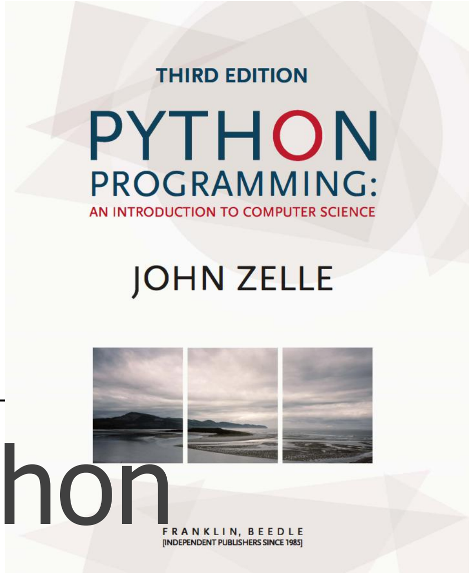

THIRD EDITION
PYTHON
PROGRAMMING:
AN INTRODUCTION TO COMPUTER SCIENCE
JOHN ZELLE
FRANKLIN, BEEDLE
[INDEPENDENT PUBLISHERS SINCE 1985]

An Introduction to Python Programming

Chapter 8: Loop Structures and Booleans

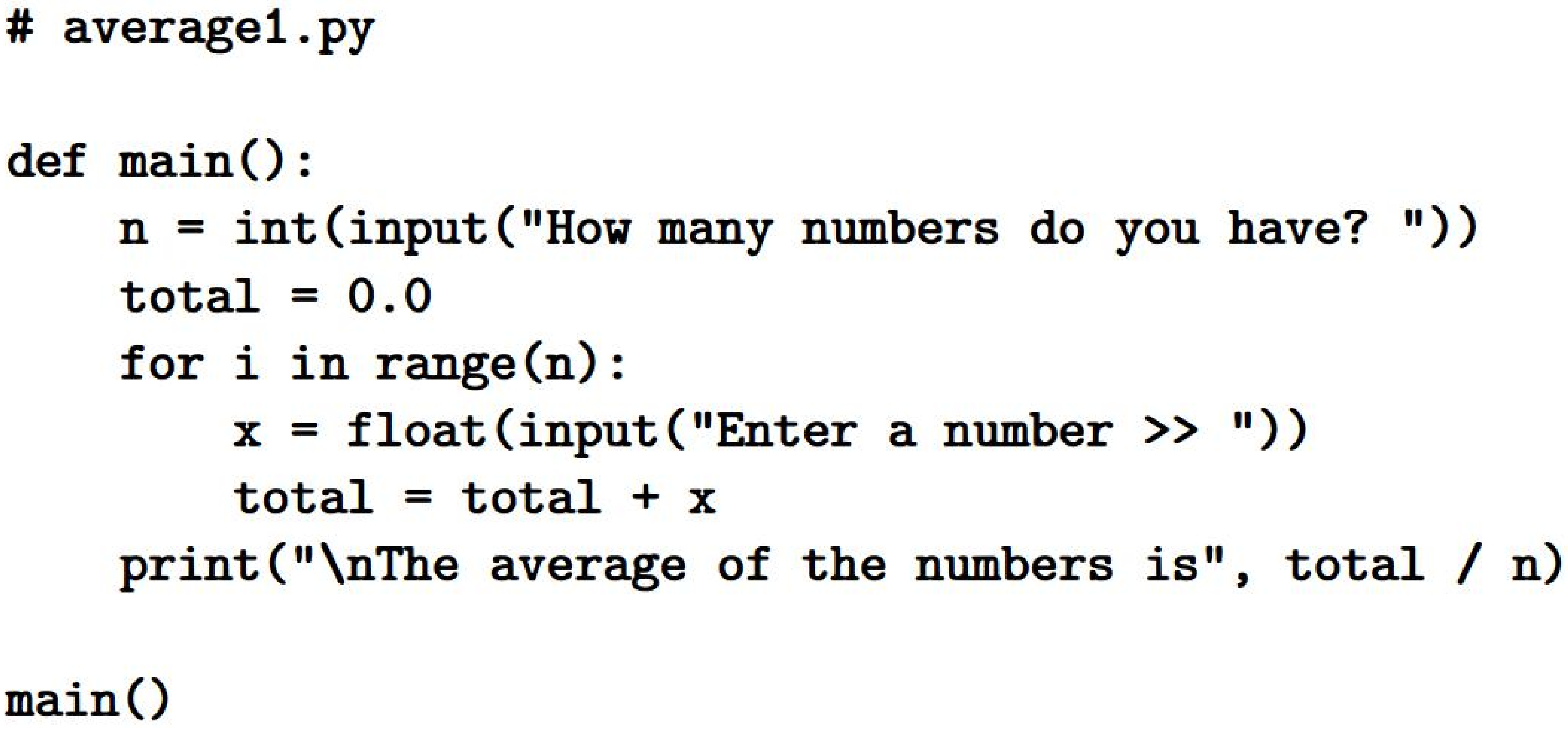
Objectives

- To understand the concepts of definite and indefinite loops as they are realized in the Python `for` and `while` statements.
- 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.

For Loops: A Quick 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 only need know the running **sum** and **how many** numbers have been added.

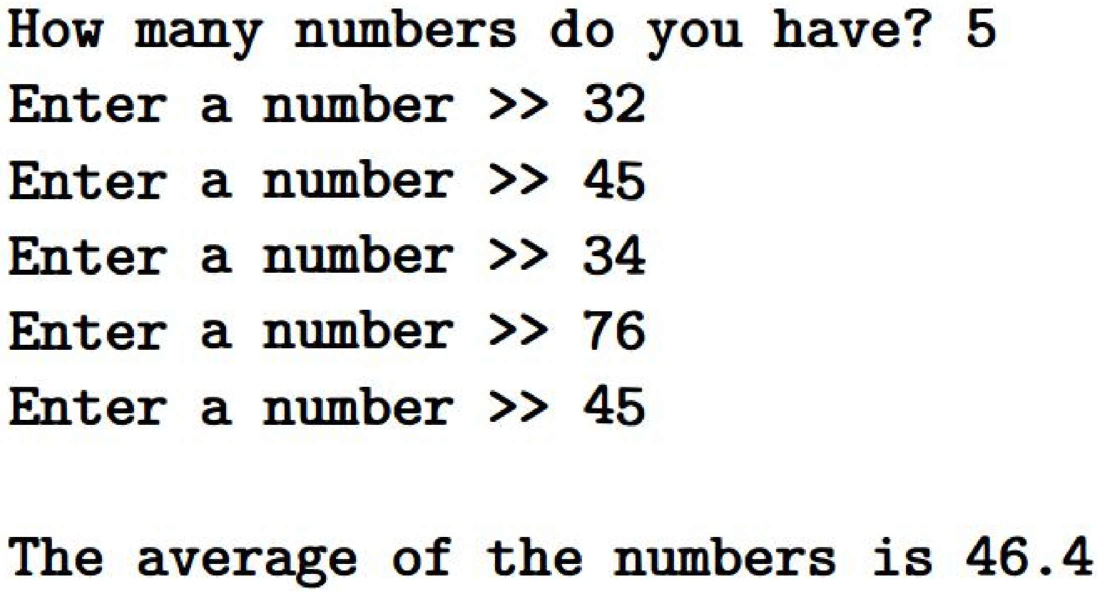
For Loops: A Quick Review



```
# average1.py

def main():
    n = int(input("How many numbers do you have? "))
    total = 0.0
    for i in range(n):
        x = float(input("Enter a number >> "))
        total = total + x
    print("\nThe average of the numbers is", total / n)

main()
```

- Note that sum is initialized to 0.0 so that sum/n returns a **float**!

For Loops: A Quick Review



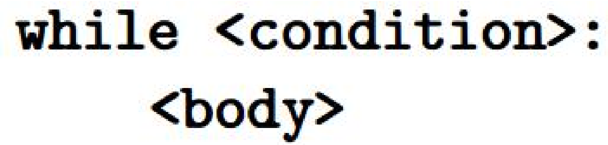
```
How many numbers do you have? 5
Enter a number >> 32
Enter a number >> 45
Enter a number >> 34
Enter a number >> 76
Enter a number >> 45

The average of the numbers is 46.4
```

Indefinite Loops

- That average program got the job done, but you need to know **ahead of time how many** numbers you'll be dealing with.
- Suppose **counting a page of numbers** is a burden.
- The `for` loop is a **definite loop**.

Indefinite Loops

- We need another tool!
- The ***indefinite* or *conditional* loop** keeps iterating until certain conditions are met.



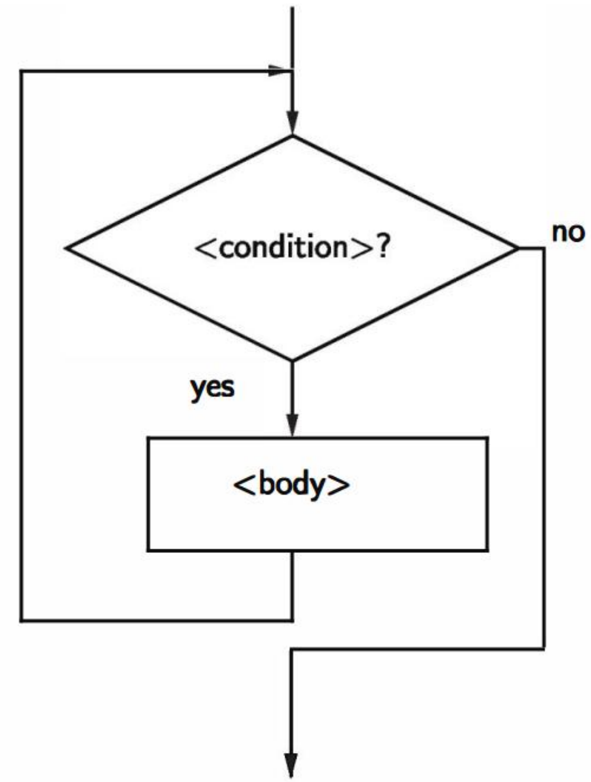
```
while <condition>:
    <body>
```

□`condition` is a Boolean expression.

