Lecture
Loops &&& Booleans
Class Administrivia
Agenda

• (In-)definite loops (for/while)
• Patterns: interactive loop and sentinel loop
• Solve problems using (possibly nested) loop patterns
• Intro to boolean algebra
Study: Max of Three
Study: Max of Three

- With decision structures, we can solve more complicated programming problems
- The negative is that writing these programs becomes harder
- Suppose we need an algorithm to find the largest of three numbers
def main():
    x1, x2, x3 = eval(input("Please enter three values: "))

    # missing code sets max to the value of the largest

    print("The largest value is", max)
Study: Max of Three
Strategy 1: Compare each to all
Study: Max of Three
Strategy 1: Compare each to all

• This looks like a three-way decision, where we need to execute one of the following:
  \[
  \text{max} = x_1 \\
  \text{max} = x_2 \\
  \text{max} = x_3 \\
  \]

• Wrap each one of these lines with the right condition
Study: Max of Three
Strategy 1: Compare each to all

• Let’s look at the case where \( x_1 \) is the largest.

\[
\text{if } x_1 \geq x_2 \geq x_3:\n\begin{align*}
\text{max} &= x_1
\end{align*}
\]

• Many languages would not allow this compound condition

• Python does allow it, though
Study: Max of Three
Strategy 1: Compare each to all

• Whenever you write a decision, there are two crucial questions:

  • **First:** When the condition is true, is executing the body of the decision the right action to take?

    • $x_1$ is at least as large as $x_2$ and $x_3$, so assigning $\text{max}$ to $x_1$ is appropriate.

    • Always pay attention to borderline values!!
Study: Max of Three
Strategy 1: Compare each to all

• **Second**: ask the converse of the first question, namely, are we certain that this condition is true in all cases where \( x_1 \) is the max?

  • Suppose the values are \( x_1=5 \), \( x_2=2 \), and \( x_3=4 \).

  • Clearly, \( x_1 \) is the largest, but does \( x_1 \geq x_2 \geq x_3 \) hold?

  • We don’t really care about the relative ordering of \( x_2 \) and \( x_3 \), so we can make two separate tests: \( x_1 \geq x_2 \) and \( x_1 \geq x_3 \).
Study: Max of Three
Strategy 1: Compare each to all

• We can separate these conditions with `and`!

```python
if x1 >= x2 and x1 >= x3:
    max = x1
elif x2 >= x1 and x2 >= x3:
    max = x2
else:
    max = x3
```

• We’re comparing each possible value against all the others to determine which one is largest.
Study: Max of Three
Strategy 2: Decision Tree
Study: Max of Three
Strategy 2: Decision Tree

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

• Suppose we start with $x_1 \geq x_2$
  
  • This knocks either $x_1$ or $x_2$ out of contention to be the max

• If the condition is true, we need to see which is larger, $x_1$ or $x_3$
Study: Max of Three
Strategy 2: Decision Tree

```python
if x1 >= x2:
    if x1 >= x3:
        max = x1
    else:
        max = x3
else:
    if x2 >= x3:
        max = x2
    else:
        max = x3
```
Study: Max of Three
Strategy 2: Decision Tree

• This approach makes exactly two comparisons, regardless of the ordering of the original three variables

• This approach is more complicated than the first.

• To find the max of four values you’d need `if-else` constructs nested three levels deep with eight assignment statements
Study: Max of Three
Strategy 3: Sequential Processing
Study: Max of Three

Strategy 3: Sequential Processing

• As a human, how would you solve the problem?

• You could probably look at three numbers and just know which is the largest
Study: Max of Three
Strategy 3: Sequential Processing

• One strategy is to scan through the list looking for a big number:

  1. When one is found, mark it, and continue looking

  2. If you find a larger value, mark it, erase the previous mark, and continue looking
Study: Max of Three
Strategy 3: Sequential Processing

max = x1

x2 > max
 True
 max = x2
 False

x3 > max
 True
 max = x3
 False

False
Study: Max of Three
Strategy 3: Sequential Processing

- This idea can easily be translated into Python

```python
max = x1
if x2 > max:
    max = x2
if x3 > max:
    max = x3
```
Study: Max of Three
Strategy 3: Sequential Processing

