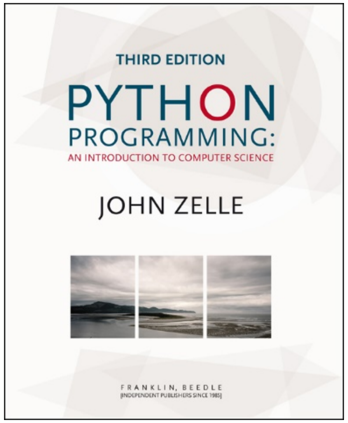
Python Programming: An Introduction To Computer Science



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 understand the programming patterns interactive loop and sentinel loop and their implementations using a Python `while` statement.

Objectives

- To understand the programming pattern end-of-file loop 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.

Objectives

- 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

- The `for` statement allows us to iterate through a sequence of values.
- ```
  for <var> in <sequence>:
      <body>
  ```
- The loop index variable `var` takes on each successive value in the sequence, and the statements in the body of the loop are executed once for each value.
  ```

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 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 Quick 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.

For Loops: A Quick Review

```
Input the count of the numbers, n
Initialize sum to 0
Loop n times
  Input a number, x
  Add x to sum
Output average as sum/n
```

For Loops: A Quick Review

```
# average1.py
#    A program to average a set of numbers
#    Illustrates counted loop with accumulator

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

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 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

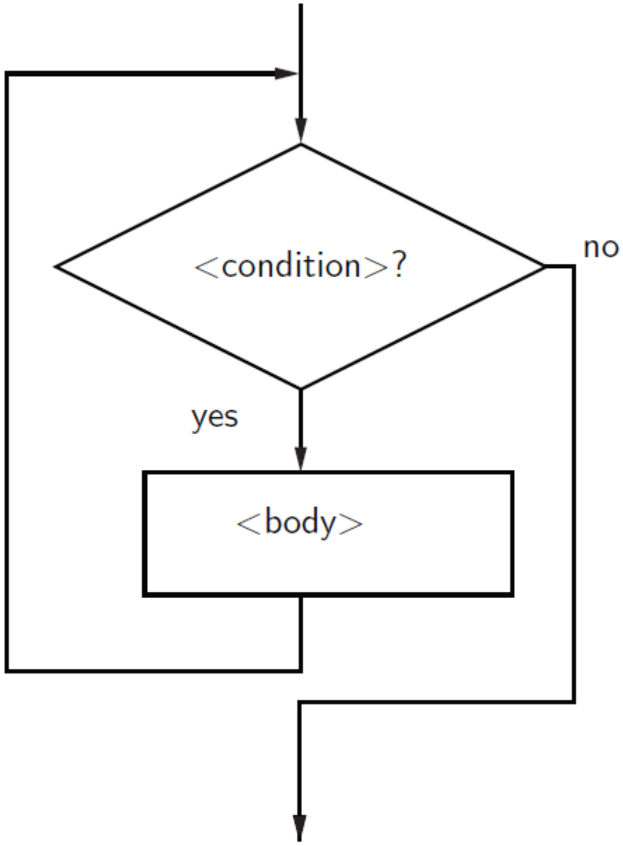
- We can’t use a definite loop unless we know the number of iterations ahead of time. We can’t know how many iterations we need until all the numbers have been entered.
- We need another tool!
- The *indefinite* or *conditional* loop keeps iterating until certain conditions are met.

Indefinite Loops

- ```
  while <condition>:
      <body>
  ```
- `condition` is a Boolean expression, just like in `if` statements. 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

- 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 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 code has the same output as this `for` loop:

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

Indefinite Loop

- 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 Loop

- The `while` statement is simple, but yet powerful and dangerous – they are a common source of program errors.
- ```
  i = 0
  while i <= 10:
      print(i)
  ```
- What happens with this code?
  ```

Indefinite Loop

- 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 Loop

- 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

- 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 sum and count:
- ```
  initialize sum to 0.0
  initialize count to 0
  set 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

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

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

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

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

- 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

- In our averaging example, assume we are averaging test scores.
- We can assume that there will be no score below 0, so a negative number will be the sentinel.

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 = float(input("Enter a number (negative to quit) >> "))
    while x >= 0:
        sum = sum + x
        count = count + 1
        x = float(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.
- There’s still a shortcoming – 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. Use a character-based sentinel.
- We could use the *empty string* (“”)!