□ The **body** is a sequence of one or more statements.

□Semantically, the body of the loop **executes repeatedly** as long as **the condition remains true**. When the condition is false, the loop terminates.

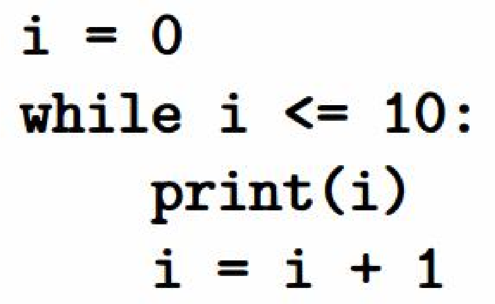
Indefinite Loops



- ***Pre-test*** **loop:** the condition is tested **at the top of the loop**. If the condition is initially false, the loop body will not execute at all.

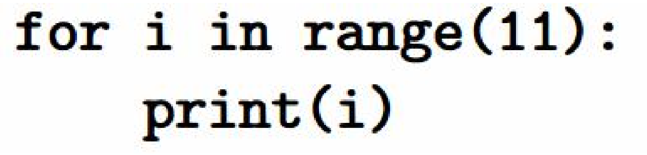
Indefinite Loop

- Here’s an example of a **`while`** loop that **counts from 0 to 10**:



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

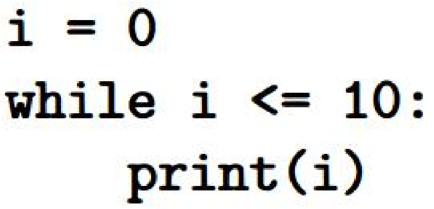
- The same output as **`for`** loop:



```
for i in range(11):
    print(i)
```

Indefinite Loop

- The loop variable **`i`** should be **initialized with 0** before the loop and **increased** at the bottom of the body.
- In the `for` loop this is handled automatically.
- The following **`while`** statement is a common source of program **errors**.

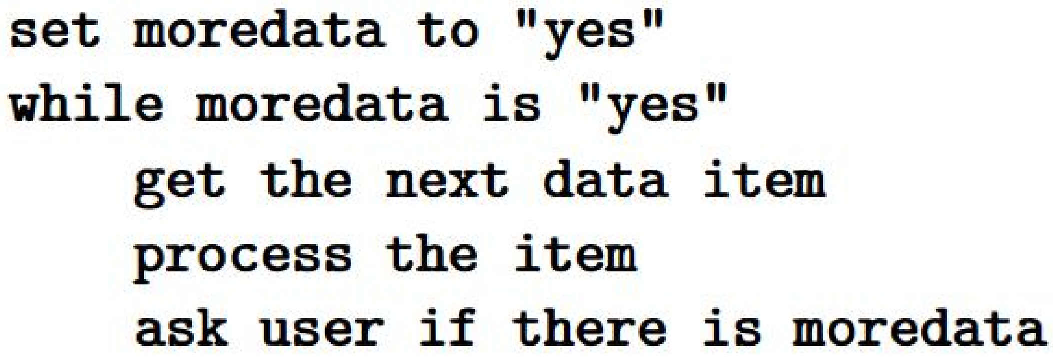


```
i = 0
while i <= 10:
    print(i)
```

- This version of the program **does nothing useful**.

Common Loop Patterns: Interactive Loops

- ***Interactive loops*** allow a user to **repeat certain portions of** a program on demand.
- Let's go to the **number-averaging** program.

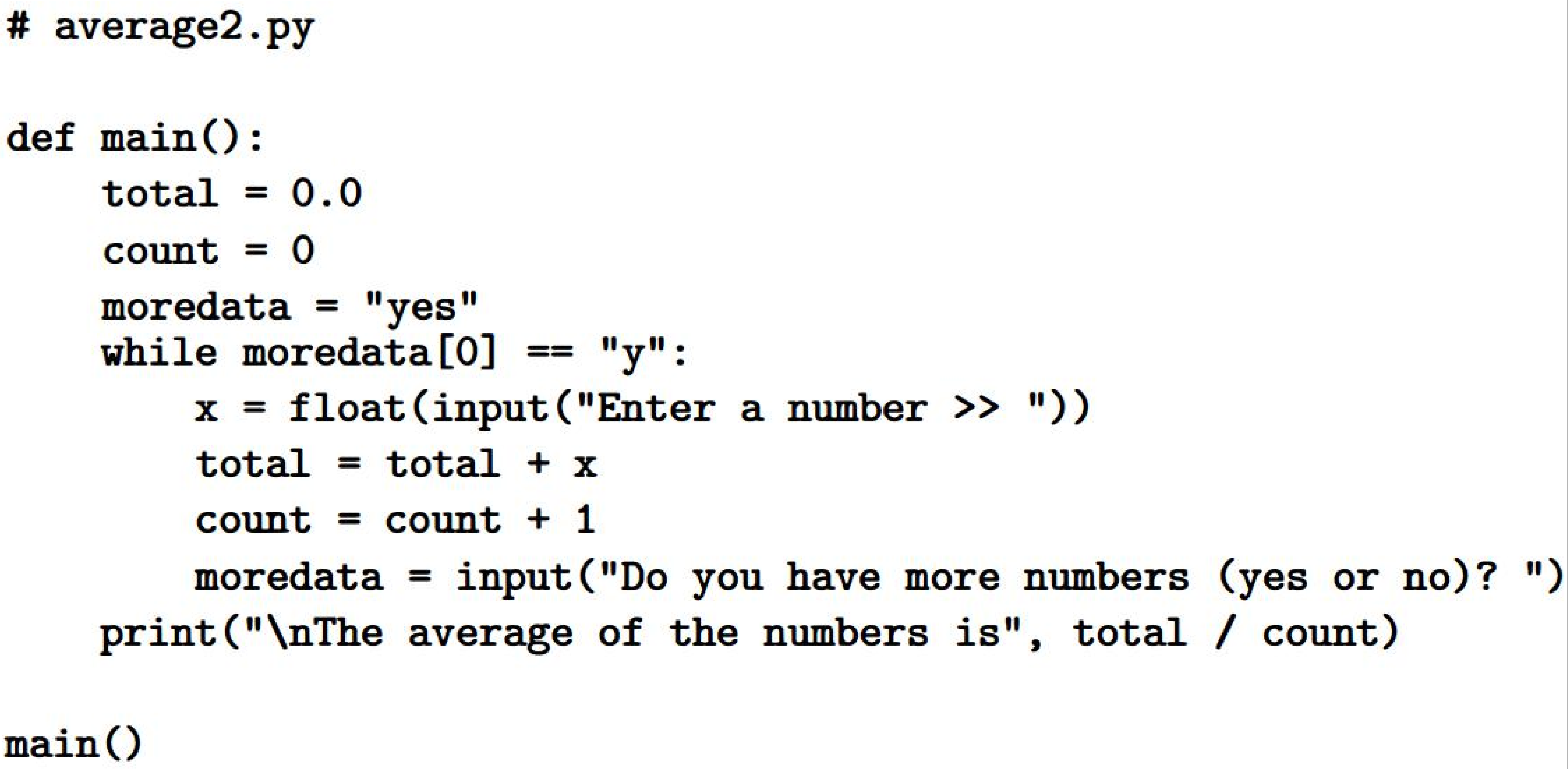


```
set moredata to "yes"
while moredata is "yes"
    get the next data item
    process the item
    ask user if there is moredata
```

Common Loop Patterns:

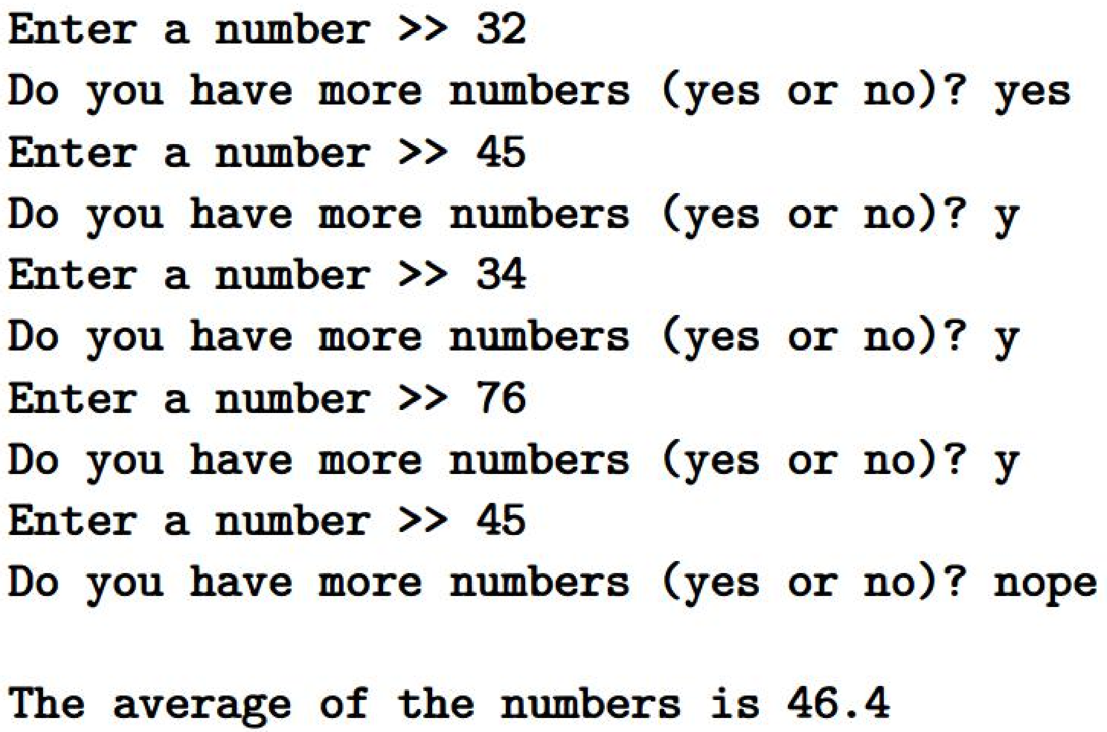
Interactive Loops

- Combining the interactive loop pattern with accumulators for **sum** and **count**:



```
# average2.py

def main():
    total = 0.0
    count = 0
    moredata = "yes"
    while moredata[0] == "y":
        x = float(input("Enter a number >> "))
        total = total + x
        count = count + 1
        moredata = input("Do you have more numbers (yes or no)? ")
    print("\nThe average of the numbers is", total / count)

main()
```