• This process is repetitive — we can use a loop:
  1. We prompt the user for a number
  2. We compare it to our current max
  3. If it is larger, we update the max value
  4. Repeat
Study: Max of Three
Strategy 3: Sequential Processing

# maxn.py
#    Finds the maximum of a series of numbers

def main():
    n = eval(input("How many numbers are there? "))

    # Set max to be the first value
    max = eval(input("Enter a number >> "))

    # Now compare the n-1 successive values
    for i in range(n-1):
        x = eval(input("Enter a number >> "))
        if x > max:
            max = x

    print("The largest value is", max)
Study: Max of Three
Strategy 4: Use Python
Study: Max of Three
Strategy 4: Use Python

- Python has a built-in function called max that returns the largest of its parameters.

```python
def main():
x1, x2, x3 = eval(input("Please enter three values: "))
print("The largest value is", max(x1, x2, x3))
```
For Loops: A Review
For Loops: A Review

• The for statement allows us to iterate through a sequence of values.

    for <var> in <sequence>:
        <body>

• The loop index variable var is successively assigned a value in the sequence

• Then, statements in the body of the loop are executed once for each of theses values
For Loops: A Review

• Suppose we want to write a program that can compute the average of a series of numbers entered by the user

• To make the program general, it should work with any size set of numbers

• We don’t need to keep track of each number entered, we only need know the running sum and how many numbers have been added
For Loops: A Review

• We’ve run into some of these things before!
  • A series of numbers could be handled by some sort of loop
    • If there are $n$ numbers, the loop should execute $n$ times
  • We need a running sum
    • This will use an accumulator
Note that sum is initialized to 0.0 so that \texttt{sum/n} returns a float!

```python
# averagel1.py
#    A program to average a set of numbers
#    Illustrates counted loop with accumulator

def main():
    n = eval(input("How many numbers do you have? "))
    sum = 0.0
    for i in range(n):
        x = eval(input("Enter a number >> "))
        sum = sum + x
    print("\nThe average of the numbers is", sum / n)
```
For Loops: A Review

How many numbers do you have? 5
Enter a number >> 32
Enter a number >> 45
Enter a number >> 34
Enter a number >> 76
Enter a number >> 45

The average of the numbers is 46.4
For Loops: zip()
Indefinite Loops
Indefinite Loops

• That last program got the job done, *but you need to know ahead of time* how many numbers you’ll be dealing with

• What we need is a way for the computer to take care of counting how many numbers there are

• The **for** loop is a *definite loop*, meaning that the number of iterations is determined when the loop starts
Indefinite Loops

• We can’t use a definite loop unless we know the number of iterations ahead of time
  • We can’t know how many iterations we need until all the numbers have been entered
• We need another loop construct!
• The indefinite or conditional loop keeps iterating until certain conditions are met
Indefinite Loops

while <condition>: 
  <body>

• A Boolean expression, <condition>, is similar to ones used in if statements
  • The body is a sequence of one or more statements
• The body of the loop executes repeatedly as long as the condition remains true
• When the condition is false, the loop terminates
Indefinite Loops

• The condition is tested at the top of the loop

• This is known as a **pre-test loop**

• If the condition is initially false, the loop body will not execute at all
Indefinite Loops

• An example of a `while` loop that counts from 0 to 10:

```python
i = 0
while i <= 10:
    print(i)
    i = i + 1
```

• The code has the same output as this `for` loop:

```python
for i in range(11):
    print(i)
```
Indefinite Loops

- The `while` loop requires us to manage the loop variable `i` by initializing it to `0` before the loop and incrementing it at the bottom of the body.

- In the `for` loop this is handled automatically.
Indefinite Loops

• The while statement is simple, but yet powerful and dangerous – they are a common source of program errors

```python
i = 0
while i <= 10:
    print(i)
```

• What happens with this code?
Indefinite Loops

• When Python gets to this loop, \( i \) is equal to 0, which is less than 10, so the body of the loop is executed, printing 0. Now control returns to the condition, and since \( i \) is still 0, the loop repeats, etc.

