Loops and Booleans
Chapter 8.1-8.4, 8.5.2

LEARNING OUTCOMES

- To understand the concepts of definite and indefinite loops as they are realized in the Python for and while statements.
- To understand the interactive loop and sentinel loop programming patterns and their implementations using a Python while statement.
- To understand the end-of-file loop programming pattern and ways of implementing such loops in Python.
- To be able to design and implement solutions to problems involving loop patterns including nested loop structures.
- To understand the basic ideas of Boolean algebra and be able to analyze and write Boolean expressions involving Boolean operators.

FLOW CONTROL

- In the last chapter, we learned about decision structures, which control the flow of the program via selection.
- In this chapter, we will learn about another mode of flow control -- repetition.

Loops

- Earlier in the semester, we learned about the definite loop.

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

- The variable *var* takes on each value in the *sequence*, and executes the statements inside *body* once for each value.

```
for i in (1, 3, 5, 7, 9):
    print("i is", i)
```

output

```
i is 1
i is 3
i is 5
i is 7
i is 9
```

EXAMPLE

- Write a program that will calculate the average of *n* numbers input by the user.
- Your program should first ask the user for the value of *n*, and then ask the user to input each number in turn. It should print out the average of the numbers at the end.

PROGRAM DESIGN

- Our program will use some common design patterns that you may already be familiar with (but just didn't know what they were called).
- We are dealing with a series of numbers.
  - Can handle this with a loop.
  - We know the number of numbers -- therefore, we can use a definite loop that "counts" up to that number.
- We need to keep track of a running sum, which will increase with each additional number entered.
  - Can handle this with a variable that accumulates with each iteration of the loop, called a loop accumulator.

PROGRAM DESIGN

1. Ask the user for the value of n
2. Initialize a variable, sum, to 0.
3. Loop n times
    i. Input a number, x.
    ii. Add x to sum
4. Output the average as sum/n

PROGRAM DESIGN

1. Ask the user for the value of n
2. Initialize a variable, sum, to 0.
3. Loop n times
    i. Input a number, x.
    ii. Add x to sum
4. Output the average as sum/n

definite loop

PROGRAM DESIGN

1. Ask the user for the value of n
2. Initialize a variable, sum, to 0.
3. Loop n times
    i. Input a number, x.
    ii. Add x to sum
4. Output the average as sum/n

loop accumulator

definite loop

My Program

```
# average1.py

def main():
      print("This program calculates the average of n numbers")
```

Take Away Point

- As you program more, you will notice that there are some common patterns that come up again and again.
  - Will always be in use regardless of the programming language.
- We have seen two of them in this last program:
  - The definite loop (or the counted loop)
  - The loop accumulator