Common Loop Patterns: Interactive Loops



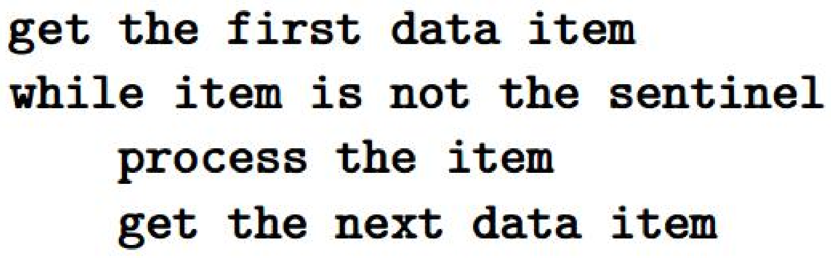
```
Enter a number >> 32
Do you have more numbers (yes or no)? yes
Enter a number >> 45
Do you have more numbers (yes or no)? y
Enter a number >> 34
Do you have more numbers (yes or no)? y
Enter a number >> 76
Do you have more numbers (yes or no)? y
Enter a number >> 45
Do you have more numbers (yes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

The average of the numbers is 46.4
```

- Is this better? The user will be annoyed by the **constant prodding for more data.**

Common Loop Patterns: 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.

Common Loop Patterns: Sentinel Loops



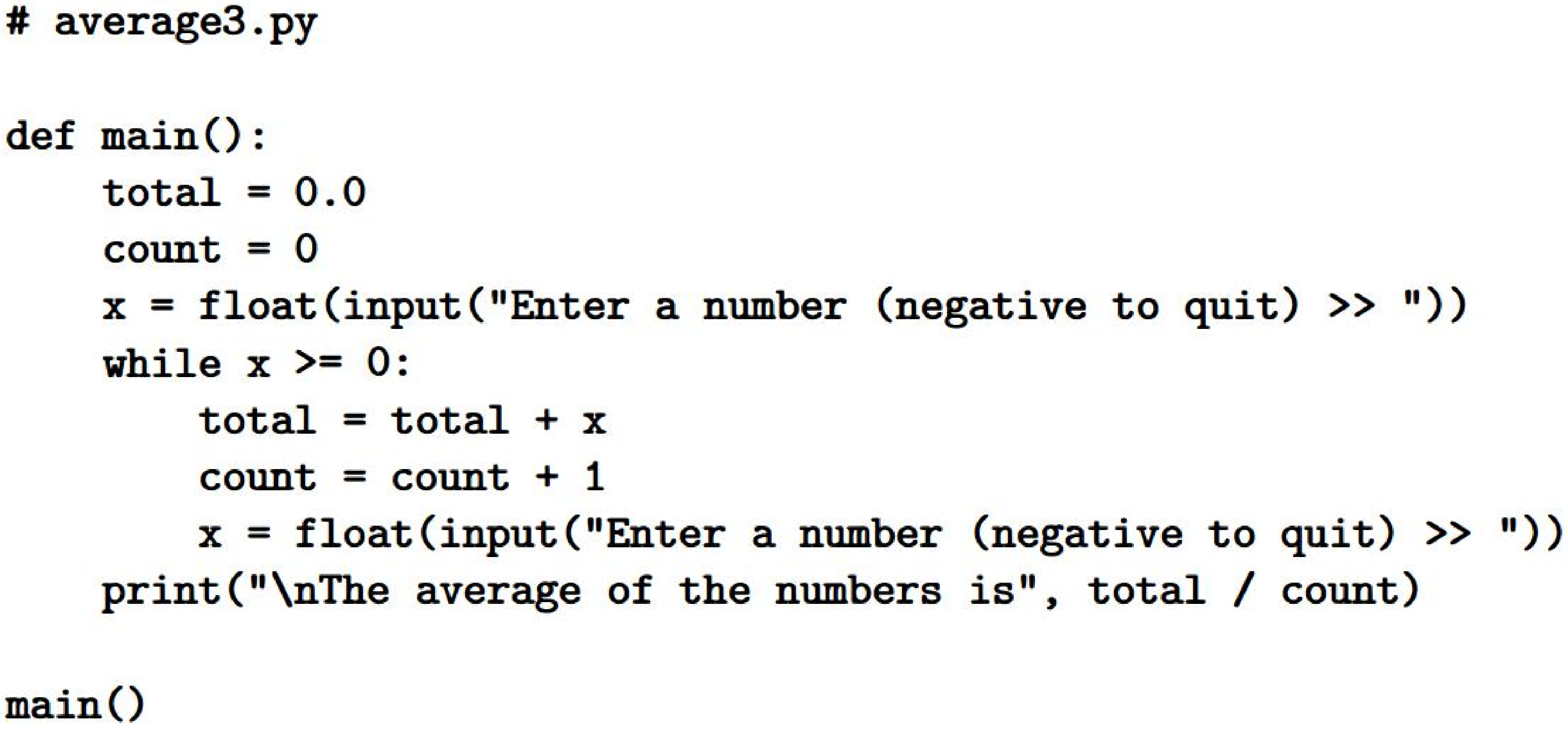
```
get the first data item
while item is not the sentinel
    process the item
    get the next data item
```

- The first item is **retrieved** before the loop starts, 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.

Common Loop Patterns: Sentinel Loops

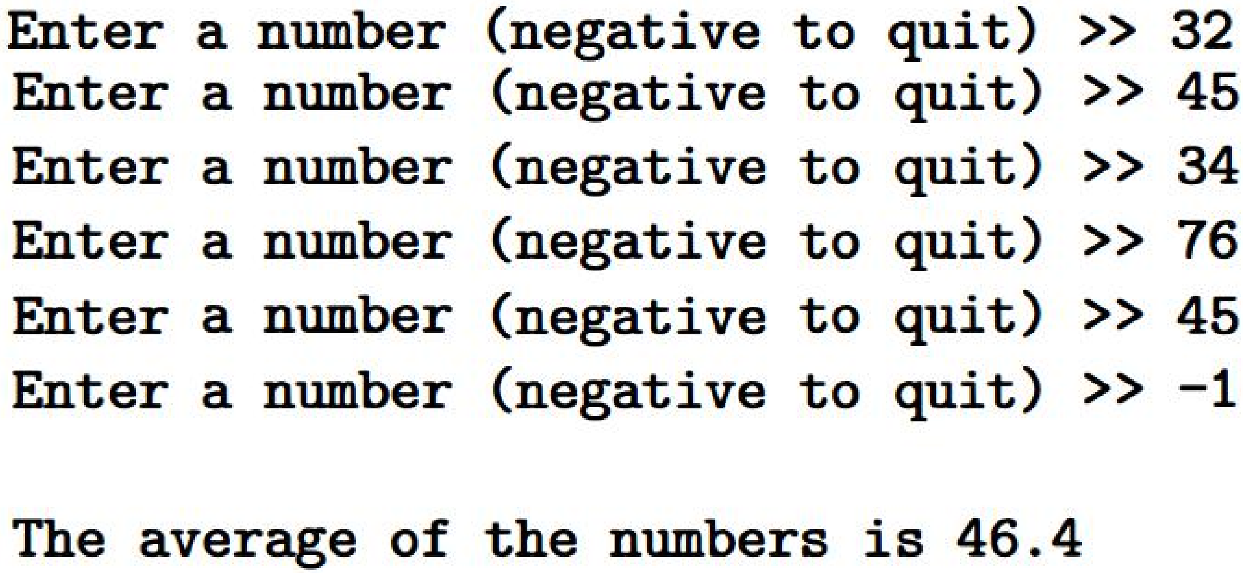
- In our averaging example, We can assume a **negative** number will be the **sentinel**.
- The user can **enter a negative number** to signal the end of the data.

