Announcements

- **Project**
  - If you haven’t yet uploaded to Canvas your final project proposal/idea, do that soon. You’ll receive partial credit.
  - Pseudocode is due on Monday.
    - Q: How “much” pseudocode must you submit?
    - A: Enough to demonstrate that you’ve started thinking about the high-level structure of your program.

- **Schedule for remainder of Quarter**
  - Monday, 7 March : last “real” lecture
  - Tuesday and Wednesday, 8 and 9 March : no lab; work on your final project
  - Wednesday, 9 March : sample final exam solutions presented in lecture
  - Friday, 11 March : no lecture
  - Sunday, 13 March : extra review session, CF125, 5-6pm

- **Final Exam**
  - 14 March (Monday), 10:30am-12:30pm, THIS room
Repetition of Lists behaves differently than repetition of Strings.

```
aList = [“prancer”, “vixen”]
aNewList = [aList] * 3
```

String

“prancer”

String

“vixen”

Repetition of a List means repetition of the reference variables that make up the list’s entries.
From Last Time

Repetition of Lists behaves differently than repetition of Strings.

\[ \text{aList} = \left[ \text{“prancer”}, \text{“vixen”} \right] \]
\[ \text{aNewList} = \left[ \text{aList} \right] \times 3 \]

When one of the entries of \text{aList} is updated to point to a “new” String object …

That has an indirect effect on the “contents” of \text{aNewList}, because \text{aNewList} points to what \text{aList} points to.

\[ \text{aList}[0] = \text{“rudolph”} \]
From Last Time

Repetition of Lists behaves differently than repetition of Strings.

```python
String
"rudolph"

aList = ["prancer", "vixen"]
aNewList = [aList] * 3

String
"vixen"

String
"prancer"

String
"rudolph"

Printing the contents of aNewList will print the contents of the list that each element of aNewList points to.

aList[0] = "rudolph"
print(aNewList)

[['rudolph','vixen'], ['rudolph','vixen'], ['rudolph','vixen']]```
Lists concatenation and appending
join
Iterating over a List’s elements

Iterating over a list’s elements works the same way as it does for strings
Iterating over a List’s elements

Iterating over a list’s elements works the same way as it does for strings

Q: For the list shown in blue, do the code snippets in yellow and green print to the screen the same thing?

```
springMonthNums = [3, 4, 5]

for month in springMonthNums:
    if month % month == 0:
        print (month)

for index in range(len(springMonthNums)):
    if springMonthNums[index] % springMonthNums[index] == 0:
        print (springMonthNums[index])
```
Iterating over a List’s elements

Iterating over a list’s elements works the same way as it does for strings.

Q: For the list shown in blue, do the code snippets in yellow and green print to the screen the same thing? **yes**

```python
springMonthNums = [3, 4, 5]
```

```python
for month in springMonthNums:
    if month % month == 0:
        print(month)
```

*month* is the iterator variable that is assigned the values 3, 4, and 5, in turn.

```python
for index in range(len(springMonthNums)):
    if springMonthNums[index] % springMonthNums[index] == 0:
        print(springMonthNums[index])
```

*index* is the iterator variable that is assigned the values 0, 1 and 2, that are the indices for the list *springMonthNums*. 
List append versus concatenate

Q: What do these pieces of code print to the screen?

```python
aList = ["almost done", 23]
aList.append(34.55)
print(aList)
```

```python
aList = ["almost done", 23]
newList = aList + [34.55]
print(newList)
```
They print the same thing. However, do these two pieces of code do the “same” thing? Do both create new lists, or modify an existing one?
Although these two programs might print the same thing, what they do in the background is quite different. Appending modifies an existing list, while concatenation creates a NEW list.

```
# Append modifies an existing list
aList = ["almost done", 23]
aList.append(34.55)
print(aList)  # ['almost done', 23, 34.55]

# Concatenation creates a NEW list
newList = aList + [34.55]
print(newList)  # ['almost done', 23, 34.55]
```

They print the same thing. However, do these two pieces of code do the “same” thing? Do both create new lists, or modify an existing one?
A new list is created, and `aList` is a variable that points “to,” or references, the new list.
Although these two programs might print the same thing, what they do in the background is quite different. Appending modifies an existing list, while concatenation creates a NEW list.

```python
aList = ['almost done', 23]
aList.append(34.55)
print(aList)
```

An element is added to the list. An entire new list is NOT created. Instead an element is added to an existing one.
Although these two programs might print the same thing, what they do in the background is quite different. Appending modifies an existing list, while concatenation creates a NEW list.

