CS1 Lecture 8 Feb. 3, 2017

- HW 2 due Mon., Feb. 6, 9:00am
  - Please include your name in a comment at the top of the file!
  - After you submit something, verify that it's actually on ICON! We won't accept homework late based on "I thought I submitted it but …"
- **NEW:** one "free" 2/2 disc. section and one "free" 10/10 homework. I.e. your lowest discussion section score will be made 2/2 and your lowest hw score will be made 10/10. You can save this for when you are too busy, forget, don't feel well, etc.

Last time

- Quick intro to beginning of Ch 8: Strings and another iteration construct, **for**
- More **while** examples

Today

- General comments on HW2
- Two **while** examples, with advice on design
  - One closely related to HW2, Q1
  - Finish printFirstNPrimes that we started last time
- Delay pushing forward through all of Ch 8 (strings and string methods, for) til HW2 done.

HW2 comments

- Q1
  - look up Python (3!) documentation for print() https://docs.python.org/3/library/functions.html#print You'll see that by default the function adds a newline but that you can add a little bit extra to make it not add a newline. In particular, see the "end" keyword argument, and play with some examples.
  - develop "top-down" (we will do a similar example in this lecture – lec8printStuff.py)

```
def printStuff(low, high):
    # initialize counter
    # while (condition involving counter):
        # print one thing
        # print a bunch of things on one line
        # update counter
```

- Q2: There is no need to use lists, string methods, or other things not yet taught in this class. Use several counters and a simple loop.
- Q3: Problems like this will occur many times in the class and you should become comfortable with it. Exams often include variants of find min/max that aren't readily solved using built-in min/max function. You need to be able to write and understand loops that maintain, e.g. "best so far" and "index for best-so-far" variables. Use the loopChars function in lec7while.py as template!
- Q3: There is no need to use Ascii values. Characters can be directly compared with Python operators like <, >, ==.
- Q4: keep things "clean" – use integer division! (Using / for divide generates floats.) Results should all be integers.

HW2: a question from a student

“How many lines of code is too much?”

It seems to me that we should only write enough code to get the job done effectively, and nothing else beyond that. I want to avoid defining and using objects the program doesn’t need.

That being said, I wrote my solution as a single function and everything works great. However, my code seems excessive. I used 18 lines to find the best character and its index.

My question is:

- Is the length of my code too far from the one the instructors have or recommend using?
- If it works perfectly as a black box, does it matter?
- Should I be thinking about code length in the context of computing resources, e.g. memory and speed?
- If my code seems too long, should I spend time trying to simplify it or break it up into a few more functions?

Thanks.

This is a great question. There are several ways to respond and I will say things related to them throughout the semester.

But first, I'll say that for this particular assignment we won't take off points if you have extraneous code. If it meets the specifications, you'll get full credit.

In the bigger picture, "too much code" or "extraneous code" can mean several things. I'll mention three interpretations here:

1. “avoid defining and using objects the program doesn't need.” Answer: it depends on what "doesn't need” means BUT, for the most part you SHOULD NOT worry about this. For example, it usually is actually a *good idea* to define temporary/intermediate variables:

```
time = distance / speed
cost = time * payrate
```

time isn't *needed* (could directly use cost = (distance / speed) * payrate) but except in very unusual situations, there is *no memory or speed penalty* for the "longer" code.

In many systems, a program called a *compiler* does a great job of simplifying code like this. You don't need to worry about it. And it is can be *much easier to read and maintain*. Fewer lines of code should not be a goal. Lines of code don't directly translate to memory code. Instead your goal should be clear correct code Conciseness can be good but clarity more important.

2.On the other hand, sometimes code is "too long" because it solves the problem in a roundabout way (say 1000 print statements instead of a loop, as discussed in class). This is generally not a good thing. Depending on the stated requirements of a specific problem, we *might* deduct a small amount for roundabout/clunky solutions, and sometimes we'll give full credit but just comment on the approach.

3. Sometimes "too long" is not actually about lines of code but instead about the efficiency of an algorithm. We will talk about algorithm efficiency later in the course. It's not measured in lines of code. *I can write a hugely inefficient 4-line algorithm to compute the nth Fibonacci number*. *I can make that algorithm HUGELY more efficient by actually adding a few lines of code* – sometimes *more* code enables a more efficient algorithm.

BOTTOM LINE: think clarity and correctness. Don't hesitate to introduce and use variables.

HW 2: an example similar to Q1

Consider function printStuff2(high, low) such that, e.g. printStuff2(4,2) produces:

```
2
**
*
3
***
**
*
4
****
***
**
*
```

lec8printStuff.py

Example started last time: 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
```

Top-down design of printFirstNPrimes

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

```
def printFirstNPrimes(n):
    # starting at 2, count upwards, testing
    #     candidate integers for primeness,
    #     printing those that are prime
    #     and stopping after n
    #     have been printed
```

Top-down design of printFirstNPrimes

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

```
def printFirstNPrimes(n):
    candidate = 2
    while (numPrimesPrinted != n):
        # test candidate for primeness
        # print, update numPrimesPrinted if prime
        candidate = candidate + 1
```

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):
        # test candidate for primeness
        # print, update numPrimesPrinted if prime
        candidate = candidate + 1
```

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)
        # print, update numPrimesPrinted if prime
        candidate = candidate + 1
```

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() !

Now: finish printFirstNPrimes by replacing stub isNumPrime with correct code

Again, develop isNumPrime in top-down fashion:

```
def isNumPrime(n):
    # presume number is prime
    # check potential divisors 2 .. n-1. If any evenly divides n
    # then n is not prime
```

Top-down design of printFirstNPrimes

Now develop isNumPrime in similar fashion:

```
def isNumPrime(n):
    isPrime = True
    # check potential divisors 2 .. n-1. If any evenly divides n
    # then n is not prime
```

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,
        #     update isPrime if it does
        potentialDivisor = potentialDivisor + 1

    return isPrime
```

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,
        #     updating isPrime if it does
        if (n % potentialDivisor) == 0:
            isPrime = False
        potentialDivisor = potentialDivisor + 1

    return isPrime
```

lec8primes.py

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

Next time

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