Common Loop Patterns: Sentinel Loops



```
# average3.py

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

main()
```

Common Loop Patterns: Sentinel Loops

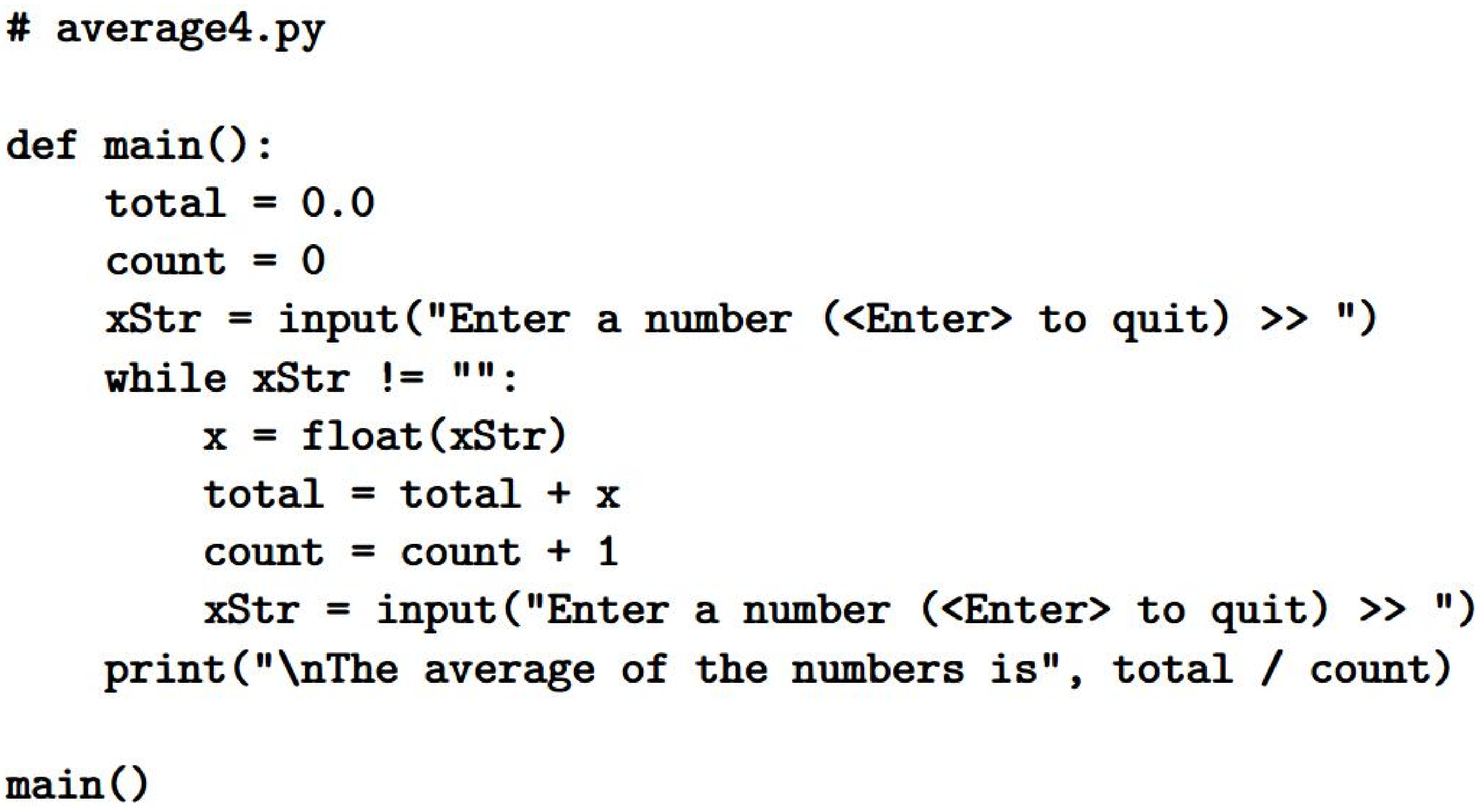


```
Enter a number (negative to quit) >> 32
Enter a number (negative to quit) >> 45
Enter a number (negative to quit) >> 34
Enter a number (negative to quit) >> 76
Enter a number (negative to quit) >> 45
Enter a number (negative to quit) >> -1

The average of the numbers is 46.4
```

Common Loop Patterns: Sentinel Loops

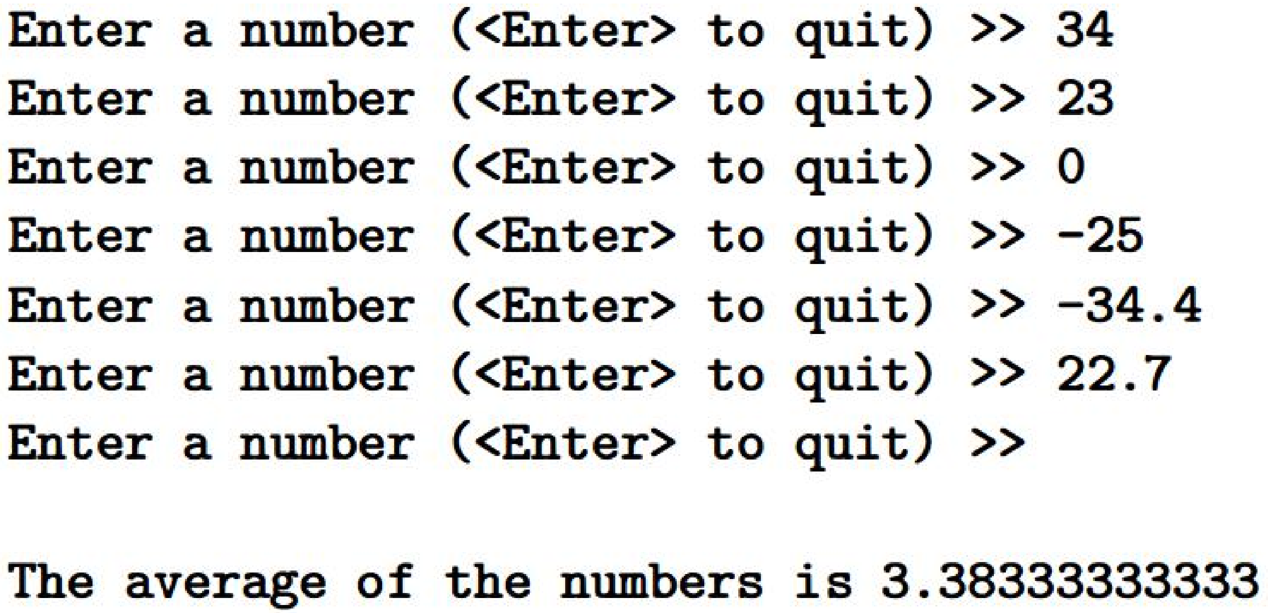
- There's still a **shortcoming** – How to generalize the program a bit?
- we **can't** average a set of **positive *and negative* numbers**.
- We could use the ***empty string* (“”)**!

Common Loop Patterns: Sentinel Loops



```
# average4.py

def main():
    total = 0.0
    count = 0
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = float(xStr)
        total = total + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit) >> ")
    print("\nThe average of the numbers is", total / count)

main()
```

Common Loop Patterns: Sentinel Loops

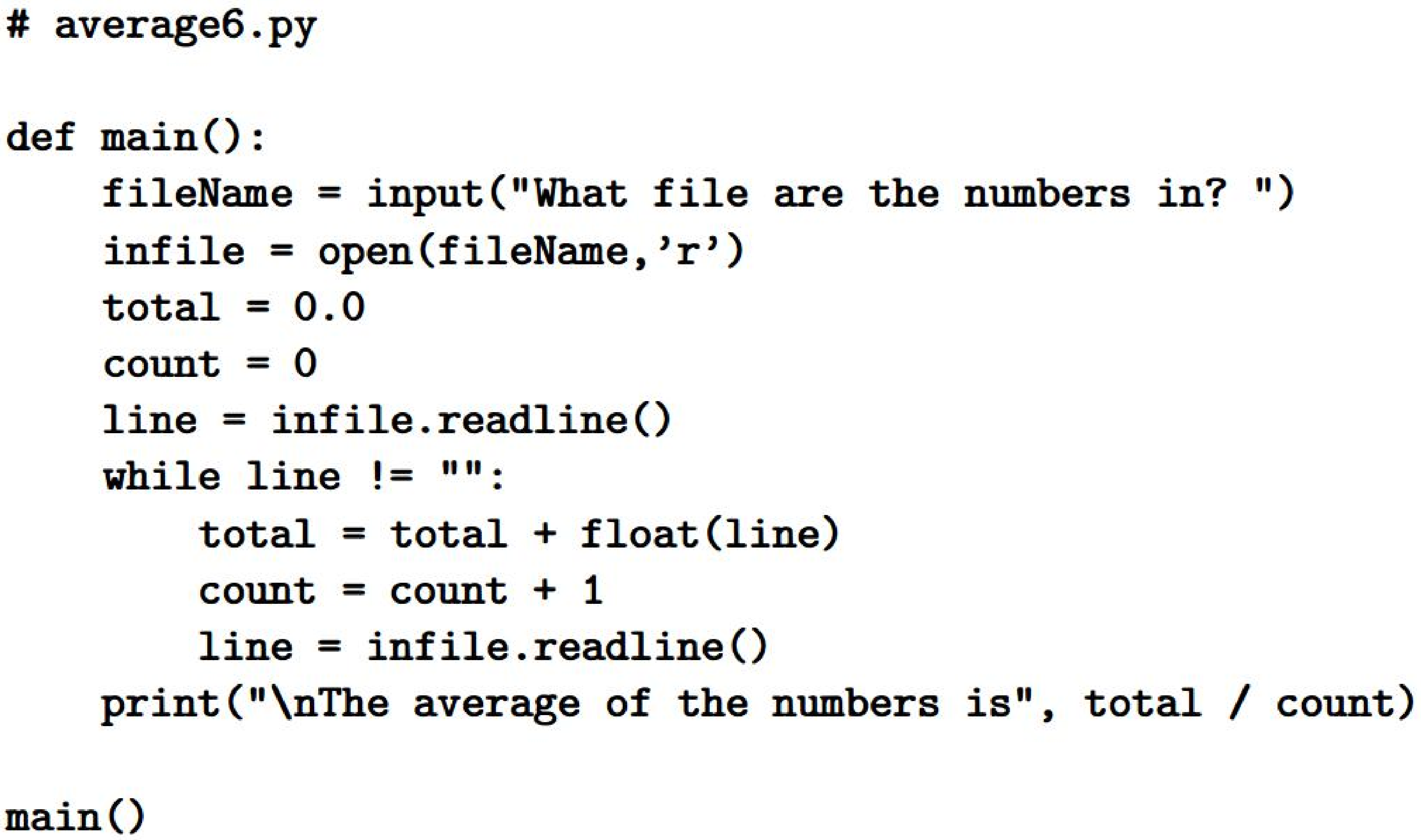


```
Enter a number (<Enter> to quit) >> 34
Enter a number (<Enter> to quit) >> 23
Enter a number (<Enter> to quit) >> 0
Enter a number (<Enter> to quit) >> -25
Enter a number (<Enter> to quit) >> -34.4
Enter a number (<Enter> to quit) >> 22.7
Enter a number (<Enter> to quit) >>

The average of the numbers is 3.38333333333
```

Common Loop Patterns: File Loops

- 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**.
- Suppose we type the numbers into the file **one per line**.