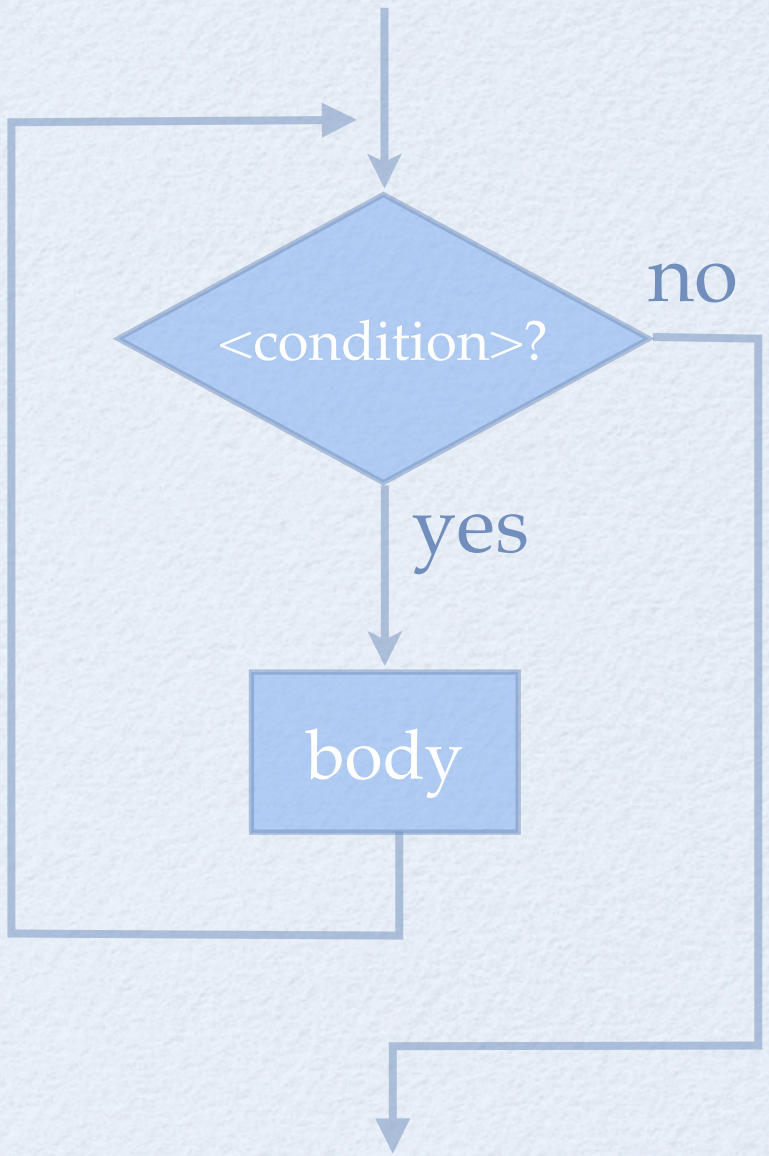
INDEFINITE LOOPS

- Our last program started out by asking the user how many numbers he/she had to average.
- Could we get the computer to take care of the counting, so we only enter the values?
- There's another kind of loop that can do this for us -- the **indefinite loop**, or **conditional loop**.
  - Keeps on iterating until certain conditions are met.

INDEFINITE LOOPS

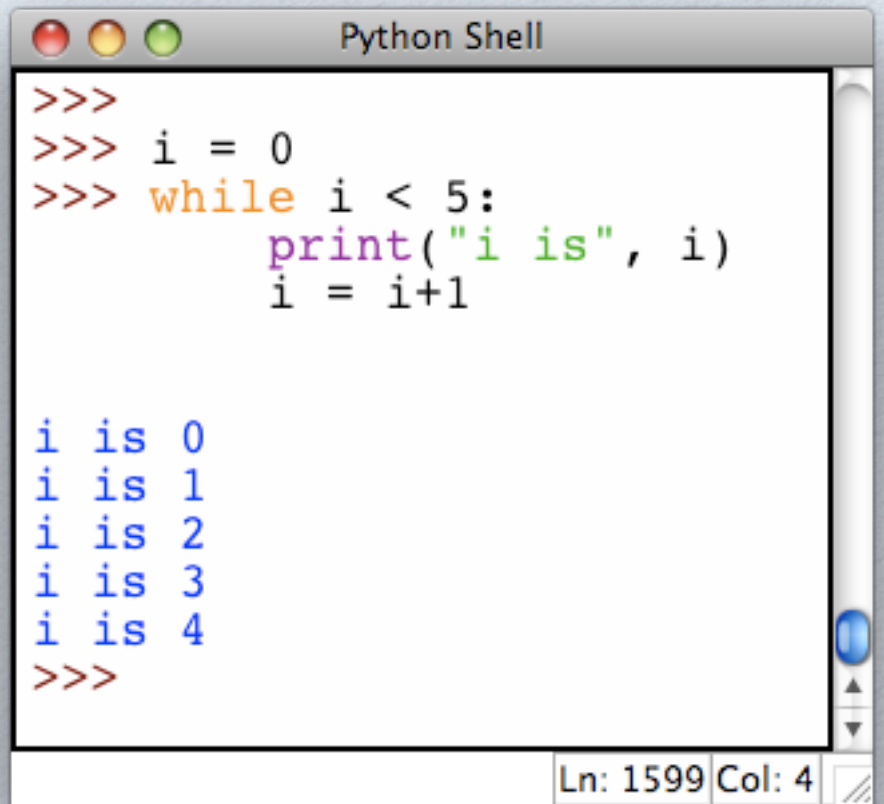
- The indefinite loop in Python is implemented using the keyword `while`.

```
while <condition>:
    <body>
```



Example of a while loop

- This loop has the same output as a `for` loop to `range(5)` and the same body.
- However, note that we need to initialize the variable `i` before the loop, and increment it ourselves on each iteration.
- In a `for` loop, this is done for us automatically.



