CS1 Lecture 9 Feb. 6, 2017

- HW 3 available Tues. morning, due 9am Monday .
- Discussion sections this week: loop and string practice
- HW1 grading done
  - Common errors: 1) copy/paste (test your code! You'll quickly see output is incorrect), 2) unnecessary use of small number for comparision: floor(time - .0001). This is not safe, doesn't always work. *Sometimes* use of small values/"epsilon"s is necessary but not here; don't use when you don't need to
  - Some probably provable cases of academic honesty violation. Don't do it – we will report to UI! And see next item…
- ***Do not post questions on Chegg, stackoverflow, etc.! It violates academic honesty policy***.

Last time

- Comments on HW2
- Two **while** examples, with advice on design
  - One related to HW2 Q1 – printStuff
  - Finished printFirstNPrimes that we started last timeHW2

Top-down design of printFirstNPrimes

Express algorithm in comments, like an outline.
Incrementally refine and implement steps.

```
def printFirstNPrimes(n):
    candidate = 2
    numPrimesPrinted = 0
    while (numPrimesPrinted != n):
        isPrime = numIsPrime(candidate)
        if isPrime:
            print(candidate)
            numPrimesPrinted = numPrimesPrinted + 1
        candidate = candidate + 1
```

```
# stub like this VERY
# USEFUL for testing!!
#
def numIsPrime(n):
    isPrime = True
    return isPrime
```

Now, just need to implement numIsPrime() BUT first test this code using "stub" numIsPrime() !

Top-down design of printFirstNPrimes

Now develop isNumPrime in similar fashion:

```
def isNumPrime(n):
    # presume number is prime
    isPrime = True
    # check potential divisors 2 .. n-1. If any evenly divides n
    potentialDivisor = 2
    while potentialDivisor < n:
        # check if potential divisor evenly divides n,
        #     upddating isPrime if it does
        if (n % potentialDivisor) == 0:
            isPrime = False
        potentialDivisor = potentialDivisor + 1

    return isPrime
```

lec9primes.py shows how basic and improved versions can be compared using timing functions (This is not required, for now. We'll discuss efficiency analysis more carefully later in the course)

Note: this can be improved:

1) When find divisor, stop searching, return False
2) Search doesn't need to go to n-1. Can stop when potential divisor reaches square root of n (if n has divisor bigger than its square root, it must also have one smaller)

BUT general rule: worry about correctness before working on optimizations like this

Today

Chapter 8: Strings and **for**, including string **method**s

Ch 8: Strings and **for**

- strings in Python are our first example of a **sequence** type. (We'll see another later this week – lists).
- As shown last week and used in HW2, you can access individual items in strings (or any sequence) using the [] operator

```
>>> word = 'hello'
>>> word[1]
'e'
```

- The len functions returns the number of items in a sequence – for strings, the number of characters

```
>>> len('hello')
5
```

Intro to Ch8: Strings and **for**

As you should know now from HW2, 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
```

Intro to Ch8: **for**

Python provides a more concise way to do the same thing as the while-loop-with-index-incrementing of the previous slide

```
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! (often tested on exam)*

Ch 8: string slices

- We've seen [] used to access one character of a string
- It can also be 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)]

Ch 8: 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

Looping on strings

```
def findChar(charToFind, stringToSearch):
    for char in stringToSearch:
        if char == charToFind:
            print('found it')
            return
```

quits loop and function immediately

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

Looping on strings

```
def findChar(charToFind, stringToSearch):
    indexOfCharSought = len(stringToSearch)
    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 result different/still correct?

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

Ch 8: 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 8: Debugging section and Exercises

- Read the debugging section for pretty good example of debugging loop errors
- 8-4 is a good exercise
  - Make sure you understand which functions are correct/incorrect. You should be able to do this without executing the code.
  - lec9ex84.py

Next time

Finish Ch 8

Some of Chapter 9.

Start Chapter 10 – important topic of **list**s