CMSC201
Computer Science I for Majors

Lecture 18 – Program Design (cont)
Last Class We Covered

• Tuples
• Dictionaries
  – Creating
  – Accessing
  – Manipulating
  – Methods
• Dictionaries vs Lists
Any Questions from Last Time?
Announcement – Survey #2

• Available now on Blackboard
• Due by Sunday, November 13, at midnight
  – Check completion under “My Grades”
• Some statistics (from Fall 2015):
  – If they had taken the surveys...
    • 9 students would have gotten an A instead of a B
    • 4 students would have gotten a B instead of a C
    • 9 students would have gotten a C instead of a D
Today’s Objectives

• To discuss the details of “good code”
• To learn how to design a program
• How to break it down into smaller pieces
  – Top Down Design
• To introduce two methods of implementation
  – Top Down and Bottom Up
• To learn more about Incremental Programming
Motivation

• We’ve talked a lot about certain ‘good habits’ we’d like you all to get in while writing code
  – What are some of them?

• There are two main reasons for this
  – Readability
  – Adaptability
“Good Code” – Readability
Readability

• Having your code be readable is important, both for your sanity and anyone else’s
  – Your TA’s sanity is important

• Having highly readable code makes it easier to:
  – Figure out what you’re doing while writing the code
  – Figure out what the code is doing when you come back to look at it a year later
  – Have other people read and understand your code
Improving Readability

• Improving readability of your code can be accomplished in a number of ways
  – Comments
  – Meaningful variable names
  – Breaking code down into functions
  – Following consistent naming conventions
  – Programming language choice
  – File organization
Readability Example

• What does the following code snippet do?

```python
def nS(p, c):
    l = len(p)
    if (l >= 4):
        c += 1
        print(p)
    if (l >= 9):
        return p, c
# FUNCTION CONTINUES...
```

• There isn’t much information to go on, is there?
Readability Example

• What if I added meaningful variable names?

```python
def nS(p, c):
    l = len(p)
    if (l >= 4):
        c += 1
        print(p)
    if (l >= 9):
        return p, c
    # FUNCTION CONTINUES...
```
Readability Example

• What if I added meaningful variable names?

```python
def nextState(password, count):
    length = len(password)
    if (length >= 4):
        count += 1
        print(password)
    if (length >= 9):
        return password, count
# FUNCTION CONTINUES...
```
Readability Example

• And replaced the magic numbers with constants?

def nextState(password, count):
    length = len(password)
    if (length >= 4):
        count += 1
        print(password)
    if (length >= 9):
        return password, count
# FUNCTION CONTINUES...
Readability Example

• And replaced the magic numbers with constants?

```python
def nextState(password, count):
    length = len(password)
    if (length >= MIN_LENGTH):
        count += 1
        print(password)
        if (length >= MAX_LENGTH):
            return password, count
    # FUNCTION CONTINUES...
```
Readability Example

• And added vertical space?

```python
def nextState(password, count):
    length = len(password)
    if (length >= MIN_LENGTH):
        count += 1
        print(password)
        if (length >= MAX_LENGTH):
            return password, count
    # FUNCTION CONTINUES...
```
Readability Example

• And added vertical space?

```python
def nextState(password, count):
    length = len(password)

    if (length >= MIN_LENGTH):
        count += 1
        print(password)

    if (length >= MAX_LENGTH):
        return password, count

# FUNCTION CONTINUES...
```
Readability Example

• Maybe even some comments?

```python
def nextState(password, count):
    length = len(password)

    if (length >= MIN_LENGTH):
        count += 1
        print(password)

    if (length >= MAX_LENGTH):
        return password, count

# FUNCTION CONTINUES...
```
Readability Example

• Maybe even some comments?

```python
def nextState(password, count):
    length = len(password)

    # if long enough, count as a password
    if (length >= MIN_LENGTH):
        count += 1
        print(password)

    # if max length, don't do any more
    if (length >= MAX_LENGTH):
        return password, count

    # FUNCTION CONTINUES...
```
Readability Example

• Now the purpose of the code is a bit clearer!
  – You can see how small, simple changes increase the readability of a piece of code

• This is actually part of a function that creates a list of the passwords for a swipe-based login system on an Android smart phone
  • Dr. Gibson wrote a paper on this, available here
Commenting
Commenting is an “Art”

