, and then writes them back to the file. Allow the user to choose various sort style and sorting based on a particular field.

Create Zip File Maker - The user enters various files from different directories and the program zips them up into a zip file. *Optional: Apply actual compression to the files. Start with Huffman Algorithm.*

PDF Generator - An application which can read in a text file, html file or some other file and generates a PDF file out of it. Great for a web based service where the user uploads the file and the program returns a PDF of the file. *Optional: Deploy on GAE or Heroku if possible.*

Mp3 Tagger - Modify and add ID3v1 tags to MP3 files. See if you can also add in the album art into the MP3 file's header as well as other ID3v2 tags.

Code Snippet Manager - Another utility program that allows coders to put in functions, classes or other tidbits to save for use later. Organized by the type of snippet or language the coder can quickly look up code. *Optional: For extra practice try adding syntax highlighting based on the language.*

1.11 Databases

SQL Query Analyzer - A utility application which a user can enter a query and have it run against a local database and look for ways to make it more efficient.

Remote SQL Tool - A utility that can execute queries on remote servers from your local computer across the Internet. It should take in a remote host, user name and password, run the query and return the results.

Report Generator - Create a utility that generates a report based on some tables in a database. Generates a sales reports based on the order/order details tables or sums up the days current database activity.

Event Scheduler and Calendar - Make an application which allows the user to enter a date and time of an event, event notes and then schedule those events on a calendar. The user can then browse the calendar or search the calendar for specific events. *Optional: Allow the application to create re-occurrence events that reoccur every day, week, month, year etc.*

Budget Tracker - Write an application that keeps track of a household's budget. The user can add expenses, income, and recurring costs to find out how much they are saving or losing over a period of time. *Optional: Allow the user to specify a date range and see the net flow of money in and out of the house budget for that time period.*

TV Show Tracker - Got a favorite show you don't want to miss? Don't have a PVR or want to be able to find the show to then PVR it later? Make an application which can search various online TV Guide sites, locate the shows/times/channels and add them to a database application.

The database/website then can send you email reminders that a show is about to start and which channel it will be on.

Travel Planner System - Make a system that allows users to put together their own little travel itinerary and keep track of the airline / hotel arrangements, points of interest, budget and schedule.

1.12 Graphics and Multimedia

Slide Show - Make an application that shows various pictures in a slide show format. *Optional: Try adding various effects like fade in/out, star wipe and window blinds transitions.*

Stream Video from Online - Try to create your own online streaming video player.

Mp3 Player - A simple program for playing your favorite music files. Add features you think are missing from your favorite music player.

Watermarking Application - Have some pictures you want copyright protected? Add your own logo or text lightly across the background so that no one can simply steal your graphics off your site. Make a program that will add this watermark to the picture. *Optional: Use threading to process multiple images simultaneously.*

Turtle Graphics - This is a common project where you create a floor of 20 x 20 squares. Using various commands you tell a turtle to draw a line on the floor. You have move forward, left or right, lift or drop pen etc. Do a search online for "Turtle Graphics" for more information. *Optional: Allow the program to read in the list of commands from a file.*

GIF Creator A program that puts together multiple images (PNGs, JPGs, TIFFs) to make a smooth GIF that can be exported. *Optional: Make the program convert small video files to GIFs as well.*

1.13 Security

Caesar cipher - Implement a Caesar cipher, both encoding and decoding. The key is an integer from 1 to 25. This cipher rotates the letters of the alphabet (A to Z). The encoding replaces each letter with the 1st to 25th next letter in the alphabet (wrapping Z to A). So key 2 encrypts "HI" to "JK", but key 20 encrypts "HI" to "BC". This simple "monoalphabetic substitution cipher" provides almost no security, because an attacker who has the encoded message can either use frequency analysis to guess the key, or just try all 25 keys

03-Final Capstone Suggested Walkthrough

April 9, 2019

1 Final Capstone Project - Suggested Walkthrough:

This is a suggested method for handling one of the Final Capstone Projects. We start by coding out the strictest requirements, and then build out from a working baseline model. Feel free to adapt this solution, and add features you think could help. Good luck!

1.1 Bank Account Manager

Under the Classes section in the list of suggested final capstone projects is a Bank Account Manager program. The goal is to create a class called Account which will be an abstract class for three other classes called CheckingAccount, SavingsAccount and BusinessAccount. Then you should manage credits and debits from these accounts through an ATM style program.

1.1.1 Project Scope

To tackle this project, first consider what has to happen. 1. There will be three different types of bank account (Checking, Savings, Business) 2. Each account will accept deposits and withdrawals, and will need to report balances

1.1.2 Project Wishlist

We might consider additional features, like: * impose a monthly maintenance fee * waive fees for minimum combined deposit balances * each account may have additional properties unique to that account: * Checking allows unlimited transactions, and may keep track of printed checks * Savings limits the number of withdrawals per period, and may earn interest * Business may impose transaction fees * automatically transfer the "change" for debit card purchases from Checking to Savings, where "change" is the amount needed to raise a debit to the nearest whole dollar * permit savings autodraft overdraft protection

1.1.3 Let's get started!

Step 1: Establish an abstract Account class with features shared by all accounts. Note that abstract classes are never instantiated, they simply provide a base class with attributes and methods to be inherited by any derived class.

```
In [1]: class Account:
            # Define an __init__ constructor method with attributes shared by all accounts:
            def __init__(self,acct_nbr,opening_deposit):
```

```
            self.acct_nbr = acct_nbr
            self.balance = opening_deposit

        # Define a __str__ mehthod to return a recognizable string to any print() command
        def __str__(self):
            return f'${self.balance:.2f}'

        # Define a universal method to accept deposits
        def deposit(self,dep_amt):
            self.balance += dep_amt

        # Define a universal method to handle withdrawals
        def withdraw(self,wd_amt):
            if self.balance >= wd_amt:
                self.balance -= wd_amt
            else:
                return 'Funds Unavailable'
```

Step 2: Establish a Checking Account class that inherits from Account, and adds Checking-specific traits.

```
In [2]: class Checking(Account):
            def __init__(self,acct_nbr,opening_deposit):
                # Run the base class __init__
                super().__init__(acct_nbr,opening_deposit)

            # Define a __str__ method that returns a string specific to Checking accounts
            def __str__(self):
                return f'Checking Account #{self.acct_nbr}\n  Balance: {Account.__str__(self)}
```

Step 3: TEST setting up a Checking Account object

```
In [3]: x = Checking(54321,654.33)

In [4]: print(x)

Checking Account #54321
  Balance: $654.33


In [5]: x.withdraw(1000)

Out[5]: 'Funds Unavailable'

In [6]: x.withdraw(30)

In [7]: x.balance

Out[7]: 624.33
```

Step 4: Set up similar Savings and Business account classes

```
In [8]: class Savings(Account):
            def __init__(self,acct_nbr,opening_deposit):
                # Run the base class __init__
                super().__init__(acct_nbr,opening_deposit)

            # Define a __str__ method that returns a string specific to Savings accounts
            def __str__(self):
                return f'Savings Account #{self.acct_nbr}\n  Balance: {Account.__str__(self)}'


        class Business(Account):
            def __init__(self,acct_nbr,opening_deposit):
                # Run the base class __init__
                super().__init__(acct_nbr,opening_deposit)

            # Define a __str__ method that returns a string specific to Business accounts
            def __str__(self):
                return f'Business Account #{self.acct_nbr}\n  Balance: {Account.__str__(self)}
```

At this point we've met the minimum requirement for the assignment. We have three different bank account classes. Each one can accept deposits, make withdrawals and report a balance, as they each inherit from an abstract Account base class.

So now the fun part - let's add some features!