• This is an example of an infinite loop
Indefinite Loops

• What should you do if you’re caught in an infinite loop?
• First, try pressing control-c
• If that doesn’t work, try control-alt-delete*
• If that doesn’t work, push the reset button!
Interactive Loops
Interactive Loops

- One good use of the indefinite loop is to write *interactive loops*
  - Interactive loops allow a user to repeat certain portions of a program on demand
- Remember how we said we needed a way for the computer to keep track of how many numbers had been entered?
  - Let’s use another accumulator, called `count`. 
Interactive Loops

- At each iteration of the loop, ask the user if there is more data to process. We need to preset it to “yes” to go through the loop the first time.

  ```
  set moredata to “yes”
  while moredata is “yes”
    get the next data item
    process the item
    ask user if there is moredata
  ```
Interactive Loops

- Combining the interactive loop pattern with accumulators for \texttt{sum} and \texttt{count}:

  ```
  initialize sum to 0.0
  initialize count to 0
  initialize moredata to "yes"
  while moredata is "yes"
    input a number, x
    add x to sum
    add 1 to count
    ask user if there is moredata
  output sum/count
  ```
Interactive Loops

• Using string indexing (moredata[0]) allows us to accept “y”, “yes”, “yeah” to continue the loop

# average2.py
#    A program to average a set of numbers
#    Illustrates interactive loop with two accumulators

def main():
    moredata = "yes"
    sum = 0.0
    count = 0

    while moredata[0] == 'y':
        x = eval(input("Enter a number >> "))
        sum = sum + x
        count = count + 1
        moredata = input("Do you have more numbers (yes or no)? ")

    print("The average of the numbers is", sum / count)
Interactive Loops

Enter a number >> 32
Do you have more numbers (yes or no)? y
Enter a number >> 45
Do you have more numbers (yes or no)? yes
Enter a number >> 34
Do you have more numbers (yes or no)? yup
Enter a number >> 76
Do you have more numbers (yes or no)? y
Enter a number >> 45
Do you have more numbers (yes or no)? nah

The average of the numbers is 46.4
Sentinel Loops
Sentinel Loops

• A *sentinel loop* continues to process data until reaching a special value that signals the end
  • This special value is called the *sentinel*
  • The sentinel must be distinguishable from the data since it is not processed as part of the data
Sentinel Loops

get the first data item
while item is not the sentinel
    process the item
    get the next data item

• The first item is retrieved before the loop starts
  • This is sometimes called the priming read, since it gets the process started
• If the first item is the sentinel, the loop terminates and no data is processed
• Otherwise, the item is processed and the next one is read
Sentinel Loops

# average3.py
#    A program to average a set of numbers
#    Illustrates sentinel loop using negative input as sentinel

def main():
    sum = 0.0
    count = 0
    x = eval(input("Enter a number (negative to quit) >> "))
    while x >= 0:
        sum = sum + x
        count = count + 1
        x = eval(input("Enter a number (negative to quit) >> "))
    print("\nThe average of the numbers is", sum / count)
Sentinel Loops

Enter a number (negative to quit) >> 32
Enter a number (negative to quit) >> 45
Enter a number (negative to quit) >> 34
Enter a number (negative to quit) >> 76
Enter a number (negative to quit) >> 45
Enter a number (negative to quit) >> -1

The average of the numbers is 46.4
Sentinel Loops

• This version provides the ease of use of the interactive loop without the hassle of typing ‘y’ all the time.

• However, using this method we can’t average a set of positive and negative numbers

• If we do this, our sentinel can no longer be a number
Sentinel Loops

• We could input all the information as strings
  • Valid input would be converted into numeric form, i.e. use a character-based sentinel
  • We could use the empty string (""")!
initialize sum to 0.0  
initialize count to 0  
input data item as a string, xStr  
while xStr is not empty  
  convert xStr to a number, x  
  add x to sum  
  add 1 to count  
  input next data item as a string, xStr  
Output sum / count
# average4.py
# A program to average a set of numbers
# Illustrates sentinel loop using empty string as sentinel

def main():
    sum = 0.0
    count = 0
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != ":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit) >> ")
    print("The average of the numbers is", sum / count)
Sentinel Loops

Enter a number (<Enter> to quit) >> 34
Enter a number (<Enter> to quit) >> 23
Enter a number (<Enter> to quit) >> 0
Enter a number (<Enter> to quit) >> -25
Enter a number (<Enter> to quit) >> -34.4
Enter a number (<Enter> to quit) >> 22.7
Enter a number (<Enter> to quit) >>