Common Loop Patterns:
File Loops



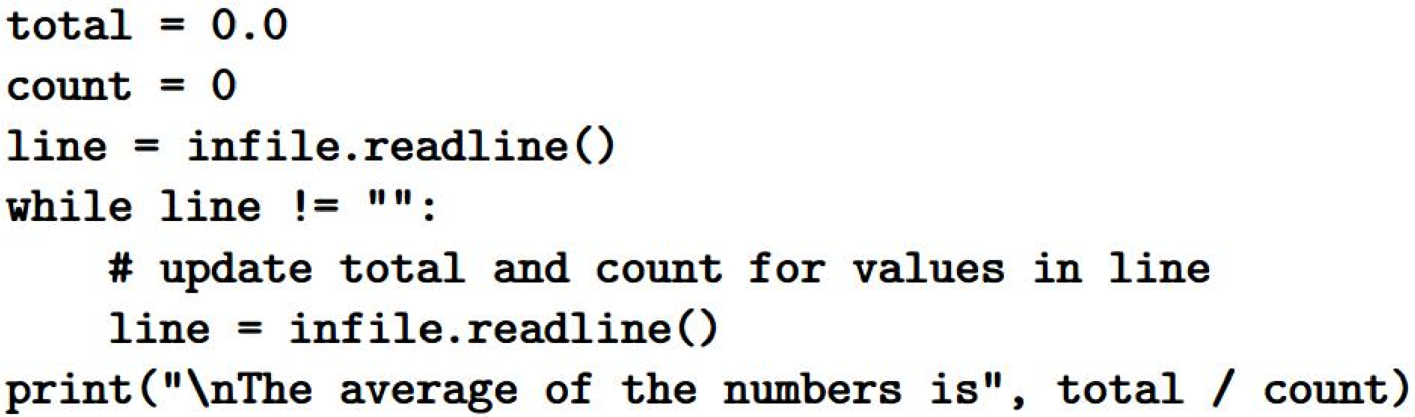
```
# average6.py

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

main()
```

Common Loop Patterns:
File Loops

- We use a **sentinel** for looping through a file.
- We could use `readline` in a loop to get the **next line** of the file.
- At the end of the file, `readline` returns an e**mpty string, “”.**
- Does this code **correctly handle** the case where there’s **a blank line** in the file?
  - □Yes. An empty line actually ends with the newline character, **“\n” != “”**.

Common Loop Patterns: Nested Loops

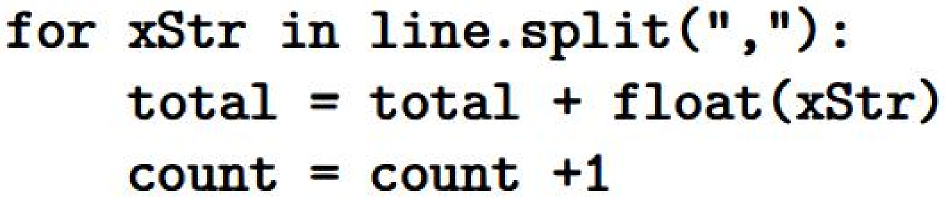
- We can nest loops.
- Suppose we allow **any number of numbers on a line** in the file (separated by commas).
- At the top level, we will use a **file-processing loop** that computes a running sum and count.



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

Common Loop Patterns: Nested Loops

- In the next level, how to update the **`sum`** and **`count`** ?
  - we can **split** the string into substrings, each of which represents a number.
  - Then loop through the substrings, **convert** each to a number, and add it to **`sum`**.
  - Update **`count`**.

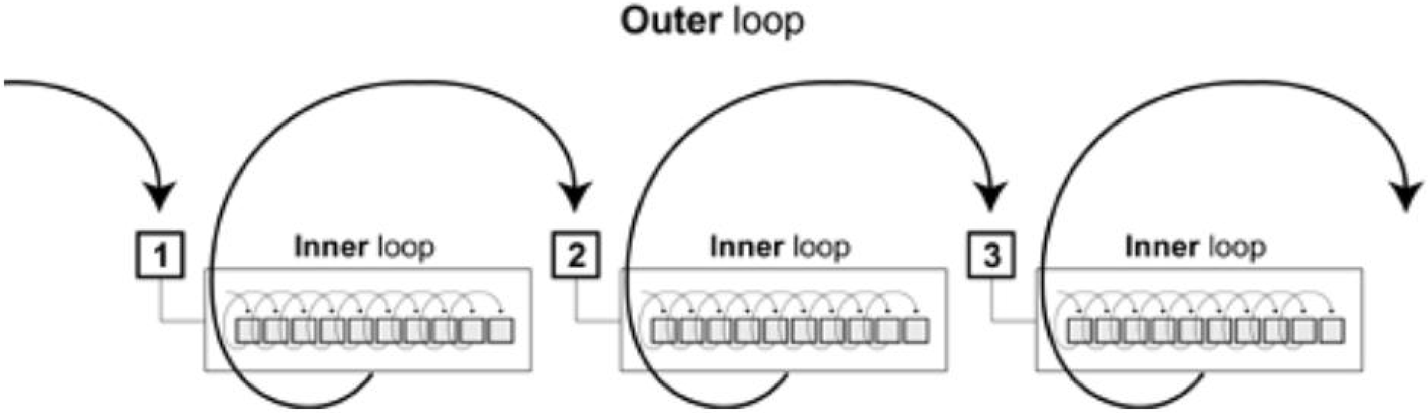


```
for xStr in line.split(","):
    total = total + float(xStr)
    count = count +1
```

Common Loop Patterns:

Nested Loops

- Then the next level loop is **indented inside** of the file processing loop.
- When the **inner loop** finishes, the **next line** of the file is read, and this process begins again.



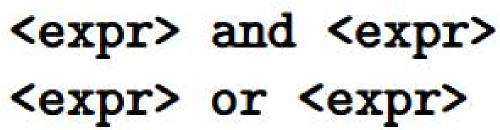
```
# average7.py

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,’r’)
    total = 0.0
    count = 0
    line = infile.readline()
    while line != "":
        # update total and count for values in line
        for xStr in line.split(","):
            total = total + float(xStr)
            count = count + 1
        line = infile.readline()
    print("\nThe average of the numbers is", total / count)

main()
```

Common Loop Patterns: 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: Boolean Operators

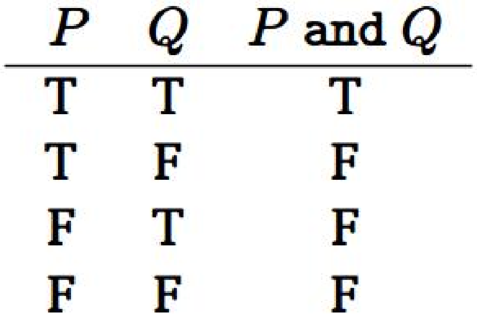
- Sometimes our simple expressions **do not** seem expressive enough.
- Let’s check out the three Boolean operators **`and`, `or`, and `not`.**



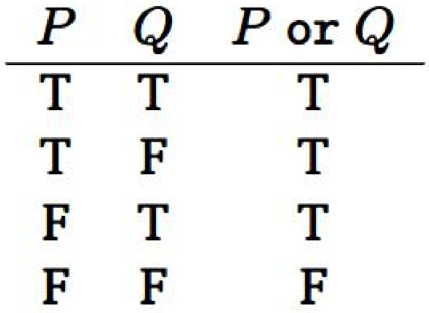
```
<expr> and <expr>
<expr> or <expr>
```

Computing with Booleans: Boolean Operators

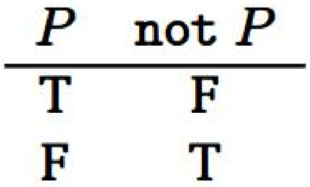
- We can represent the `and & or & not` in the ***truth tables***.