Step 5: Create a Customer class For this next phase, let's set up a Customer class that holds a customer's name and PIN and can contain any number and/or combination of Account objects.

```
In [9]: class Customer:
            def __init__(self, name, PIN):
                self.name = name
                self.PIN = PIN

                # Create a dictionary of accounts, with lists to hold multiple accounts
                self.accts = {'C':[],'S':[],'B':[]}

            def __str__(self):
                return self.name

            def open_checking(self,acct_nbr,opening_deposit):
                self.accts['C'].append(Checking(acct_nbr,opening_deposit))

            def open_savings(self,acct_nbr,opening_deposit):
                self.accts['S'].append(Savings(acct_nbr,opening_deposit))

            def open_business(self,acct_nbr,opening_deposit):
                self.accts['B'].append(Business(acct_nbr,opening_deposit))
```

```
    # rather than maintain a running total of deposit balances,
    # write a method that computes a total as needed
    def get_total_deposits(self):
        total = 0
        for acct in self.accts['C']:
            print(acct)
            total += acct.balance
        for acct in self.accts['S']:
            print(acct)
            total += acct.balance
        for acct in self.accts['B']:
            print(acct)
            total += acct.balance
        print(f'Combined Deposits: ${total}')
```

Step 6: TEST setting up a Customer, adding accounts, and checking balances

```
In [10]: bob = Customer('Bob',1)

In [11]: bob.open_checking(321,555.55)

In [12]: bob.get_total_deposits()

Checking Account #321
  Balance: $555.55
Combined Deposits: $555.55


In [13]: bob.open_savings(564,444.66)

In [14]: bob.get_total_deposits()

Checking Account #321
  Balance: $555.55
Savings Account #564
  Balance: $444.66
Combined Deposits: $1000.21


In [15]: nancy = Customer('Nancy',2)

In [16]: nancy.open_business(2018,8900)

In [17]: nancy.get_total_deposits()

Business Account #2018
  Balance: $8900.00
Combined Deposits: $8900
```

Wait! Why don't Nancy's combined deposits show a decimal? This is easily fixed in the class definition (mostly copied from above, with a change made to the last line of code):

```
In [18]: class Customer:
             def __init__(self, name, PIN):
                 self.name = name
                 self.PIN = PIN
                 self.accts = {'C':[],'S':[],'B':[]}

             def __str__(self):
                 return self.name

             def open_checking(self,acct_nbr,opening_deposit):
                 self.accts['C'].append(Checking(acct_nbr,opening_deposit))

             def open_savings(self,acct_nbr,opening_deposit):
                 self.accts['S'].append(Savings(acct_nbr,opening_deposit))

             def open_business(self,acct_nbr,opening_deposit):
                 self.accts['B'].append(Business(acct_nbr,opening_deposit))

             def get_total_deposits(self):
                 total = 0
                 for acct in self.accts['C']:
                     print(acct)
                     total += acct.balance
                 for acct in self.accts['S']:
                     print(acct)
                     total += acct.balance
                 for acct in self.accts['B']:
                     print(acct)
                     total += acct.balance
                 print(f'Combined Deposits: ${total:.2f}') # added precision formatting here
```

So it's fixed, right?

```
In [19]: nancy.get_total_deposits()

Business Account #2018
  Balance: $8900.00
Combined Deposits: $8900
```

Nope! Changes made to the class definition do *not* affect objects created under different sets of instructions.To fix Nancy's account, we have to build her record from scratch.

```
In [20]: nancy = Customer('Nancy',2)
         nancy.open_business(2018,8900)
         nancy.get_total_deposits()
```

```
Business Account #2018
  Balance: $8900.00
Combined Deposits: $8900.00
```

This is why testing is so important!

Step 7: Let's write some functions for making deposits and withdrawals. Be sure to include a docstring that explains what's expected by the function!

```
In [21]: def make_dep(cust,acct_type,acct_num,dep_amt):
             """
             make_dep(cust, acct_type, acct_num, dep_amt)
             cust      = variable name (Customer record/ID)
             acct_type = string 'C' 'S' or 'B'
             acct_num  = integer
             dep_amt   = integer
             """
             for acct in cust.accts[acct_type]:
                 if acct.acct_nbr == acct_num:
                     acct.deposit(dep_amt)

In [22]: make_dep(nancy,'B',2018,67.45)

In [23]: nancy.get_total_deposits()

Business Account #2018
  Balance: $8967.45
Combined Deposits: $8967.45


In [24]: def make_wd(cust,acct_type,acct_num,wd_amt):
             """
             make_dep(cust, acct_type, acct_num, wd_amt)
             cust      = variable name (Customer record/ID)
             acct_type = string 'C' 'S' or 'B'
             acct_num  = integer
             wd_amt    = integer
             """
             for acct in cust.accts[acct_type]:
                 if acct.acct_nbr == acct_num:
                     acct.withdraw(wd_amt)

In [25]: make_wd(nancy,'B',2018,1000000)

In [26]: nancy.get_total_deposits()

Business Account #2018
  Balance: $8967.45
Combined Deposits: $8967.45
```

What happened?? We seemed to successfully make a withdrawal, but nothing changed!This is because, at the very beginning, we had our Account class *return* the string 'Funds Unavailable' instead of print it. If we change that here, we'll have to also run the derived class definitions, and Nancy's creation, but *not* the Customer class definition. Watch:

```
In [27]: class Account:
             def __init__(self,acct_nbr,opening_deposit):
                 self.acct_nbr = acct_nbr
                 self.balance = opening_deposit

             def __str__(self):
                 return f'${self.balance:.2f}'

             def deposit(self,dep_amt):
                 self.balance += dep_amt

             def withdraw(self,wd_amt):
                 if self.balance >= wd_amt:
                     self.balance -= wd_amt
                 else:
                     print('Funds Unavailable')  # changed "return" to "print"

In [30]: class Checking(Account):
             def __init__(self,acct_nbr,opening_deposit):
                 super().__init__(acct_nbr,opening_deposit)

             def __str__(self):
                 return f'Checking Account #{self.acct_nbr}\n  Balance: {Account.__str__(self)}


         class Savings(Account):
             def __init__(self,acct_nbr,opening_deposit):
                 super().__init__(acct_nbr,opening_deposit)

             def __str__(self):
                 return f'Savings Account #{self.acct_nbr}\n  Balance: {Account.__str__(self)}


         class Business(Account):
             def __init__(self,acct_nbr,opening_deposit):
                 super().__init__(acct_nbr,opening_deposit)

             def __str__(self):
                 return f'Business Account #{self.acct_nbr}\n  Balance: {Account.__str__(self)}

In [31]: nancy = Customer('Nancy',2)
         nancy.open_business(2018,8900)
         nancy.get_total_deposits()
```

```
Business Account #2018
  Balance: $8900.00
Combined Deposits: $8900.00
```

```
In [32]: make_wd(nancy,'B',2018,1000000)
```

```
Funds Unavailable
```

```
In [33]: nancy.get_total_deposits()
```

```
Business Account #2018
  Balance: $8900.00
Combined Deposits: $8900.00
```

1.2 Good job!

01-Collections Module

April 9, 2019

1 Collections Module

The collections module is a built-in module that implements specialized container data types providing alternatives to Python's general purpose built-in containers. We've already gone over the basics: dict, list, set, and tuple.

Now we'll learn about the alternatives that the collections module provides.

1.1 Counter

Counter is a *dict* subclass which helps count hashable objects. Inside of it elements are stored as dictionary keys and the counts of the objects are stored as the value.

Let's see how it can be used:

```
In [1]: from collections import Counter
```

Counter() with lists

```
In [2]: lst = [1,2,2,2,2,3,3,3,1,2,1,12,3,2,32,1,21,1,223,1]

        Counter(lst)

Out[2]: Counter({1: 6, 2: 6, 3: 4, 12: 1, 21: 1, 32: 1, 223: 1})
```

Counter with strings

```
In [3]: Counter('aabsbsbsbhshhbbsbs')

Out[3]: Counter({'a': 2, 'b': 7, 'h': 3, 's': 6})
```

Counter with words in a sentence

```
In [4]: s = 'How many times does each word show up in this sentence word times each each word'

        words = s.split()

        Counter(words)
```

```
Out[4]: Counter({'How': 1,
                 'does': 1,
                 'each': 3,
                 'in': 1,
                 'many': 1,
                 'sentence': 1,
                 'show': 1,
                 'this': 1,
                 'times': 2,
                 'up': 1,
                 'word': 3})

In [5]: # Methods with Counter()
        c = Counter(words)

        c.most_common(2)

Out[5]: [('each', 3), ('word', 3)]
```

1.2 Common patterns when using the Counter() object

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
c += Counter()                  # remove zero and negative counts
```

1.3 defaultdict

defaultdict is a dictionary-like object which provides all methods provided by a dictionary but takes a first argument (default_factory) as a default data type for the dictionary. Using defaultdict is faster than doing the same using dict.set_default method.

A defaultdict will never raise a KeyError. Any key that does not exist gets the value returned by the default factory.

