Repetition Structures - Loops

Reading – Chapter 4

Topics

- Introduction to Repetition Structures
- The `while` Loop: a Condition-Controlled Loop
- The `for` Loop: a Count-Controlled Loop
- Calculating a Running Total
- Sentinels
- Input Validation Loops
- Nested Loops

Introduction to Repetition Structures

- Often have to write code that performs the same task multiple times
  - Disadvantages to duplicating code
    - Makes program large
    - Time consuming
    - May need to be corrected in many places
- <u>Repetition structure</u>: makes computer repeat included code as necessary
  - Includes condition-controlled loops and count-controlled loops

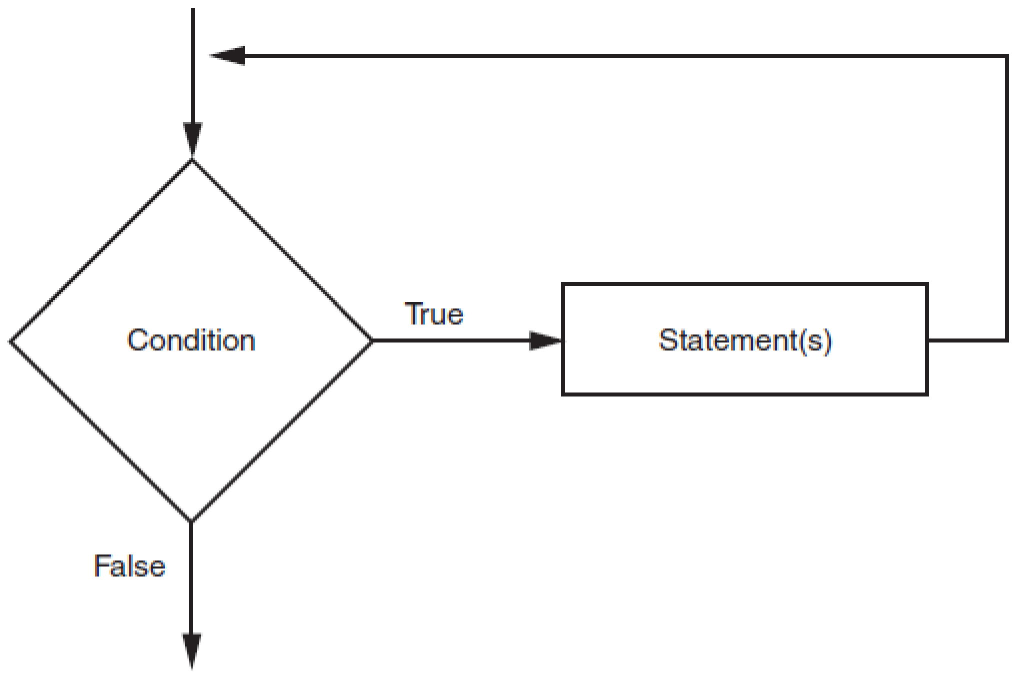
The `while` Loop: a Condition-Controlled Loop

- `while` loop: while condition is true, do something
  - Two parts:
    - Condition tested for true or false value
    - Statements repeated as long as condition is true
  - In flow chart, line goes back to previous part
  - General format:

```
while condition:
    statements
```

The `while` Loop: a Condition-Controlled Loop (cont'd.)

Figure 4-1 The logic of a `while` loop



The `while` Loop: a Condition-Controlled Loop (cont'd.)

- In order for a loop to stop executing, something has to happen inside the loop to make the condition false
- Iteration: one execution of the body of a loop
- `while` loop is known as a *pretest* loop
  - Tests condition before performing an iteration
    - Will never execute if condition is false to start with
    - Requires performing some steps prior to the loop

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.

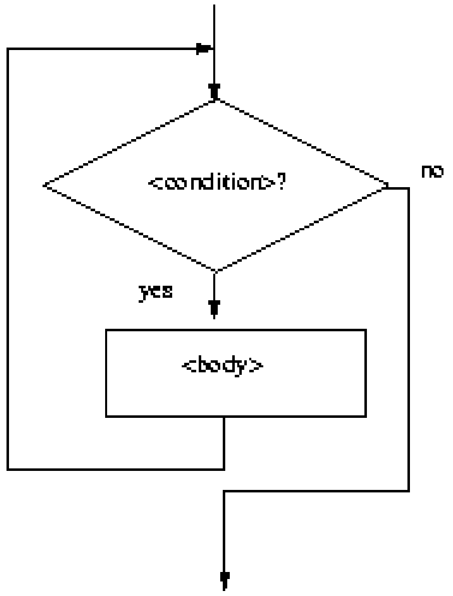
Indefinite Loops

- 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 (later...):

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

Simple Example

```
total = 0
while total < 25:
    num = float(input('Enter number '))
    if num > 0:
        total = total + num**2
```

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`.