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

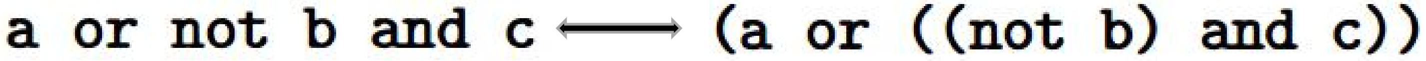


P	Q	P or Q
T	T	T
T	F	T
F	T	T
F	F	F



P	not P
T	F
F	T

Computing with Booleans: Boolean Operators

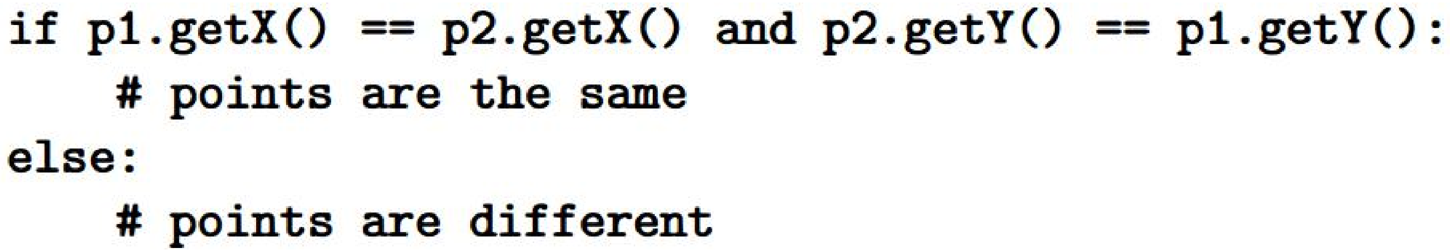


```
a or not b and c ⟷ (a or ((not b) and c))
```

- The order of precedence, from **high** to **low**, is **`not`, `and`, `or`.**
- We can use **parentheses** to prevent confusion.

Computing with Booleans: Boolean Operators

- Suppose you need to **determine whether two points are in the same position** – their *x* coordinates are equal and their *y* coordinates are equal.



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

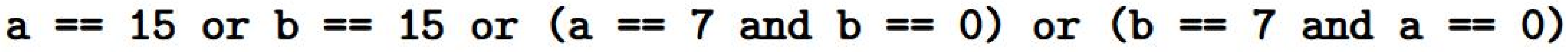
- This is much simpler and clearer than the nested ifs!

Computing with Booleans: Boolean Operators

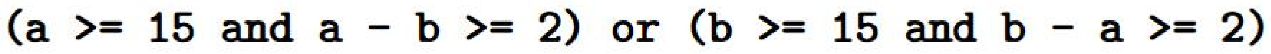
- Applications:
  - In a racquetball simulation,the game is over as soon as either player has scored 15 points. In shutout condition, if one player has scored 7 points and the other person hasn’t scored yet, the game is over.
  - Let’s look at volleyball scoring. To win, a volleyball team needs to win by at least two points.

Computing with Booleans: Boolean Operators

- You can take the following of the game-over condition as your loop condition!

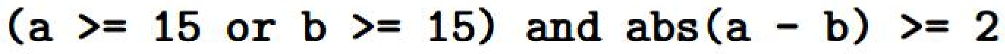


```
a == 15 or b == 15 or (a == 7 and b == 0) or (b == 7 and a == 0)
```



```
(a >= 15 and a - b >= 2) or (b >= 15 and b - a >= 2)
```

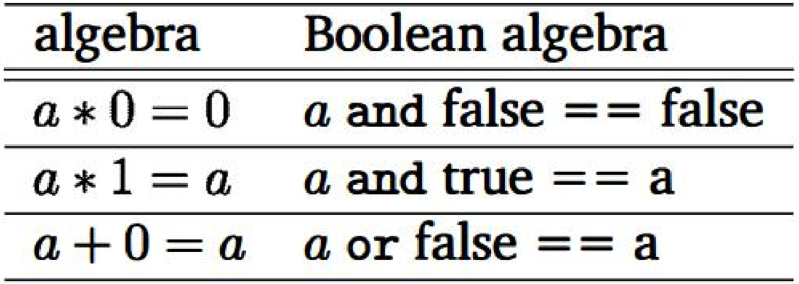
□**another way:**



```
(a >= 15 or b >= 15) and abs(a - b) >= 2
```

Computing with Booleans: Boolean Algebra

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

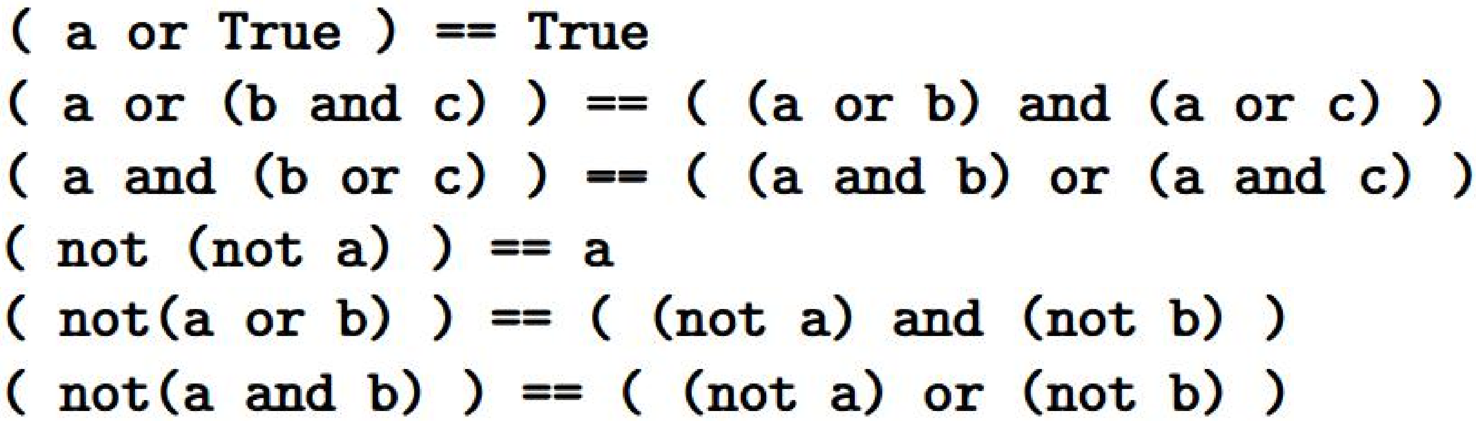


algebra	Boolean algebra
$a * 0 = 0$	a and false == false
$a * 1 = a$	a and true == a
$a + 0 = a$	a or false == a

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

Computing with Booleans: Boolean Algebra

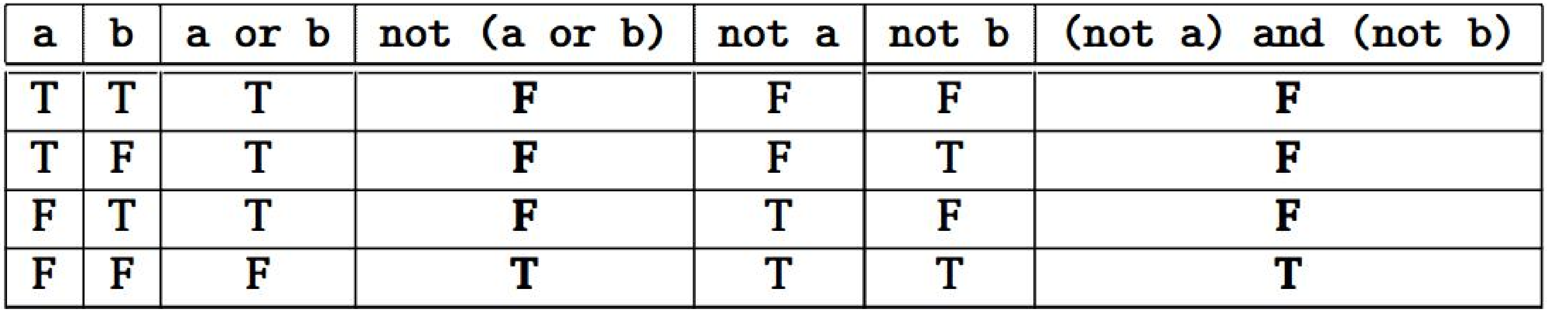
- Here are some properties of Boolean operations.



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

Computing with Booleans: Boolean Algebra

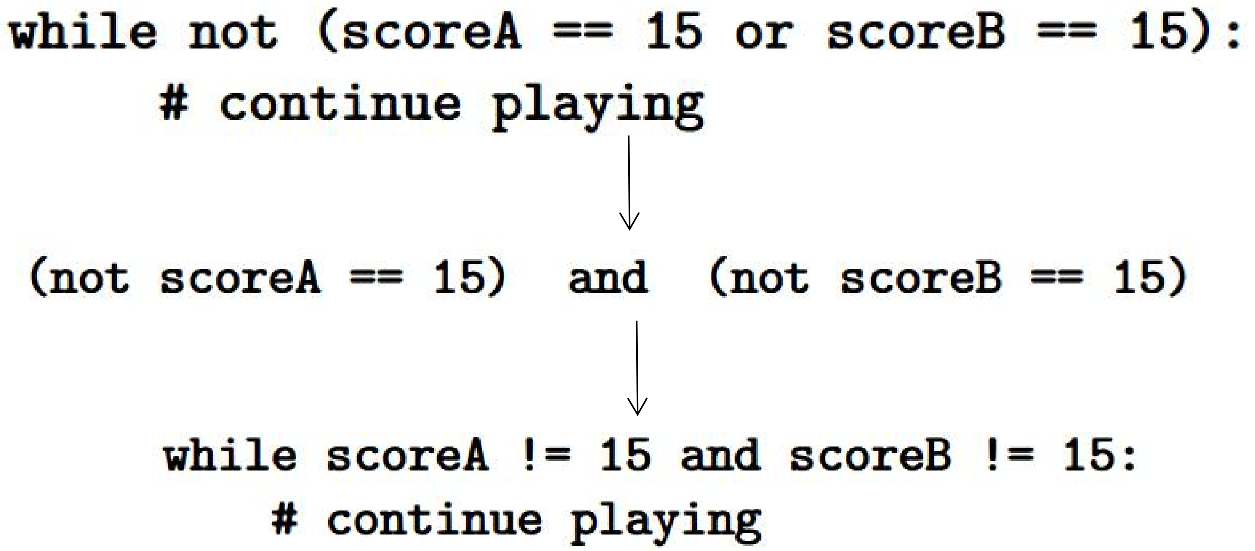
- listing all of the possibilites and computing the value of the expressions demonstrates DeMorgan's first law:



a	b	a or b	not (a or b)	not a	not b	(not a) and (not b)
T	T	T	**F**	F	F	**F**
T	F	T	**F**	F	T	**F**
F	T	T	**F**	T	F	**F**
F	F	F	**T**	T	T	**T**

Computing with Booleans: Boolean Algebra

- We can use Boolean algebra to simplify our Boolean expressions.