```
In [6]: from collections import defaultdict

In [7]: d = {}

In [8]: d['one']

        ---------------------------------------------------------------------------

        KeyError                                  Traceback (most recent call last)
```

```
        <ipython-input-8-924453a5f45e> in <module>()
    ----> 1 d['one']


        KeyError: 'one'
```

```
In [9]: d  = defaultdict(object)

In [10]: d['one']

Out[10]: <object at 0x1792df202e0>

In [11]: for item in d:
             print(item)

one
```

Can also initialize with default values:

```
In [12]: d = defaultdict(lambda: 0)

In [13]: d['one']

Out[13]: 0
```

1.4 OrderedDict

An OrderedDict is a dictionary subclass that remembers the order in which its contents are added.

For example a normal dictionary:

```
In [14]: print('Normal dictionary:')

         d = {}

         d['a'] = 'A'
         d['b'] = 'B'
         d['c'] = 'C'
         d['d'] = 'D'
         d['e'] = 'E'

         for k, v in d.items():
             print(k, v)

Normal dictionary:
a A
b B
c C
d D
e E
```

An Ordered Dictionary:

```
In [15]: from collections import OrderedDict

         print('OrderedDict:')

         d = OrderedDict()

         d['a'] = 'A'
         d['b'] = 'B'
         d['c'] = 'C'
         d['d'] = 'D'
         d['e'] = 'E'

         for k, v in d.items():
             print(k, v)
```

```
OrderedDict:
a A
b B
c C
d D
e E
```

1.5 Equality with an Ordered Dictionary

A regular dict looks at its contents when testing for equality. An OrderedDict also considers the order the items were added.

A normal Dictionary:

```
In [16]: print('Dictionaries are equal?')

         d1 = {}
         d1['a'] = 'A'
         d1['b'] = 'B'

         d2 = {}
         d2['b'] = 'B'
         d2['a'] = 'A'

         print(d1==d2)
```

```
Dictionaries are equal?
True
```

An Ordered Dictionary:

```
In [17]: print('Dictionaries are equal?')

         d1 = OrderedDict()
         d1['a'] = 'A'
         d1['b'] = 'B'


         d2 = OrderedDict()

         d2['b'] = 'B'
         d2['a'] = 'A'

         print(d1==d2)
```

```
Dictionaries are equal?
False
```

2 namedtuple

The standard tuple uses numerical indexes to access its members, for example:

```
In [18]: t = (12,13,14)

In [19]: t[0]

Out[19]: 12
```

For simple use cases, this is usually enough. On the other hand, remembering which index should be used for each value can lead to errors, especially if the tuple has a lot of fields and is constructed far from where it is used. A namedtuple assigns names, as well as the numerical index, to each member.

Each kind of namedtuple is represented by its own class, created by using the namedtuple() factory function. The arguments are the name of the new class and a string containing the names of the elements.

You can basically think of namedtuples as a very quick way of creating a new object/class type with some attribute fields. For example:

```
In [20]: from collections import namedtuple

In [21]: Dog = namedtuple('Dog','age breed name')

         sam = Dog(age=2,breed='Lab',name='Sammy')

         frank = Dog(age=2,breed='Shepard',name="Frankie")
```

We construct the namedtuple by first passing the object type name (Dog) and then passing a string with the variety of fields as a string with spaces between the field names. We can then call on the various attributes:

```
In [22]: sam

Out[22]: Dog(age=2, breed='Lab', name='Sammy')

In [23]: sam.age

Out[23]: 2

In [24]: sam.breed

Out[24]: 'Lab'

In [25]: sam[0]

Out[25]: 2
```

2.1 Conclusion

Hopefully you now see how incredibly useful the collections module is in Python and it should be your go-to module for a variety of common tasks!

02-Datetime

April 9, 2019

1 datetime

Python has the datetime module to help deal with timestamps in your code. Time values are represented with the time class. Times have attributes for hour, minute, second, and microsecond. They can also include time zone information. The arguments to initialize a time instance are optional, but the default of 0 is unlikely to be what you want.

1.1 time

Let's take a look at how we can extract time information from the datetime module. We can create a timestamp by specifying datetime.time(hour,minute,second,microsecond)

```
In [1]: import datetime

        t = datetime.time(4, 20, 1)

        # Let's show the different components
        print(t)
        print('hour  :', t.hour)
        print('minute:', t.minute)
        print('second:', t.second)
        print('microsecond:', t.microsecond)
        print('tzinfo:', t.tzinfo)
```

```
04:20:01
hour  : 4
minute: 20
second: 1
microsecond: 0
tzinfo: None
```

Note: A time instance only holds values of time, and not a date associated with the time.
We can also check the min and max values a time of day can have in the module:

```
In [2]: print('Earliest  :', datetime.time.min)
        print('Latest    :', datetime.time.max)
        print('Resolution:', datetime.time.resolution)
```

```
Earliest  : 00:00:00
Latest    : 23:59:59.999999
Resolution: 0:00:00.000001
```

The min and max class attributes reflect the valid range of times in a single day.

1.2 Dates

datetime (as you might suspect) also allows us to work with date timestamps. Calendar date values are represented with the date class. Instances have attributes for year, month, and day. It is easy to create a date representing today's date using the today() class method.

Let's see some examples:

```
In [3]: today = datetime.date.today()
        print(today)
        print('ctime:', today.ctime())
        print('tuple:', today.timetuple())
        print('ordinal:', today.toordinal())
        print('Year :', today.year)
        print('Month:', today.month)
        print('Day  :', today.day)
```

```
2018-02-05
ctime: Mon Feb  5 00:00:00 2018
tuple: time.struct_time(tm_year=2018, tm_mon=2, tm_mday=5, tm_hour=0, tm_min=0, tm_sec=0, tm_w
ordinal: 736730
Year : 2018
Month: 2
Day  : 5
```

As with time, the range of date values supported can be determined using the min and max attributes.

```
In [4]: print('Earliest  :', datetime.date.min)
        print('Latest    :', datetime.date.max)
        print('Resolution:', datetime.date.resolution)
```

```
Earliest  : 0001-01-01
Latest    : 9999-12-31
Resolution: 1 day, 0:00:00
```

Another way to create new date instances uses the replace() method of an existing date. For example, you can change the year, leaving the day and month alone.

```
In [5]: d1 = datetime.date(2015, 3, 11)
        print('d1:', d1)

        d2 = d1.replace(year=1990)
        print('d2:', d2)
```

```
d1: 2015-03-11
d2: 1990-03-11
```

2 Arithmetic

We can perform arithmetic on date objects to check for time differences. For example:

```
In [6]: d1

Out[6]: datetime.date(2015, 3, 11)

In [7]: d2

Out[7]: datetime.date(1990, 3, 11)

In [8]: d1-d2

Out[8]: datetime.timedelta(9131)
```

This gives us the difference in days between the two dates. You can use the timedelta method to specify various units of times (days, minutes, hours, etc.)

Great! You should now have a basic understanding of how to use datetime with Python to work with timestamps in your code!

03-Python Debugger (pdb)

April 9, 2019

1 Python Debugger

You've probably used a variety of print statements to try to find errors in your code. A better way of doing this is by using Python's built-in debugger module (pdb). The pdb module implements an interactive debugging environment for Python programs. It includes features to let you pause your program, look at the values of variables, and watch program execution step-by-step, so you can understand what your program actually does and find bugs in the logic.

This is a bit difficult to show since it requires creating an error on purpose, but hopefully this simple example illustrates the power of the pdb module. *Note: Keep in mind it would be pretty unusual to use pdb in an iPython Notebook setting.*

Here we will create an error on purpose, trying to add a list to an integer

```
In [1]: x = [1,3,4]
        y = 2
        z = 3

        result = y + z
        print(result)
        result2 = y+x
        print(result2)
```

```
5
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-1-e7e4af986cb2> in <module>()
          5 result = y + z
          6 print(result)
    ----> 7 result2 = y+x
          8 print(result2)
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Hmmm, looks like we get an error! Let's implement a set_trace() using the pdb module. This will allow us to basically pause the code at the point of the trace and check if anything is wrong.

