CS1210 Lecture 7 Sep. 8, 2021

- DS2 assignment due tomorrow morning
- HW2 due Tuesday morning, Sep 14
- Quiz 1 Wednesday, Sep 15 in class

Last time

- Finished conditional execution, if-elif-else, Chapter 7
- Began Ch 8 - iteration and while loops

Today

- Before continuing with loops, some basic string operations and methods
- More on while loops

Terminology: what are Python **method**s?

- Python **method**s are special kinds of functions that are called using a different syntax. We'll see them a lot when working with strings.

- Methods are invoked/called using "dot notation". Some people find it confusing:

```
>>> 'AbCd'.lower()
'abcd'
```

invokes the String method lower, a function specific to strings *BUT you can think of it as lower('AbCd')*

```
>>> 'abcde'.index('d')
3
```

invokes the String method index. You can think of it as index('abcde', 'd')

(In fact, it is even possible to invoke methods in the "usual" function call style. E.g. str.lower("AbCd") and str.index('abcde', 'd'))

WE'LL TALK A LOT MORE ABOUT METHODS AS WE PROGRESS THROUGH THE COURSE.

Intro to Ch9: Strings

- So far, we've used strings to print things and only discussed one string operator, +. There are many other string operators and functions
- 9.3 covers +, * on strings

  >>> "hi " + "jim"

  "hi jim"

  >>> "go" * 3

  "gogogo"
- And len() provides the number of characters

  >>> len(myString)

  6
- 9.4 You can get individual characters within a string via the [] operator:

  >>> myString = "python"

  >>> myString[0]

  'p'

  >>> myString[3]

  'h'

  The expression inside the brackets is called an **index**

  – In Python the first element has index 0 !!! ← IMPORTANT

9.4 String operations and indexing

- Can also use negative indices!

  s[-1] for any string s is the last character of string

  >>> ‘hello’[-1]

  ‘o’

  Equivalent to, but more convenient than,
  s[len(s)-1]

Ch 9.5: string **method**s

- We use strings a lot in Python. Python provides many special built-in functions, called **method**s, for strings.
- Methods are called/invoked using a different syntax, dot notation (some people find it confusing):

```
>>> 'abcd'.upper()
'ABCD'
```

invokes the built-in string upper function

NOTE: You can think of it as upper('abcd')

- There are a quite a few: Look them up – I won't go over many of them in detail.

Ch 9.5: string **method**s

```
>>> myString = 'hello'
>>> myString.count('l')
2
>>> 'ababcab'.count('ab')
3
>>> 'eeeeeee'.count('ee')
?


>>> myString.index('l')
2
>>> myString.find('l')
2
```

Again, you can think of it as: count(myString, 'l')

index and find nearly the same but look up in docs! (give different result when not found)

Ch 9.5: string **method**s

```
>>> 'This is a sentence.'.split()
['This', 'is', 'a', 'sentence.']
```

*a **list** (we'll study lists soon)*

```
>>> '1,2,104,7,12'.split(',')
['1', '2', '104', '7', '12']
```

```
>>> '    non-whitespace  '.strip()
'non-whitespace'
```

```
>>> '.'.join(['www', 'uiowa', 'edu'])
'www.uiowa.edu'
```

Note: these three are very commonly used when reading in data from files

Ch 9.7: string slices

- We've seen [] used to access one character of a string
- It can also be used to extract a series of elements from a string. This is called **slicing.**

```
>>> 'abcdefghij'[4:7]
'efg'
```

yields a new string consisting of the index 4, 5, and 6 chars

- 'abcdefghij'[:5] is the same as 'abcdefghij'[0:5]
- myString[3:] is the same as myString[3:len(mystring)]

9.8 string comparison

- ==, <, > work on strings
  - but understand that results might not be what you expect

  'a'== 'A' -> False

  'apple' < 'Banana' -> False

  'apple' < 'Apple' -> False

  'A' < 'a' -> True

  'a' < 'b' -> True

  'A' < 'B' -> True

- Note: Python has ord() and chr() functions to turn characters into integer values (Unicode point values)
  - ord('a') -> 97
  - chr(97) -> 'a'
  - chr(8986) -> '⌚'
  - chr(9096) -> '❈'

  DO NOT use ord/chr in homework (unless explicitly told to do so)! Usually, there is no reason to. Usually you should just compare characters directly.

Ch 9.9: strings are immutable!

- You might want to do:

  >>> myString = ‘fun’

  >>> myString[0] = ‘s’ ← ERROR

  hoping to change ‘fun’ to ‘sun’

  You cannot do this in Python. Strings cannot be modified! (this is different than several other popular programming languages)

- You can easily build new strings but you can’t directly modify strings.