Interactive Loops

- We can make use of the indefinite loop to write **interactive loops**.
  - Allows the user to repeat certain parts of the program on demand.

EXAMPLE

- Let's rewrite the average program to keep on asking the user for numbers until he/she decides to stop.
- We need an indefinite loop for this.
- We also need a loop accumulator to keep track of the sum of values that the user has entered.
  - Call it `sum`
- We also need another accumulator variable to keep track of the number of values that the user has entered.
  - Call it `count`

General Interactive Loop Pattern

1. set moredata to "yes"
2. while moredata is "yes"
   i) Get the next data item
   ii)Process the data item
   iii)Ask user if there is moredata

OUR ALGORITHM

1. Initialize sum to 0
2. Initialize count to 0
3. Set moredata to "yes"
4. While moredata is "yes"
   i) Input a number, x
   ii) Add x to sum
   iii) Add 1 to count
   iv) Ask user if there are any more numbers
5. Output sum/count

OUR ALGORITHM

1. Initialize sum to 0
2. Initialize count to 0
3. Set moredata to "yes"
4. While moredata is "yes"
   i) Input a number, x
   ii) Add x to sum
   iii) Add 1 to count
   iv) Ask user if there are any more numbers
5. Output sum/count

OUR ALGORITHM

1. Initialize sum to 0
2. Initialize count to 0
3. Set moredata to "yes"
4. While moredata is "yes"
   i) Input a number, x
   ii) Add x to sum
   iii) Add 1 to count
   iv) Ask user if there are any more numbers
5. Output sum/count

loop accumulator

accumulator

OUR ALGORITHM

1. Initialize sum to 0
2. Initialize count to 0
3. Set moredata to "yes"
4. While moredata is "yes"
   i) Input a number, x
   ii) Add x to sum
   iii) Add 1 to count
   iv) Ask user if there are any more numbers
5. Output sum/count

loop accumulator

accumulator

indefinite loop

PYTHON PROGRAM

```
# average2.py

def main():
      print("This program calculates the average of some numbers")

      sum = 0
      count = 0
      moredata = "yes"
      while moredata[0] == "y":
          x = eval(input("Enter a number >> "))
          sum = sum + x
          count = count + 1
          moredata = input("Any more numbers? (yes/no) ")
      print("The average of", count, "numbers is", sum/count)

main()
```

PROGRAM OUTPUT

```
J:\> python average2.py
This program calculates the average of some numbers
Enter a number >> 3
Any more numbers? (yes/no) y
Enter a number >> 5
Any more numbers? (yes/no) yes
Enter a number >> 10
Any more numbers? (yes/no) yes
Enter a number >> 7
Any more numbers? (yes/no) nope
The average of 4 numbers is 6.25
J:\>
```

program allows for varied responses (not just yes/no). How did we do it?

Sentinel Loops

- Our interactive loop program kept on asking the user whether there was a new value, before asking for the value.
- Can't we simply "signal" the end of the list of values by using a "special" value?
- That is the idea behind a **sentinel loop**.
  - The special value is called the **sentinel**, and **is not processed** with the rest of the data.
  - Any value may be chosen for the sentinel, but it must be distinguishable from the real data values.

General Sentinel Loop Pattern

The first item is read in before the loop starts.

Also called the **priming read** as it gets the loop started.

If the first item is the sentinel value, the loop never executes.