In [ ]:
```
import pdb

x = [1,3,4]
y = 2
z = 3

result = y + z
print(result)

# Set a trace using Python Debugger
pdb.set_trace()

result2 = y+x
print(result2)
```

Great! Now we could check what the various variables were and check for errors. You can use 'q' to quit the debugger. For more information on general debugging techniques and more methods, check out the official documentation: https://docs.python.org/3/library/pdb.html

04-Timing your code - timeit

April 9, 2019

1 Timing your code

Sometimes it's important to know how long your code is taking to run, or at least know if a particular line of code is slowing down your entire project. Python has a built-in timing module to do this.

This module provides a simple way to time small bits of Python code. It has both a Command-Line Interface as well as a callable one. It avoids a number of common traps for measuring execution times.

Let's learn about timeit!

```
In [1]: import timeit
```

Let's use timeit to time various methods of creating the string '0-1-2-3-.....-99'

We'll pass two arguments: the actual line we want to test encapsulated as a string and the number of times we wish to run it. Here we'll choose 10,000 runs to get some high enough numbers to compare various methods.

```
In [2]: # For loop
        timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)

Out[2]: 0.21865416520477374

In [3]: # List comprehension
        timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)

Out[3]: 0.19484614421698643

In [4]: # Map()
        timeit.timeit('"-".join(map(str, range(100)))', number=10000)

Out[4]: 0.15291817337139246
```

Great! We see a significant time difference by using map()! This is good to know and we should keep this in mind.

Now let's introduce iPython's magic function **%timeit** *NOTE: This method is specific to jupyter notebooks!*

iPython's %timeit will perform the same lines of code a certain number of times (loops) and will give you the fastest performance time (best of 3).

Let's repeat the above examinations using iPython magic!

```
In [5]: %timeit "-".join(str(n) for n in range(100))

20.4 ţs ś 269 ns per loop (mean ś std. dev. of 7 runs, 10000 loops each)


In [6]: %timeit "-".join([str(n) for n in range(100)])

18.1 ţs ś 56.2 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)


In [7]: %timeit "-".join(map(str, range(100)))

14.4 ţs ś 64.1 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)
```

Great! We arrive at the same conclusion. It's also important to note that iPython will limit the amount of *real time* it will spend on its timeit procedure. For instance if running 100000 loops took 10 minutes, iPython would automatically reduce the number of loops to something more reasonable like 100 or 1000.

Great! You should now feel comfortable timing lines of your code, both in and out of iPython. Check out the documentation for more information: https://docs.python.org/3/library/timeit.html

Regular Expressions

Regular expressions are text-matching patterns described with a formal syntax. You'll often hear regular expressions referred to as 'regex' or 'regexp' in conversation. Regular expressions can include a variety of rules, from finding repetition, to text-matching, and much more. As you advance in Python you'll see that a lot of your parsing problems can be solved with regular expressions (they're also a common interview question!).

If you're familiar with Perl, you'll notice that the syntax for regular expressions are very similar in Python. We will be using the `re` module with Python for this lecture.

Let's get started!

Searching for Patterns in Text

One of the most common uses for the re module is for finding patterns in text. Let's do a quick example of using the search method in the re module to find some text:

In [1]:
```
import re

# List of patterns to search for
patterns = ['term1', 'term2']

# Text to parse
text = 'This is a string with term1, but it does not have the other term.'

for pattern in patterns:
    print('Searching for "%s" in:\n "%s"\n' %(pattern,text))

    #Check for match
    if re.search(pattern,text):
        print('Match was found. \n')
    else:
        print('No Match was found.\n')
```

```
Searching for "term1" in:
 "This is a string with term1, but it does not have the other term."

Match was found.

Searching for "term2" in:
 "This is a string with term1, but it does not have the other term."

No Match was found.
```

Now we've seen that `re.search()` will take the pattern, scan the text, and then return a **Match** object. If no pattern is found, **None** is returned. To give a clearer picture of this match object, check out the cell below:

In [2]:
```
# List of patterns to search for
pattern = 'term1'

# Text to parse
text = 'This is a string with term1, but it does not have the other term.'

match = re.search(pattern,text)

type(match)
```

Out[2]:
```
_sre.SRE_Match
```

This **Match** object returned by the search() method is more than just a Boolean or None, it contains information about the match, including the original input string, the regular expression that was used, and the location of the match. Let's see the methods we can use on the match object:

In [3]:
```
# Show start of match
match.start()
```

Out[3]:
```
22
```

In [4]:
```
# Show end
match.end()
```

Out[4]:
```
27
```

Split with regular expressions

Let's see how we can split with the re syntax. This should look similar to how you used the split() method with strings.

In [5]:
```
# Term to split on
split_term = '@'

phrase = 'What is the domain name of someone with the email: hello@gmail.com'

# Split the phrase
re.split(split_term,phrase)
```

Out[5]:
```
['What is the domain name of someone with the email: hello', 'gmail.com']
```

Note how `re.split()` returns a list with the term to split on removed and the terms in the list are a split up version of the string. Create a couple of more examples for yourself to make sure you understand!

Finding all instances of a pattern

You can use `re.findall()` to find all the instances of a pattern in a string. For example:

In [6]:
```
# Returns a list of all matches
re.findall('match','test phrase match is in middle')
```

Out[6]:
```
['match']
```

re Pattern Syntax

This will be the bulk of this lecture on using re with Python. Regular expressions support a huge variety of patterns beyond just simply finding where a single string occurred.

We can use *metacharacters* along with re to find specific types of patterns.

Since we will be testing multiple re syntax forms, let's create a function that will print out results given a list of various regular expressions and a phrase to parse:

In [7]:
```
def multi_re_find(patterns,phrase):
    '''
    Takes in a list of regex patterns
    Prints a list of all matches
    '''
    for pattern in patterns:
        print('Searching the phrase using the re check: %r' %(pattern))
        print(re.findall(pattern,phrase))
        print('\n')
```

Repetition Syntax

There are five ways to express repetition in a pattern:

1. A pattern followed by the meta-character `*` is repeated zero or more times.
2. Replace the `*` with `+` and the pattern must appear at least once.
3. Using `?` means the pattern appears zero or one time.
4. For a specific number of occurrences, use `{m}` after the pattern, where **m** is replaced with the number of times the pattern should repeat.
5. Use `{m,n}` where **m** is the minimum number of repetitions and **n** is the maximum. Leaving out **n** `{m,}` means the value appears at least **m** times, with no maximum.

Now we will see an example of each of these using our multi_re_find function:

In [8]:
```
test_phrase = 'sdsd..sssddd...sdddsddd...dsds...dssss...sdddd'

test_patterns = [ 'sd*',     # s followed by zero or more d's
                  'sd+',          # s followed by one or more d's
                  'sd?',          # s followed by zero or one d's
                  'sd{3}',        # s followed by three d's
                  'sd{2,3}',      # s followed by two to three d's
                  ]

multi_re_find(test_patterns,test_phrase)
```

```
Searching the phrase using the re check: 'sd*'
['sd', 'sd', 's', 's', 'sddd', 'sddd', 'sddd', 'sd', 's', 's', 's', 's', '
```

```
s', 's', 'sdddd']


Searching the phrase using the re check: 'sd+'
['sd', 'sd', 'sddd', 'sddd', 'sddd', 'sd', 'sdddd']


Searching the phrase using the re check: 'sd?'
['sd', 'sd', 's', 's', 'sd', 'sd', 'sd', 'sd', 's', 's', 's', 's', 's', 's
', 'sd']


Searching the phrase using the re check: 'sd{3}'
['sddd', 'sddd', 'sddd', 'sddd']


Searching the phrase using the re check: 'sd{2,3}'
['sddd', 'sddd', 'sddd', 'sddd']
```

Character Sets

Character sets are used when you wish to match any one of a group of characters at a point in the input. Brackets are used to construct character set inputs. For example: the input `[ab]` searches for occurrences of either **a** or **b**. Let's see some examples:

In [9]:
```
test_phrase = 'sdsd..sssddd...sdddsddd...dsds...dssss...sdddd'

test_patterns = ['[sd]',    # either s or d
                 's[sd]+']   # s followed by one or more s or d

multi_re_find(test_patterns,test_phrase)
```

```
Searching the phrase using the re check: '[sd]'
['s', 'd', 's', 'd', 's', 's', 's', 'd', 'd', 'd', 's', 'd', 'd', 'd', 's'
, 'd', 'd', 'd', 'd', 's', 'd', 's', 'd', 's', 's', 's', 's', 's', 's', 'd
', 'd', 'd', 'd']


Searching the phrase using the re check: 's[sd]+'
['sdsd', 'sssddd', 'sdddsddd', 'sds', 'sssss', 'sdddd']
```

It makes sense that the first input `[sd]` returns every instance of s or d. Also, the second input `s[sd]+` returns any full strings that begin with an s and continue with s or d characters until another character is reached.

