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.

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 = int(input("Enter a number >> "))
        sum = sum + x
    print("The average of the numbers is", sum / n)

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

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.

---

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:

```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 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.

- \( i = 0 \)
  
  ```python
  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

```python
# 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 = input("Enter a number >> ")
        sum = sum + x
        count = count + 1
        moredata = raw_input("Do you have more numbers (yes or no)? ")
    print "The 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.

```python
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.

```python
# 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 = input("Enter a number (negative to quit) >> ")
    while x >= 0:
        sum = sum + x
        count = count + 1
        x = input("Enter a number (negative to quit) >> ")
    print "The average of the numbers is", sum / count
```

Python Programming, 1/e 27
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 (""")!

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

```
# averaged4.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 = raw_input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = raw_input("Enter a number (<Enter> to quit) >> ")
    print "The average of the numbers is", sum / count

Python Programming, 1/e
33
```

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 3.3833333333
```

Python Programming, 1/e
34
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.

```python
# averages.py
# Computes the average of numbers listed in a file.

def main():
    fileName = raw_input("What file are the numbers in? ")
    inFile = open(fileName,'r')
    sum = 0.0
    count = 0
    for line in inFile.readlines():
        sum += eval(line)
        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, ""

```python
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 = raw_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