Logistics

- No class on Monday, Feb 20th
- First midterm is Wednesday, March 1st, in class!
- Start reviewing. Ask me or tutors for help on concepts you don't understand. We'll review things in class as well.
Accumulator Variables and Augmented Assignment Operators
Many programming tasks require you to calculate the total of a series of numbers or the number of times you iterate through a loop.

We can utilize an “accumulator” variable to do this.
Using Accumulator Variables

- Set up your accumulator variables outside of your loops. Usually it's good style to initialize your accumulator variables right before beginning a repetition structure.

- Decide on a value you want to start your accumulator values at. 0 or 0.0 is generally a good starting point depending on whether you are counting whole numbers or numbers with fractional values.

- Use a self-referential assignment statement when incrementing an accumulator variable. Example:
  - counter = counter + 1
Self-referential assignment statements

```plaintext
# default x to 5
x = 5

x = x + 1

5 + 1 = 6
```
The self-referential assignment statement that we just used is extremely useful, and can be extended to use any of the other math operations we have covered in class so far.

- \( a = a + 1 \)
- \( b = b \times 2 \)
- \( c = c / 3 \)
- \( d = d - 4 \)
Augmented Assignment Operators

- However, Python (and most other programming languages) contains a series of “shortcuts” that can be used to cut down the amount of typing when working with self-referential assignment statements.

- We call these shortcuts the “augmented assignment operators”
### Augmented Assignment Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Equal to</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>c += 5</td>
<td>c = c + 5</td>
</tr>
<tr>
<td>-=</td>
<td>c -= 2</td>
<td>c = c - 2</td>
</tr>
<tr>
<td>*=</td>
<td>c *= 3</td>
<td>c = c * 2</td>
</tr>
<tr>
<td>/=</td>
<td>c /= 3</td>
<td>c = c / 3</td>
</tr>
<tr>
<td>%=</td>
<td>c %= 3</td>
<td>c = c % 3</td>
</tr>
</tbody>
</table>
Programming Challenge: Grocery Checkout Calculator

- Write a program that asks the user to enter in a series of price values
- Calculate a running total of these values
- Calculate sales tax (7%) on the total bill and display the result to the user at the end of the program
Programming Challenge: Coin Flips

- Write a program that simulates a coin flipping 1 million times
- Count the # of heads and tails that result, and display the result to the user after you have finished running the simulation
Programming Challenge: Rock, Paper, Scissors Tournament

- Write a program that lets the user play a game of Rock, Paper, Scissors against the computer
- End the game when either the player or the computer earns 3 points
Sentinels

Imagine that you want to ask your users to enter in a large number of items that need to be calculated in a certain way.

You don’t know how many values the user will be entering.

Given our current toolset we really only have ways to handle this kind of scenario:

- Ask the user at the end of each iteration if they want to continue. This can be annoying and make your program cumbersome if you will be entering in hundreds or thousands of values.
- Ask the user ahead of time how many items they will be entering. This can be difficult since the user may not know at the beginning of the loop how many items they will be working with.
Sentinels

- A sentinel value is a pre-defined value that the user can type in to indicate that they are finished entering data.

Example:

- >> Enter a test score (type -1 to end): 100
- >> Enter a test score (type -1 to end): 80
- >> Enter a test score (type -1 to end): -1
- >> Your test average is: 90 %

- In the example above the value -1 is considered a sentinel -- it indicates to the program that the user is finished entering data.

- Sentinels must be distinctive enough that they will not be mistaken for regular data (in the previous example the value -1 was used – there is no way that a “real” test value could be -1).

- An everyday example: hitting the "return" key tells the IDLE interpreter that you have finished entering a line of code.
Programming Challenge: Adding Machine

- Write a program that continually asks the user for an integer
- Add the supplied integer to a total variable
- When the user enters a 0 value end the program and display the sum for the user
Repetition Flow Control
The “break” command

- The “break” command is a special Python command that can be used to immediately end a loop.

- It will not end your program – it simply ends the current repetition structure and allows the program to pick up from the line directly after the end of your loop.

- Note that when the break command runs it will immediately terminate the current loop, which prevents any commands in the loop block after the break command from running.

- Best to avoid this if possible: do you really have two completely unrelated conditions for terminating your loop?
Trace the Output

```python
x = 0

while x < 10:
    if x >= 3:
        break
    print (x)

x += 1
```
Prime Number Tester

- Write a program that asks the user for an integer.
- Test to see if the number is prime. A prime number is any number that is evenly divisible only by 1 and itself.
- Extension on this: can you write a program to find all prime numbers between 1 and 100?
Simple Data Validation
Simple Data Validation

- Often we need to ask the user to supply a value in our programs

- But as you know you can't always trust the user to supply you with usable data!

- One strategy you can use to ensure that you get "good" data is to "validate" the user's input. This involves asking the user for a value – if it meets our criteria we can continue. If not we will need to ask the user to re-supply the value.
Nested Loops

- Just as you can nest select statements, you can also nest loops.

- One example: you want to have an input validation loop inside a loop that accepts inputs repeatedly and does calculations.

- Consider this:
  ```python
  while (x < 200):
      inp = -1
      while (inp < 0):
          inp = int(input("Enter a positive integer: "))
      x += inp
  print("Now x =",x)
  ```
break and Nested Loops

while (x < 200):
    inp = -1
while (True):
    inp = int(input("Enter a positive integer: ")
    if (inp < 1):
        print("That's not a positive int.")
    else:
        break
x += inp
print("Now x =",x)
Programming Challenge

- Write a program that asks the user for a positive integer
- Do not accept a negative value (or zero) – if the user supplies an invalid value you should re-prompt them
- Once you have a positive integer you can print that number of stars to the screen. For example:

  Enter a positive integer:  -5
  Invalid, try again!
  Enter a positive integer:  0
  Invalid, try again!
  Enter a positive integer:  5

  *****
Sources of Error in Loops
If your loop isn't working right...

- Are you using the correct condition to control it?
  - Counting (off-by-one) errors
  - Assignment instead of equality

- Does the condition change when you expect it to?

- Are you updating your loop variables in the right place?

- Will it ever end?