CMSC201
Computer Science I for Majors

Lecture 12 – Program Design and Modularity
Last Class We Covered

• Functions
  – Returning values
  – Matching parameters
  – Matching return assignments

• Mutability
  – Immutability
  – Effect on functions
Any Questions from Last Time?
Today’s Objectives

• To understand shallow copy

• To practice program design
  – With the max of three example

• To better understand the purpose of modularity, functions, and incremental development
  – Through a design example
Review: Mutability in Functions

Function is called, and formal parameter B is assigned the actual parameter A

A is **mutable** (lists, or dicts)

- B is modified in place
  - B.append(2)
- A changes if B changes

A is **immutable** (int, string, tuple)

- A doesn’t change if B changes

B is assigned to something else
- B = [0, 1]
- A doesn’t change if B changes

From http://stackoverflow.com/a/25670170
Shallow (and Deep) Copies
Copying Lists

• When you assign one list to another, it is by default a “shallow” copy of the list

• A *shallow copy* is when the new variable actually points to the old variable, rather than making an actual copy

• A *deep copy* is the opposite, creating an entirely new list for the new variable
  – This is what you probably want to be happening!
Shallow Copy

- When we make a shallow copy, we are essentially just giving the same list two different variable names.
  - This only happens to **mutable** data types, like lists, and only if we alter them in-place.

```
list1
list2
["red", "blue"]
```
Shallow Copy Example

• A shallow copy and its effects on the original:

```python
list1 = ['red', 'blue']  # original list
list2 = list1  # shallow copy made
list2.append('green')  # update shallow copy
list2[1] = 'yellow'  # and again
print("list1 (end): ", list1)
print("list2 (end): ", list2)
```

```
list1 (start): ['red', 'blue']
list1 (end):   ['red', 'yellow', 'green']
list2 (end):   ['red', 'yellow', 'green']
```
Deep Copy

• There are two easy ways to do a deep copy:
  – Use slicing, and “slice” out the **entire** list
  – **Cast** the original as a list when assigning

• With these, Python returns an entirely new list that you can then assigned to the new variable
  – Now you have two separate lists!
Deep Copy Example

```
list1 = ["red", "blue"]
list2 = list1[:] # use slicing to copy
list2[1] = "yellow"
list3 = list(list1) # use casting to copy
list3.append("purple")
print("original:     ", list1)
print("deep copy1:   ", list2)
print("deep copy2:   ", list3)
```

```
original:      ['red', 'blue']
deep copy1:    ['red', 'yellow']
deep copy2:    ['red', 'blue', 'purple']
```
Deep Copy

- Creates a copy of the entire list's contents, not just of the list itself
- Each variable now has its own individual list

```
list1: ["red", "blue"]
list2: ["red", "yellow"]
list3: ["red", "blue", "purple"]
```
Program Design Example
Study in Design: Max of Three

• You know about a lot of tools at this point in the semester, but knowing when and how to apply them may still be difficult sometimes

• Let’s create an algorithm to find the largest of three numbers

• Start off by writing the code to get the input from the user, and to print the final maximum
Max of Three: Code Framework

• Here’s the “easy” part of our code completed:

```python
def main():
    x1 = int(input("Please enter a value: "))
    x2 = int(input("Please enter a value: "))
    x3 = int(input("Please enter a value: "))

    # we need to write the missing code that sets
    # "maximum" to the value of the largest number

    print("The largest value is ", maximum)

main()
```
Max of Three: Strategies

• Spend a few minutes thinking about the different ways you could compare these three numbers to find the maximum

• Don’t write code right away – brainstorm first!

• Your first idea might not be your best idea, so be prepared to be flexible
Strategy 1: Compare Each to All

• This looks like a three-way decision, where we need to execute one of the following:

  \[
  \begin{align*}
  \text{maximum} &= x_1 \\
  \text{maximum} &= x_2 \\
  \text{maximum} &= x_3
  \end{align*}
  \]

• What we need to do now is preface each one of these with the right condition
Strategy 1: Solution

- Here’s our completed code:

```python
def main():
    # getting input goes here
    if x1 >= x2 and x1 >= x3:
        maximum = x1
    elif x2 >= x1 and x2 >= x3:
        maximum = x2
    else:
        maximum = x3

    print("The largest value is ", maximum)
main()
```
Strategy 1: Downsides