1. Get the first data item

2. while item is not the sentinel value

   i) Process the data item

   ii) Get the next data item

Sentinel Loops

- Our first step is to pick a sentinel.
- This requires that we have some idea of what our data will be like.
- Supposing we're averaging quiz scores for COMP 201.
  - Nobody should get a negative quiz score!
- Let's set the sentinel to any negative number.

PYTHON PROGRAM

```
# average3.py

def main():
      print("This program calculates the average of some numbers")

      sum = 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("The average of", count, "numbers is", sum/count)

main()
```

PROGRAM OUTPUT

```
J:\> python average3.py
This program calculates the average of some numbers
Enter a number (negative to quit) >> 3
Enter a number (negative to quit) >> 5
Enter a number (negative to quit) >> 10
Enter a number (negative to quit) >> 7
Enter a number (negative to quit) >> -1
The average of 4 numbers is 6.25
J:\>
```

- The sentinel loop is another useful pattern, especially when dealing with data processing

IMPROVING OUR PROGRAM

- Our program is much improved, but some courses do give negative scores for quizzes!
- We need a sentinel value that is not a number at all for this.
- Recall: user input (the returned value from the `input()` function) is always a string.
  - We put the `eval()` around it to turn it into a number.
- So we can use a string as the sentinel value, and `eval()` the input only if it doesn't match the sentinel value.
  - Simple solution: use the empty string (user just hits `enter` without typing anything) as the sentinel.

OUR IMPROVED ALGORITHM

1. Initialize sum to 0
2. Initialize count to 0
3. Input data item as a string, xString
4. While xString is not the empty string
   i) Convert xString to a number, x
   ii) Add x to sum
   iii) Add 1 to count
   iv) Input next data item as a string, xString
5. Output sum/count

OUR PROGRAM

```
# average4.py

def main():
      print("This program calculates the average of some numbers")

      sum = 0
      count = 0
      xString = input("Enter a number (<Enter> to quit) >> ")
      while xString != "":
          x = eval(xString)
          sum = sum + x
          count = count + 1
          xString = input("Enter a number (<Enter> to quit) >> ")
      print("The average of", count, "numbers is", sum/count)

main()
```

The empty string

PROGRAM OUTPUT

- We can now average arbitrary numbers.

```
J:\> python average4.py
This program calculates the average of some numbers
Enter a number (<Enter> to quit) >> 5
Enter a number (<Enter> to quit) >> 23
Enter a number (<Enter> to quit) >> 0
Enter a number (<Enter> to quit) >> 28
Enter a number (<Enter> to quit) >> -5
Enter a number (<Enter> to quit) >> 99
Enter a number (<Enter> to quit) >> 103.5
Enter a number (<Enter> to quit) >>
The average of 7 numbers is 36.2142857143
J\:>
```

Remember these techniques and apply them to your own programs!

COMMON LOOP ERRORS

- What is wrong with the program fragments below?

```
while i < 10:
    print(i)
```

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

```
product = 0
for i in range(5):
    x = eval(input("Number?"))
    product = product * x
print("The product is", product)
```

EXERCISES

- What will be the output of the program fragment below?

```
x = 12
while x > 7:
    print(x, end=" ")
    x = x-1
```

```
y = 0
x = 1
total = 0
while x <= 5:
      y = x*x
      print(y)
      total = total+y
      x = x+1
print("total is", total)
```

EXERCISE

- Modify your odd/even program from last chapter to keep on asking the user for values, until the user enters a blank string.
- Your program must include a function called `isEven()`, which will return `True` if the parameter is a even number, and `False` otherwise.

```
Value? 9
Odd
Value? 12
Even
Value? 24
Even
Value?
Bye!
```

File Loops

- Loops can be used to read from files as well.
- Imagine if we were trying to average the Quiz 1 scores of all the COMP 201 students...
  - ... we type in 99 of them, and then make a mistake on the 100th one.
- Large sets of data are usually read in from files rather than input by hand.