The average of the numbers is 3.38333333333
File Loops
# average4.py
# A program to average a set of numbers
# Illustrates sentinel loop using empty string as sentinel

def main():
    sum = 0.0
    count = 0
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != ":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit) >> ")
    print("The average of the numbers is", sum / count)
File Loops

- The biggest disadvantage of our program at this point is that they are interactive.
- What happens if you make a typo on number 43 out of 50?
- A better solution for large data sets is to read the data from a file.
File Loops

• Many languages don’t have a mechanism for looping through a file directly, like Python can
  • Rather, they use a sentinel

• We could use `readline()` in a loop to get the next line of the file

• At the end of the file, `readline()` returns an empty string, ""
File Loops

# average5.py
# Computes the average of numbers listed in a file.

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    for line in infile:
        sum = sum + eval(line)
        count = count + 1
    print("\nThe average of the numbers is", sum / count)
    infile.close()
File Loops

- Does this code correctly handle the case where there’s a blank line in the file?

```python
line = infile.readline()
while line != ""
    #process line
    line = infile.readline()
```
File Loops

• Yes

• An empty line in a file actually ends with the newline character, and `readline()` includes the newline: “\n” != “”
File Loops

# average6.py
# Computes the average of numbers listed in a file.

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    line = infile.readline()
    while line != "":
        sum = sum + eval(line)
        count = count + 1
        line = infile.readline()
    print("\nThe average of the numbers is", sum / count)
Nested Loops
Nested Loops

• We know we can nest if statements
  • We can also nest loops!

• Suppose we change our program specification to allow any number of numbers on a line in the file (separated by commas), rather than one per line
Nested Loops

numbers1.txt
144
15
20
12
10
12
17
14
30
64

d/articles/numbers1.txt

numbers2.txt
10,33,352,37,13
42,182,22,156,13
22,20,15,11,14
26,10,10,110,43
28,108,90,10,15
24,73,46,20,12
14,10,30,18,15
87,34,44,164,13
433,11,29,187,26
28,11,23,35,28

d/articles/numbers2.txt
Nested Loops

- At the top level, we will use a file-processing loop that computes a running `sum` and `count`

```python
sum = 0.0
count = 0
line = infile.readline()
while line != "":
    #update sum and count for values in line
    line = infile.readline()
print("\nThe average of the numbers is", sum/count)
```
Nested Loops

• In the next level in we need to update the sum and count in the body of the loop.

• Since each line of the file contains one or more numbers separated by commas, we can split the string into substrings, each of which represents a number.

• Then we need to loop through the substrings, convert each to a number, and add it to sum.

• We also need to update count.
Nested Loops

# average7.py
# Computes the average of numbers listed in a file.
# Works with multiple numbers on a line.

import string

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    line = infile.readline()
    while line != "":
        for xStr in line.split("",""):
            sum = sum + eval(xStr)
            count = count + 1
        line = infile.readline()
    print("\nThe average of the numbers is", sum / count)
Nested Loops

• The loop that processes the numbers in each line is indented inside of the file processing loop.

• The outer while loop iterates once for each line of the file.

• For each iteration of the outer loop, the inner for loop iterates as many times as there are numbers on the line.

• When the inner loop finishes, the next line of the file is read, and this process begins again.
Nested Loops

• Designing nested loops
  • Design the outer loop without worrying about what goes inside
  • Design what goes inside, ignoring the outer loop
  • Put the pieces together, preserving the nesting
Computing with Booleans
Computing with Booleans

- *if* and *while* both use *Boolean expressions*
- Boolean expressions evaluate to the Python *True* or *False* data types
- So far we’ve used Boolean expressions to compare two values, e.g. *while* *x* >= 0:
Boolean Operators
Boolean Operators

• Sometimes our simple expressions do not seem expressive enough

• Suppose you need to determine whether two points are in the same position – their x coordinates are equal and their y coordinates are equal
Boolean Operators

• This is an awkward way to evaluate multiple Boolean expressions

• Let’s check out the three Boolean operators **and**, **or**, and **not**.

```python
if p1.getX() == p2.getX():
    if p1.getY() == p2.getY():
        # points are the same
    else:
        # points are different
else:
    # points are different
```
Boolean Operators

- **Conjunction**: The and of two expressions is true exactly when both of the expressions are true.

