CS2110 Lecture 10 Feb. 15, 2021

- HW2 scores have been posted
- Test 1 has been graded

Score	1-3	4-6	7-9	10-12	13-15
# of people	2	4	4	3	6

median: 8

high: 15 (1 person, 5 at 14)

- DS Assignment 3, due tomorrow/Tuesday, 10pm
  - Again, optional to attend BUT this time TA will walk students through some of the examples
- HW3 is available, due next Monday, 10pm

Today

- HW3 Q1 overview and advice
- An alternative looping construct, **for**

We seen looping over strings with **while**

Using [] and len, you can write while loops that do things with each character in a string:

```
currentIndex = 0
while currentIndex < len(myString):
    currentChar = myString[currentIndex]
    ….
    …. (loop body – typically does something
    …. with character, currentChar)
    ….
    currentIndex = currentIndex + 1
```

We'll continue to practice this over the next couple of weeks; it is very important that you understand this general pattern of stepping through a string (or, later, other sequence) by using a loop and incrementing an index

for loops and strings

Python provides an alternative, more concise way to iterate over strings

```
for currentChar in myString:
        ….
        …. (loop body – typically does something
        …. with character, currentChar)
        ….
```

The body of the **for** loop gets executed once for each character in the string, myString. On the first iteration, currentChar is bound to the first (i.e. index 0) character of myString. On the second iteration, currentChar is bound to the second (i.e. index 1) character, etc. This loop and the one on the previous page are equivalent! *You need to be able to convert* ***for*** *loops to equivalent* ***while*** *loop! It's a simple "robotic" process (and I almost always test this on the first exam)*

lec10loopchars.py

Looping on strings with **for**

```
def findChar(charToFind, stringToSearch):
    for char in stringToSearch:            lec10findChar.py
        if char == charToFind:
            print('found it')
            return      <— leaves function immediately
```

But what if specification is instead to return the index of character if found, and length of given string if not?

Looping on strings with **for**

```
def findChar(charToFind, stringToSearch):
    indexOfCharSought = len(stringToSearch)          lec10findChar.py
    currentIndex = 0
    for char in stringToSearch:
        if char == charToFind:
            indexOfCharSought = currentIndex
            break                                   <— exits loop immediately*
        currentIndex = currentIndex + 1
    return indexOfCharSought
```

What is different if we remove break Is the result different/still correct?

** Be careful; if not used well **break** can yield confusing code*

Ch 9.13: string **in** operator

- ‘a’ in myString — returns True if ‘a’ is in myString, False otherwise

Write function inBoth(string1, string2) that prints all characters that appear in both:

```
def inBoth(string1, string2):
    for c in string1:
        if c in string2:
            print(c)
```

Demo/exercises

- lec10exercises.py debugging exercises involving **for** and strings

Problem like HW3 Q1

Suppose goal is to find second and third smallest letters, and most common letter

A two-part approach (you *can* do it "all at once" if you want but many people will find separating the two easier):

```
# find second and third smallest
    # go through string char by char updating values for
    # three simple variables:
    #   smallest, secondSmallest, and thirdSmallest

# find most common
    # presume you have a function howMany(c, s) that
    #       returns the number of times c occurs in s
    # Using a loop simply go through string char by char,
    #       calling howMany(char, s) for each char and comparing result with a   #
maxOccurrencesSoFar variable, updating when appropriate

# print results
```

howMany(c, s)
is easy to write!

Hint: consider using **None** for initializing variables

HW3 Q1

```
# find second and third smallest
    # go through string char by char updating values for
    # three simple variables:
    #    smallest, secondSmallest, and thirdSmallest
```

smallest: ~~?~~ ~~e~~ ~~c~~ ~~b~~ a

secondSmallest: ~~?~~ ~~e~~ ~~d~~ ~~c~~ b

thirdSmallest: ~~?~~ ~~e~~ ~~d~~ c

e c d b f a

Ch 10: **lists**

- **list** is another Python sequence type (**string** was our first)
- In a string, each item of the sequence is a character
- In a list, each item can be a value of any type!
- The most basic way to create a **list** is to enclose a comma-separated series of values with brackets:

```
>>> [1, ‘a’, 2.4]
[1, ‘a’, 2.4]

>>> myList = [1, ‘a’, 2.4]
>>> len(myList)
 3
>>> myList[0]
1
```