Sentinel Loops

```
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
```

Sentinel 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 = float(xStr)
        sum = sum + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit) >> ")
    print("\nThe 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

- 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

```
# 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 + float(line)
        count = count + 1
    print("\nThe average of the numbers is", sum / count)
```

File Loops

- Many languages don't have a mechanism for looping through a file like this. 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

- ```
  line = infile.readline()
  while line != ""
      #process line
      line = infile.readline()
  ```
- 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, 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 + float(line)
        count = count + 1
        line = infile.readline()
    print("\nThe average of the numbers is", sum / count)
```

Nested Loops

- In the last chapter we saw how we could nest `if` statements. We can also nest loops.
- Suppose we change our specification to allow any number of numbers on a line in the file (separated by commas), rather than one per line.

Nested Loops

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

```
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

- ```
  for xStr in line.split(","):
      sum = sum + float(xStr)
      count = count + 1
  ```
- Notice that this `for` statement uses `line`, which is also the loop control variable for the outer loop.
  ```

Nested Loops

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

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    line = infile.readline()
    while line != "":
        # update sum and count for values in line
        for xStr in line.split(","):
            sum = sum + float(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

- `if` and `while` both use Boolean expressions.
- Boolean expressions evaluate to `True` or `False`.
- So far we've used Boolean expressions to compare two values, e.g. (`while x >= 0`)

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

- ```
  if p1.getX() == p2.getX():
      if p1.getY() == p2.getY():
          # points are the same
      else:
          # points are different
  else:
      # points are different
  ```
- Clearly, this is an awkward way to evaluate multiple Boolean expressions!
- Let's check out the three Boolean operators `and`, `or`, and `not`.
  ```

Boolean Operators

- The Boolean operators `and` and `or` are used to combine two Boolean expressions and produce a Boolean result.
- `<expr> and <expr>`
- `<expr> or <expr>`

Boolean Operators

- The and of two expressions is true exactly when both of the expressions are true.
- We can represent this in a *truth table*.

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

Boolean Expressions

- 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` for each combination.

Boolean Expressions

- The `or` of two expressions is true when either expression is true.

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

Boolean Expressions

- 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

- The `not` operator computes the opposite of a Boolean expression.
- `not` is a *unary* operator, meaning it operates on a single expression.

P	not P
T	F
F	T

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.

Boolean Operators

- Consider `a or not b and c`
- How should this be evaluated?
- The order of precedence, from high to low, is `not`, `and`, `or`.
- This statement is equivalent to
  `(a or ((not b) and c))`
- Since most people don't memorize the Boolean precedence rules, use parentheses to prevent confusion.

Boolean Operators

- To test for the co-location of two points, we could use an and.
- ```
  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?
- `scoreA == 15 or 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!
- ```
  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.
- ```
  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 >= 15 and a - b >= 2) or (b >= 15 and b - a >= 2)`
- `(a >= 15 or b >= 15) and abs(a - b) >= 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

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.

Boolean Algebra

- Anything `or`ed with true is true:
  `a or true == true`
- Both `and` and `or` distribute:
  `a or (b and c) == (a or b) and (a or c)`
  `a and (b or c) == (a and b) or (a and c)`
- Double negatives cancel out:
  `not(not a) == a`
- DeMorgan's laws:
  `not(a or b) == (not a) and (not b)`
  `not(a and b) == (not a) or (not b)`

Boolean Algebra

- We can use these rules to simplify our Boolean expressions.
- ```
  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:

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

Boolean Algebra

- This becomes:

```
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

- Sometimes it's easier to figure out 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

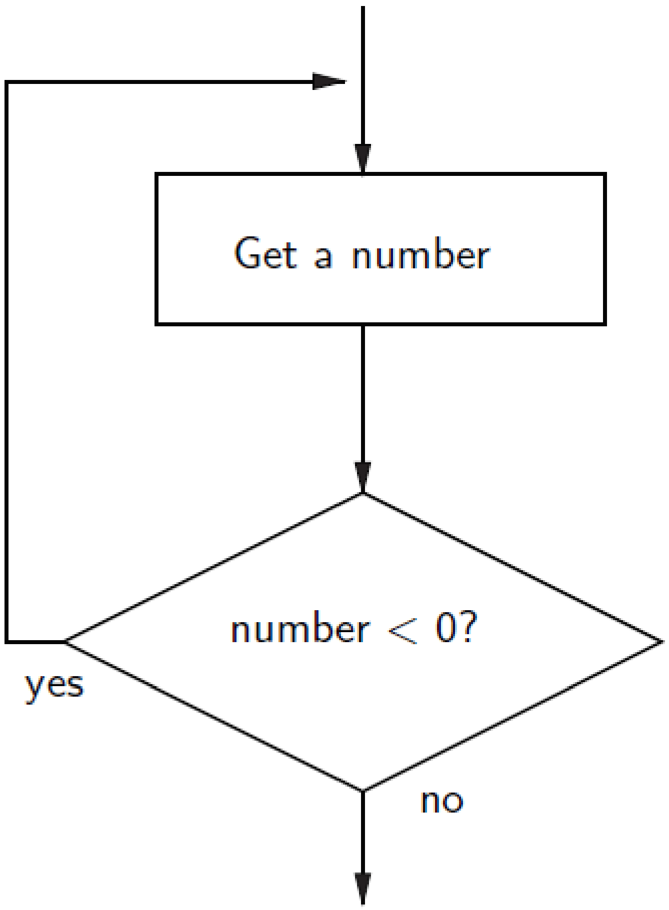
- The `if` and `while` can be used to express every conceivable algorithm.
- For certain problems, an alternative structure can be convenient.

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