File Loops -- The Python Way

- Back in the Strings chapter (Chapter 4), we learned of one way to read in a file, using a `for` loop.
- The following program reads in a file containing the scores of all the students in COMP 201, and averages them.
- Note the similarity between this and the manual user input version.

```
def main():
    fileName = input("What is the file name? ")
    infile = open(fileName, "r")
    sum = 0
    count = 0
    for line in infile:
        sum = sum + eval(line)
        count = count + 1
    print("The average of the numbers is", sum/count)
```

Reads data in from file

File Loops -- The Usual Way

- In Python, the for loop includes the mechanism for reading in files. (That's why our program worked.)
- Most programming languages do not have this capability.
- Instead, we need to use a sentinel loop to check for the end of file.
  - At the end of every file, there is a special character -- which in Python is equivalent to the empty string ("")

File Loops -- The Normal Way

- Rewrite the average score program using a sentinel loop to read in the file values:
  - Hint: `file.readline()` reads in one line.

```
def main():
    fileName = input("What is the file name? ")
    infile = open(fileName, "r")
    sum = 0
    count = 0
```

Nested Loops

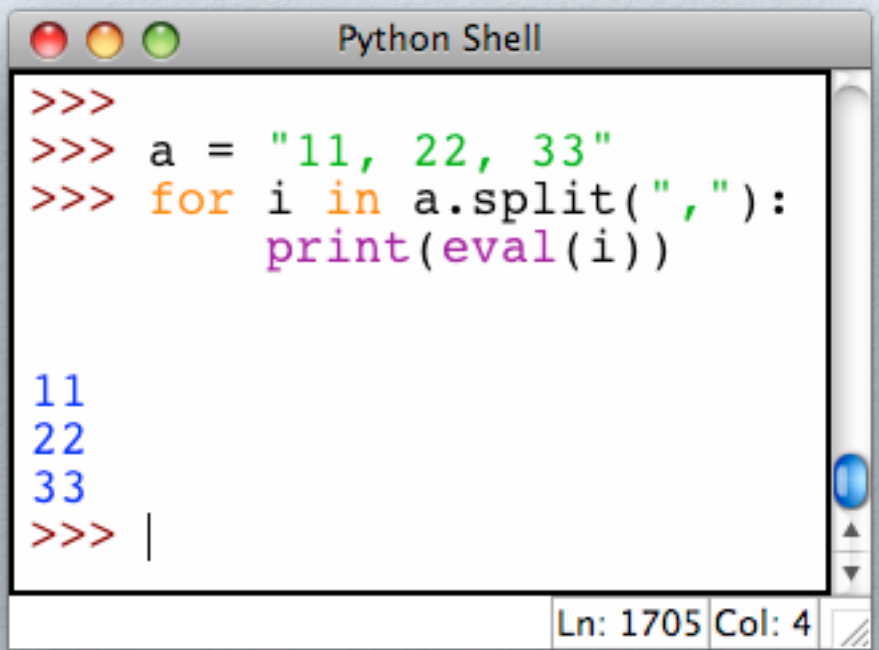
- In the last chapter, we saw an `if-else` inside an `if-else`, and an `if-else` inside a loop.
- Similarly, loops can also be nested inside loops.

Example: Averaging Scores

- Suppose that we need to average all the quiz scores for COMP 201.
- Also, suppose that the file containing the data is not as tidy as we would like.
  - Some lines have only one number.
  - Some lines have multiple scores, separated by commas.
- How would we modify our program to accommodate this?

AVERAGING SCORES

```
89
90, 33, 46, 27
91, 44
20, 57, 89
12
46
```



- Python can only read in one line at a time.
  - This will be handled either by a `for` loop or a `while` loop.
- We have the `split()` function for strings, which will split a string into substrings.
  - The `split()` function returns a list, which we can use a `for` loop to iterate over.

AVERAGING SCORES

- We have two loops: one to read in the lines of a file, another one to process the data items in the lines.
- One of these loops will have to go inside the other.
  - Which one is the outer loop?
  - Which one will be the inner loop?

ALGORITHM

1. Open file
2. Initialize count, sum
3. For each line in the file
    i) For each item in the line
        i) Update count, sum
4. Calculate and print out average

- Now try and write the program yourself.