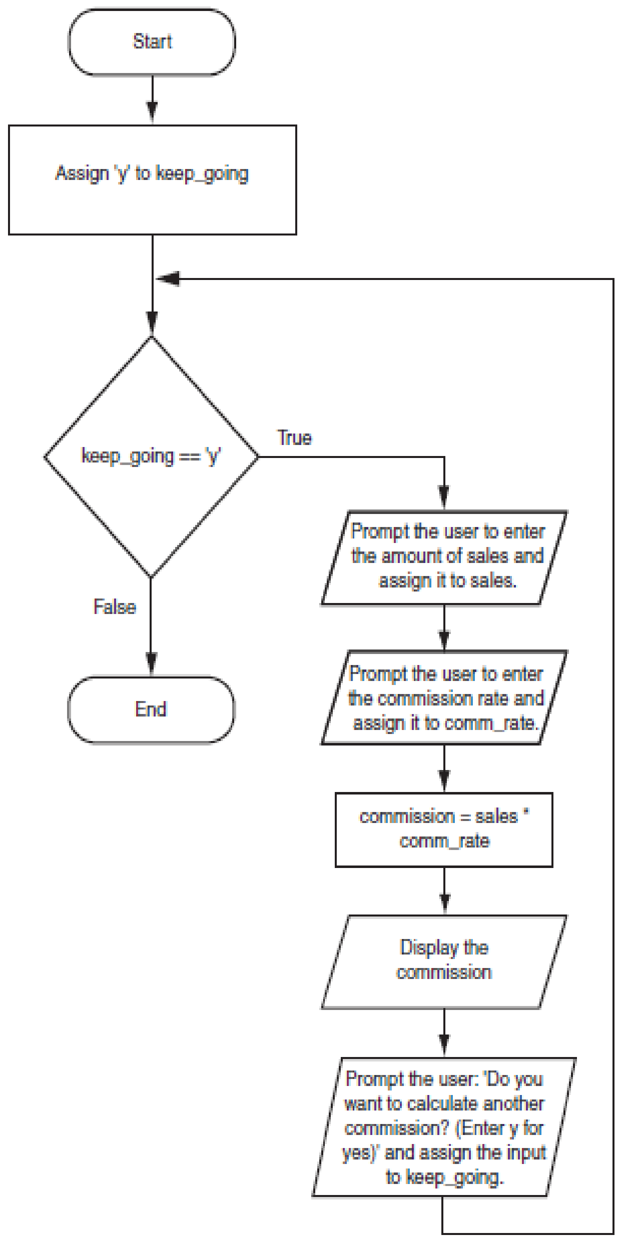
Interactive Loops

- 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
  ```

```
# This program calculates sales commissions.

# Create a variable to control the loop.
    keep_going = 'y'

    # Calculate a series of commissions.
    while keep_going == 'y':
        # Get a salesperson's sales and commission rate.
        sales = float(input('Enter the amount of sales: '))
        comm_rate = float(input('Enter the commission rate: '))

        # Calculate the commission.
        commission = sales * comm_rate

        # Display the commission.
        print('The commission is $', \
              format(commission, ',.2f'), sep='')

        # See if the user wants to do another one.
        keep_going = input('Do you want to calculate another ' +
                           'commission (Enter y for yes): ')
```

Figure 4-3 Flowchart for Program 4-1


Start
Assign 'y' to keep_going
keep_going == 'y'
True
False
End
Prompt the user to enter the amount of sales and assign it to sales.
Prompt the user to enter the commission rate and assign it to comm_rate.
commission = sales * comm_rate
Display the commission
Prompt the user: 'Do you want to calculate another commission? (Enter y for yes)' and assign the input to keep_going.

Infinite Loops

- Loops must contain within themselves a way to terminate
  - Something inside a `while` loop must eventually make the condition false
- <u>Infinite loop</u>: loop that does not have a way of stopping
  - Repeats until program is interrupted
  - Occurs when programmer forgets to include stopping code in the loop

Infinite Loops

```
# This program demonstrates an infinite loop.