```python
aList = ['almost done', 23]
aList.append(34.55)
print(aList)
```

This would then print

```plaintext
['almost done', 23, 34.55]
```
This scenario begins the same way. A new list is created, and aList is a reference that is made to point “to” the just-created list.
List append versus concatenate

Although these two programs might print the same thing, what they do in the background is quite different. Appending modifies an existing list, while concatenation creates a NEW list.

```python
aList = ["almost done", 23]
newList = aList + [34.55]
print(newList)
```

a new list is created ...
List append versus concatenate

Although these two programs might print the same thing, what they do in the background is quite different. Appending modifies an existing list, while concatenation creates a NEW list.

```
aList = ["almost done", 23]
newList = aList + [34.55]
print(newList)
```

Because aList is being concatenated with another list, a copy of what aList points to is created.

- String: "almost done"
- Integer: 23
- Float: 34.55
- AList
Although these two programs might print the same thing, what they do in the background is quite different. Appending modifies an existing list, while concatenation creates a NEW list.
And finally a new variable is made that refers to the just-created new list.
Although these two programs might print the same thing, what they do in the background is quite different. Appending modifies an existing list, while concatenation creates a NEW list.

This would then print

```
[‘almost done’, 23, 34.55]
```

And finally a new variable is made that refers to the just-created new list.
List append versus concatenate

Keep in mind that to save resources, the two objects highlighted in green are actually “consolidated” into one. There is only ONE of those objects, and both \texttt{aList[1]} and \texttt{newList[1]} point to it. The same for the float and string objects.

\begin{verbatim}
    aList = ["almost done", 23]
aList.append(34.55)
print(aList)

    aList = ["almost done", 23]
newList = aList + [34.55]
print(newList)
\end{verbatim}
List append versus concatenate

Keep in mind that to save resources, the two objects highlighted in green are actually “consolidated” into one. There is only ONE of those objects, and both `aList[1]` and `newList[1]` point to it. The same for the float and string objects.

```
aList = ["almost done", 23]
aList.append(34.55)
print(aList)

aList = ["almost done", 23]
newList = aList + [34.55]
print(newList)
```

The “real” object relationship diagram is shown right. There are two variables that reference different lists, which have the “same” elements.
List append versus concatenate

Keep in mind that to save resources, the two objects highlighted in green are actually “consolidated” into one. There is only ONE of those objects, and both `aList[1]` and `newList[1]` point to it. The same for the float and string objects.

```python
aList = ["almost done", 23]
aList.append(34.55)
print(aList)
```

```python
aList = ["almost done", 23]
newList = aList + [34.55]
print(newList)
```

This code modified an existing list.
Keep in mind that to save resources, the two objects highlighted in green are actually “consolidated” into one. There is only ONE of those objects, and both `aList[1]` and `newList[1]` point to it. The same for the float and string objects.

This code created a new one.
Lists and Parameters

When a function receives an argument that “is” a list, what is actually assigned as the value of the function’s parameter?
Lists and Parameters

When a function receives an argument that “is” a list, what is actually assigned as the value of the function’s parameter?

Once again the “answer” to this question is best answered if you think of lists, strings, etc. as objects.

On-the-board explanation
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```python
def addAWord(aList, aWord):
    aList.append(aWord)

def addToEnd(anotherWord):
    myList.append(anotherWord)

myList = ["CSCI", 145]
addAWord(myList, "WWU")
print(myList)
addToEnd("Bellingham")
print(myList)
```

Q: What is the output of the code on the left?

Task: Be able to execute the code line-by-line.
Lists and Parameters

Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```python
def addAWord(aList, aWord):
    aList.append(aWord)

def addToEnd(anotherWord):
    myList.append(anotherWord)

myList = ["CSCI", 145]
addAWord(myList, "WWU")
print(myList)
addToEnd("Bellingham")
print(myList)
```

When this code is executed, a new list is created, that is made up of 2 elements, the first a String and the second an integer.
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```python
def addAWord(aList, aWord):
aList.append(aWord)
def addToEnd(anotherWord):
    myList.append(anotherWord)

myList = ["CSCI", 145]
addAWord(myList, "WWU")
print(myList)
addToEnd("Bellingham")
print(myList)
```

