Administrivia

- Checkoff queue procedure:
  - We read your feedback and understand that there is often logistical difficulties when LAs are calling names off the queue. We're going to need your help to make this process better. First, when you sign up for the queue, please make sure your names are numbered so we know where to start calling names. Second, when your name is one of the next 5 to be called in either the help queue or the checkoff queue, please go to the “A” cubicle area so that LAs know where to search for you. After that, you will be able to move to a different location so that area does not become too crowded.

- Interview 2 will be offered on Wednesday. You will be able to complete it without Thursday and Friday’s lecture. We are not expecting to extend the Interview 2 deadline past Friday 5PM. Guidelines will be posted “tonight”.

- If you haven’t filled out the Week 1 feedback form, please do so. We are reading every response and will try to address them ASAP. bit.ly/6149-feedback1

Loops review

For loop review
Up to this point, we’ve used for loop to iterate over a list or a string. For example:

```
email_addresses = ["mszucs@mit.edu", "jessk@mit.edu", "knjohnso@mit.edu"]
for email in email_addresses:
    print email
```

or we can iterate over the indices of the list

```
for email_index in range(len(email_addresses)):
    print email_addresses[email_index]
```

While loop review
We can similarly use a while loop to iterate over a list or string, but the process is less intuitive. Remember, we use while loops to execute a block of code while a certain condition is True. Therefore, we can translate the our for loop that iterated over a list’s indices into a while loop that runs until the index we are at is out of bounds for our list.

```
email_index = 0
```
while email_index < len(email_addresses):
    print email_addresses[email_index]
    email_index += 1

Nested loops
Sometimes, we want to write loops within loops. Say we have the following information:

work_email_addresses = ["mszucs@mit.edu", "jessk@mit.edu", "knjohnso@mit.edu"]
friend_email_addresses = ["neibloom@mit.edu", "npatki@mit.edu", "liberamental@gmail.com"]
my_contacts = [work_email_addresses, friend_email_addresses]
print my_contacts # a list of lists

I can print each element of the list my_emails, which is just another list

for email_list in my_contacts:
    print email_list # a list of email addresses

I can also print the individual emails in each list

for email_list in my_contacts:
    for email_address in email_list:
        print email_address # one address in a list of addresses

Side note: a common pattern to create nested for loops:

nested_list = []
for i in range(10):
    nested_list.append([])
    for j in range(5):
        nested_list[i].append("i = " + str(i) + ", j = " + str(j))
print nested_list

A prettier print:
for i in range(len(nested_list)):
    print nested_list[i]

This may be useful later for our second project, or you may choose to use a different data structure that we will talk about in a few minutes.

**Flow control**
Sometimes you don’t want to iterate through everything in a list.

Maybe after some condition, you want to stop the loop. So let’s take my_contacts and try to see if there contains @mit.edu email address.

for email_list in my_contacts:
    for email_address in email_list:
        if email_address[-8:] == "@mit.edu":
            print "found an MIT email address!"

We printed that we found an MIT email address every time we found one. What if we only care if we found one MIT email address per email list?

for email_list in my_contacts:
    for email_address in email_list:
        if email_address[-8:] == "@mit.edu":
            print "found an MIT email address! in list",
        email_list
        break

Let’s say we have weird inputs and only want to check the last 8 characters if the email has 8 characters. Otherwise, we will skip to the next iteration of the loop

e-mail_list[0].insert(0,"harlin")
for email_list in my_contacts:
    for email_address in email_list:
        if len(email_address) < 8:
            continue
        if email_address[-8:] == "@mit.edu":
            print "found an MIT email address! in list",
e-mail_list
In summary:
break takes you out of the innermost loop only
continue stops running the current iteration of the loop and goes to the beginning of the loop for the next iteration.

Dictionaries
Let's take a look at our nested list of classes from earlier:

```python
work_email_addresses = ["mszucs@mit.edu", "jessk@mit.edu", "knjohnso@mit.edu"]
friend_email_addresses = ["neibloom@mit.edu", "npatki@mit.edu", "liberamental@gmail.com"]
my_contacts = [work_email_addresses, friend_email_addresses]
```

Are lists the best way we can represent this data? It doesn't seem that intuitive for us to use `work_emails[0]` to get Michelle's email address. Rather, it would be awesome if we could access the data using "Michelle". We can do so using a different data structure. Instead of a nested list, let's use a dictionary.

Think of a dictionary as a phone book or (to be more modern) your contacts in your phone/email. So for example, I might need to email Michelle. I can do so by searching for "Michelle" in my contacts and next to her name, I will find her contact information, mszucs@mit.edu. The syntax for this:

```python
contacts = {}  # can initialize an empty dictionary
contacts = {"Michelle": "mszucs@mit.edu", "Jessica": "jessk@mit.edu", "Katy": "knjohnso@mit.edu"}
print contacts["Michelle"]
```

We mapped keys to values. For example, key Michelle maps to `mszucs@mit.edu`. Say we hired another member of the 6.149 staff. We can add their contact information to the my_contacts dictionary as well:

```python
contacts["Tomas"] = "tomascg@mit.edu"
```
What happens we try to add Jessica Q (jessq)?

```python
print contacts[“Jessica”]
contacts[“Jessica”] = “jessq@mit.edu”
print contacts[“Jessica”]
print contacts
```