Exclusion

We can use `^` to exclude terms by incorporating it into the bracket syntax notation. For example:

`[^...]` will match any single character not in the brackets. Let's see some examples:

In [10]:
```
test_phrase = 'This is a string! But it has punctuation. How can we remove
 it?'
```

Use `[^!.? ]` to check for matches that are not a !,.,?, or space. Add a `+` to check that the match appears at least once. This basically translates into finding the words.

In [11]:
```
re.findall('[^!.? ]+',test_phrase)
```

Out[11]:
```
['This',
 'is',
 'a',
 'string',
 'But',
 'it',
 'has',
 'punctuation',
 'How',
 'can',
 'we',
 'remove',
 'it']
```

Character Ranges

As character sets grow larger, typing every character that should (or should not) match could become very tedious. A more compact format using character ranges lets you define a character set to include all of the contiguous characters between a start and stop point. The format used is `[start-end]`.

Common use cases are to search for a specific range of letters in the alphabet. For instance, `[a-f]` would return matches with any occurrence of letters between a and f.

Let's walk through some examples:

In [12]:
```
test_phrase = 'This is an example sentence. Lets see if we can find some l
etters.'

test_patterns=['[a-z]+',      # sequences of lower case letters
               '[A-Z]+',      # sequences of upper case letters
               '[a-zA-Z]+',   # sequences of lower or upper case letters
               '[A-Z][a-z]+'] # one upper case letter followed by lower ca
se letters

multi_re_find(test_patterns,test_phrase)
```

```
Searching the phrase using the re check: '[a-z]+'
['his', 'is', 'an', 'example', 'sentence', 'ets', 'see', 'if', 'we', 'can'
, 'find', 'some', 'letters']


Searching the phrase using the re check: '[A-Z]+'
```

```
['T', 'L']


Searching the phrase using the re check: '[a-zA-Z]+'
['This', 'is', 'an', 'example', 'sentence', 'Lets', 'see', 'if', 'we', 'ca
n', 'find', 'some', 'letters']


Searching the phrase using the re check: '[A-Z][a-z]+'
['This', 'Lets']
```

Escape Codes

You can use special escape codes to find specific types of patterns in your data, such as digits, non-digits, whitespace, and more. For example:

Code	Meaning
`\d`	a digit
`\D`	a non-digit
`\s`	whitespace (tab, space, newline, etc.)
`\S`	non-whitespace
`\w`	alphanumeric
`\W`	non-alphanumeric

Escapes are indicated by prefixing the character with a backslash `\`. Unfortunately, a backslash must itself be escaped in normal Python strings, and that results in expressions that are difficult to read. Using raw strings, created by prefixing the literal value with `r`, eliminates this problem and maintains readability.

Personally, I think this use of `r` to escape a backslash is probably one of the things that block someone who is not familiar with regex in Python from being able to read regex code at first. Hopefully after seeing these examples this syntax will become clear.

In [13]:
```
test_phrase = 'This is a string with some numbers 1233 and a symbol #hasht
ag'

test_patterns=[ r'\d+', # sequence of digits
                r'\D+', # sequence of non-digits
                r'\s+', # sequence of whitespace
                r'\S+', # sequence of non-whitespace
                r'\w+', # alphanumeric characters
                r'\W+', # non-alphanumeric
                ]

multi_re_find(test_patterns,test_phrase)
```

```
Searching the phrase using the re check: '\\d+'
['1233']
```

```
Searching the phrase using the re check: '\\D+'
['This is a string with some numbers ', ' and a symbol #hashtag']


Searching the phrase using the re check: '\\s+'
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']


Searching the phrase using the re check: '\\S+'
['This', 'is', 'a', 'string', 'with', 'some', 'numbers', '1233', 'and', 'a
', 'symbol', '#hashtag']


Searching the phrase using the re check: '\\w+'
['This', 'is', 'a', 'string', 'with', 'some', 'numbers', '1233', 'and', 'a
', 'symbol', 'hashtag']


Searching the phrase using the re check: '\\W+'
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' #']
```

Conclusion

You should now have a solid understanding of how to use the regular expression module in Python. There are a ton of more special character instances, but it would be unreasonable to go through every single use case. Instead take a look at the full documentation if you ever need to look up a particular pattern.

You can also check out the nice summary tables at this source.

Good job!

06-StringIO

April 9, 2019

1 StringIO Objects and the io Module

Back in **Lecture 24 - Files** we opened files that exist outside of python, and streamed their contents into an in-memory file object. You can also create in-memory file-like objects within your program that Python treats the same way. Text data is stored in a StringIO object, while binary data would be stored in a BytesIO object. This object can then be used as input or output to most functions that would expect a standard file object.

Let's investigate StringIO objects. The best way to show this is by example:

```
In [1]: import io

In [2]: # Arbitrary String
        message = 'This is just a normal string.'

In [3]: # Use StringIO method to set as file object
        f = io.StringIO(message)
```

Now we have an object f that we will be able to treat just like a file. For example:

```
In [4]: f.read()

Out[4]: 'This is just a normal string.'
```

We can also write to it:

```
In [5]: f.write(' Second line written to file like object')

Out[5]: 40

In [6]: # Reset cursor just like you would a file
        f.seek(0)

Out[6]: 0

In [7]: # Read again
        f.read()

Out[7]: 'This is just a normal string. Second line written to file like object'
```

```
In [8]: # Close the object when contents are no longer needed
        f.close()
```

Great! Now you've seen how we can use StringIO to turn normal strings into in-memory file objects in our code. This kind of action has various use cases, especially in web scraping cases where you want to read some string you scraped as a file.

For more info on StringIO check out the documentation: https://docs.python.org/3/library/io.html

01-Advanced Numbers

April 9, 2019

1 Advanced Numbers

In this lecture we will learn about a few more representations of numbers in Python.

1.1 Hexadecimal

Using the function hex() you can convert numbers into a hexadecimal format:

```
In [1]: hex(246)
```

```
Out[1]: '0xf6'
```

```
In [2]: hex(512)
```

```
Out[2]: '0x200'
```

1.2 Binary

Using the function bin() you can convert numbers into their binary format.

```
In [3]: bin(1234)
```

```
Out[3]: '0b10011010010'
```

```
In [4]: bin(128)
```

```
Out[4]: '0b10000000'
```

```
In [5]: bin(512)
```

```
Out[5]: '0b1000000000'
```

1.3 Exponentials

The function pow() takes two arguments, equivalent to `x^y`. With three arguments it is equivalent to `(x^y)%z`, but may be more efficient for long integers.

```
In [6]: pow(3,4)
```

```
Out[6]: 81
```

```
In [7]: pow(3,4,5)
```

```
Out[7]: 1
```

1.4 Absolute Value

The function abs() returns the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned.

```
In [8]: abs(-3.14)

Out[8]: 3.14

In [9]: abs(3)

Out[9]: 3
```

1.5 Round

The function round() will round a number to a given precision in decimal digits (default 0 digits). It does not convert integers to floats.

```
In [10]: round(3,2)

Out[10]: 3

In [11]: round(395,-2)

Out[11]: 400

In [12]: round(3.1415926535,2)

Out[12]: 3.14
```

Python has a built-in math library that is also useful to play around with in case you are ever in need of some mathematical operations. Explore the documentation here!

