WELCOME BACK!
Questions?

Just two weeks left in our class!
<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Flow Control</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>print()</td>
<td>branching</td>
<td>def</td>
</tr>
<tr>
<td>float</td>
<td>type()</td>
<td>loops</td>
<td>return</td>
</tr>
<tr>
<td>string</td>
<td>help()</td>
<td>break</td>
<td>None</td>
</tr>
<tr>
<td>bool</td>
<td>min()/max()</td>
<td>continue</td>
<td>import</td>
</tr>
<tr>
<td></td>
<td>int()/float()</td>
<td></td>
<td>if/elif/else</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td></td>
<td>True/False</td>
</tr>
<tr>
<td></td>
<td>len()</td>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>Input()</td>
<td></td>
<td>for</td>
</tr>
<tr>
<td></td>
<td>&lt;user defined&gt;</td>
<td></td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>&lt;turtle functions&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;random functions&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We're ready to explore our first data structure: the list.

Python has 4 built-in data structures:

- Lists `[]`
- Tuples `()`
- Sets `{}`
- Dicts `{a:b}`

Lists are *heterogenous mutable* collections of *ordered* data
Tuples are *heterogenous immutable* collections of *ordered* data
Sets are *heterogenous mutable* collections of *unordered unique* data
Dicts are *heterogenous mutable unordered mappings* of *unique keys to values*
What do these terms mean?

*heterogenous*- can have different *types* of data in them (strings, ints, floats, lists, bools…all in the same list if you want)

*mutable*- talk about this later (we need to spend some time on this, it's a biggy)

*ordered*- objects added to the list in a certain order stay in the same order when the list is examined
Let's say you wanted to write a function to check passwords against a list of too common passwords.

```python
def pswrd_check(password):
    if password == "password":
        return False
    elif password == "abc123":
        return False
    elif password == "":
        return False
    ...

    else:
        return True
```

How do we update this code to account for a new common password?
Here's a much better bit of code

```python
common_password = ["abc123", "password", "]

def pswrd_check(password):
    if password in common_passwords:
        return False
    else:
        return True

def update_commons(new_password, add_or_subtract):
    if add_or_subtract:
        common_passwords.append(password)
    else:
        common_passwords.remove(password)

Much more compact and easy to update.
```
List Basics

an empty list can be created by using the [] brackets like this:

```
myLst = []
```

similarly a list can also be created with elements in it already:

```
myLst = [1, "two", 3]
```

notice elements are separated by commas

Having created a list we can add elements to it with append()

```
myLst.append("four")
```

insert() allows you to place an item at a certain place in a list

elements in a list have positions and can be retrieved from the list with them:

```
myLst[1]
```

will grab the second element from a list (remember zero indexing!)

Note- this is exactly the slicing we used with strings. You can do all the same tricks (negative numbers, slicing out sequences of elements).
List Basics - keywords

+ concatenates two lists
myLst = [1] + ["two", True]
+= works as expected, - does not work with lists

* works as with strings, repeating a list an integer number of times
myLst * 2
*= works as expected, / does not work with lists

len() is a built in function that returns the number of elements in the list
len(myLst)

min and max are built in functions that return the smallest and largest elements in the list using > and < operations
min(myLst)
max(myLst)
List Basics - keywords

```
in tests if any element of the list matches the criteria
"two" in myLst

del() destroys a list
del(myLst)

sum() will add up all elements of a list using + operators
sum(myList)

count() is a list function that counts the number of instances of the argument in the list
myLst.count("two")

index() is a list function that finds the index of the first instance of the argument in the list
myLst.count(True)
```
WELCOME BACK!
Questions?
<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Flow Control</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>print()</td>
<td>branching</td>
<td>def</td>
</tr>
<tr>
<td>float</td>
<td>type()</td>
<td>loops</td>
<td>return</td>
</tr>
<tr>
<td>string</td>
<td>help()</td>
<td>return</td>
<td>None</td>
</tr>
<tr>
<td>bool</td>
<td>min()/max()</td>
<td>break</td>
<td>import</td>
</tr>
<tr>
<td>list</td>
<td>int()/float()</td>
<td>continue</td>
<td>if/elif/else</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td></td>
<td>True/False</td>
</tr>
<tr>
<td></td>
<td>len()</td>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>Input()</td>
<td></td>
<td>for</td>
</tr>
<tr>
<td></td>
<td>&lt;user defined&gt;</td>
<td></td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>&lt;turtle functions&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;random functions&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lists are Different

Lists work differently than everything else we've dealt with so far because they are mutable, which means they can change in place.