- We can represent this in a truth table:

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P and Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Boolean Operators

• In the truth table, $P$ and $Q$ represent smaller Boolean expressions

• Since each expression has two possible values, there are four possible combinations of values

• The last column gives the value of $P$ and $Q$
• **Disjunction:** The or of two expressions is true when either expression is true

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P or Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Boolean Operators

• The only time or is false is when both expressions are false

• Also, note that or is true when both expressions are true
  • This isn’t how we normally use “or” in language
Boolean Operators

- **Negation:** The \textit{not} operator computes the opposite of a Boolean expression.

- The \textit{not} operator is a \textit{unary}, meaning it operates on a single expression.

<table>
<thead>
<tr>
<th>P</th>
<th>not P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
Boolean Operators

• We can put these operators together to make arbitrarily complex Boolean expressions

• The interpretation of the expressions relies on the precedence rules for the operators
• Consider:

   \[ a \text{ or} \ \text{not} \ b \ \text{and} \ c \]

• How should this be evaluated?

• The order of precedence, from high to low, is \text{not}, \text{and}, then \text{or}
Boolean Operators

• Most people don’t memorize the Boolean precedence rules, so using parentheses to prevent confusion is certainly encouraged.

• This statement is equivalent to:

\[(a \text{ or } ((\text{not } b) \text{ and } c))\]
Boolean Operators

• To test for the co-location of two points, we could use an `and`

```python
if p1.getX() == p2.getX() and p2.getY() == p1.getY():
    # points are the same
else:
    # points are different
```

• The entire condition will be true only when both of the simpler conditions are true
Boolean Operators

• Say you’re writing a racquetball simulation
  • The game is over as soon as either player has scored 15 points
• How can you represent that in a Boolean expression?

  \[
  \text{scoreA} == 15 \text{ or } \text{scoreB} == 15
  \]

• When either of the conditions becomes true, the entire expression is true, if neither condition is true, the expression is false
Boolean Operators

• We want to construct a loop that continues as long as the game is not over

• You can do this by taking the negation of the game-over condition as your loop condition!

```python
while not (scoreA == 15 or scoreB == 15):
    #continue playing
```
Boolean Operators

• Some racquetball players also use a shutout condition to end the game, where if one player has scored 7 points and the other person hasn’t scored yet, the game is over

```python
while not(scoreA == 15 or scoreB == 15 or 
  (scoreA == 7 and scoreB == 0) or 
  (scoreB == 7 and scoreA == 0)):
    # continue playing
```
Boolean Operators

• Let’s look at volleyball scoring
  • To win, a volleyball team needs to win by at least two points
  • In volleyball, a team wins at 15 points
  • If the score is 15 – 14, play continues, just as it does for 21 – 20

\[
(a \geq 15 \text{ and } a - b \geq 2) \text{ or } (b \geq 15 \text{ and } b - a \geq 2)\\
(a \geq 15 \text{ or } b \geq 15) \text{ and } \text{abs}(a - b) \geq 2
\]
Boolean Algebra
The ability to formulate, manipulate, and reason with Boolean expressions is an important skill.

Boolean expressions obey certain algebraic laws called *Boolean logic* or *Boolean algebra*.
Boolean Algebra

• **and** has properties similar to multiplication
• **or** has properties similar to addition
• 0 and 1 correspond to false and true, respectively.

<table>
<thead>
<tr>
<th>Algebra</th>
<th>Boolean algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a * 0 = 0$</td>
<td>$a$ and False == False</td>
</tr>
<tr>
<td>$a * 1 = a$</td>
<td>$a$ and True == $a$</td>
</tr>
<tr>
<td>$a + 0 = a$</td>
<td>$a$ or False == $a$</td>
</tr>
</tbody>
</table>
Boolean Algebra

• Anything or-ed with True is true:
  \((a \text{ or True}) == \text{true}\)

• Both and and or distribute:
  \((a \text{ or (b and c)}) == ((a \text{ or b}) \text{ and (a or c)})\)
  \((a \text{ and (b or c)}) == ((a \text{ and b}) \text{ or (a and c)})\)

• Double negatives cancel out:
  \(\text{not(not a)} == a\)

• DeMorgan’s laws:
  \(\text{not(a or b)} == (\text{not a}) \text{ and (not b)}\)
  \(\text{not(a and b)} == (\text{not a}) \text{ or (not b)}\)
• We can use these rules to simplify our Boolean expressions

```python
while not (scoreA == 15 or scoreB == 15):
    #continue playing
```

• This is saying something like “While it is not the case that player A has 15 or player B has 15, continue playing.”