Keys must be unique in dictionaries! So maybe instead we would have:

```python
contacts[“Jessica K”] = “jessk@mit.edu”
contacts[“Jessica Q”] = “jessq@mit.edu”
```

To avoid confusion, let’s delete “Jessica” from our contacts using the del keyword

```python
del contacts[“Jessica”]
```

But Jessica K also goes by “Jess”. While all keys in a dictionary must be unique, not all values have to be.

```python
contacts[“Jess”] = “jessk@mit.edu”
print contacts
```

What happens when we try to access “Louis”?

```python
print contacts[“Louis”]
```

Side note: may be useful in the future or for making code more concise

Dictionaries have a get method, where you can try to access a particular key, and if the key does not exist, the get method will return None or a pre-specified value.

```python
print contacts.get(“Louis”)
print contacts.get(“Louis”, “cannot find this person’s email address”)
```

Are dictionaries mutable? Absolutely. We’ve been changing a dictionary this whole time.

Mutability revisited: A variable is just a name.

An assignment associated the name with the thing on the right hand side of the parentheses.

An object can have one, multiple, or no names associated with it.

name can be associated with a function name, just like any other variable

aliasing = 2 paths to the same object - multiple ways to mutate an object, and changes will be visible to both paths.
Because dictionaries are mutable, we need to be careful about aliasing. In English, alias is defined as “a false or assumed identity,” so a spy may adopt an alias (another name). Similarly in Python, we can assign one object multiple (variable) names. Having a different name does not change what the object is.

```python
aliased_contacts = contacts
aliased_contacts["Louis"] = "loulamia@mit.edu"
print aliased_contacts
print contacts
```

How do we fix this? To avoid aliasing lists, instead of saying l2 = l1, we said l2 equals a copy of l1, using [:]. Similarly, we want to create a copy of the dictionary and dictionaries happen to have a built-in copy function:

```python
contacts_copy = contacts.copy()
contacts_copy["Harlin"] = "harlin@mit.edu"
print contacts_copy
print contacts
```

Side note for an extra challenge: .copy() will almost always be sufficient for the material presented in this class. However, it creates what we call a shallow copy. Sometimes, we may need to use something called a deep copy. Take this example:

```python
email_addresses = {"work": ["mszucs@mit.edu", "jessk@mit.edu", "knjohnso@mit.edu"], "friends": ["neibloom@mit.edu", "npatki@mit.edu", "liberamental@gmail.com"]}
shallow_copy = email_addresses.copy()
from copy import deepcopy
deep_copy = deepcopy(email_addresses)

print shallow_copy["work"]
shallow_copy["work"].append("harlin@mit.edu")
print shallow_copy
print email_addresses

print deep_copy
deep_copy["work"].append("jessq@mit.edu")
print deep_copy
print shallow_copy
```
print email_addresses

We can iterate through dictionaries using the dictionary method .keys(), which returns a list of the dictionary’s keys.

    for name in my_contacts.keys():
        print my_contacts[name]

We can alternatively do:

    for name in my_contacts:
        print my_contacts[name]

One final note about dictionaries: what can the keys be? So far, I’ve only been using strings. We can use integers, floats. What happens when we try to use a list? We can’t use a list because it is “unhashable” (which means it can’t be converted to a constant, unique value) - that’s a side effect of being mutable. Instead of using a list, we can instead use a tuple.

**Tuples**
Tuples are, for our purposes right now, immutable lists. We create them using parentheses, not square brackets, and because they are tuple objects, not list objects, we cannot use list methods such as .append(), .extend(), etc.

    my_tuple = (1, 2)
    my_tupe[0] = 0  # doesn’t work
    my_tuple = (0, 2)  # have to create a new one

Tuples are commonly used for easy, efficient variable assignment; you can omit the parentheses when doing assignment with tuples.

    a, b = 5, “hello”

They can also be used for swapping values. The hard way:

    temp = a
    a = b
    b = temp

    The easy way:
    a, b = b, a  # swap values of a and b
And of course because they are immutable, we can use them as dictionary keys:

```python
some_dict = {}
some_dict[(a, b)] = “some value”
```

Finally, we can use tuples when we want a function to return multiple items by storing them in a tuple. Just as we used tuples above to assign multiple variables to values at once earlier, we can use them to assign multiple variables to the output of a function.

We can also iterate over tuples similarly to how we iterate over a list.

```python
tup = (1, 2, 3)
for item in tup:
    print item

for index in range(len(tup)):
    print tup[index]
```

**Functions**

**REVIEW:**

If element is less than 10, multiply it by two.

```python
def modify_list(my_list):
    for index in range(len(my_list)):
        if my_list[index] < 10:
            my_list[index] *= 2
    print my_list
```

How we call the function:

```python
modify_list([1, 11, 21, 31, 41])
print modify_list
```

Let’s change print to return and see what happens.

```python
so if we have print and try to do
greater_than_10()
greater_than_10() = greater_than_10()
print filtered_list
```
Try again with return. Can you see some similarities to what we expect .append() and .extend() would do?

Scope notes: https://gist.github.com/jasonpr59/6606e7cce706010e1814