[] operator and len() function work on both strings and lists

Ch 10: **lists**

I said the items in a list be any type. So, can lists be elements of lists? YES!

```
>>> myList = [1, 2, [‘a’, 3]]
>>> len(myList)
3
>>> myList[2]
[‘a’, 3]
>>> myList[2][1]
3
>>> myList[1][2]
Error
```

we call this a “nested list”

Ch 10: **lists**

A list can have no elements!

```
>>> myList = []
>>> len(myList)
0
>>> myList[0]
Error
```

we call this an
"empty list"

Ch 10: list operations

slices, +, * work similarly to how they work on strings

```
>>> myList = [1, 2, 3, 4, 5]
>>> myList[1:3]
[2,3]
>>> myList + myList
[1,2,3,4,5,1,2,3,4,5]
>>> myList = myList + [6]
[1,2,3,4,5,6]
>>> myList = myList + 6
Error
>>> myList = myList + [[6]]
[1,2,3,4,5,6,[6]]
>>> 2 * myList
[1,2,3,4,5,6,[6],1,2,3,4,5,6,[6]]
```

Ch 10: traversing lists

Just like we often want to iterate through the characters of a string, we often want to "traverse" lists, doing some computation on each list item in turn. Like they are for string, **for** loops are again concise and useful

```
for element in ['a', 2, 'word', ['1,2', 3]]:
        if type(element) == list:
            print('list of length:', len(element))
        else:
            print(element)
```

yields:

```
a
2
word
list of length: 2
```

Traversing lists with **for**

```
for number in l:
    if number < 0:
        print(“negative”)
    else:
        print(“not negative”)
```

The **range** function

Python's **range** function is very useful. There is no one clear place in the text where it is presented. It is first mentioned in 4.7 of the Turtle chapter, and then used in examples in Ch 9 and 10.

The range function produces values of a **range** type

The range type is another sequence type, like **list** and **string**.

range(9) is a sequence of the integers 0, 1, …, 8

range(2,6) is sequence 2, 3, 4, 5

range(2,13,3) is sequence 2, 5, 8, 11

Since range is a sequence type, (most of) the standard sequence operations apply (not nicely specified anywhere in text – go to Python sequence docs on-line)

range – standard sequence ops

```
>>> 5 in range(9)
True
>>> 5 in range(2,10,2)
?
>>> len(range(2,10,2))
?
>>> myRange = range(2,20,2)
>>> myRange[3:6]
?
>>> range(5) + range(5)
?
```

Ch 10: **range** – Python 3 vs Python 2

In Python 2, range is just a function that produces a list:

```
>>> range(9)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

In Python3, range(9) is an object that represents the same sequence of numbers, but it *not* a list.

```
>>> range(9)
range(9)
```

Note: in Python 3, you can still use range to build an ordered list of numbers:

```
>>> list(range(9))
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Next time

- Examples looping with lists and range lec11.py

The rest of Ch 10. Much of it is related to important property of lists:

- lists are *mutable!*

It is very important to understand the consequences of list mutability. It can be confusing if you don't take time to understand it!

Exercise to think about. Lists make it easy to implement printLetterCounts(inputString, letters) that prints the number of occurrences in inputString of each letter in letters

```
>>> printLetterCounts("This is a sentence containing a variety of letters", "aeiouy")
'This is a sentence containing a variety of letters' has:
        4 'a's
        6 'e's
        5 'i's
        2 'o's
        0 'u's
        1 'y's
        and 32 other letters
```

Next time

- Examples looping with lists and range lec11.py

The rest of Ch 10. Much of it is related to important property of lists:

- lists are *mutable!*

It is very important to understand the consequences of list mutability. It can be confusing if you don’t take time to understand it!

Exercise to think about. Lists make it easy to implement printLetterCounts(inputString, letters) that prints the number of occurrences in inputString of each letter in letters

```
>>> printLetterCounts("This is a sentence containing a variety of letters", "aeiouy")
'This is a sentence containing a variety of letters' has:
        4 'a's
        6 'e's
        5 'i's
        2 'o's
        0 'u's
        1 'y's
        and 32 other letters
```