When the `addAWord` function is invoked, the first argument is `myList`, which is a reference to a list. The argument that is passed is the address (or reference) to the list, and NOT a copy of the list. The second argument is a reference to a String.
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```python
def addAWord(aList, aWord):
    aList.append(aWord)

def addToEnd(anotherWord):
    myList.append(anotherWord)

myList = ["CSCI", 145]
addAWord(myList, "WWU")
print(myList)
addToEnd("Bellingham")
print(myList)
```

When the `addAWord` function is executing, the local variables `aList` and `aWord` are references to the list and String objects created OUTSIDE of the function.
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```
def addAWord(aList, aWord):
    aList.append(aWord)

def addToEnd(anotherWord):
    myList.append(anotherWord)

myList = ["CSCI", 145]
addAWord(myList, "WWU")
print(myList)
addToEnd("Bellingham")
print(myList)
```

Q: What does this code accomplish?
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```python
def addAWord(aList, aWord):
    aList.append(aWord)

myList = ["CSCI", 145]
addAWord(myList, "WWU")
print(myList)
addToEnd("Bellingham")
print(myList)
```

The list referred to as `aList` is appended with the String object referred to as `aWord`.

Notice that the function does not return anything, but the list `aList` HAS been modified.
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```python
def addAWord(aList, aWord):
    aList.append(aWord)

def addToEnd(anotherWord):
    myList.append(anotherWord)

myList = ["CSCI", 145]
addAWord(myList, "WWU")
addToEnd("Bellingham")
print(myList)
```

When the function `addAWord` terminates, `myList` now points to a modified version of the original list, and `print(myList)` generates?
Lists and Parameters

Q: Can a function modify a list that is created outside of the function (in main, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```
def addAWord(aList, aWord):
    aList.append(aWord)
def addToEnd(anotherWord):
    myList.append(anotherWord)

myList = ["CSCI", 145]
addAWord(myList, "WWU")
print(myList)
addToEnd("Bellingham")
print(myList)
```

When the function `addAWord` terminates, `myList` now points to a modified version of the original list, and `print(myList)` generates:

```
['CSCI', 145, 'WWU']
```
Lists and Parameters

Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global

```
def addAWord(aList, aWord):
    aList.append(aWord)
def addToEnd(anotherWord):
    myList.append(anotherWord)
```

When the function `addToEnd` is invoked, the single argument being provided is a new String object with contents `Bellingham`.
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```
def addAWord(aList, aWord):
    aList.append(aWord)

def addToEnd(anotherWord):
    myList.append(anotherWord)

myList = ["CSCI", 145]
addAWord(myList, "WWU")
print(myList)
addToEnd("Bellingham")
print(myList)
```

When the function `addToEnd` is running, the parameter `anotherWord` is referring to the object that was passes as the first argument.

Values:
- String "CSCI"
- Integer 145
- String "WWU"
- String "Bellingham"
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```python
myList = ["CSCI", 145]
def addAWord(aList, aWord):
    aList.append(aWord)
def addToEnd(anotherWord):
    myList.append(anotherWord)
addAWord(myList, "WWU")
print(myList)
addToEnd("Bellingham")
print(myList)
```

`myList` is not a local variable, but the function still knows about it because it is a global variable declared outside of the function.
Lists and Parameters

Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```python
def addAWord(aList, aWord):
    aList.append(aWord)

def addToEnd(anotherWord):
    myList.append(anotherWord)

myList = ["CSCI", 145]
addAWord(myList, "WWU")
print(myList)
addToEnd("Bellingham")
print(myList)
```

`myList` is not a local variable, but the function still knows about it because it is a global variable declared outside of the function.

The function `addToEnd` modifies the list referred to as `myList` and appends to it the String object with the contents `Bellingham`.

String
"CSCI"

Integer
145

String
"WWU"

String
"Bellingham"
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```python
myList = ["CSCI", 145]
addAWord(myList, "WWU")
addToEnd("Bellingham")
print(myList)
```

After `addToEnd` terminates, the reference `myList` points to a List made up of 4 elements, and when `print(myList)` is invoked, the output is ...
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```python
def addAWord(aList, aWord):
    aList.append(aWord)

def addToEnd(anotherWord):
    myList.append(anotherWord)

myList = ['CSCI', 145]
addAWord(myList, 'WWU')
addToEnd('Bellingham')
print(myList)
```

After `addToEnd` terminates, the reference `myList` points to a List made up of 4 elements, and when `print(myList)` is invoked, the output is ...

```
['CSCI', 145, 'WWU', 'Bellingham']
```
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

```python
def addAWord(aList, aWord):
    aList.append(aWord)
def addToEnd(anotherWord):
    myList.append(anotherWord)
```

Therefore the output of this program is:

```
['CSCI', 145, 'WWU', 'Bellingham']
```
Q: Can a function modify a list that is created outside of the function (in `main`, for example)? And if so, how is that done?