  >>> myString = ‘fun’

  >>> myString = ‘s’ + myString[1:]

  >>> myString

  ‘sun’ ← THIS A NEW STRING. ‘fun’ didn’t change. Remember: myString is just a name

(last time) Ch 8.2: Iteration – the **while** statement

- Many computations involve repetition, doing the same (or nearly the same) things repeatedly (perhaps a few times, perhaps billions of times)
- You could already write a program to, say, print out the first 1000 integers

```
def printFirstThousand():
    print(1)
    print(2)
    …
    print(1000)
```

- But Python (and other languages) provide statements to conveniently describe and control repetitive computations.

(last time) Ch 8 – the while statement

The **while** statement provides a very general mechanism for describing repetitive tasks.

```
…
… (B1: code before while)
…
while boolean expression:
    …
    …      (B2: code in while body)
    …
…
… (B3: code after while)
…
```

What happens?

1. Execute B1 code
2. Evaluate Boolean exp
3. If True, do
   3a. eval B2 code.
   3b. jump to step 2 again
   If False, ignore B2 code and simply continue with step 4
4. Execute B3 code

Designing **while** loops

When designing loops:

- Typically, just before while statement, set variables that ensure truth of boolean condition (also called “loop guard”)
- Within loop, update one or more variables of the loop guard expression. In many simple loops, this is often done at the end of the loop body.
- Carefully reason about your loop body and loop guard, convincing yourself that eventually the loop guard will become false (Note: sometimes, people formally prove these things. E.g. when validating software for, say, flight control systems)

lec7while.py **while** examples

- whileFn: sometimes loop body never executes
- whileFn2: sometimes loop body never terminates. Often this is due to a design error, but sometimes on purpose.
- whileFn3: using a while loop to process user input
- countDownBy3From:
  - Different behavior on different inputs? For what inputs does countDownBy3 from not terminate?
- loopChars: strings and while. Looping over strings

Strings and **while**

Using [] and len, you can write while loops that do things with each character in a string. (loopChars in lec7while.py)

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

We'll talk about 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

Any questions on Q3 of DS2?

```
# Q3.Complete function sumDigitsOf(n) that returns the sums the digits of a
#    the given positive number n
#
# For example, sumDigitsOf(5137) should return 16
#
# SOLUTION METHOD:
#
# How can we extract and sum the digits of a non-negative integer
# ONLY using math?
#
# Some people might know that in Python we first convert the number to a
# string of digits and then look at each character in the string. BUT, the goal
# here is to do it with just math - NO STRING OPERATIONS ARE ALLOWED!
#
# Use this approach:
#
#    1. get the last digit of an integer by using % 10
#       digit = number % 10 (E.g. if number was 432, digit will be 2)
#       Add this value to a variable you use for the ultimate sum.
#
#    2. get a new number with all but the last digit via
#       newNumber = number // 10  (E.g. if number was 432, newNumber will be 43)
#
#    3. repeat those steps until nothing's left (i.e. when newNumber becomes 0)
#
#    USE A WHILE LOOP TO ACCOMPLISH THE ABOVE STEPS
#
#  HINT: First, you should try examples of steps 1 and 2 at a Python prompt to
#  make sure you understand. E.g. enter 432 % 10 and 432 // 10
#
def sumDigitsOf(number):
    # add code here
    return #result
```

REMEMBER! MATH ONLY - **NO STRING OPERATIONS** ARE ALLOWED!!

HW2 Q1 hint

q1(origString, repeatCount, lettersToRepeat)

Hint: use a simple loop to iterate over chars on origString. Initialize a variable for new/result string. On each iteration of the loop, add something to the new string

```
result = ""
while ….
    if …
        update result string in some way
    else:
        update result string in a different way

return result string
```

A looping example helpful for Q3 of HW2

- Q3 asks you to write a function that takes four inputs:
  - a string, inputString, of lower case characters
  - a string, minLetter
  - a string, lettersToIgnore
  - a string, specialLetter

  and returns 1) the "smallest" character in inputString that is both greater than minLetter and not in lettersToIgnore, 2) the highest index at which that letter occurs, and 3) True if specialLetter occurs an odd number of times in inputString

- Study the three functions in charExamples.py to help you get started. They use simple loops and no string operations other than "in". It's good to practice this kind of loop pattern: iterating through a string maintaining one or more variables related to a "best" or "minimum" or "largest"

Next time

More on iteration, Ch 8, including development of function printFirstNPrimes:

- A prime number is an integer greater than one that has no divisors other than 1 and itself.
  - 2, 3, 5, etc.
- Goal: implement function printFirstNPrimes(n) that takes integer n as input and prints the first n prime numbers.

```
>>> printFirstNPrimes(4)
2
3
5
7
```