- ```
  repeat
      get a number from the user
  until number is >= 0
  ```


  ```

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

- We seed the loop condition so we’re guaranteed to execute the loop once.
- ```
  number = -1       # start with an illegal value
  while number < 0: # to get into the loop
      number = float(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.

Post-Test Loop

- The same algorithm implemented with a `break`:

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

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

Post-Test Loop

- When the value of *x* is nonnegative, the `break` statement executes, which terminates the loop.
- If the body of an `if` is only one line long, you can place it right after the `:`!
- Wouldn’t it be nice if the program gave a warning when the input was invalid?

Post-Test Loop

- In the `while` loop version, this is awkward:

```
number = -1
while number < 0:
    number = float(input("Enter a positive number: "))
    if number < 0:
        print("The number you entered was not positive")
```

- We're doing the validity check in two places!

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 x >= 0:
        break # Exit loop if number is valid
    else:
        print("The number you entered was not positive.")
```

Loop and a Half

- Stylistically, some programmers prefer the following approach:

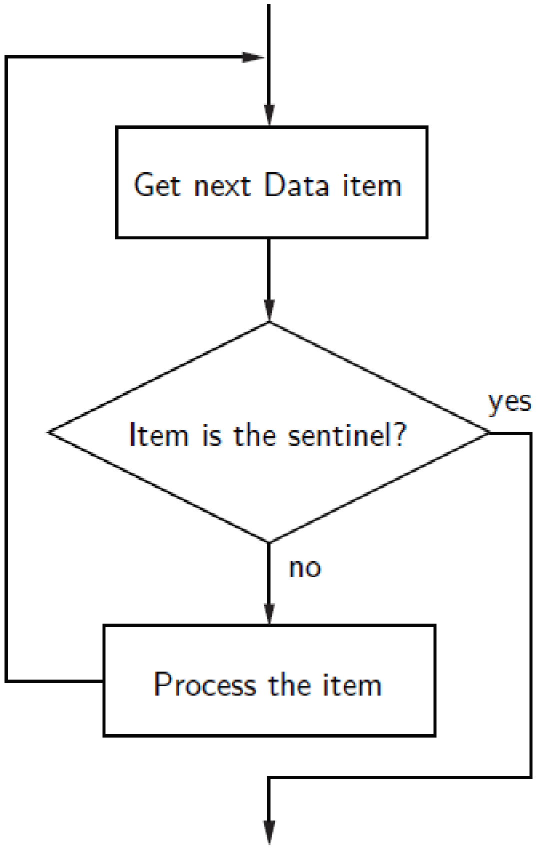
```
while True:
    number = float(input("Enter a positive number: "))
    if x >= 0: break # Loop exit
    print("The number you entered was not positive")
```

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

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
  ```
- This method is faithful to the idea of the sentinel loop, the sentinel value is not processed!
  ```

Loop and a Half



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 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:

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

Boolean Expressions as Decisions

- Be careful! You can’t take shortcuts:
```
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 (int, float, and long ints), zero is considered `False`, anything else is considered `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

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.

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" or "Y"`
- The Boolean operator is combining two operations.
- Here's an equivalent expression:
`(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.

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

- ```
  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!

```
ans = input("What flavor fo 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!

```
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!

Example: A Simple Event Loop

- Modern programs incorporating graphical user interfaces (GUIs) are generally written in an event-driven style.
- The program displays a graphical user interface and then “waits” for the user events such as clicking on a menu or pressing a key on the keyboard.

Example: A Simple Event Loop

- The mechanism that drives this style of program is a so-called *event loop*.

```
Draw the GUI
While True:
   get next event
   if event is “quit signal”
      break
   process the event
clean up and exit
```

Example: A Simple Event Loop

- Consider a program that opens a graphics window and allows the user to change its color by typing different keys – “r” for red, etc.
- The user can quit at any time by pressing “q”

Example: A Simple Event Loop

```
# event_loop1.py -- keyboard-driven color changing window

from graphics import *

def main():
    win = GraphWin("Color Window", 500, 500)

    # Event Loop: handle key presses until user
    # presses the "q" key.
    while True:
        key = win.getKey()
        if key == "q": # loop exit
            break
```

Example: A Simple Event Loop

```
    #process the key
    if key == "r":
        win.setBackground("pink")
    elif key == "w":
        win.setBackground("white")
    elif key == "g":
        win.setBackground("lightgray")
    elif key == "b":
        win.setBackground("lightblue")

# exit program
win.close()
```

Example: A Simple Event Loop

- Each time through the event loop this program waits for the user to press a key on the keyboard.
- A more flexible user interface might allow the user to interact in various ways – typing on the keyboard, selecting a menu item, hovering over an icon, clicking a button, etc.