• What would happen if we were trying to find the max of five values?
  – We would need four Boolean expressions, each consisting of four conditions and’ded together

• What about twenty values?
  – We would need nineteen Boolean expressions, with nineteen conditions each

• There has to be a better way!
Strategy 2: Decision Tree

• We can avoid the redundant tests of the previous algorithm by using a decision tree

• Suppose we start with checking if $x_1 \geq x_2$
  – This knocks either $x_1$ or $x_2$ out of the running to be the maximum value
  – If the condition is True, then we move on to check whether $x_1$ or $x_3$ is larger
Strategy 2: Decision Tree Flowchart

Start

Get the 3 numbers

x1 >= x2

FALSE

x1 >= x3

x2 >= x3

TRUE

maximum = x3

FALSE

maximum = x1

TRUE

maximum = x1

FALSE

maximum = x3

FALSE

maximum = x2

FALSE

maximum = x3

End
Strategy 2: Decision Tree Code

• Here’s the code for the previous flowchart

```python
if x1 >= x2:
    if x1 >= x3:
        maximum = x1
    else:
        maximum = x3
else:
    if x2 >= x3:
        maximum = x2
    else:
        maximum = x3
```
Strategy 2: (Dis)advantages

• This approach makes exactly two comparisons between the three variables

• However, this approach is more complicated than the first
  – To find the max of four values you’d need if-else statements nested three levels deep with eight assignment statements
  – This isn’t much better than the last method!
Strategy 3: Sequential Processing

• How would you solve the problem?

• Since you’re not a computer, you could look at three numbers and know which is the largest
  – But what if there were one hundred numbers?

• One strategy is to scan the list for a big number
  – When one is found, mark it, and continue looking
  – If you find a larger value, mark it, erase the previous mark, and continue looking
Strategy 3: Sequential Processing

Start

Get the 3 numbers

maximum = x1

x2 > maximum

FALSE

TRUE

maximum = x2

x3 > maximum

FALSE

TRUE

maximum = x3

End
Strategy 3: Sequential Processing Code

• This idea can be easily done in Python code

```python
maximum = x1
if x2 >= maximum:
    maximum = x2
if x3 >= maximum:
    maximum = x3
```

Why do we use two `if` statements?

What would happen if we used an `if-elif` statement?
Strategy 3: Sequential Processing

• This process is pretty repetitive
  – Which means we could use a loop!

• We would repeat the following steps:
  1. Prompt the user for a number
  2. Compare it to the current maximum
  3. If it is larger, update the max value
  – Repeat until the user is done entering numbers

• Or combine it with a list of given numbers
Strategy 4: Take Advantage of Python

- Python has a built-in function called **max**
  - It takes in numbers and returns the max value

```python
def main():
    # getting input goes here
    maximum = max(x1, x2, x3)
    print("The largest value is ", maximum)
main()
```

- This is why we called our variable “**maximum**” instead of **max** – because **max** is already defined!
Modularity
Modularity

• A program being *modular* means that it is:
  • Made up of individual pieces (modules)
    – That can be changed or replaced
    – Without affecting the rest of the system

• So if we replace or change one function, the rest should still work, even after the change
Modularity

• With modularity, you can also reuse and repurpose your code

• What are some pieces of code you’ve had to write multiple times?
  – Getting input between some min and max
  – Using a sentinel loop to create a list
  – What else?
Functions and Program Structure

• So far, functions have been used as a mechanism for reducing code duplication

• Another reason to use functions is to make your programs more modular

• As the algorithms you design get increasingly complex, it gets more and more difficult to make sense out of the programs
Functions and Program Structure

• One option to handle this complexity is to break it down into smaller pieces

• Each piece makes sense on their own

• You can easily combine them together to form the complete program
Program Design Example
Vending Machine

• We want to write a program that simulates a vending machine

• How do we even start!?

• With questions:
  – What things do we want our program to be able to do?
  – What info does it need?
  – How will we store data?
Announcements

• Homework 5 is/was due Wednesday

• Homework 6 does not come out this week
  – It will come out the night of October 20th

• The midterm exam is when?
  – During class on October 19th and 20th!

• Review packets will be available in class on October 17th and 18th