• Though it may sound pretentious, it’s true

• There are NO hard and fast rules for when a piece of code should be commented
  – Only guidelines
  – NOTE: This doesn’t apply to required comments like file headers and function headers!
General Guidelines

• If you have a complex conditional, give a brief overview of what it accomplishes
  
  # check if car fits customer criteria
  if color == "black" and int(numDoors) > 2 \ 
  and float(price) < 27000:

• If you did something you think was clever, comment that piece of code
  – So that “future you” will understand it!
General Guidelines

• If you have a complex conditional, give a brief overview of what it accomplishes

```python
# check if car fits customer criteria
if color == "black" and int(numDoors) > 2 \ 
    and float(price)
```

• If you did something you think was clever, comment that piece of code
  – So that “future you” will understand it!

This backslash symbol tells Python that the code will continue on the next line.
General Guidelines

• Don’t write obvious comments
  # iterate over the list
  for item in myList:

• Don’t comment every line
  # initialize the loop variable
  choice = 1
  # loop until user chooses 0
  while choice != 0:
General Guidelines

• **Do** comment “blocks” of code

```python
# calculate tip and total - if a party is
# large, set percent to minimum of 15%
if (numGuests > LARGE_PARTY):
    percent = MIN_TIP

tip = bill * percent
total = bill + tip
```
General Guidelines

• **Do** comment nested loops and conditionals

```python
listFib = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
listPrime = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

# iterate over both lists, checking to see if each
# fibonacci number is also in the prime list
for num1 in listFib:
    for num2 in listPrime:
        if (num1 == num2):
            print(num1, "is both a prime and a \ Fibonacci number!")
```

General Guidelines

• **Do** comment very abbreviated variables names (especially those used for constants)
  – You can even put the comment at the end of the line!

```plaintext
MIN_CH = 1  # minimum choice at menu
MAX_CH = 5  # maximum choice at menu
MENU_EX = 5  # menu choice to exit (stop)
P1_MARK = "x"  # player 1's marker
P2_MARK = "o"  # player 2's marker
```
Side Note: Global Constants

•Globals are variables declared outside of any function (including `main()`)
•Accessible to all functions and code in the file

•Your programs may **not** have global variables
•Your programs **may** use global **constants**
  – In fact, constants should generally be global
Side Note: Global Constants

• A constant defines a number (or string) once, and we use the constant instead of the value

• Constants are often used in multiple functions
  – Being global means they’re available to all functions

• A global variable will show up in a different font color from regular variables or code

```python
GLOBAL_VAR = 7

def main():
    localVar = 7
main()
```
“Good Code” – Adaptability
Adaptability

• Often, what a program is supposed to do evolves and changes as time goes on
  – Well-written flexible programs can be easily altered to do something new
  – Rigid, poorly written programs often take a lot of work to modify

• When coding, keep in mind that you might want to change or extend something later
Adaptability: Example

- Remember how we talked about not using “magic numbers” in our code?

Bad:

```python
def makeSquareGrid():
    temp = []
    for i in range(0, 10):
        temp.append([0] * 10)
    return temp
```

Good:

```python
def makeSquareGrid():
    temp = []
    for i in range(0, GRID_SIZE):
        temp.append([0] * GRID_SIZE)
    return temp
```

0 and 1 are not “magic” numbers – why?
Adaptability: Example

• We can change `makeSquareGrid()` to be an even more flexible function

Better:

```python
def makeSquareGrid(size):
    temp = []
    for i in range(0, size):
        temp.append([0] * size)
    return temp
```

Good:

```python
def makeSquareGrid():
    temp = []
    for i in range(0, GRID_SIZE):
        temp.append([0] * GRID_SIZE)
    return temp
```

# call makeSquareGrid
grid = makeSquareGrid(GRID_SIZE)
Solving Problems
Simple Algorithms

• Input
  – What information we will be given, or will ask for

• Process
  – The steps we will take to reach our specific goal

• Output
  – The final product that we will produce
More Complicated Algorithms

• We can apply the same principles of input, process, output to more complicated algorithms and programs

• There may be multiple sets of input/output, and we may perform more than one process
Complex Problems