• Applying DeMorgan’s law:

```python
while (not scoreA == 15) and (not scoreB == 15):
    #continue playing
```
Boolean Algebra

• This becomes:

```python
while scoreA != 15 and scoreB != 15
    # continue playing
```

• Isn’t this easier to understand?

• “While player A has not reached 15 and player B has not reached 15, continue playing.”
Boolean Algebra

• It can be easier to specify when a loop should *stop*, rather than when the loop should *continue*.

• In this case, write the loop termination condition and put a *not* in front of it.

  • After a couple applications of DeMorgan’s law you are ready to go with a simpler but equivalent expression.
Other Common Structures
Other Common Structures

- The **if** and **while** can be used to express every conceivable algorithm.
- For certain problems, an alternative structure can be convenient.
Post-Test Loop
Post-Test Loop

• A **post-test loop** is a construct that exists in other languages

• Python only has a pre-test loop — **while**

• Some languages have explicit post-test loops, where the condition is checked after the first iteration, usually called *do-while/do-until loops*
Post-Test Loop

• When the condition test comes after the body of the loop it’s called a post-test loop

• A post-test loop always executes the body of the code at least once.

• Python doesn’t have a built-in statement to do this, but we can do it with a slightly modified while loop.
Post-Test Loop

repeat
    get a number from the user
until number is >= 0
Indefinite Loops

- A post-test loop, in the C language

```c
int counter = 5;
int factorial = 1;

do {
    factorial *= counter--; /* Multiply, then decrement. */
} while (counter > 0);
printf("factorial of 5 is %d\n", factorial);
```
Post-Test Loop

- A post-test loop, made with a `while` loop, in Python

```python
counter, factorial=5, 1
while True:
    factorial *= counter
    counter -= 1
if not (counter>0):
    break
print(factorial)
```
Post-Test Loop

• Say we want to write a program that is supposed to get a nonnegative number from the user

• If the user types an incorrect input, the program asks for another value

• This process continues until a valid value has been entered

• This process is *input validation*
Post-Test Loop

• We seed the loop condition so we’re guaranteed to execute the loop once

```python
number = -1
while number < 0:
    number = eval(input("Enter a positive number: "))
```

• By setting `number` to –1, we force the loop body to execute at least once.
Post-Test Loop

• Some programmers prefer to simulate a post-test loop by using the Python `break` statement

• Executing `break` causes Python to immediately exit the enclosing loop

• `break` is sometimes used to exit what looks like an infinite loop
• The same algorithm implemented with a break

• A `while` loop continues as long as the expression evaluates to `True`. Since `True` always evaluates to `True`, it looks like an infinite loop!

```python
while True:
    number = eval(input("Enter a positive number: "))
    if x >= 0:
        break  # Exit loop if number is valid
```
• When the value of \( x \) is nonnegative, the \texttt{break} statement executes, which terminates the loop.

• Wouldn’t it be nice if the program gave a warning when the input was invalid?
Post-Test Loop

• If the body of an `if` is only one line long, you can place it right after the `:`

```python
while True:
    number = eval(input("Enter a positive number: "))
    if x >= 0: break  # Exit loop if number is valid
```
Post-Test Loop

- Adding the warning to the `break` version only adds an `else` statement:

```python
while True:
    number = eval(input("Enter a positive number: "))
    if x >= 0:
        break  # Exit loop if number is valid
    if:
        print("The number you entered was not positive.")
```
while (not edge) {
    run();
}

do {
    run();
} while (not edge);
Loop-and-a-Half
Loop-and-a-Half

- Stylistically, some programmers prefer the following approach
- Here the loop exit is in the middle of the loop body
- This is what we mean by a loop and a half

```python
while True:
    number = eval(input("Enter a positive number: "))
    if x >= 0: break  # Loop exit
print("The number you entered was not positive")
```
Loop-and-a-Half

• The loop and a half is an elegant way to avoid the priming read in a sentinel loop

• This method is faithful to the idea of the sentinel loop, the sentinel value is not processed

```python
while True:
    get next data item
    if the item is the sentinel: break
    process the item
```
Loop-and-a-Half

1. Get next data item
2. Check if item is the sentinel:
   - True: Stop
   - False: Process the item

The diagram illustrates a loop that processes items until a sentinel is reached.
Loop-and-a-Half

• To use or not use `break`, that is the question!