# Create a variable to control the loop.
    keep_going = 'y'

    # Warning! Infinite loop!
    while keep_going == 'y':
        # Get a salesperson's sales and commission rate.
        sales = float(input('Enter the amount of sales: '))
        comm_rate = float(input('Enter the commission rate: '))

        # Calculate the commission.
        commission = sales * comm_rate

        # Display the commission.
        print('The commission is $', \
              format(commission, ',.2f'), sep='')
```

Note that there is nothing in the loop that allows the user to change the value of control variable

The `for` Loop: a Count-Controlled Loop

- Count-Controlled loop: iterates a specific number of times
  - Use a `for` statement to write count-controlled loop
    - Designed to work with sequence of data items
      - Iterates once for each item in the sequence
    - General format:

```
for variable in [val1, val2, etc]:
    statements
```

val1, val2, val3, etc, will be assigned to the target variable

target variable

- Target variable: the variable which is the target of the assignment at the beginning of each iteration

Python programmers commonly refer to the variable that is used in the for clause as the *target variable* because it is the target of an assignment at the beginning of each loop iteration

The first line is called the *for clause*. In the for clause, *variable* is the name of a variable (created inside of the for clause). Inside the brackets a sequence of values appears, with a comma separating each value. In Python, a comma-separated sequence of data items that are enclosed in a set of brackets is called a *list*.

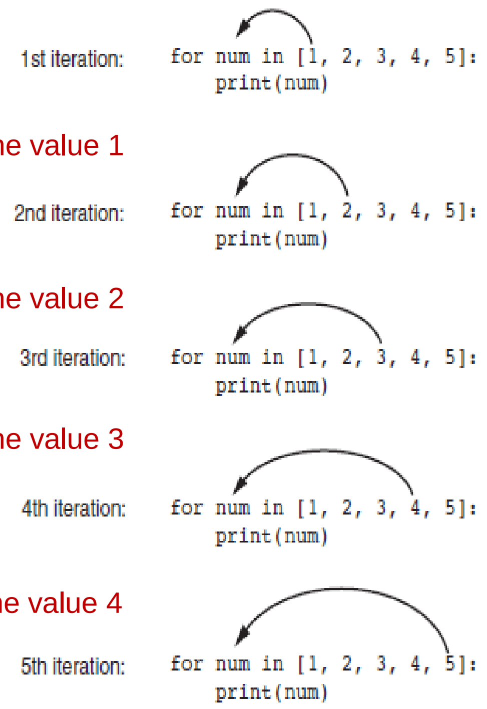
The `for` Loop con’t

```
# This program demonstrates a simple for loop
# that uses a list of numbers.

print('I will display the numbers 1 through 5.')
for num in [1, 2, 3, 4, 5]:
      print(num)
```

Program Output

```
I will display the numbers 1 through 5.
1
2
3
4
5
```

Figure 4-4 The `for` loop



1st iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

num is assigned the value 1

2nd iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

num is assigned the value 2

3rd iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

num is assigned the value 3

4th iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

num is assigned the value 4

5th iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

num is assigned the value 5

The for statement executes in the following manner: The *variable* is assigned the first value in the list, and then the statements that appear in the block are executed. Then, *variable* is assigned the next value in the list, and the statements in the block are executed again. This continues until *variable* has been assigned the last value in the list. Program 5-5 shows a simple example that uses a for loop to display the numbers 1 through 5.

The `for` Loop con’t

```
# This program also demonstrates a simple for
# loop that uses a list of numbers.

print('I will display the odd numbers 1 through 9.')
    for num in [1, 3, 5, 7, 9]:
        print(num)
```

Program Output

```
I will display the odd numbers 1 through 9.
1
3
5
7
9
```

```
# This program also demonstrates a simple for
# loop that uses a list of strings.

def main():
    for name in ['Winken', 'Blinken', 'Nod']:
        print(name)

# Call the main function.
main()
```

During each iteration, the variable *name* is assigned to target variables in the list which are *strings.*

Program Output

```
Winken
Blinken
Nod
```

Using the `range` Function with the `for` Loop

- The `range` function simplifies the process of writing a `for` loop
  - `range` returns an iterable object
    - Iterable: contains a sequence of values that can be iterated over
- `range` characteristics:
  - One argument: used as ending limit

```
for num in range(5):
print(num)
```

Notice that instead of using a list of values, we call to the range function passing 5 as an argument. In this statement the range function will generate an iterable sequence of integers in the range of 0 up to (but not including) 5.

  - Two arguments: starting value and ending limit

```
for num in range(1, 5):
print(num)
```

This code will display the following:

```
1
2
3
4
```

  - Three arguments: third argument is step value

```
for num in range(1, 10, 2):
print(num)
```

This code will display the following:

```
1
3
5
7
9
```

For Loops: Examples

- 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: Accumulator

- 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: Accumulator

- 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: Compute Average

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

eval() is another Python function to convert input but we won't use it in this course.

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

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

For Loops: Compute Average