02-Advanced Strings

April 9, 2019

1 Advanced Strings

String objects have a variety of methods we can use to save time and add functionality. Let's explore some of them in this lecture:

```
In [1]: s = 'hello world'
```

1.1 Changing case

We can use methods to capitalize the first word of a string, or change the case of the entire string.

```
In [2]: # Capitalize first word in string
        s.capitalize()
```

```
Out[2]: 'Hello world'
```

```
In [3]: s.upper()
```

```
Out[3]: 'HELLO WORLD'
```

```
In [4]: s.lower()
```

```
Out[4]: 'hello world'
```

Remember, strings are immutable. None of the above methods change the string in place, they only return modified copies of the original string.

```
In [5]: s
```

```
Out[5]: 'hello world'
```

To change a string requires reassignment:

```
In [6]: s = s.upper()
        s
```

```
Out[6]: 'HELLO WORLD'
```

```
In [7]: s = s.lower()
        s
```

```
Out[7]: 'hello world'
```

1.2 Location and Counting

```
In [9]: s.count('o') # returns the number of occurrences, without overlap

Out[9]: 2

In [10]: s.find('o') # returns the starting index position of the first occurence

Out[10]: 4
```

1.3 Formatting

The center() method allows you to place your string 'centered' between a provided string with a certain length. Personally, I've never actually used this in code as it seems pretty esoteric...

```
In [11]: s.center(20,'z')

Out[11]: 'zzzzhello worldzzzzz'
```

The expandtabs() method will expand tab notations <̂/code> into spaces:

```
In [12]: 'hello\thi'.expandtabs()

Out[12]: 'hello   hi'
```

1.4 is check methods

These various methods below check if the string is some case. Let's explore them:

```
In [13]: s = 'hello'
```

isalnum() will return True if all characters in **s** are alphanumeric

```
In [14]: s.isalnum()

Out[14]: True
```

isalpha() will return True if all characters in **s** are alphabetic

```
In [15]: s.isalpha()

Out[15]: True
```

islower() will return True if all cased characters in **s** are lowercase and there is at least one cased character in **s**, False otherwise.

```
In [16]: s.islower()

Out[16]: True
```

isspace() will return True if all characters in **s** are whitespace.

```
In [17]: s.isspace()
```

```
Out[17]: False
```

istitle() will return True if **s** is a title cased string and there is at least one character in **s**, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. It returns False otherwise.

```
In [18]: s.istitle()
```

```
Out[18]: False
```

isupper() will return True if all cased characters in **s** are uppercase and there is at least one cased character in **s**, False otherwise.

```
In [19]: s.isupper()
```

```
Out[19]: False
```

Another method is endswith() which is essentially the same as a boolean check on s[-1]

```
In [20]: s.endswith('o')
```

```
Out[20]: True
```

1.5 Built-in Reg. Expressions

Strings have some built-in methods that can resemble regular expression operations. We can use split() to split the string at a certain element and return a list of the results. We can use partition() to return a tuple that includes the first occurrence of the separator sandwiched between the first half and the end half.

```
In [21]: s.split('e')
```

```
Out[21]: ['h', 'llo']
```

```
In [22]: s.partition('l')
```

```
Out[22]: ('he', 'l', 'lo')
```

Great! You should now feel comfortable using the variety of methods that are built-in string objects!

03-Advanced Sets

April 9, 2019

1 Advanced Sets

In this lecture we will learn about the various methods for sets that you may not have seen yet. We'll go over the basic ones you already know and then dive a little deeper.

```
In [1]: s = set()
```

2 add

add elements to a set. Remember, a set won't duplicate elements; it will only present them once (that's why it's called a set!)

```
In [2]: s.add(1)

In [3]: s.add(2)

In [4]: s

Out[4]: {1, 2}
```

2.1 clear

removes all elements from the set

```
In [5]: s.clear()

In [6]: s

Out[6]: set()
```

2.2 copy

returns a copy of the set. Note it is a copy, so changes to the original don't effect the copy.

```
In [7]: s = {1,2,3}
        sc = s.copy()

In [8]: sc
```

```
Out[8]: {1, 2, 3}

In [9]: s

Out[9]: {1, 2, 3}

In [10]: s.add(4)

In [11]: s

Out[11]: {1, 2, 3, 4}

In [12]: sc

Out[12]: {1, 2, 3}
```

2.3 difference

difference returns the difference of two or more sets. The syntax is:

```
set1.difference(set2)
```

For example:

```
In [13]: s.difference(sc)

Out[13]: {4}
```

2.4 difference_update

difference_update syntax is:

```
set1.difference_update(set2)
```

the method returns set1 after removing elements found in set2

```
In [14]: s1 = {1,2,3}

In [15]: s2 = {1,4,5}

In [16]: s1.difference_update(s2)

In [17]: s1

Out[17]: {2, 3}
```

2.5 discard

Removes an element from a set if it is a member. If the element is not a member, do nothing.

```
In [18]: s

Out[18]: {1, 2, 3, 4}

In [19]: s.discard(2)

In [20]: s

Out[20]: {1, 3, 4}
```

2.6 intersection and intersection_update

Returns the intersection of two or more sets as a new set.(i.e. elements that are common to all of the sets.)

```
In [21]: s1 = {1,2,3}

In [22]: s2 = {1,2,4}

In [23]: s1.intersection(s2)

Out[23]: {1, 2}

In [24]: s1

Out[24]: {1, 2, 3}
```

intersection_update will update a set with the intersection of itself and another.

```
In [25]: s1.intersection_update(s2)

In [26]: s1

Out[26]: {1, 2}
```

2.7 isdisjoint

This method will return True if two sets have a null intersection.

```
In [27]: s1 = {1,2}
         s2 = {1,2,4}
         s3 = {5}

In [28]: s1.isdisjoint(s2)

Out[28]: False

In [29]: s1.isdisjoint(s3)

Out[29]: True
```

2.8 issubset

This method reports whether another set contains this set.

```
In [30]: s1

Out[30]: {1, 2}

In [31]: s2

Out[31]: {1, 2, 4}

In [32]: s1.issubset(s2)

Out[32]: True
```

2.9 issuperset

This method will report whether this set contains another set.

```
In [33]: s2.issuperset(s1)

Out[33]: True

In [34]: s1.issuperset(s2)

Out[34]: False
```

2.10 symmetric_difference and symmetric_update

Return the symmetric difference of two sets as a new set.(i.e. all elements that are in exactly one of the sets.)

```
In [35]: s1

Out[35]: {1, 2}

In [36]: s2

Out[36]: {1, 2, 4}

In [37]: s1.symmetric_difference(s2)

Out[37]: {4}
```

2.11 union

Returns the union of two sets (i.e. all elements that are in either set.)

```
In [38]: s1.union(s2)

Out[38]: {1, 2, 4}
```

2.12 update

Update a set with the union of itself and others.

```
In [39]: s1.update(s2)

In [40]: s1

Out[40]: {1, 2, 4}
```

Great! You should now have a complete awareness of all the methods available to you for a set object type. This data structure is extremely useful and is underutilized by beginners, so try to keep it in mind!

Good Job!

04-Advanced Dictionaries

April 9, 2019

1 Advanced Dictionaries

Unlike some of the other Data Structures we've worked with, most of the really useful methods available to us in Dictionaries have already been explored throughout this course. Here we will touch on just a few more for good measure:

1.1 Dictionary Comprehensions

Just like List Comprehensions, Dictionary Data Types also support their own version of comprehension for quick creation. It is not as commonly used as List Comprehensions, but the syntax is:

```
In [1]: {x:x**2 for x in range(10)}
```

```
Out[1]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

One of the reasons it is not as common is the difficulty in structuring key names that are not based off the values.

1.2 Iteration over keys, values, and items

Dictionaries can be iterated over using the keys(), values() and items() methods. For example:

```
In [2]: d = {'k1':1,'k2':2}
```

```
In [3]: for k in d.keys():
            print(k)
```

```
k1
k2
```

```
In [4]: for v in d.values():
            print(v)
```

```
1
2
```

```
In [5]: for item in d.items():
            print(item)

('k1', 1)
('k2', 2)
```

2 Viewing keys, values and items

By themselves the keys(), values() and items() methods return a dictionary *view object*. This is not a separate list of items. Instead, the view is always tied to the original dictionary.

```
In [6]: key_view = d.keys()

        key_view

Out[6]: dict_keys(['k1', 'k2'])

In [7]: d['k3'] = 3

        d

Out[7]: {'k1': 1, 'k2': 2, 'k3': 3}

In [8]: key_view

Out[8]: dict_keys(['k1', 'k2', 'k3'])
```

Great! You should now feel very comfortable using the variety of methods available to you in Dictionaries!

05-Advanced Lists

April 9, 2019

1 Advanced Lists

In this series of lectures we will be diving a little deeper into all the methods available in a list object. These aren't officially "advanced" features, just methods that you wouldn't typically encounter without some additional exploring. It's pretty likely that you've already encountered some of these yourself!

Let's begin!