• The use of `break` is mostly a matter of style and taste

• Avoid using `break` often within loops, because the logic of a loop is hard to follow when there are multiple exits
Boolean Expressions as Decisions
Boolean Expressions as Decisions

- Boolean expressions can be used as control structures themselves.
- Suppose you’re writing a program that keeps going as long as the user enters a response that starts with ‘y’ (like our interactive loop).
- One way you could do it:

```python
while response[0] == "y" or response[0] == "Y":
```
Boolean Expressions as Decisions

• Be careful! You can’t take shortcuts:

```python
while response[0] == "y" or "Y":
```

• Why doesn’t this work?

• Python has a `bool` type that internally uses 1 and 0 to represent `True` and `False`, respectively

• The Python condition operators, like `==`, always evaluate to a value of type `bool`
Boolean Expressions as Decisions

- However, Python will let you evaluate any built-in data type as a Boolean

- For numbers (\texttt{int()}, \texttt{float()}), zero is considered \texttt{False}, anything else is considered \texttt{True}
Boolean Expressions as Decisions

>>> bool(0)
False
>>> bool(1)
True
>>> bool(32)
True
>>> bool("Hello")
True
>>> bool(""")
False
>>> bool([1,2,3])
True
>>> bool([])
False
Boolean Expressions as Decisions

- An empty sequence is interpreted as **False** while any non-empty sequence is taken to mean **True**
- The Boolean operators have operational definitions that make them useful for other purposes
# Boolean Expressions as Decisions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operational definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x and y</td>
<td>If x is false, return x. Otherwise, return y.</td>
</tr>
<tr>
<td>x or y</td>
<td>If x is true, return x. Otherwise, return y.</td>
</tr>
<tr>
<td>not x</td>
<td>If x is false, return True. Otherwise, return False.</td>
</tr>
</tbody>
</table>
Boolean Expressions as Decisions

• Consider \(x\) and \(y\)
  
  • In order for this to be true, both \(x\) and \(y\) must be true.

• As soon as one of them is found to be false, we know the expression as a whole is false and we don’t need to finish evaluating the expression

• So, if \(x\) is false, Python should return a false result, namely \(x\)
Boolean Expressions as Decisions

- If \( x \) is true, then whether the expression as a whole is true or false depends on \( y \).
- By returning \( y \), if \( y \) is true, then true is returned.
  - If \( y \) is false, then false is returned.
Boolean Expressions as Decisions

• These definitions show that Python’s Booleans are *short-circuit operators*, meaning that a true or false is returned as soon as the result is known.

• In an **and** where the first expression is false and in an **or**, where the first expression is true, Python will not evaluate the second expression.
Boolean Expressions as Decisions

$response[0] == "y" \text{ or } "Y"

• The Boolean operator is combining two operations.

• Here’s an equivalent expression:

$(response[0] == "y") \text{ or } ("Y")$

• By the operational description of or, this expression returns either \textbf{True}, if $response[0]$ equals “y”, or “Y”, both of which are interpreted by Python as true.
Boolean Expressions as Decisions

• Sometimes we write programs that prompt for information but offer a default value obtained by simply pressing <Enter>.

• Since the string used by ans can be treated as a Boolean, the code can be further simplified.
Boolean Expressions as Decisions

```python
ans = input("What flavor of you want [vanilla]: ")
if ans:
    flavor = ans
else:
    flavor = "vanilla"

• If the user just hits <Enter>, `ans` will be an empty string, ",", which Python interprets as false
Boolean Expressions as Decisions

• We can code this even more succinctly!

```python
ans = input("What flavor of you want [vanilla]: ")
flavor = ans or "vanilla"
```

• Remember, any non-empty answer is interpreted as True

• This exercise could be boiled down into one line!

```python
flavor = input("What flavor do you want [vanilla]: ") or "vanilla"
```
Boolean Expressions as Decisions

- Again, if you understand this method, feel free to utilize it
  - Just make sure that if your code is tricky, that it’s well documented!
Questions?