```
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
```

The Augmented Assignment Operators

- In many assignment statements, the variable on the left side of the = operator also appears on the right side of the = operator
- Augmented assignment operators: special set of operators designed for this type of job
  - Shorthand operators

The Augmented Assignment Operators (cont’d.)

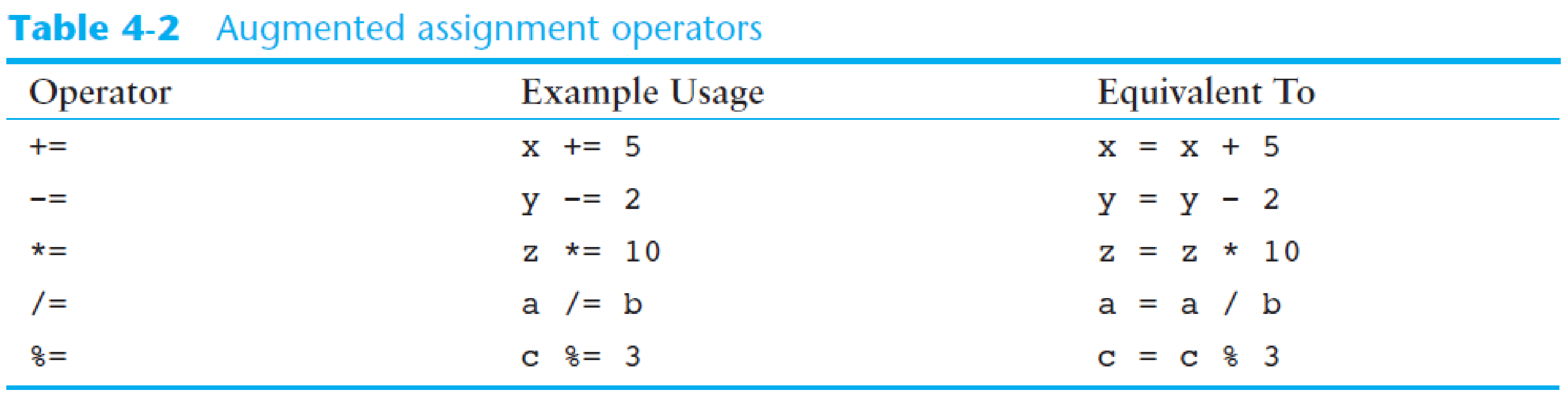


Table 4-2 Augmented assignment operators

Operator	Example Usage	Equivalent To
`+=`	`x += 5`	`x = x + 5`
`-=`	`y -= 2`	`y = y - 2`
`*=`	`z *= 10`	`z = z * 10`
`/=`	`a /= b`	`a = a / b`
`%=`	`c %= 3`	`c = c % 3`

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

equivalent to i= i + 1.

Using the Target Variable Inside the Loop

```
# This program uses a loop to display a
# table showing the numbers 1 through 10
# and their squares.

# Print the table headings.
    print('Number\tSquare')
    print('---------------------------')

    # Print the numbers 1 through 10
    # and their squares.
    for number in range(1, 11):
        square = number**2
        print(number, '\t', square)
```

Program Output

```
Number          Square
----------------------------------
1               1
2               4
3               9
4               16
5               25
6               36
7               49
8               64
9               81
10              100
```

Nested Loops

- Nested loop: loop that is contained inside another loop
  - Example: analog clock works like a nested loop
    - Hours hand moves once for every twelve movements of the minutes hand: for each iteration of the "hours," do twelve iterations of "minutes"
    - Seconds hand moves 60 times for each movement of the minutes hand: for each iteration of "minutes," do 60 iterations of "seconds"

Nested Loops: Clock Example

```
for hours in range(24):
        for minutes in range(60):
                for seconds in range(60):
                        print(hours, ':', minutes, ':', seconds)
```

This code's output would be:

```
0:0:0
0:0:1 0:0:2
```

(The program will count through each second of 24 hours.)

```
23:59:59
```

The innermost loop will iterate 60 times for each iteration of the middle loop. The middle loop will iterate 60 times for each iteration of the outermost loop. When the outermost loop has iterated 24 times, the middle loop will have iterated 1,440 times and the innermost loop will have iterated 86,400 times!

More Examples

```
# This program displays a rectangular pattern
# of asterisks.
rows = int(input('How many rows? '))
cols = int(input('How many columns? '))

for r in range(rows):
    for c in range(cols):
        print('*', end='')
    print()
```

Program Output (with input shown in bold)

```
How many rows?  5
How many columns? 10
**********
**********
**********
**********
**********
```

```
# This program displays a triangle pattern.
BASE_SIZE = 8

for r in range(BASE_SIZE):
    for c in range(r + 1):
        print('*', end='')
    print()
```

Program Output

```
*
**
***
****
*****
******
*******
********
```

Nested Loops (cont'd.)

- Key points about nested loops:
  - Inner loop goes through all of its iterations for each iteration of outer loop
  - Inner loops complete their iterations faster than outer loops
  - Total number of iterations in nested loop:

```
number_iterations_inner  x  number_iterations_outer
```

Loops Activity

- Get in teams of two
  - Do the loops Activity (Posted on Canvas)