…which means what?
Remember before we said that variables don't change value unless you have an assignment statement?

i.e.
\[
x = 3 \\
x + 1 \\
\text{print}(x)
\]

prints out 3 not 4
This was true for ints, floats, strings, bools…

but this isn't (always) true for lists!
ints, floats, strings, bools are all immutable, which means they can't change.

but that's stupid, of course they can change:

```python
x = "foo"
x += "bar"
print(x)
```

right? Didn't we just change a string from "foo" to "foobar"?

Well...no.
Look under the hood

Remember the id() function? Here's where it's really pretty useful.

i.e.

\[
\begin{align*}
x &= 1000000 \\
id(x) \\
x &+= 1 \\
id(x)
\end{align*}
\]

We didn't change the value of the int that \(x\) points toward. What we did was store a new int and made \(x\) point at it. The new int lives at a very different address from the old int.

Same thing with strings and floats.
Hey a rabbit hole!

Try the same thing with a list.

i.e.
\[
x = [1] \\
id(x) \\
x += [2] \\
id(x)
\]

The lists are in the same place because we didn't create a new list, we changed the old list \textit{in place} (i.e. where it was).
Great, who cares?

Here's (one reason) why it matters.

```python
x = [1]
y = x
x += [2]
print(x)
print(y)
```
Great, who cares?

Here's (one reason) why it matters.

```python
x = [1]
y = x
x += [2]
print(x)
print(y)
```

When we set `y = x` we didn't create a new copy of the list, we just pointed `y` at the same copy. So when we modified `x` we modified `y`.

Completely different than

```python
x = 1
y = x
x += 1
print(x)
print(y)
```
Why don't we just make a new copy of the list like we do with ints and floats, strings, and bools?

Lists are often very large, and take lots of memory. You don't make copies of them unless you need to.

BTW if you do need to you can with either of these commands:

\[
y = x.copy()
\]
or
\[
y = x[:]
\]
(slicing creates a new list)
So you need to understand what operations create new lists and which operations change a list in place.

concatenation creates a new list:

```python
x = []
y = x
x + [1]
print(x)
print(y)
```

meanwhile append() modifies a list in place:

```python
x = []
y = x
x .append(1)
print(x)
print(x)
print(y)
```
Spot the error

What will this give us?

```python
x = [1, "two"]
x = x.append(3)
print(x)
```
Spot the error

What will this give us?

```python
x = [1, "two"]
x = x.append(3)
print(x)
```

append() changes a list in place, and hence it returns None. When you set `x = x.append(3)` you nuke your list and replace it with the value None.

You probably didn't want to do that.
Good to remember

• if a function sorts in-place then it will return None (because it did its work on the list itself)
• similarly, if a function does not work on the list in-place then it will return a new list (because otherwise you couldn't use it)
Two ways to sort a list:

```python
myLst = [2, 1, 6, 0]
#method 1
print( sorted(myLst) )
print( myLst )
#method 2
myLst.sort()
print(myLst)
```

notice one of these returns a new list and one sorts in place.
WELCOME BACK!
Questions?
# Where We Are

<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Flow Control</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>print()</td>
<td>branching</td>
<td>def</td>
</tr>
<tr>
<td>float</td>
<td>type()</td>
<td>loops</td>
<td>return</td>
</tr>
<tr>
<td>string</td>
<td>help()</td>
<td>return</td>
<td>None</td>
</tr>
<tr>
<td>bool</td>
<td>min() / max()</td>
<td>break</td>
<td>import</td>
</tr>
<tr>
<td>list</td>
<td>int() / float()</td>
<td>continue</td>
<td>if / elif / else</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td></td>
<td>True / False</td>
</tr>
<tr>
<td></td>
<td>len()</td>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>Input()</td>
<td></td>
<td>for</td>
</tr>
<tr>
<td></td>
<td>&lt;user defined&gt;</td>
<td></td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>&lt;turtle functions&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;random functions&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We covered the basics of lists Monday and the hazards or tricky aspects yesterday. Today we do some wrap up on a few different concepts related to lists.
Split method

You can make lists out of strings quite easily with the string split() method

```python
x = "Hello, my name is Jason!"
y = x.split()
print(y)
```

So that's kind of cool.
x = "Hello, my name is Jason!"
y = x.split()
print(y)

What's more by splitting the string up into pieces and storing them in a list it becomes easy to then loop through the pieces

count = 0
for item in y:
    if item.islower():
        count += 1
print(count)
We don't have to split on white space.

```python
x = "Hello, my name is Jason!"
y = x.split("","
print(y)
```

splitting on commas is particularly useful when you are looking at .csv files (comma separated values)

We'll talk tomorrow about handling text files including csv files with Python.
What if we wanted to double each item of a list? Will this work:

```python
example = [1, 2, 3, 4]
for item in example:
    item = item * 2
```

let's look at it in the visualizer…
Changing values

What if we wanted to double each item of a list? Will this work:

```python
example = [1, 2, 3, 4]
for item in example:
    item = item * 2
```

Well that sucked.