A function can be defined to have a parameter that is a list, or a function can modify a list that is global.

Both of these functions are called `modifier` functions because they modify a list that they don’t themselves create. The function `addAWord` modifies a list whose address it receives as the first argument, and the second function `addToEnd` modifies a list that has global scope.

The changes that these two functions make to the list are called `side effects`.
Technical: When a list is passed as reference to a function, the list can be mutated using existing list methods. However, if the list in the method is changed (reassigned), then the reassignment does not modify the original (received as an argument) list.

```python
def iModifyLists(aList):
    aList.append("eat")

def iCreateANewList(aList):
    aList = ["WWU", "CS"]

meals = ["breakfast", "lunch", "dinner"]
iModifyLists(meals)
print(meals)
iCreateANewList(meals)
print(meals)
```

Q: What is the output of this program?

On the board explanation
Q: How do we write a function that generates no side effects, but which can still “modify” the contents of a list?
Pure Functions

Task: be able to explain why the function `changeTheList` is not a modifier function.

```python
def changeTheList(aList, aWord):
    aNewList = []
    for item in aList:
        aNewList.append(item)
    aNewList.append(aWord)
    return aNewList

myList = ["CSCI", 145]
myList = changeTheList(myList, "WWU")
print(myList)
```
The function `changeTheList` is called a pure function because it does not generate any side effects. Instead, it returns a list that it created.

The reference variable `myList` is reassigned to refer to the list (an object) returned by `changeTheList` function.

Q: What does this program print to the screen?

```python
def changeTheList(aList, aWord):
    aNewList = []
    for item in aList:
        aNewList.append(item)
    aNewList.append(aWord)
    return aNewList

myList = ["CSCI", 145]
myList = changeTheList(myList, "WWU")
print(myList)
```
Join

The opposite of `split` is to `join`. Just like `split` needs to know about the delimiter (by default it is the empty space), when you `join` 2 items, you must specify what is the “glue” that will hold two things together when they are joined.

```python
aList = ['fred', 'susan']
myGlue = 'hello'
aNewList = myGlue.join(aList)
print(aList)
print(aNewList)
```

Q: What is the output of the above code?
Join

The opposite of \texttt{split} is to \texttt{join}. Just like \texttt{split} needs to know about the delimiter (by default it is the empty space), when you \texttt{join} 2 items, you must specify what is the “glue” that will hold two things together when they are joined.

```python
aList = ["fred", "susan"]
myGlue = "hello"
aNewList = myGlue.join(aList)
print(aList)
print(aNewList)
```

Q: What is the output of the above code?

Q: What does “join” do?

If you don’t know ... where do you look?
The opposite of **split** is to **join**. Just like **split** needs to know about the delimiter (by default it is the empty space), when you **join** 2 items, you must specify what is the “glue” that will hold two things together when they are joined.

```python
aList = ["fred", "susan"]
myGlue = "hello"
aNewList = myGlue.join(aList)
print(aList)
print(aNewList)
```
The opposite of \texttt{split} is to \texttt{join}. Just like \texttt{split} needs to know about the delimiter (by default it is the empty space), when you \texttt{join} 2 items, you must specify what is the “glue” that will hold two things together when they are joined.

On the board explanation and demo

```python
aList = ["fred", "susan"]
myGlue = "hello"

aNewList = myGlue.join(aList)
print(aList)
print(aNewList)
```

Description

The method \texttt{join()} returns a string in which the string elements of sequence have been joined by \texttt{str separator}.

Syntax

Following is the syntax for \texttt{join()} method:

```
str.join(sequence)
```
The opposite of \texttt{split} is to \texttt{join}. Just like \texttt{split} needs to know about the delimiter (by default it is the empty space), when you \texttt{join} 2 items, you must specify what is the “glue” that will hold two things together when they are joined.

On the board explanation and demo

```python
aList = ["fred", "susan"]
myGlue = "hello"
aNewList = myGlue.join(aList)
print(aList)
print(aNewList)
```

\texttt{fredhellosusan}
Task: Complete the code (add at most 2 lines):

```python
typeOfBeetles = ["stink", "yellow", "dung", "squashed"]

So that the output of the 3 line program is:

```
stinkBeetleyellowBeetledungBeetlesquashed
```

Live demo
Up next

Dictionaries
Finishing up