CS1210 Lecture 11 Sep. 17, 2021

- HW3 available, due next Friday, 8pm
- DS4 will be made available by Monday morning, due Tuesday 8pm
- Quiz 1 will be graded within a few days. Don’t panic if you think you did poorly. Keep working. You can still master things if you keep working hard

Last time

- A debugging example
- Looping with **for**
- Discussion of Quiz problem types

Today

- A example with some programming advice
- Discussion of HW3 Q1
- Start Chapter 10: **list**s

Programming advice

Be careful with variable names:

- Don’t use ..index.. when it’s bound to a value other an index!
- Don’t change type of thing variable is bound to – use a different variable!

```
cost1 = 23.0
cost2 = 143.
for index1 in string1:                                <—    index1 is not an index
    index2 = 0
    while index2 < len(string2):
        if string1[index1] == string2[index2]:        <—    error here
            cost1 = “The cost is:” + str(cost1)       <—    dangerous to change
        index2 = index2 + 1                                   change type of
…                                                             object bound to var.
…                                                             cost1 was a number,
print(cost1)                                                  now a string
if (cost1 < cost2):
    print(“Option 1 is the better one!”)                      oops, error. Forgot
                                                              cost1 now a string
```

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
- In a string, each item of the sequence is a character
- In a list, each item can be a value of any type! (and can be as long as you want)
- 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]
>>> myList
[1,2,3,4,5,6]
>>> myList = myList + 6
Error
>>> myList = myList + [[6]]
>>> myList
[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
```

lec11.py

Traversing lists with **for**

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

Last time: 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]
```

Ch 10: lists are mutable!

- Strings are immutable. You can’t change them.

```
>>> myString = ‘hello’
>>> myString[0] = ‘j’
```

← Error

- But lists are mutable! You can update lists

```
>>> myList = [1, 2, ‘hello’, 9]
```

```
>>> myList[1] = 53
 >>> myList
[1, 53, ‘hello’, 9]
```

you can replace a item in a list with a new value

```
>>> myList.append(‘goodbye’)
>>> myList
[1, 53, ‘hello’, 9, ‘goodbye’]
```

you can add new items to the end of a list

```
>>> myList2 = [3, 99, 1, 4]
>>> myList2.sort()
>>> myList2
[1, 3, 4, 99]
```

you can even sort! Note: Python’s sort rearranges the items directly within the given list. It doesn’t yield a new list with same items in sorted order (different function**, sorted**, yields new sorted list)

Next Time

for-while loop conversion

More Chapter 10

- more on list mutability
- + vs append
- *“aliasing”*
- ***Is** operator and object identity (vs ==)*
- *lists as arguments to functions*