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

```
(not scoreA == 15)  and  (not scoreB == 15)
```

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

- Sometimes it’s **easier** to figure out when a loop should **stop**, rather than when the loop should continue.

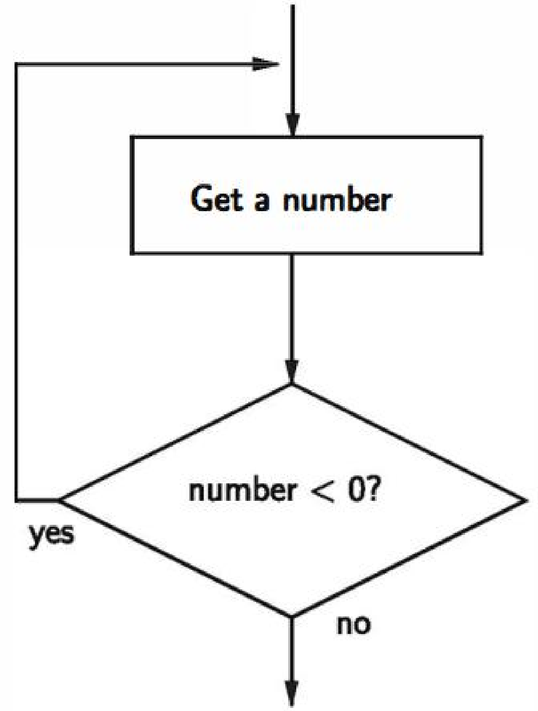
Other Common Structures:

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 and continues until a valid value has been entered.
- This process is ***input validation***.
- Well-engineered programs validate inputs whenever possible.

Other Common Structures: Post-Test Loop

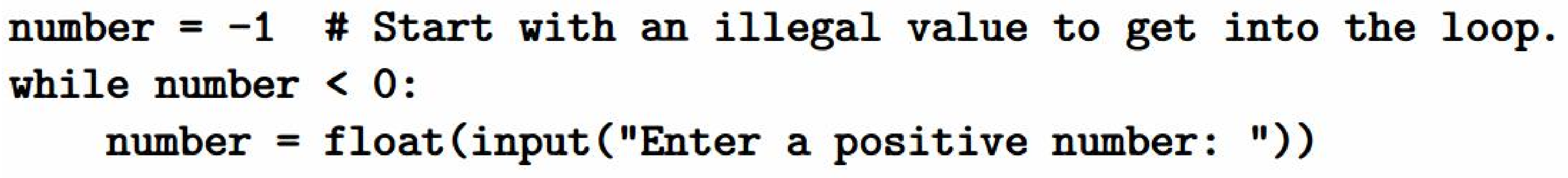
- The flowchart depicting this design in shown



Other Common Structures:

Post-Test Loop

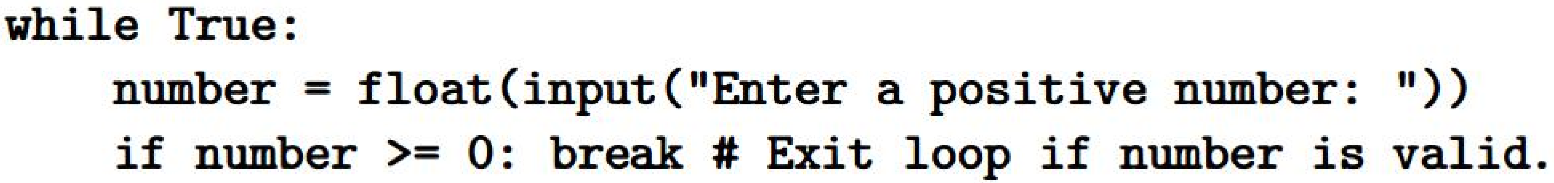
- When the condition test comes after the body of the loop it’s called a ***post-test loop***.
- A post-test loop always executes the body of the code **at least once**.
- In python,this algorithm can be implemented with a **while** by **"seeding" the loop condition** for the first iteration:



```
number = -1  # Start with an illegal value to get into the loop.
while number < 0:
    number = float(input("Enter a positive number: "))
```

Other Common Structures:
Post-Test Loop

- By using the Python `break` **statement**.Python immediately exit the enclosing loop.
- `Break` is sometimes used to exit what looks like an infinite loop.

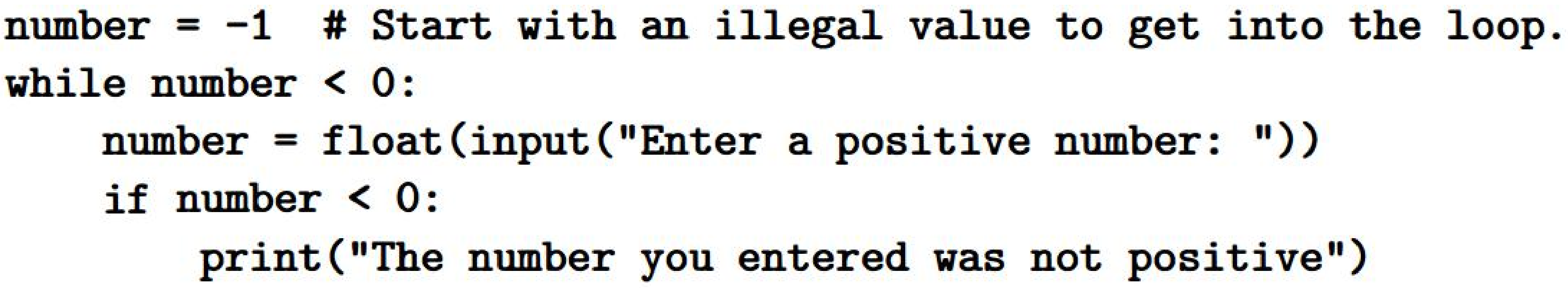


```
while True:
    number = float(input("Enter a positive number: "))
    if number >= 0: break # Exit loop if number is valid.
```

  - Since `True` *always* evaluates to true, it looks like an infinite loop!

Other Common Structures: Post-Test Loop

- It would be nice if the program issued a warning **explaining why** the input was invalid.

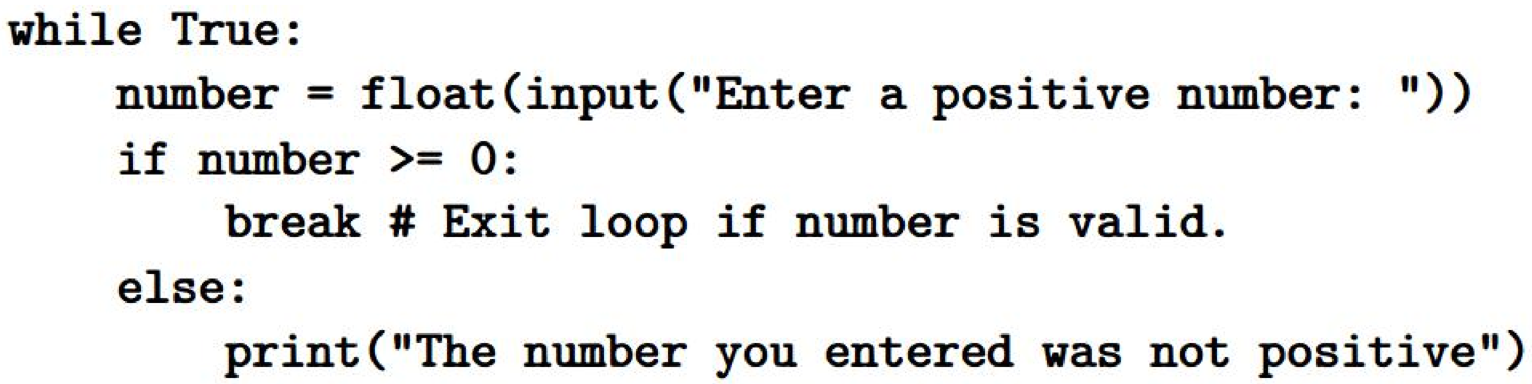


```
number = -1  # Start with an illegal value to get into the loop.
while number < 0:
    number = float(input("Enter a positive number: "))
    if number < 0:
        print("The number you entered was not positive")
```

The validity check gets repeated in two places!