```
In [1]: list1 = [1,2,3]
```

1.1 append

You will definitely have used this method by now, which merely appends an element to the end of a list:

```
In [2]: list1.append(4)

        list1

Out[2]: [1, 2, 3, 4]
```

1.2 count

We discussed this during the methods lectures, but here it is again. count() takes in an element and returns the number of times it occurs in your list:

```
In [3]: list1.count(10)

Out[3]: 0

In [4]: list1.count(2)

Out[4]: 1
```

1.3 extend

Many times people find the difference between extend and append to be unclear. So note:
append: appends whole object at end:

```
In [5]: x = [1, 2, 3]
        x.append([4, 5])
        print(x)

[1, 2, 3, [4, 5]]
```

extend: extends list by appending elements from the iterable:

```
In [6]: x = [1, 2, 3]
        x.extend([4, 5])
        print(x)

[1, 2, 3, 4, 5]
```

Note how extend() appends each element from the passed-in list. That is the key difference.

1.4 index

index() will return the index of whatever element is placed as an argument. Note: If the the element is not in the list an error is raised.

```
In [7]: list1.index(2)

Out[7]: 1

In [8]: list1.index(12)

        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-8-56b94ada72bf> in <module>()
    ----> 1 list1.index(12)


        ValueError: 12 is not in list
```

1.5 insert

insert() takes in two arguments: insert(index,object) This method places the object at the index supplied. For example:

```
In [9]: list1

Out[9]: [1, 2, 3, 4]

In [10]: # Place a letter at the index 2
         list1.insert(2,'inserted')

In [11]: list1

Out[11]: [1, 2, 'inserted', 3, 4]
```

1.6 pop

You most likely have already seen pop(), which allows us to "pop" off the last element of a list. However, by passing an index position you can remove and return a specific element.

```
In [12]: ele = list1.pop(1)  # pop the second element

In [13]: list1

Out[13]: [1, 'inserted', 3, 4]

In [14]: ele

Out[14]: 2
```

1.7 remove

The remove() method removes the first occurrence of a value. For example:

```
In [15]: list1

Out[15]: [1, 'inserted', 3, 4]

In [16]: list1.remove('inserted')

In [17]: list1

Out[17]: [1, 3, 4]

In [18]: list2 = [1,2,3,4,3]

In [19]: list2.remove(3)

In [20]: list2

Out[20]: [1, 2, 4, 3]
```

1.8 reverse

As you might have guessed, reverse() reverses a list. Note this occurs in place! Meaning it affects your list permanently.

```
In [21]: list2.reverse()

In [22]: list2

Out[22]: [3, 4, 2, 1]
```

1.9 sort

The sort() method will sort your list in place:

```
In [23]: list2

Out[23]: [3, 4, 2, 1]

In [24]: list2.sort()

In [25]: list2

Out[25]: [1, 2, 3, 4]
```

The sort() method takes an optional argument for reverse sorting. Note this is different than simply reversing the order of items.

```
In [26]: list2.sort(reverse=True)

In [27]: list2

Out[27]: [4, 3, 2, 1]
```

1.10 Be Careful With Assignment!

A common programming mistake is to assume you can assign a modified list to a new variable. While this typically works with immutable objects like strings and tuples:

```
In [28]: x = 'hello world'

In [29]: y = x.upper()

In [30]: print(y)

HELLO WORLD
```

This will NOT work the same way with lists:

```
In [31]: x = [1,2,3]
```

```
In [32]: y = x.append(4)

In [33]: print(y)

None
```

What happened? In this case, since list methods like append() affect the list *in-place*, the operation returns a None value. This is what was passed to **y**. In order to retain **x** you would have to assign a *copy* of **x** to **y**, and then modify **y**:

```
In [34]: x = [1,2,3]
         y = x.copy()
         y.append(4)

In [35]: print(x)

[1, 2, 3]


In [36]: print(y)

[1, 2, 3, 4]
```

Great! You should now have an understanding of all the methods available for a list in Python!

06-Advanced Python Objects Test

April 9, 2019

1 Advanced Python Objects Test

1.1 Advanced Numbers

Problem 1: Convert 1024 to binary and hexadecimal representation

In [ ]:

Problem 2: Round 5.23222 to two decimal places

In [ ]:

1.2 Advanced Strings

Problem 3: Check if every letter in the string s is lower case

```
In [ ]: s = 'hello how are you Mary, are you feeling okay?'
```

Problem 4: How many times does the letter 'w' show up in the string below?

```
In [ ]: s = 'twywywtwywbwhsjhwuwshshwuwwwjdjdid'
```

1.3 Advanced Sets

Problem 5: Find the elements in set1 that are not in set2:

```
In [ ]: set1 = {2,3,1,5,6,8}
        set2 = {3,1,7,5,6,8}
```

Problem 6: Find all elements that are in either set:

In [ ]:

1.4 Advanced Dictionaries

Problem 7: Create this dictionary: {0: 0, 1: 1, 2: 8, 3: 27, 4: 64} using a dictionary comprehension.

In [ ]:

1.5 Advanced Lists

Problem 8: Reverse the list below:

```
In [ ]: list1 = [1,2,3,4]
```

Problem 9: Sort the list below:

```
In [ ]: list2 = [3,4,2,5,1]
```

2 Great Job!

07-Advanced Python Objects Test - Solutions

April 9, 2019

1 Advanced Python Objects Test

1.1 Advanced Numbers

Problem 1: Convert 1024 to binary and hexadecimal representation

```
In [1]: print(bin(1024))
        print(hex(1024))
```

```
0b10000000000
0x400
```

Problem 2: Round 5.23222 to two decimal places

```
In [2]: round(5.23222,2)
```

```
Out[2]: 5.23
```

1.2 Advanced Strings

Problem 3: Check if every letter in the string s is lower case

```
In [3]: s = 'hello how are you Mary, are you feeling okay?'

        s.islower()
```

```
Out[3]: False
```

Problem 4: How many times does the letter 'w' show up in the string below?

```
In [4]: s = 'twywywtwywbwhsjhwuwshshwuwwwjdjdid'
        s.count('w')
```

```
Out[4]: 12
```

1.3 Advanced

Problem 5: Find the elements in set1 that are not in set2:

```
In [5]: set1 = {2,3,1,5,6,8}
        set2 = {3,1,7,5,6,8}

        set1.difference(set2)
```

```
Out[5]: {2}
```

Problem 6: Find all elements that are in either set:

```
In [6]: set1.union(set2)
```

```
Out[6]: {1, 2, 3, 5, 6, 7, 8}
```

1.4 Advanced Dictionaries

Problem 7: Create this dictionary: {0: 0, 1: 1, 2: 8, 3: 27, 4: 64} using a dictionary comprehension.

```
In [7]: {x:x**3 for x in range(5)}
```

```
Out[7]: {0: 0, 1: 1, 2: 8, 3: 27, 4: 64}
```

1.5 Advanced Lists

Problem 8: Reverse the list below:

```
In [8]: list1 = [1,2,3,4]

        list1.reverse()

        list1
```

```
Out[8]: [4, 3, 2, 1]
```

Problem 9: Sort the list below:

```
In [9]: list2 = [3,4,2,5,1]

        list2.sort()

        list2
```

```
Out[9]: [1, 2, 3, 4, 5]
```

2 Great Job!

08-BONUS - With Statement Context Managers

April 9, 2019

1 With Statement Context Managers

When you open a file using `f = open('test.txt')`, the file stays open until you specifically call `f.close()`. Should an exception be raised while working with the file, it remains open. This can lead to vulnerabilities in your code, and inefficient use of resources.

A context manager handles the opening and closing of resources, and provides a built-in `try/finally` block should any exceptions occur.

The best way to demonstrate this is with an example.

1.0.1 Standard `open()` procedure, with a raised exception:

```
In [1]: p = open('oops.txt','a')
        p.readlines()
        p.close()


        ---------------------------------------------------------------------------

        UnsupportedOperation                      Traceback (most recent call last)

        <ipython-input-1-ad7a2000735b> in <module>()
          1 p = open('oops.txt','a')
    ----> 2 p.readlines()
          3 p.close()


        UnsupportedOperation: not readable
```

Let's see if we can modify our file:

```
In [2]: p.write('add more text')
```

```
Out[2]: 13
```

Ouch! I may not have wanted to do that until I traced the exception! Unfortunately, the exception prevented the last line, `p.close()` from running. Let's close the file manually:

```
In [3]: p.close()
```

1.0.2 Protect the file with `try/except/finally`

A common workaround is to insert a `try/except/finally` clause to close the file whenever an exception is raised:

```
In [4]: p = open('oops.txt','a')
        try:
            p.readlines()
        except:
            print('An exception was raised!')
        finally:
            p.close()
```

```
An exception was raised!
```

Let's see if we can modify our file this time:

```
In [5]: p.write('add more text')
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-5-1209a18e617d> in <module>()
    ----> 1 p.write('add more text')


        ValueError: I/O operation on closed file.
```

Excellent! Our file is safe.

1.0.3 Save steps with `with`

Now we'll employ our context manager. The syntax follows `with [resource] as [target]: do something`

```
In [6]: with open('oops.txt','a') as p:
            p.readlines()
```

```
        ---------------------------------------------------------------------------

        UnsupportedOperation                      Traceback (most recent call last)

        <ipython-input-6-7ccc44e332f9> in <module>()
          1 with open('oops.txt','a') as p:
    ----> 2     p.readlines()
```

```
UnsupportedOperation: not readable
```

Can we modify the file?

```
In [7]: p.write('add more text')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-7-1209a18e617d> in <module>()
----> 1 p.write('add more text')

ValueError: I/O operation on closed file.
```

Great! With just one line of code we've handled opening the file, enclosing our code in a `try/finally` block, and closing our file all at the same time.

Now you should have a basic understanding of context managers.

01-Advanced Object Oriented Programming

April 9, 2019

1 Advanced Object Oriented Programming

In the regular section on Object Oriented Programming (OOP) we covered:

- Using the *class* keyword to define object classes
- Creating class attributes
- Creating class methods
- Inheritance - where derived classes can inherit attributes and methods from a base class
- Polymorphism - where different object classes that share the same method can be called from the same place
- Special Methods for classes like `__init__`, `__str__`, `__len__` and `__del__`

In this section we'll dive deeper into * Multiple Inheritance * The `self` keyword * Method Resolution Order (MRO) * Python's built-in `super()` function

1.1 Inheritance Revisited

Recall that with Inheritance, one or more derived classes can inherit attributes and methods from a base class. This reduces duplication, and means that any changes made to the base class will automatically translate to derived classes. As a review:

```
In [1]: class Animal:
            def __init__(self, name):    # Constructor of the class
                self.name = name

            def speak(self):              # Abstract method, defined by convention only
                raise NotImplementedError("Subclass must implement abstract method")


        class Dog(Animal):
            def speak(self):
                return self.name+' says Woof!'

        class Cat(Animal):
            def speak(self):
                return self.name+' says Meow!'
```

```
        fido = Dog('Fido')
        isis = Cat('Isis')

        print(fido.speak())
        print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

In this example, the derived classes did not need their own `__init__` methods because the base class `__init__` gets called automatically. However, if you do define an `__init__` in the derived class, this will override the base:

```
In [2]: class Animal:
            def __init__(self,name,legs):
                self.name = name
                self.legs = legs

        class Bear(Animal):
            def __init__(self,name,legs=4,hibernate='yes'):
                self.name = name
                self.legs = legs
                self.hibernate = hibernate
```

This is inefficient - why inherit from Animal if we can't use its constructor? The answer is to call the Animal `__init__` inside our own `__init__`.

```
In [3]: class Animal:
            def __init__(self,name,legs):
                self.name = name
                self.legs = legs

        class Bear(Animal):
            def __init__(self,name,legs=4,hibernate='yes'):
                Animal.__init__(self,name,legs)
                self.hibernate = hibernate

        yogi = Bear('Yogi')
        print(yogi.name)
        print(yogi.legs)
        print(yogi.hibernate)
```

```
Yogi
4
yes
```

1.2 Multiple Inheritance

Sometimes it makes sense for a derived class to inherit qualities from two or more base classes. Python allows for this with multiple inheritance.

```
In [4]: class Car:
            def __init__(self,wheels=4):
                self.wheels = wheels
                # We'll say that all cars, no matter their engine, have four wheels by default

        class Gasoline(Car):
            def __init__(self,engine='Gasoline',tank_cap=20):
                Car.__init__(self)
                self.engine = engine
                self.tank_cap = tank_cap # represents fuel tank capacity in gallons
                self.tank = 0

            def refuel(self):
                self.tank = self.tank_cap


        class Electric(Car):
            def __init__(self,engine='Electric',kWh_cap=60):
                Car.__init__(self)
                self.engine = engine
                self.kWh_cap = kWh_cap # represents battery capacity in kilowatt-hours
                self.kWh = 0

            def recharge(self):
                self.kWh = self.kWh_cap
```

So what happens if we have an object that shares properties of both Gasolines and Electrics? We can create a derived class that inherits from both!

```
In [5]: class Hybrid(Gasoline, Electric):
            def __init__(self,engine='Hybrid',tank_cap=11,kWh_cap=5):
                Gasoline.__init__(self,engine,tank_cap)
                Electric.__init__(self,engine,kWh_cap)


        prius = Hybrid()
        print(prius.tank)
        print(prius.kWh)
```

```
0
0
```

```
In [6]: prius.recharge()
        print(prius.kWh)
```

```
5
```

1.3 Why do we use `self`?

We've seen the word "self" show up in almost every example. What's the deal? The answer is, Python uses `self` to find the right set of attributes and methods to apply to an object. When we say:

```
prius.recharge()
```

What really happens is that Python first looks up the class belonging to `prius` (Hybrid), and then passes `prius` to the `Hybrid.recharge()` method.

It's the same as running:

```
Hybrid.recharge(prius)
```

but shorter and more intuitive!

1.4 Method Resolution Order (MRO)

Things get complicated when you have several base classes and levels of inheritance. This is resolved using Method Resolution Order - a formal plan that Python follows when running object methods.

To illustrate, if classes B and C each derive from A, and class D derives from both B and C, which class is "first in line" when a method is called on D?Consider the following:

```
In [7]: class A:
            num = 4

        class B(A):
            pass

        class C(A):
            num = 5

        class D(B,C):
            pass
```

Schematically, the relationship looks like this:

```
     A
   num=4
  /     \
 /       \
 B       C
pass    num=5
 \       /
  \     /
     D
    pass
```

Here `num` is a class attribute belonging to all four classes. So what happens if we call `D.num`?

```
In [8]: D.num

Out[8]: 5
```

You would think that `D.num` would follow `B` up to `A` and return **4**. Instead, Python obeys the first method in the chain that *defines* num. The order followed is `[D, B, C, A, object]` where *object* is Python's base object class.

In our example, the first class to define and/or override a previously defined `num` is `C`.

1.5 `super()`

Python's built-in `super()` function provides a shortcut for calling base classes, because it automatically follows Method Resolution Order.

In its simplest form with single inheritance, `super()` can be used in place of the base class name :

```
In [9]: class MyBaseClass:
            def __init__(self,x,y):
                self.x = x
                self.y = y

        class MyDerivedClass(MyBaseClass):
            def __init__(self,x,y,z):
                super().__init__(x,y)
                self.z = z
```

Note that we don't pass `self` to `super().__init__()` as `super()` handles this automatically.

In a more dynamic form, with multiple inheritance like the "diamond diagram" shown above, `super()` can be used to properly manage method definitions:

```
In [10]: class A:
             def truth(self):
                 return 'All numbers are even'

         class B(A):
             pass

         class C(A):
             def truth(self):
                 return 'Some numbers are even'

In [11]: class D(B,C):
             def truth(self,num):
                 if num%2 == 0:
                     return A.truth(self)
                 else:
                     return super().truth()
```

```
d = D()
d.truth(6)
```

```
Out[11]: 'All numbers are even'
```

```
In [12]: d.truth(5)
```

```
Out[12]: 'Some numbers are even'
```

In the above example, if we pass an even number to `d.truth()`, we'll believe the `A` version of `.truth()` and run with it. Otherwise, follow the MRO and return the more general case.

For more information on `super()` visit https://docs.python.org/3/library/functions.html#super and https://rhettinger.wordpress.com/2011/05/26/super-considered-super/

Great! Now you should have a much deeper understanding of Object Oriented Programming!