Nested Loop Design

- Nested loops can get very intricate when they are put together, but taken separately, the individual loops are usually quite simple.
- The design of nested loops almost always follows the same steps:
  - Decide where the loops are.
  - Design the outer loop without worrying about the inner loop.
  - Design the inner loop, ignoring the outer loop.
  - Put the pieces together, preserving the nesting.

EXERCISE

- What is the output of the following program?

```
for i in range(2):
   print("Outer", i)
   for j in range(3):
      print("Inner", i, j)
   k = 2
   while k > 0:
      print("Inner", i, k)
      k = k - 1
```

EXERCISE

- Write a program that will ask the user for a number between 1 and 9 (inclusive), and then print the following pattern.
- Your program should have a function, `triangle()`, that will do the actual printing. Only the user input should be in `main()`.

```
1
12
123
1234
12345
123456
```

input = 6

```
1
12
123
1234
```

input = 4

EXERCISE

- Modify your program to create the following output.
- Hint: You need one or two (possible with one) inner loops inside the outer loop.

```
     1
    12
   123
  1234
 12345
123456
```

input = 6

```
   1
  12
 123
1234
```

input = 4

EXERCISE

- Further modify your program to output this pattern.

```
   1
  121
 12321
1234321
 12321
  121
   1
```

input = 4

input = 6

```
     1
    121
   12321
  1234321
 123454321
12345654321
 123454321
  1234321
   12321
    121
     1
```

BOOLEANS

- We have worked a little with Boolean expressions already:
  - `x > 0`, `xString != ""`, etc
- The comparison is between two values of the same type (numbers to numbers, strings to strings)

Comparing > 2 values

- Sometimes, we want to compare more than two values.
- Suppose we have two points, p1 and p2.
  - We want to know if they are located in the same place -- if their (x,y) coordinates are identical.
  - Need `p1.getX()` == `p2.getX()`, and `p1.getY()` == `p2.getY()`.
- We can use the **Boolean operators** to combine two Boolean expressions.

BOOLEAN OPERATORS

- Python provides three Boolean operators: `and`, `or` and `not`.
- `and` and `or` combine two Boolean expressions to produce a Boolean result.

```
<expr> and <expr>
<expr> or <expr>
```

- The definitions of `and` and `or` can be expressed by a **truth table**.

TRUTH TABLE FOR and AND or

- P, Q are smaller Boolean expressions.
- Two possible values for each expression -- four possible combinations in all.

P	Q	P and Q	P or Q
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

TRUTH TABLE FOR not

- The Boolean operator not takes one Boolean expression, and computes the opposite of the expression.

P	not P
T	F
F	T

Operator Precedence

- As we start writing more complex Boolean expressions, we run into the problem of **operator precedence**.
  - Arithmetic operators have the same problem too -- x and ÷ take place before + and -
- How to evaluate something like this:

```
a or not b and c
```

- Is it this:

```
(a or (not b)) and c
```

- Or is it this:

```
(a or (not (b and c))
```

Operator Precedence

- The order of operation is that not goes first, followed by and, followed by or.
- So our expression is equivalent to this:

```
(a or ((not b) and c))
```

- However, to avoid mistakes (because you probably won't remember the order), it is suggested to always parenthesize complex Boolean expressions.

Example: Two Points

- Given the Boolean operators we learned, we can now write the Boolean expression that will check if p1 and p2 are in the same location:

```
p1.getX() == p2.getX() and p1.getY() == p2.getY()
```

EXAMPLE: RACQUETBALL

- In racquetball, the game continues until one of the players has reached 15 points.
  - In other words, when `scoreA == 15 or scoreB == 15` is `true`
- Or, as a loop:

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

RACQUETBALL

- There's a "mercy rule" in racquetball that states that when one of the players hasn't scored a single point, the game will also end when the other player reaches 7 points.

```
scoreA == 15 or scoreB == 15 or (scoreA == 7 and
scoreB == 0) or (scoreB == 7 and scoreA == 0)
```

- As a loop, we have:

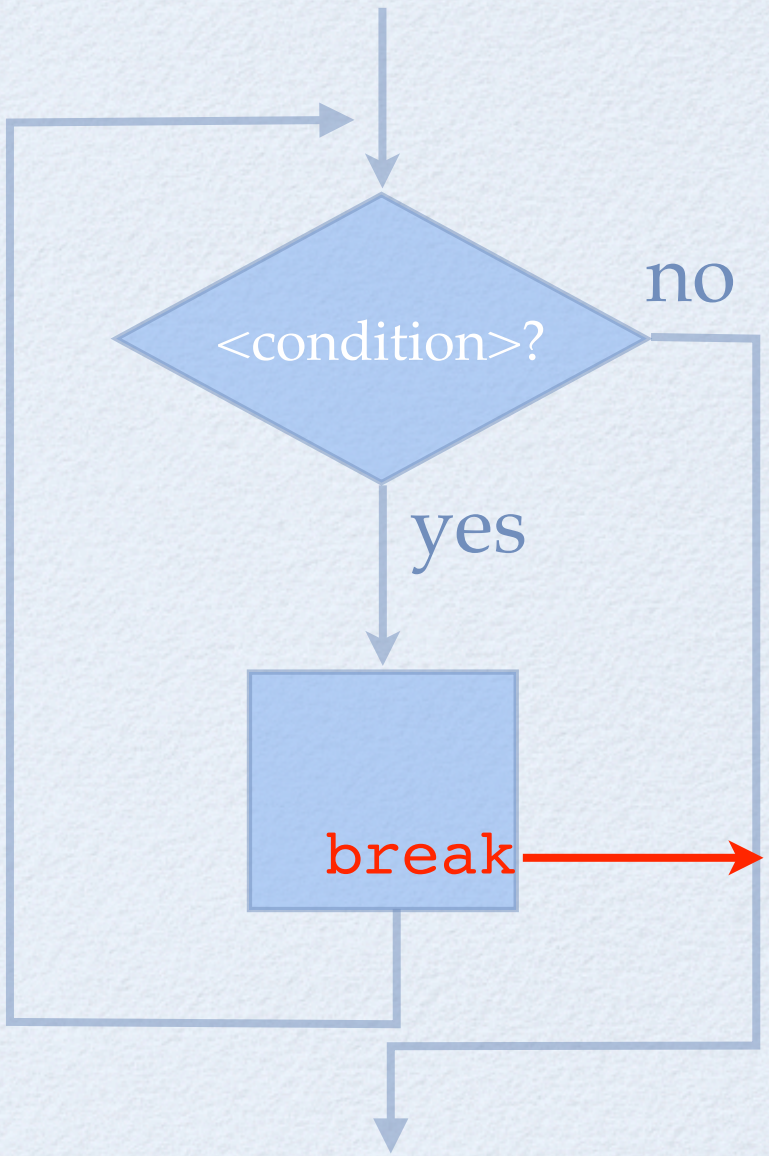
```
while not(scoreA == 15 or scoreB == 15 or (scoreA
== 7 and scoreB == 0) or (scoreB == 7 and scoreA
== 0)):
    # continue playing
```

Volleyball

- Volleyball games end when one team reaches at least 25 points and is at least 2 points ahead of the other team.
- Write the Boolean condition that will return `true` when a volleyball game is over.

BREAK STATEMENT

- The last of the control structures that we will learn about is the break statement.
- The break statement terminates the loop that encloses it.



BREAK EXAMPLE

- Suppose we need the user to enter a positive number for a program (e.g. if we need the number of students in a class).
- We can make use of an infinite loop plus a break statement to do error checking:

```
while True:
    number = input("How many students?")
    if number >= 0:
        break
    print("Number of students must be positive!")
```

You now know...

- What definite and indefinite loops are, and how they are realized in Python with the for and while statements.
- What interactive loops and sentinel loops are, and how to implement them using a Python while statement.
- What end-of-file loops do, and how to implement such loops in Python.
- How to design and implement solutions to problems involving loop patterns including nested loop structures.
- The basic ideas of Boolean algebra and how to analyze and write Boolean expressions involving Boolean operators.