Other Common Structures: Post-Test Loop

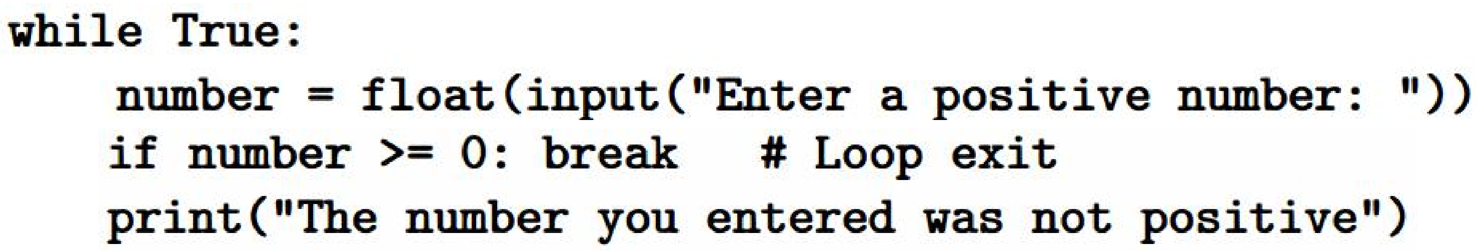
- Adding the warning to the **`break`** version only adds an **`else`** statement:



```
while True:
    number = float(input("Enter a positive number: "))
    if number >= 0:
        break # Exit loop if number is valid.
    else:
        print("The number you entered was not positive")
```

Other Common Structures: Loop and a Half

- Some programmers prefer the following approach:

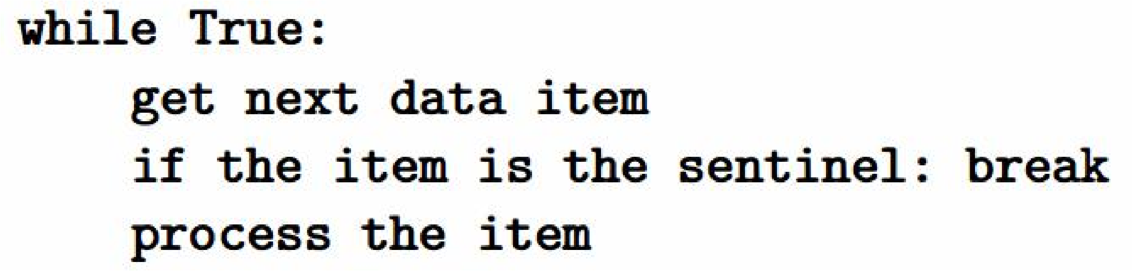


```
while True:
    number = float(input("Enter a positive number: "))
    if number >= 0: break   # Loop exit
    print("The number you entered was not positive")
```

- Here the **loop exit** is in the **middle** of the loop body. This is what we mean by **a *loop and a half***.

Other Common Structures: Loop and a Half

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

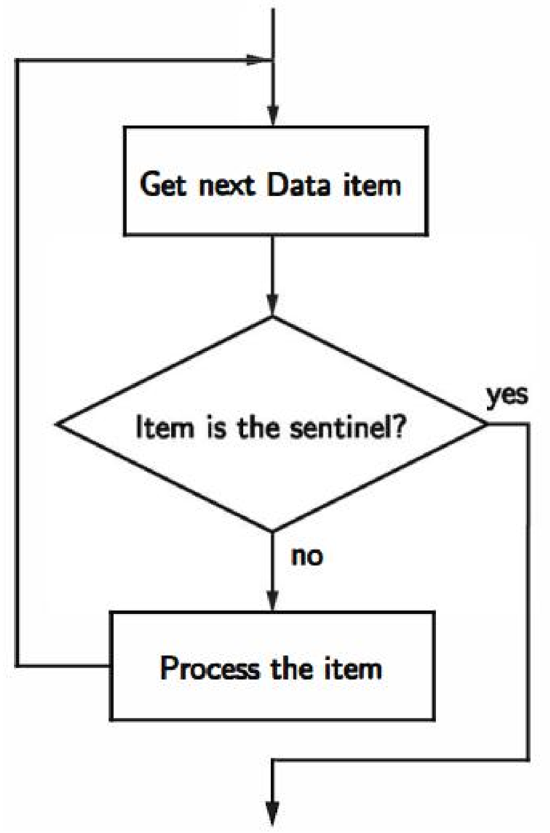


```
while True:
    get next data item
    if the item is the sentinel: break
    process the item
```

Other Common Structures:

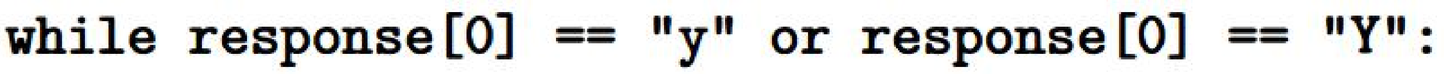
Loop and a Half

- The logic of a loop is easily lost when there are multiple exits.
- However, there are times when even this rule should be broken to provide the most elegant solution to a problem.

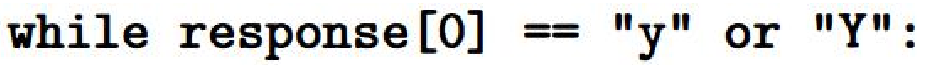


Other Common Structures: Boolean Expressions as Decisions

- Suppose you’re writing a program that keeps going as long as the user enters a response that starts with ‘y’ *(To allow the user to type either an upper- or lowercase response).*



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



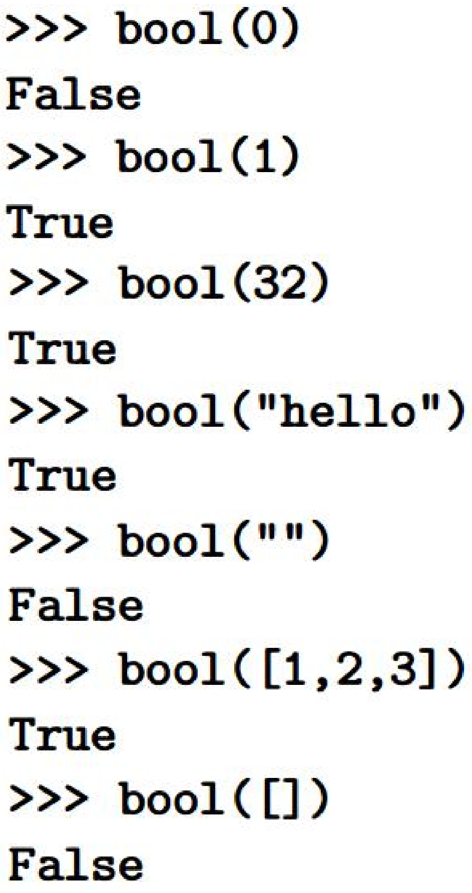
```
while response[0] == "y" or "Y":
```

✗

Other Common Structures: Boolean Expressions as Decisions

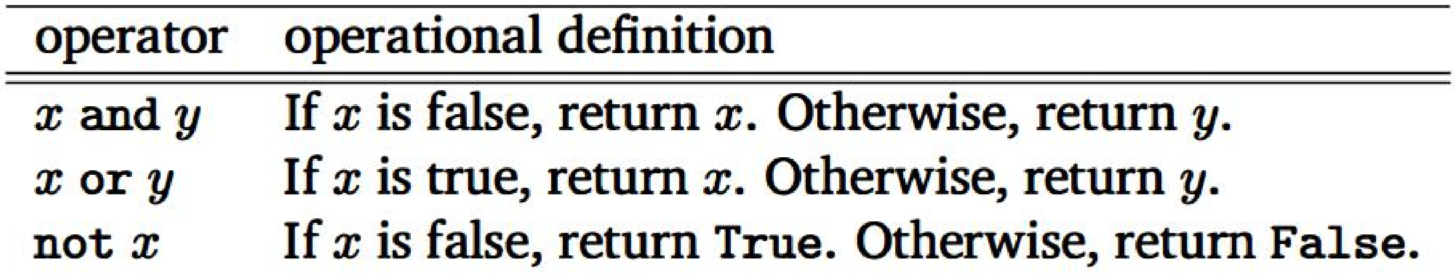
- Why?
- Python has a **`bool`** type that internally uses 1 and 0 to represent **`True`** and **`False`**, respectively.
- The Python condition operators, like **`==`**, always evaluate to a value of type **`bool`**.
- For **numbers** (int, float, and long ints), zero is considered **`False`**, anything else is considered **`True`**.
- An **empty sequence** is interpreted as **`False`** while any non-empty sequence is taken to mean **`True`**.

Other Common Structures: 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
```

Other Common Structures: Boolean Expressions as Decisions



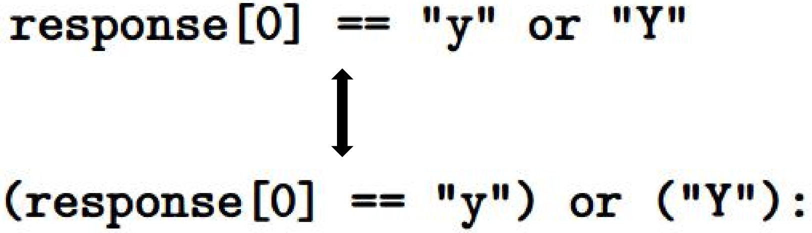
operator	operational definition
x and y	If x is false, return x. Otherwise, return y.
x or y	If x is true, return x. Otherwise, return y.
not x	If x is false, return True. Otherwise, return False.

Other Common Structures: Boolean Expressions as Decisions

- Python’s Booleans are ***short-circuit*** operators:
    - Consider ***x*** **`and`** ***y***. In order for this to be true, both *x* and *y* must be true.
    - 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.

Other Common Structures: Boolean Expressions as Decisions

- The Boolean operator is combining two operations.



```
response[0] == "y" or "Y"
```

```
(response[0] == "y") or ("Y"):
```

- By the operational description of **`or`**, this expression returns either **`True`**, if response[0] equals “y”, or “Y”, both of which are interpreted by Python as true.

Programming Exercises

- The 14th exercise on P281