ECS10

10/22

Midterm

- Grading this afternoon and tonight.
- Should appear on SmartSite soon.
- Solutions up today.
- If you did poorly, take it as a wake-up call. You need to master Chapters 1-3 in the book, ASAP.

Today and onwards

- We now know the basics!
- Now we look at ways to organize and compute with data.
- New data type – **list**
- New kind of loop – **for**
- More about strings
- New assignment will be up tomorrow, due Tues Oct 27.

List

```
dayList = ['Mon', 'Tues', 'Wds', 'Thur', 'Fri']
```

- dayList is a list.
- dayList[0] is the string 'Mon', dayList[2] is the string 'Wds'
- You can have a list of any kind of data.

```
BP = [140,80]
letters = ['c', 'o', 'w']
```

Lists and indexing

- A list is a sequence of anything, numbered starting from zero.

```
intList = [5,0,3,5,6,8,9,0,4,4,4]
```

- Index into a list.

```
intList[4]    # This is the integer 6
```

- First thing in the list has **INDEX ZERO**!

Indexing into strings

- A string is like a list of single characters.
- We can index into a string, too.

```
beast = "cow"
print beast[2]
```

Function len()

- Tells you how many things are in a list.

```
dayList = ['Mon', 'Tue', 'Wds', 'Thur', 'Fri']
print  len(days)
```

- Prints 5
- The elements have indecies 0,1,2,3,4
- Also works with strings

```
print len('cow')
```

- Prints 3

Sequences

- Lists and strings are two kinds of **sequences**.
- The len() function works on sequences (not on floats, or ints, or Booleans).
- You can index into sequences to get **elements**.
- There is another kind of sequence in Python, the **tuple,** which is very like a list (does fewer things, but does them faster). We will only use tuples when we start to use other people's modules that use them.

Loop over a list

```
beastList = ["cow", "sheep",  "duck"]
 j = 0
 while j < len(beastList):
   beast = beastList[j]
   print  "A", beast, "is an animal."
   j = j+1
```

- i is the index variable. Index variables are always integers.
- beast is the string at index j.

Loop over a string

```
strIn = "5,236,320"
i = 0
strOut = ""
while i < len(strIn):
    char = strIn[i]
   if char != ",":
        strOut = strOut+char
   i = i+1
```

Eliminates the commas.

Understand the variables

```
strIn = "5,236,320"
i = 0
strOut = ""
while i < len(strIn):
    char = strIn[i]
   if char != ",":
        strOut = strOut+char
   i = i+1
```

- i is the index variable – what position in the list are we working on?
- char is the character at that position.

Very common structure

- You have a big list, and you want to do the same thing for every item in the list.
  - Add up scores for all students.
  - Draw all visible objects in a game.
  - Check all sectors of the sky for supernovas.
  - Convert all strings to integers.
  - ….
- Use a loop to go through the list, like the two previous examples.
- for loop is shorthand for this.

For loop

```
beastList = ["cow","sheep","duck"]
for beast in beastList:
   print “A”,beast, “is an animal”
```

- Shorter and sweeter, but exactly the same effect.
- Goes through items in list, starting with **beastList[0]**, then **beastList[1]**, **beastList[2]**….
- String variable **beast** takes on each of these values in turn
- Eliminates the index variable**!**

for loop over a string

```
strIn = “5,342,750”
strOut = “”
for char in strIn:
   if char != “,”:
        strOut = strOut+char
```

- Exactly the same effect as version using while.
- Prettier, shorter.
- char takes on values “5”, then “,”, then “3”…..

for vs while

- Anything you can do with a for loop, you could also do with a while.
- for loops can only be used when you know how many times they will run before you start (length of list,…)
- while loops are much more versatile, since you don’t need to know how many times it will loop.
- for loop will be a little shorter and tidier.

The replace method

```
string = “2,407,018”
popString = string.replace(“,”,  “”)
population = int(popString)
```

- Replaces all copies of the first argument with the second.
- Here, replaces all commas with the empty string; that is, eliminates commas.

```
s = ‘Flinch’
s = s.replace(“Fl”, “Gr”)
```

methods

- If replace was a normal function we’d expect the name of the function to come first:

```
num = int(“45”)
length = len(“A”, “B”, “C”)
name = raw_input(“What is your name? “)
```

- Instead, the name of the string the function works on comes first:

```
popString = string.replace(“,”, “”)
```

String methods

- Google “Python string methods”

separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings. New in version 2.5.

replace(*old*, *new*[, *count*])
Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

rfind(*sub* [,*start* [,*end*]])
Return the highest index in the string where substring *sub* is found, such that *sub* is contained within

- There are lots of them!