What about this:

```python
for i in range(len(example)):
    example[i] = example[i] * 2
# or
#example *= 2
```
I want to write the ultimate popular sci-fi show but I'm not that creative, so I've written a program to help me come up with a title based on previous examples.

```python
def mashup():
    firsts = ['fire', 'battle', 'star', 'far']
    seconds = ['wars', 'trek', 'star', 'fly', 'scape']

    for afirst in firsts:
        for asecond in seconds:
            print(afirst + asecond)
```
I have a list of names and I want to just get first initials. How hard is that?

```python
def initials_from_list(alist):
    for i in range(len(alist)):
        print(alist[i][0])
```
I have a list of names and I want to just get first initials, but now the list consists of first names followed by a space and then the last name all as a single string.
I have a list of names and I want to just get first initials, but now the list consists of first names followed by a space and then the last name all as a single string.

What if instead of printing we want to return a list of the initials?
WELCOME BACK!
Questions?
<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Flow Control</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>print()</td>
<td>branching</td>
<td>def</td>
</tr>
<tr>
<td>float</td>
<td>type()</td>
<td>loops</td>
<td>return</td>
</tr>
<tr>
<td>string</td>
<td>help()</td>
<td>return</td>
<td>None</td>
</tr>
<tr>
<td>bool</td>
<td>min() / max()</td>
<td>break</td>
<td>import</td>
</tr>
<tr>
<td>list</td>
<td>int() / float()</td>
<td>continue</td>
<td>if / elif / else</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td></td>
<td>True / False</td>
</tr>
<tr>
<td></td>
<td>len()</td>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>Input()</td>
<td></td>
<td>for</td>
</tr>
<tr>
<td></td>
<td>&lt;user defined&gt;</td>
<td></td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>&lt;turtle functions&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;random functions&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
One big thing we haven't looked at is input/output in terms of files.

Guess what? Python makes this easy, at least for text files.
Reading Files

We can easily open a file for reading like this:

```
fin = open("afile.txt", "r")
```

"afile.txt" is the name of the file to be opened. "r" indicates we are opening the file in read only mode.

What is the type of fin?
We can easily open a file for reading like this:

```python
fin = open("afile.txt", "r")
```

"afile.txt" is the name of the file to be opened. "r" indicates we are opening the file in read only mode.

To actually read the file we have several options.

```python
fin.read()
fin.readline()
fin.readlines()
```
Pointers

To understand how files are read you have to understand that Python maintains a pointer that keeps track of where you are in a file. To begin with this is set at the start of the file.
Reading

fin.read()  # reads the entire file into a single txt file

fin.readline()  # reads 1 line from the file moving the pointer forward

fin.readlines()  # reads every line and puts each as a separate string into a list

all of them can be useful but I find readlines() the most useful.
Opening a file for writing is easy too:

fout = open("afile.txt", "w")

or

fout = open("afile.txt", "a")

"afile.txt" is the name of the file to be opened. "w" indicates we are opening the file in write mode. The pointer is at the start of the file. "a" indicates we are opening the file in append mode. The pointer is at the end of the file.
Once the file is open writing is as easy as:

```python
text = "This is some text to write"
fout.write(text)
```

or

```python
txt = "Here's some text"
fout.write(txt)
```

and then

```python
fout.close()
```

`close()`, *shocker*, closes the file and makes the changes.
Let's Write Poetry

The other day upon the stair
I saw a man who wasn't there
He wasn't there again today
Oh, how I wish he'd go away

Let's get python to write our poem into a txt file called "python_poetry.txt"
The other day upon the stair
I saw a man who wasn't there
He wasn't there again today
Oh, how I wish he'd go away

Let's get python to write our poem into a txt file called "python_poetry.txt"

Now let's get python to read the poetry, except let's make it SHOUT.
Why Extra Spaces?

each line had a "\n" new line character at the end so print added a return, but print() also has a built in "\n" at the end of every print, so we end up with doubled returns.

We can fix this with the strip() function for strings.
Where'd It Save the File?

If no file path is supplied then it saves in the default location, the same place your .py files are saved.

In addition to the name you can provide a filepath as such:

```python
fin = open("C:\Documents\Cool_Python_Stuff\python_poetry.txt", "r")
```

often I do something like this:

```python
filepath = "C:\Documents\Cool_Python_Stuff\"
fin = open(filepath + "python_poetry.txt", "r")
```