• If we only take a problem in one piece, it may seem too complicated to even begin to solve
  – A program that recommends classes to take based on availability, how often the class is offered, and the professor’s rating
  – Creating a video game from scratch
Top Down Design
Top Down Design

• Computer programmers often use a *divide and conquer* approach to problem solving:
  – Break the problem into parts
  – Solve each part individually
  – Assemble into the larger solution

• One component of this technique is known as *top down design*
Top Down Design

• Breaking the problem down into pieces makes it more manageable to solve

• *Top-down design* is a process in which:
  – A big problem is broken down into small sub-problems
  • Which can themselves be broken down into even smaller sub-problems
    – And so on and so forth...
Top Down Design: Illustration

• First, start with a clear statement of the problem or concept

• A single big idea
Top Down Design: Illustration

• Next, break it down into several parts
Top Down Design: Illustration

• Next, break it down into several parts
• If any of those parts can be further broken down, then the process continues...
Top Down Design: Illustration

- And so on...
Top Down Design: Illustration

• Your final design might look like this chart, which shows the overall structure of the smaller pieces that together make up the “big idea” of the program.
Top Down Design: Illustration

• This is like an upside-down “tree,” where each of the nodes represents a process (or a function)
Top Down Design: Illustration

- The bottom nodes are “leaves” that represent pieces that need to be developed.
- They are then recombined to create the solution to the original problem.
Analogy: Paper Outline

• Think of it as an outline for a paper you’re writing for a class assignment

• You don’t just start writing things down!
  – You come up with a plan of the important points you’ll cover, and in what order
  – This helps you to formulate your thoughts as well
Implementing a Design in Code
Bottom Up Implementation

• Develop each of the modules separately
  – Test that each one works as expected
• Then combine into their larger parts
  – Continue until the program is complete
Bottom Up Implementation

• To test your functions, you will probably use `main()` as a (temporary) test bed
  – You can even call it `testMain()` if you want

• Call each function with different test inputs
  – How does function ABC handle zeros?
  – Does this `if` statement work right if XYZ?
  – Ensure that functions “play nicely” together
Top Down Implementation

• Create “dummy” functions that fulfill the requirements, but don’t perform their job
  – For example, a function that is supposed to take in a file name and return the weighted grades; it takes in a filename, but then simply returns a 1

• Write up a “functional” `main()` that calls these dummy functions
  – Helps to pinpoint other functions you may need
Which To Choose?

• Top down? Or bottom up?

• It’s up to you!
  – As you do more programming, you will develop your own preference and style

• For now, just use something – don’t code up everything at once without testing anything!
Design Example
Questions when Designing

• What is the “big picture” problem?

• What sort of tasks do you need to handle?
  – What functions would you make?
  – How would they interact?
  – What does each function take in and return?

• What will your `main()` look like?
In-Class Example

• A program that recommends classes to take based on availability, how often the class is offered, and the professor’s rating

• Spend a few minutes brainstorming now
  – “Big picture” problem
  – Tasks that need to be handled
  – What `main()` looks like
In-Class Example

• Specifics:
  – Get underlying data:
    • Availabilities (probably read in from a file)
    • Class offering frequency (again, from a file)
    • Professor rating (from, you guessed it, a file)
    • How to obtain this information in the first place?
  – Ask user what courses they want to take
  – Find out how many semesters they have left
  – etc...
Incremental Development
What is Incremental Development?

- Developing your program in small increments
  
  1. Program a small piece of the program
  2. Run and test your program
  3. Ensure the recently written code works
  4. Address any errors and fix any bugs
  5. Return to step 1
Why Use Incremental Development?

• Incremental development:
  – Makes a large project more manageable
  – Leads to higher quality code
  – Makes it easier to find and correct errors
  – Is faster for large projects

• May seem like you’re taking longer since you test at each step, but faster in the long run
Debugging Woes

• Writing code is easy...

• Writing code that works correctly is HARD

• Sometimes the hardest part of debugging is finding out where the error is coming from
  – And solving it is the easy part (sometimes!)

• If you only wrote one function, you can start by looking there for the error
Announcements

• Survey #2 is out
  – Due Sunday, Nov 13 @ 11:59 PM

• Project 1 is due next Wednesday
  – It is much harder than the homeworks
  – No collaboration allowed
  – Start early
  – Think before you code
  – Come to office hours