Example: A Simple Event Loop

- The event loop would have to check for multiple types of events rather than waiting for one specific event.
- Let's add the ability for the user to click the mouse to position and type strings into the window, a souped-up version of chapter 4's click-and-type example.

Example: A Simple Event Loop

- When mixing mouse and keyboard control, we run into a problem...
  - We can no longer rely on `getMouse` and `getKey`!
  - Why????
  - If we call `win.getKey` then the program pauses until the user types a key. What if the user decided to use the mouse instead?

Example: A Simple Event Loop

- These are *modal* input methosd, because they lock the user into a certain mode of interaction.
- We can make the event loop nonmodal (i.e. the user is in control of how to interact) by using `checkKey` and `checkMouse`.

Example: A Simple Event Loop

- These methods are similar to `getKey` and `getMouse`, but they don't wait for the user to do something.
- `key = win.checkKey()`
- Python will check to see whether a key has been pressed
  - If one has, it will return a string that represents that key.
  - If not, it returns the empty string.

Example: A Simple Event Loop

```
Draw the GUI
while True:
   key = checkKey()
   if key is quit signal: break
   if key is valid key:
      process key

   click = checkMouse()
   if click is valid:
      process click
Clean up and Exit
```

Example: A Simple Event Loop

- Each time through the loop the program looks for a key press or a mouse click and handles them appropriately.
- If there is no event to process, it does not wait, instead it just spins around the loop and checks again!

Example: A Simple Event Loop

```
# event_loop2.py -- color changing window

from graphics import *

def handleKey(k, win):
    if k == "r":
         win.setBackground("pink")
    elif k == "w":
        win.setBackground("white")
    elif k == "g":
        win.setBackground("lightgray")
    elif k == "b":
        win.setBackground("lightblue")
```

Example: A Simple Event Loop

```
def handleClick(pt, win):
    pass
```

- Since we haven't decided what to do with mouse clicks yet, `handleClick` has a `pass` statement.
- A `pass` statement does nothing – it simply fills in the spot where Python is syntactically expecting a statement.

Example: A Simple Event Loop

```
def main():
    win = GraphWin("Click and Type", 500, 500)
    # Event Loop: handle key presses and mouse clicks until user
    #    presses the "q" key.
    while True:
        key = win.checkKey()
        if key == "q": # loop exit
            break
        if key:
            handleKey(key, win)
        pt = win.checkMouse()
        if pt:
            handleClick(pt, win)
    win.close()
```

Example: A Simple Event Loop

- When there is no input, `checkKey()` and `checkMouse()` both return values that Python interprets as false.
- We can type `if key:` rather than `if key != ""`
  - You can read this as “If I got a key…”

Example: A Simple Event Loop

- Clicking on the window initiates a basic 3 step algorithm:
  1. Display an `Entry` box where the user clicked.
  2. Allow the user to type text into the box; typing is terminated by hitting the return key (<Enter>).
  3. The `Entry` box disappears and the typed text appears directly in the window.

Example: A Simple Event Loop

- In step 2, we want the text the user types to show up in the `Entry` box, but we don't want them interpreted as top-level commands (a 'q' here shouldn't quit!)
- The program should be modal – it should switch to text-entry mode until the user hits a return key.

Example: A Simple Event Loop

- How do we do this?
  - Inside the main loop we nest another loop that consumes all the keypresses until the user hits the return key.
  - Once the return key is pressed, the inner loop terminates and the program continues on.

Example: A Simple Event Loop

```
def handleClick(pt, win):
    # create an Entry for user to type in
    entry = Entry(pt, 10)
    entry.draw(win)

    # Go modal: loop until user types Return key
    while True:
        key = win.getKey()
        if key == "Return":
            break
```

Example: A Simple Event Loop

```
# undraw the entry and create and draw Text
entry.undraw()
typed = entry.getText()
Text(pt, typed).draw(win)

# clear (ignore) any mouse click that occurred
#   during text entry
win.checkMouse()
```

Example: A Simple Event Loop

- The body of this loop literally does nothing.
- It could have been rewritten as

```
while win.getKey() != "Return":
   pass
```

- The last line ensures the text entry is truly modal.

Example: A Simple Event Loop

- Mouse clicks before the return key was pressed should be ignored.
- Since `checkMouse` only returns mouse clicks that have happened since the last call to `checkMouse`, calling the function here has the effect of clearing any click that may have occurred but not yet been checked for.